

Politecnico di Torino

Politecnico di Torino

Corso di Laurea Magistrale in Cybersecurity

A.a. 2025/2026

Graduation Session September 2026

Privacy-Preserving Federated Learning with Homomorphic Encryption on GPU

Supervisors:

Valentino Peluso

Erich Malan

Erkay Savas

Candidate:

Jacopo Giglioli

Abstract

Federated Learning is a distributed machine learning paradigm that enables multiple clients to jointly train a global model while keeping each client's dataset local and private. In each training round, clients download the current global model, fine-tune it locally using their data and send the resulting updates back to the server, which aggregates them to produce a new version of the global model. Although not transmitting clients' raw data already represents a significant improvement for privacy, some threats still exist, as an attacker can still infer sensitive information about clients' datasets from the shared updates.

Homomorphic Encryption, which enables computation on encrypted data without requiring prior decryption, offers a robust solution to this concern. The communication channel between clients and server is encrypted, preventing an attacker from reading the updates shared by the nodes. In this scenario clients send their updates encrypted and the server performs the aggregation phase homomorphically. Unfortunately, HE introduces important computational and communication overhead.

The goal of this thesis is to quantify the computational and communication overhead introduced by the use of HE in an FL workflow. We additionally assess the impact of HE on the FL framework by adopting a new library in the simulator, which exploits GPU acceleration to speed up HE computations: HEonGPU.

Experiments show that, when TenSEAL, a popular HE library, is employed, server-side aggregation time increases by a factor of 35x to 120x, depending on the encryption parameters, compared to the plaintext scenario. The total amount of data exchanged throughout the training process also grows by a factor of 8x to 21x, depending on the encryption parameters, compared to the scenario without HE.

On the other hand, thanks to GPU acceleration, the HEonGPU scenario achieves a speedup of 5x to 14x in aggregation time, 52x to 152x in decryption time and 62x to 84x in encryption time compared to the TenSEAL experiments, depending on the encryption parameters.

Table of Contents

List of Tables IV

List of Figures V

Glossary VI

1 Introduction 1
- 1.1 Context: Federated Learning and Homomorphic Encryption 1
- 1.2 Problem Statement: HE Overhead in Federated Learning 2
- 1.3 Proposed Solution: Adopting HEonGPU for GPU-Accelerated HE . 4
- 1.4 Results Obtained 4
- 1.5 Thesis Structure 5

2 Background 6
- 2.1 The CKKS Scheme 6
  - 2.1.1 The Ring Learning With Errors (RLWE) problem 6
  - 2.1.2 Algebraic Preliminaries 7
  - 2.1.3 The CKKS algorithm 9
- 2.2 The Federated Averaging Algorithm 18

3 Related Works 21
- 3.1 Alternative Fully Homomorphic Encryption Schemes 21
- 3.2 Fully Homomorphic Encryption Libraries 25

4 Method 26
- 4.1 System Architecture and Threat Model 26
  - 4.1.1 Threat Model 27
- 4.2 Federated Learning Training Protocol 29
  - 4.2.1 Data Partitioning across Clients 29
  - 4.2.2 Local Training 30
  - 4.2.3 Model Aggregation 32

4.2.4 Global Model Evaluation . . . 34
4.3 Research Questions and Experimental Scenarios . . . 36
4.4 CKKS Parameter Configuration . . . 38
4.4.1 Polynomial Modulus Degree . . . 38
4.4.2 Coefficient Modulus Chain . . . 39
4.4.3 Scaling Factor . . . 40

5 Experimental Setup 41
5.1 Dataset: CIFAR-10 . . . 41
5.1.1 Preprocessing . . . 42
5.2 Models . . . 42
5.2.1 Convolutional Neural Networks . . . 43
5.2.2 Residual Networks . . . 45
5.3 Evaluation Metrics . . . 50
5.4 Parameters . . . 53

6 Results 55
6.1 Plaintext Scenario . . . 55
6.2 TenSEAL Scenario . . . 59
6.2.1 Evaluation of the Temporal and Communication Overhead Introduced by TenSEAL . . . 65
6.3 HEonGPU Scenario . . . 68
6.3.1 Accuracy and Training Time . . . 69
6.3.2 Communication Cost . . . 71
6.3.3 Computational Costs . . . 73
6.3.4 Impact of Number of Global Rounds and Data Distribution on Model Performance . . . 78

7 Conclusions 80

List of Tables

1.1 Computational and Communication Overhead introduced by HE . . 3

3.1 Overview of FHE Libraries 25

5.1 Architecture of the CNN5 model 45
5.2 Architecture of a `BasicBlock`. 47
5.3 Federated Learning and Encryption Hyperparameters used in the Experiments 53

6.1 Plaintext Scenario: `round_clients` and Model Architecture sweep . 56
6.2 TenSEAL Scenario : n and q_{chain} sweep, $\Delta = 2^{30}$ 61
6.3 TenSEAL Scenario : n and q_{chain} sweep, $\Delta = 2^{40}$ 61
6.4 TenSEAL Benchmark 64
6.5 HEonGPU Scenario: Accuracy and Training Time 69
6.6 HEonGPU Benchmark 77

List of Figures

2.1 End-to-end pipeline of the CKKS encryption scheme. 12

3.1 Overview of FHE Schemes and Their Properties 24

4.1 Workflow of Federated Learning with Homomorphic Encryption . . 33

5.1 Illustration of a convolution filter applied to a multi-channel input . 44
5.2 Residual Block with Skip Connection 48
5.3 Architecture of the ResNet20 model 49

6.1 Plaintext Scenario: Further `round_clients` sweep with fixed model architecture 58
6.2 Impact of Homomorphic Encryption on Aggregation Time 66
6.3 Impact of Homomorphic Encryption on Exchanged Data 67
6.4 HEonGPU vs TenSEAL: Total Exchanged Data over All Training Rounds 71
6.5 HEonGPU vs TenSEAL: Aggregation Time 73
6.6 HEonGPU vs TenSEAL: Encryption Time 74
6.7 HEonGPU vs TenSEAL: Decryption Time 75
6.8 Learning curve with $\alpha = 0.3$ 78
6.8 Learning curve with $\alpha = 10$ 79

Glossary

FL
: Federated Learning

HE
: Homomorphic Encryption

RLWE
: Ring Learning With Errors

LWE
: Learning With Errors

CNN
: Convolutional Neural Network

FHE
: Fully Homomorphic Encryption

Chapter 1

Introduction

1.1 Context: Federated Learning and Homomorphic Encryption

Federated Learning Federated Learning is a distributed learning paradigm that enables the training of a global model using heterogeneous data from various clients. By aggregating information from participants, this approach develops an algorithm that reflects the unique features of every node.

The core innovation of this approach consists of keeping data on owners' devices. In fact, it is not necessary to share raw records with the central server responsible for training the network.

In FL, there is a global model maintained by the server whose specifications are shared at the beginning of each round with the clients participating in that training round. Then, each node performs a local refinement of the model using its own dataset. At the end of this fine-tuning process , every participant sends to the server only the parameter updates, that is, the difference between the initially received configuration and the one obtained after local training . Consequently, raw data are never transmitted; instead, only information related to network parameter updates is exchanged. Once the server has received these local updates, it performs an aggregation phase of all contributions, obtaining a new version of the global model, ready for the next optimization round.

This approach represents a significant advance in data protection and user privacy.

However FL strategy alone does not resolve all confidentiality concerns. Some threats like gradient inversion [1] or inference attacks [2] can still permit malicious adversaries to retrieve private information from user data. These attacks can be mounted only by having access to local updates in transit between clients and server.

Homomorphic Encryption Homomorphic Encryption is a cryptographic technique that allows computation on encrypted data without the need to decrypt it beforehand. More specifically, a user who does not have access to the private key required to decrypt two ciphertexts can still add or multiply them (or perform any other homomorphic operation permitted by the scheme), while being sure that once the result of the computation is recovered, it will correspond to the sum or the product of the two related plaintexts.

Combining HE with FL can offer a solution to some of the security issues mentioned before. In this scenario, clients own a private key, which they use to encrypt the local model updates sent to the server. The central entity then performs aggregation in a homomorphic fashion, computing on encrypted data without being able to directly access the information shared by the nodes. This strategy enables us to treat the central server as an untrusted entity in our threat model. Additionally, the data transmitted over the network is protected against malicious users, who are unable to read it, thus protecting the system against eavesdropping attacks. The server always stores the global model in encrypted form and never has access to its plaintext representation. When the server shares the global model with the users, it sends the already encrypted model, thus providing protection also for this communication channel. The clients decrypt the model before performing the local training in clear, speeding up the entire process.

HE use is particularly recommended in sectors where privacy is essential. In industries that handle confidential data, such as healthcare [3] [4] or recommendation systems [5], HE must be taken into consideration among the solutions to prevent information leakage.

1.2 Problem Statement: HE Overhead in Federated Learning

However, HE protection comes with a cost. These types of algorithms are extremely resource-intensive. First, communication costs increase dramatically, as encrypted updates are considerably larger than their plaintext counterparts, with an increase ranging from 8.22× to 21.33× depending on the encryption parameters, according to our experiments. Furthermore, on the server side, homomorphic operations during the aggregation phase are slower than standard operations, resulting in an overhead between 35.29× and 119.54× depending on the encryption parameters for the total aggregation phase, according to our experiments. Finally, on the client side, nodes need to perform encryption before sending local updates and decryption after downloading the global model, routines that are not present in the plaintext scenario. These overheads pose significant challenges that need to be addressed in future research.

In addition, it is important to note that FL is often deployed in environments involving resource-constrained clients [6] [4], such as IoT devices [7]. These systems typically have limited computational power and resources, making them unsuitable for the execution of computationally heavy routines as the HE primitives. As a result, implementing HE on the clients'side is often challenging.

Table 1.1: Computational and Communication Overhead introduced by HE

	Aggregation Time (s)	Encryption Time (s)	Decryption Time (s)	Exchanged Data (GB)
Plaintext	0.28	0	0	8.92
HE	20.87	16.65	3.70	131.55
Increase	74.54×	/	/	14.75×

Table 1.1 shows the computational and communication overhead introduced by HE in the FL workflow. The experiments were conducted using the same training parameters to ensure a fair comparison between the two scenarios (100 clients, 100 global training rounds and 15 participating clients per round). The aggregation time refers to the total time spent by the server aggregating the clients' updates over all 100 rounds, while the encryption and decryption times refer to the average time a client spends encrypting or decrypting data over all 100 rounds. The TenSEAL library [8] was used in the HE scenario, implementing the CKKS encryption scheme [9], with the following parameters: $n = 8192$, $q_{\text{chain}} = [60,40,40]$ and $\Delta = 2^{30}$.

Although the the absolute execution costs introduced by HE may appear acceptable, it is important to note that these measurements were obtained in a small-scale experimental setting, involving a limited number of clients and training rounds. In real-world FL scenarios, where the number of participating devices and communication rounds can be larger, these costs may explode.

The main goal of this thesis is to quantify the temporal and computational overhead introduced by the use of HE in a FL workflow.

We will analyze the results of several experiments conducted using a simulator of a FL framework under various configurations. The first scenario considered is the plain scenario, where no HE algorithms are applied. Then, the same environment with the same training parameters will be evaluated using the TenSEAL library, where both the local updates sent by the clients and the global model parameters distributed by the server will be encrypted. These two scenarios will be compared across multiple metrics, such as the aggregation time required by the server to compute the new model after the local training or the volume of exchanged data between the nodes and the central entity.

1.3 Proposed Solution: Adopting HEonGPU for GPU-Accelerated HE

We aim to assess the impact of HE on the FL framework by implementing a promising new library in the simulator we are using, which exploits GPU acceleration to speed up HE computations, HEonGPU [10].

A Graphics Processing Unit (GPU) is a processor architecture optimized to achieve a high degree of parallelism. This technology is able to compute a large number of independent operations simultaneously.

However, GPU cores are much simpler than CPU ones and run at a lower clock frequency. A CPU has a lower number of cores, it is designed to minimize latency and perform more complex operations. On the other hand, a GPU consists of more cores and it is designed to maximize throughput, achieving the best performance when executing simple and repetitive operations on a large amount of data.

In HE schemes, operations are independent from each other and repetitive over large amounts of data, so they can be executed in parallel. This suits perfectly the use of GPUs.

Finally, also this last HEonGPU scenario will be compared with the TenSEAL one using the same encryption parameters, to evaluate the temporal and computational overhead reduction obtained by GPU acceleration.

1.4 Results Obtained

When quantifying the temporal and computational overhead introduced by the use of HE in a FL workflow, we measured an increase by a factor ranging from 35.29× to 119.54×, depending on the encryption parameters, in the aggregation time, that is the time the server spends aggregating the updates shared by the clients at the end of the local training phase in order to compute a new version of the global model. This is due to the fact that performing homomorphic operations on encrypted data is significantly more computationally expensive than performing the corresponding operations on plaintext data. We also observed an increase by a factor ranging from 8.22× to 21.33×, depending on the encryption parameters, in the total amount of exchanged data across all rounds during the whole training process. This can be explained by the fact that homomorphically encrypted data are considerably larger than their plaintext counterpart.

In addition, in a scenario where HE is employed to protect the communication channel between clients and the server, we found that nodes spend between 12.93s and 21.89s encrypting their updates before sharing them with the server and between 1.73s and 5.72s decrypting the global model's weights received from the server at the beginning of the local training phase, across 100 training rounds.

These measurements were performed using the TenSEAL library [8] implementing the CKKS encryption scheme for HE.

Moving to the scenario where the HEonGPU library [10] is employed to achieve GPU acceleration for HE, we measured that HEonGPU achieves a speedup ranging from 4.72× to 14.30× in the aggregation time compared to the scenario using TenSEAL to implement HE, depending on the encryption parameters. As for the encryption time and the decryption time, we observed a speedup factor ranging between 62.17× and 84.13×, and between 52.37× and 151.67× respectively, achieved by HEonGPU. At the same time, we found that the GPU-accelerated setting does not introduce additional communication costs, keeping the amount of exchanged data constant compared to the TenSEAL scenario.

1.5 Thesis Structure

The structure of this thesis is defined as follows:

- **Chapter 2: Background** introduces some fundamental concepts for our work, such as the CKKS encryption scheme used to implement HE in the communication channel and the FedAvg algorithm used during the FL training process.
- **Chapter 3: Related Works** presents Fully Homomorphic Encryption schemes other than CKKS and libraries implementing FHE other than TenSEAL.
- **Chapter 4: Method** shows how the tool we used to simulate FL works, together with our threat model. In addition, it contains our research questions.
- **Chapter 5: Experimental Setup** goes into the details of our setting configuration: the dataset used, the architecture of the trained model and the hyperparameters used during the training process.
- **Chapter 6: Results** presents the outcomes of our experiments, examining three different scenarios: the Plaintext Scenario, the TenSEAL Scenario and the HEonGPU Scenario.
- **Chapter 7: Conclusions** briefly reports the final considerations of our work.

Chapter 2

Background

2.1 The CKKS Scheme

CKKS is a homomorphic encryption algorithm proposed in 2017 by Cheon,Andrey Kim,Miran Kim and Song [9].

Unlike other HE schemes, CKKS is natively designed to handle complex and real numbers. Moreover, CKKS works with approximate arithmetic: computations performed on encrypted data do not yield exact results, instead a slight error is introduced into resulting data after decryption.

These characteristics make CKKS a natural choice for private machine learning applications and it will be the algorithm used in all our experiments requiring HE.

2.1.1 The Ring Learning With Errors (RLWE) problem

Let $R_q = Z_q[X]/(f_n(x))$ be a polynomial ring with modulus q, where $f_n(x)$ is a polynomial of degree n. Therefore, elements of R_q are polynomials of degree at most $n-1$ with coefficients in Z_q [1]. Any polynomial can be represented in R_q by reducing it modulo $f_n(x)$, i.e. , taking the remainder of the division by $f_n(x)$, and then reducing the coefficients modulo q , i.e. , taking the remainder of the division by q.

[1] Z_q denotes the set of integers modulo q, containing the elements $0,1,\ldots,q-1$, where arithmetic operations are performed modulo q.

RLWE distribution Let χ_s denote a secret distribution over R_q, and let χ_e denote an error distribution over R_q.

- Sample $s \in R_q$ according to χ_s,
- Sample $a \in R_q$ uniformly at random,
- Sample $e \in R_q$ according to χ_e,
- Compute $b := a \cdot s + e \in R_q$,

The pair (a, b) is a RLWE sample.

Decision RLWE The Decision RLWE problem is to distinguish whether samples (a, b) are drawn from the RLWE distribution or uniformly at random from $R_q \times R_q$.

Search RLWE The Search RLWE problem is to recover the secret s given m samples

$$\{(a_i, b_i = a_i \cdot s + e_i) : i = 1, \ldots, m\}$$

drawn from the RLWE distribution.

The security of CKKS scheme relies on the believed hardness of the RLWE problems. RLWE is a larger version of the Learning with Errors (LWE) problem, specialized to polynomial rings over finite fields. LWE is essentially the same problem, but defined over vectors of integers instead of polynomials, so that a, b, s, e are vector of integers . In general, to resolve RLWE instances the problem can be reconducted to more LWE instances [2]. Solving LWE instances is equivalent to solving lattice-based problems, so CKKS can be defined as a lattice-based cryptographic scheme.

2.1.2 Algebraic Preliminaries

μ-th Root of Unity A μ-th root of unity is a complex number ζ that satisfies the equation $\zeta^\mu = 1$.

Primitive μ-th Root of Unity A primitive μ-th root of unity is a μ-th root of unity that satisfies the additional condition $\zeta^k \neq 1 \ \forall k < \mu$.

[2]Each RLWE sample $(a, b = a \cdot s + e) \in R_q^2$ can be interpreted as n independent LWE samples, since elements of the ring R_q are polynomials of degree $n-1$ and therefore consist of n coefficients

μ-th Cyclotomic Polynomial The μ-th cyclotomic polynomial $\Phi_\mu(x)$ is the polynomial whose roots are the primitive μ-th roots of unity.

The μ-th cyclotomic polynomial has degree $d = \phi(\mu)$ and d roots in $\mathbb{C}$. Here, ϕ is the Euler totient function, which counts the positive integers less than μ that are coprime to μ.

Isomorphism between Polynomials and Vectors Now, let's define a mapping between the polynomial ring $Z_q[X]/(F(x))$ and the vector space Z_q^d.

$$\sigma : f(x) \in Z_q[X]/(F(x)) \rightarrow (f(x_0), f(x_1), \ldots, f(x_{d-1})) \tag{2.1}$$

where $F(x)$ is the μ-th cyclotomic polynomial of degree d and $x_0, x_1, \ldots, x_{d-1}$ are the d primitive μ-th roots of unity.

The mapping σ is important because it allows us to convert the data to be encrypted—such as model weights—into polynomials, which are the elements on which the RLWE paradigm and consequently the CKKS scheme works.

To recover the polynomial form from its vector representation, it is necessary to interpolate the polynomial using the primitive μ-th roots of unity and the components of the vector.

The mapping σ is an isomorphism: it is a bijective one-to-one mapping and it preserves the operations of addition and multiplication, it is a homomorphism with respect to addition and multiplication.

$$\begin{aligned} \sigma(f_a(x) + f_b(x)) &= \sigma(f_a(x)) + \sigma(f_b(x)) \\ \sigma(f_a(x) \cdot f_b(x)) &= \sigma(f_a(x)) \odot \sigma(f_b(x)) \end{aligned} \tag{2.2}$$

where $\odot$ is the Hadamard product.

Therefore, this isomorphism allows us to switch between the vector and polynomial representations: numerical vectors can be represented as elements of Z_q^d and encoded as polynomials in $Z_q[X]/(F(x))$, while preserving both addition and multiplication. As a result, computations performed on polynomials correspond exactly to element-wise computations on their vector representations, making this algebraic structure suitable for processing encrypted data. This correspondence represents the mathematical foundation for the encoding phase in CKKS. However, in real implementations, the CKKS encoding procedure differs from this representation. The goal of this isomorphism is only to provide the underlying intuition that allows the vector space Z_q^d and its operations to be interpreted through the polynomial ring $Z_q[X]/(F(x))$ and its corresponding operations. The CKKS encoding procedure does not follow exactly the isomorphism and it will be inspected in more detail in the following sections.

Furthermore, the choice of a cyclotomic polynomial as the modulus for the polynomial ring is not arbitrary. This class of rings possesses favorable algebraic properties and enhances the security of the RLWE problem. An example of these advantageous features is the bound that they impose on the noise growth within the RLWE samples. Following a polynomial multiplication, the magnitude of the noise is amplified by an Expansion Factor γ, which depends on the chosen polynomial modulus. For generic polynomials γ can be high, leading to decryption errors due to the exponential growth of the noise. On the other hand, selecting a cyclotomic polynomial as the modulus guarantees a strictly bounded expansion, scaling as $O(n)$ in the worst-case scenario and $O(\sqrt{n})$ in the average case, where n denotes the degree of the cyclotomic polynomial [11].

We will make a last assumption: $F(x)$ is of the form $x^n + 1$ and n is a power of 2.

When μ is a power of 2, the μ-th cyclotomic polynomial is always of the form $x^n + 1$ where $n = \mu/2$. These conditions accelerate polynomial multiplication via the Number - Theoretic Transform (NTT), an algorithm for polynomial evaluation. Thanks to this technique, the computational complexity is reduced from $O(n^2)$ to $O(n \log n)$. [12].

2.1.3 The CKKS algorithm

Encoding

The input is a vector of complex numbers of length $n/2$, where n is the degree of the cyclotomic polynomial $x^n + 1$, used as the modulus for the polynomial ring. This is because, as mentioned before, the isomorphism σ that we are using handles vectors of length $d = \phi(\mu)$, and when $\mu = 2n$ it can be proved that $\phi(\mu) = n$.

However, the size of the input vector has to be halved because the first step of the algorithm is to transform the input vector $\vec{v} = (v_0, v_1, \ldots, v_{\frac{n}{2}-1})$ into a Hermitian vector $\vec{v'}$ by appending the complex conjugates, as follows:

$$\vec{v'} = (v_0, v_1, \ldots, v_{\frac{n}{2}-1}, \overline{v_0}, \overline{v_1}, \ldots, \overline{v_{\frac{n}{2}-1}}) \tag{2.3}$$

In our case, we will work with real numbers, as we need to encode the model weights or their updates, but the procedure remains the same.

Next, the vector $\vec{v'}$ is mapped to a real vector $\vec{m} \in R^n$. The details of this transformation are beyond the scope of this thesis, the important aspect of this step is that we move from a complex-valued vector to a real-valued one.

Finally the vector $\vec{m}$ is scaled by a factor Δ and rounded to the nearest integer component-wise:

$$\Delta m = (\lfloor \Delta \cdot m_0 \rceil, \lfloor \Delta \cdot m_1 \rceil, \ldots, \lfloor \Delta \cdot m_{n-1} \rceil) \tag{2.4}$$

The scaling factor Δ is a parameter of the CKKS scheme that controls the precision of computations. The rounding operation $\lfloor \Delta m_i \rceil$ introduces an encoding error. This the reason why CKKS scheme is considered an approximate encryption algorithm. A larger Δ allows for a more precise encoding, since more bits are used to represent the fractional part. We discuss into more detail about the choice of Δ later.
The encoded plaintext polynomial is defined as:

$$\Delta M(X) = \sum_{i=0}^{n-1} \Delta m_i X^i \in Z_q[X]/(X^n + 1) \tag{2.5}$$

Decoding

From the plaintext polynomial $\Delta M(X)$ recover the vector $\vec{m} \in R^n$ from its coefficients:

$$\vec{m} = (\frac{c_0}{\Delta}, \frac{c_1}{\Delta}, \ldots, \frac{c_{n-1}}{\Delta}) \tag{2.6}$$

where c_i is the coefficient of X^i in $\Delta M(X)$.

After that, the real vector $\vec{m}$ is mapped back to a complex vector $\vec{v'} \in C^n$ through the inverse of the previous transformation. Again, as before, we will not analyze the details of this inverse transformation, as they are beyond the scope of this thesis. The important aspect of this step is that it recovers the complex-valued representation from the real-valued one through a transformation that is the inverse of the one previously used to move from a complex-valued vector to a real-valued one.

Finally, extract only the first $\frac{n}{2}$ elements of vector $\vec{v'}$ to obtain the original input vector $\vec{v}$.

Key Generation

Secret Key The secret key is a small polynomial sk in $Z_q[X]/(X^n + 1)$. sk is small in the sense that its coefficients are significantly smaller than the modulus q.

The construction of this small polynomial depends on the specific implementation of the CKKS scheme. For instance, TenSEAL uses an IID (Independent and Identically Distributed) ternary distribution over {-1,0,1}, meaning that each coefficient has the same probability to assume one of these values.

On the other hand, HEonGPU employs a sparse polynomial representation, where only a small and fixed number of coefficients are different from zero. These non-zero coefficients assume with the same probability the value of 1 or -1.

Public Key

- Sample $a \in Z_q[X]/(X^n + 1)$ uniformly at random
- Sample $e \in Z_q[X]/(X^n + 1)$ according to an error distribution χ_e
  The error polynomial is a small polynomial, in the sense defined before. The error distribution χ_e depends on the specific implementation of the CKKS scheme. Both TenSEAL and HEonGPU use a discrete Gaussian distribution over Z_q to sample the coefficients of the error polynomial. A value is drawn from the Gaussian distribution, then it is rounded to the nearest integer and finally reduced modulo q. Both libraries use mean $= 0$ and $\sigma = 3.19 \approx \frac{8}{\sqrt{2\pi}}$. These parameters ensure the smallness of the error polynomial.
- Compute $b = -a \cdot sk + e$

The public key is $pk = (b, a)$.

It should be noted that the public key pair is a RLWE sample. Therefore, it follows the hardness assumption of the search RLWE problem, ensuring that recovering the secret key from the public key is computationally infeasible.

Encryption

Before encrypting, an ephemeral small polynomial $h \in Z_q[X]/(X^n + 1)$ is sampled. h is drawn from an IID ternary distribution over $\{-1,0,1\}$ for both TenSEAL and HEonGPU. Also two additional polynomial errors e_0, e_1 are sampled from χ_e.

The formula for encrypting with $pk = (b, a)$ a plaintext message $msg \in Z_q[X]/(X^n + 1)$ after encoding is:

$$\begin{aligned} c_0 &= h \cdot b + msg + e_0 \\ c_1 &= h \cdot a + e_1 \\ ct &= (c_0, c_1) \in (Z_q[X]/(X^n + 1))^2 \end{aligned} \tag{2.7}$$

Decryption

The formula for decrypting with sk a ciphertext $ct = (c_0, c_1)$ is:

$$\begin{aligned} \tilde{msg} &= c_0 + c_1 \cdot sk = h \cdot b + msg + e_0 + (h \cdot a + e_1) \cdot sk \\ &= -a \cdot h \cdot sk + h \cdot e + msg + e_0 + a \cdot h \cdot sk + e_1 \cdot sk \\ &= msg + \underbrace{e_0 + e_1 \cdot sk + h \cdot e}_{e_{ct}} \\ &= msg + e_{ct} \approx msg \end{aligned} \tag{2.8}$$

After decryption, $\tilde{msg}$ needs to be decoded to recover the original input vector.

During encryption, some error terms are introduced and remain present after decryption, in the form of an aggregate error e_{ct}. However, under the assumption that e, e_0, e_1, h and sk are sufficiently small, this accumulated error remains small as well, so the decrypted value stays close to the original plaintext, ensuring the correctness of the scheme.

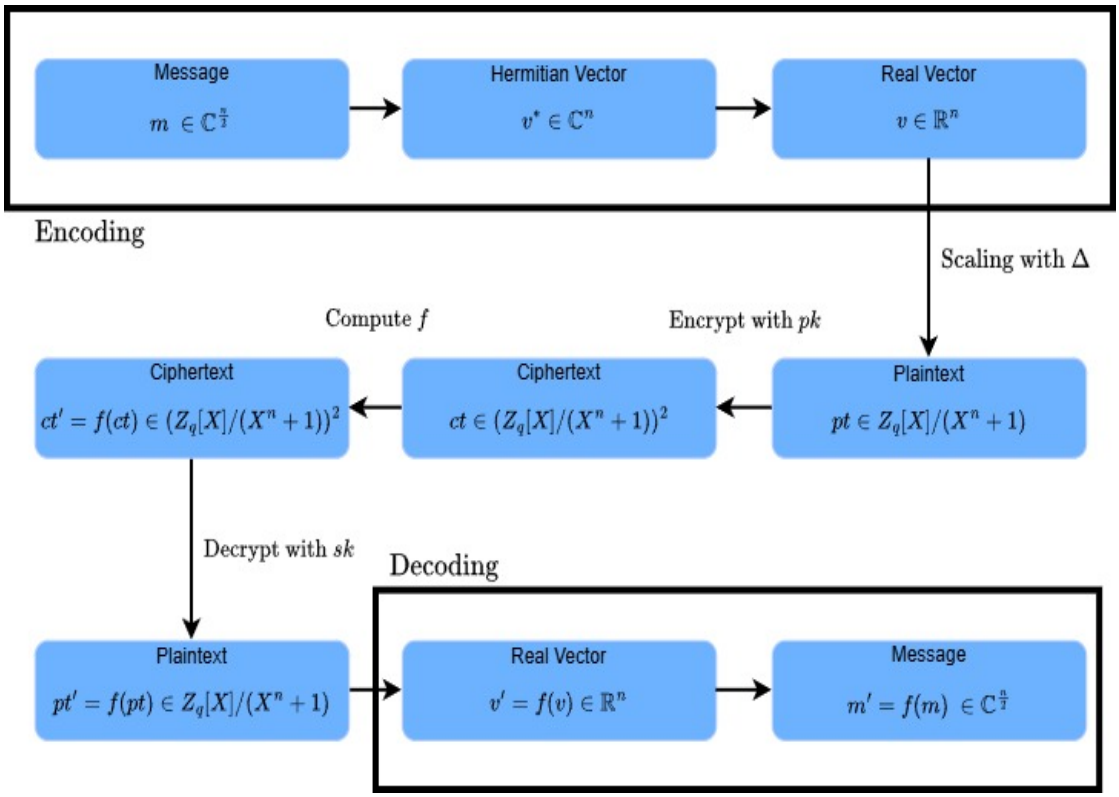


Figure 2.1: End-to-end pipeline of the CKKS encryption scheme.

Homomorphic Addition

CKKS supports homomorphic addition, allowing ciphertexts to be added without prior decryption. The result of this operation corresponds, after decryption, to the sum of the original plaintexts.

A method is also defined for adding a plaintext to a ciphertext.

Ciphertext-to-Plaintext Addition Suppose we have a message msg_1 encrypted as $ct_1 = (c_{0,1}, c_{1,1})$ and a plaintext message msg_2 encoded as $m\tilde{s}g_2$.

The homomorphic addition of the ciphertext and the plaintext is defined as:

$$ct_{add} = ct_1 + (m\tilde{s}g_2, 0) = (c_{0,1} + m\tilde{s}g_2, c_{1,1}) \tag{2.9}$$

The decryption of ct_{add} yields:

$$\begin{aligned}
&c_{add,0} + c_{add,1} \cdot sk = \\
&= c_{0,1} + \tilde{msg}_2 + c_{1,1} \cdot sk \\
&= (h \cdot b + \tilde{msg}_1 + e_0) + \tilde{msg}_2 + (h \cdot a + e_1) \cdot sk \\
&= (h \cdot (-a \cdot sk + e) + \tilde{msg}_1 + e_0) + \tilde{msg}_2 + a \cdot h \cdot sk + e_1 \cdot sk \\
&= -a \cdot h \cdot sk + \tilde{msg}_1 + \tilde{msg}_2 + a \cdot h \cdot sk + \underbrace{h \cdot e + e_0 + e_1 \cdot sk}_{e_{ct}} \\
&= \tilde{msg}_1 + \tilde{msg}_2 + e_{ct} \\
&\approx msg_1 + msg_2
\end{aligned} \tag{2.10}$$

The plaintext message must be encoded before being added to the ciphertext.

Ciphertext-to-Ciphertext Addition Suppose we have two messages msg_1 and msg_2 encrypted as $ct_1 = (c_{0,1}, c_{1,1})$ and $ct_2 = (c_{0,2}, c_{1,2})$. The homomorphic addition of these two ciphertexts is defined as:

$$ct_{add} = ct_1 + ct_2 = (c_{0,1} + c_{0,2}, c_{1,1} + c_{1,2}) \tag{2.11}$$

The decryption of ct_{add} yields:

$$\begin{aligned}
&c_{add,0} + c_{add,1} \cdot sk = c_{0,1} + c_{0,2} + (c_{1,1} + c_{1,2}) \cdot sk \\
&= \underbrace{c_{0,1} + c_{1,1} \cdot sk}_{msg_1 + e_{ct,1}} + \underbrace{c_{0,2} + c_{1,2} \cdot sk}_{msg_2 + e_{ct,2}} \\
&= msg_1 + msg_2 + e_{ct,1} + e_{ct,2} \\
&\approx msg_1 + msg_2 + 2e_{ct} \approx msg_1 + msg_2
\end{aligned} \tag{2.12}$$

The error terms $e_{ct,1}$ and $e_{ct,2}$ are assumed to be similar. Their sum can still be considered small.

This illustrates the correctness of the homomorphic addition operation.

Homomorphic Multiplication

CKKS supports homomorphic multiplication, allowing ciphertexts to be multiplied without prior decryption. The result of this operation corresponds, after decryption, to the product of the original plaintexts.

A method is also defined for multiplying a plaintext by a ciphertext.

Ciphertext-to-Plaintext Multiplication The homomorphic multiplication of a ciphertext $ct = (c_0, c_1)$, which encrypts a message msg_1, and a plaintext $pt \in Z_q[X]/(X^n + 1)$, corresponding to an already encoded msg_2, is defined as:

$$ct_{mult} = (c_0 \cdot pt, c_1 \cdot pt) \tag{2.13}$$

The decryption of ct_{mult} yields:

$$\begin{aligned}
&c_{mult,0} + c_{mult,1} \cdot sk = \\
&= c_0 \cdot pt + c_1 \cdot pt \cdot sk \\
&= (h \cdot b + msg_1 + e_0) \cdot pt + (h \cdot a + e_1) \cdot pt \cdot sk \\
&= (h \cdot (-a \cdot sk + e) + msg_1 + e_0) \cdot pt + a \cdot h \cdot pt \cdot sk + e_1 \cdot pt \cdot sk \\
&= -a \cdot h \cdot pt \cdot sk + msg_1 \cdot pt + a \cdot h \cdot pt \cdot sk + \underbrace{h \cdot e \cdot pt + e_0 \cdot pt + +e_1 \cdot pt \cdot sk}_{e_{mult}} \\
&= msg_1 \cdot pt + e_{mult} \approx msg_1 \cdot msg_2
\end{aligned} \tag{2.14}$$

After this operation, the result has a scale of Δ^2 and needs to be restored to the original scale Δ. To achieve this goal, a process called rescaling is performed, which will be discussed in detail later.

Ciphertext-to-Ciphertext Multiplication The homomorphic multiplication of two ciphertexts, $ct_1 = (c_{0,1}, c_{1,1})$ and $ct_2 = (c_{0,2}, c_{1,2})$, corresponding to the plaintext messages msg_1 and msg_2, respectively, is defined as:

$$ct_{mult} = (c_{0,1} \cdot c_{0,2}, c_{0,1} \cdot c_{1,2} + c_{1,1} \cdot c_{0,2}, c_{1,1} \cdot c_{1,2}) = (d_0, d_1, d_2) \tag{2.15}$$

It can be proven that the correct decryption of ct_{mult} is obtained as:

$$d_0 + d_1 \cdot sk + d_2 \cdot sk^2 = msg_1 \cdot msg_2 + e_{mult} \approx msg_1 \cdot msg_2 \tag{2.16}$$

However, a problem now arises: the resulting ciphertext ct_{mult} has three components, which breaks the CKKS structure for two-component ciphertexts. For example, two multiplications in a row are no longer possible, more generally, after such a multiplication, the ciphertext cannot be processed further. To address this issue, relinearization is introduced.

Relinearization

Relinearization is a procedure that transforms the three-component ciphertext resulting from a homomorphic multiplication into a standard two-component CKKS ciphertext, without affecting the correctness of the computations.

Relinearization is performed using a special key pair called the relinearization keys $rk = (r_0, r_1)$. The relinearization keys encrypt the square of the secret key sk^2 under the public key pk.

$$rk = (sk^2 - a \cdot sk + e, a) \tag{2.17}$$

where a is sampled uniformly at random from $Z_q[X]/(X^n + 1)$ and e is sampled according to the error distribution χ_e.

By definition we have:

$$r_0 + r_1 \cdot sk = sk^2 \tag{2.18}$$

However, the construction of the relinearization keys is not as simple as described above, due to the noise growth in the relinearization keys during later relinearization steps. In practice, relinearization keys are generated using some form of modulus switching techniques [13].

Relinearization of a three-component ciphertext $ct_{mult} = (d_0, d_1, d_2)$ is performed as follows:

$$ct_{relin} = (d_0 + d_2 \cdot r_0, d_1 + d_2 \cdot r_1) = (c'_0, c'_1) \tag{2.19}$$

Finally, ct_{relin} is a standard two-component CKKS ciphertext and can be decrypted in the usual way:

$$\begin{aligned} &c'_0 + c'_1 \cdot sk = d_0 + d_2 \cdot r_0 + d_1 \cdot sk + d_2 \cdot r_1 \cdot sk \\ &= d_0 + d_1 \cdot sk + d_2 \cdot (r_0 + r_1 \cdot sk) \\ &= d_0 + d_1 \cdot sk + d_2 \cdot sk^2 = msg_1 \cdot msg_2 + e_{mult} \approx msg_1 \cdot msg_2 \end{aligned} \tag{2.20}$$

It must be specified that the noise introduced after a homomorphic multiplication, e_{mult}, is considerably higher than the one introduced after a homomorphic addition. A sequence of repeated multiplications can cause the noise to grow beyond the allowed bound, making decryption infeasible or computationally incorrect. We will later analyze some countermeasures that can be taken into account to address this problem.

Rescaling

When multiplying two CKKS ciphertexts with scale Δ, the resulting ciphertext has scale Δ^2. Therefore, after each homomorphic multiplication, a rescaling operation is required to bring the scale back to Δ.

Conceptually, rescaling can be viewed as dividing the ciphertext by a factor Δ in order to restore the original scale. However, the actual implementation is more complex, here we only provide an intuitive description of the process.

In practical implementations of the CKKS scheme, rescaling is achieved by removing one level from the modulus chain.

The modulus q of the polynomial ring $Z_q[X]/(X^n + 1)$ is defined as the product of several moduli, $q = q_0 \cdot q_1 \cdot \cdots \cdot q_L$.

The central moduli of the modulus chain are chosen to be approximately the same size as the scaling factor Δ. Consequently, after a homomorphic multiplication, one level of the modulus chain is removed. More precisely, a ciphertext that was previously represented modulo $q_i \cdot q_{i+1} \cdot \cdots \cdot q_L$ is rescaled and represented modulo $q_{i+1} \cdot q_{i+2} \cdot \cdots \cdot q_L$.

This operation effectively divides the ciphertext by q_i, which is approximately equal to Δ, thus restoring the original scale.

As a consequence, the number of levels L in the modulus chain determines the maximum multiplicative depth that can be evaluated. Since each homomorphic multiplication consumes one level through the subsequent rescaling step, after all levels have been exhausted no further ciphertext multiplications can be performed.

As we will see in the next chapters, our simulator and our FL workflow do not use homomorphic multiplication, therefore, we do not need to consider the length of the modulus chain. This will not represent a limitation for our purposes.

In conclusion, we can summarize the steps needed to perform a homomorphic multiplication of two ciphertexts as follows:

- Execute the homomorphic multiplication to obtain a three-component ciphertext: $ct_{mult} = (c_{0,1} \cdot c_{0,2}, c_{0,1} \cdot c_{1,2} + c_{1,1} \cdot c_{0,2}, c_{1,1} \cdot c_{1,2}) = (d_0, d_1, d_2)$
- Relinearize ct_{mult} using rk to return to the standard two-component notation: $ct_{relin} = (d_0 + d_2 \cdot r_0, d_1 + d_2 \cdot r_1) = (c'_0, c'_1)$
- Rescale the output by removing a level from the modulus chain.
- Decrypt the relinearized ciphertext ct_{relin} in the usual fashion: $ct_{dec} = c'_0 + c'_1 \cdot sk$

In high-level modern HE libraries, like TenSEAL [8], relinearization and rescaling are performed automatically within the function that performs the homomorphic multiplication. The user does not need to explicitly call the relinearization and rescaling functions. Contrarily, the HEonGPU library [10] does not handle relinearization and rescaling operations automatically within the function that performs homomorphic multiplication, the user needs to call them explicitly.

As we will see in the next chapters, our simulator and our FL workflow do not use homomorphic multiplication, so we will not need to worry about relinearization and rescaling operations. The description of homomorphic multiplication, relinearization and rescaling has been included only for completeness in the description of the CKKS scheme.

Bootstrapping

In general, every homomorphic operation executed on encrypted data adds noise to the ciphertext. After several operations, this noise can explode, making the decryption process unfeasible and leading to an incorrect result.

Furthermore, only a limited number of sequential multiplications can be performed. This is because each multiplication consumes a level from the modulus chain, consequently, after a certain number of multiplications, the ciphertext runs out of available levels.

To address both of these problems—the growth of noise in ciphertexts and the limited capacity for multiplications—the bootstrapping technique is introduced.

The main idea of bootstrapping is to homomorphically evaluate the decryption circuit, using as the decryption key an encrypted version of the secret key, encrypted with its own corresponding public key.

We know that applying a homomorphic function $F(x)$ to an encrypted input returns a ciphertext inside which the function F has been applied to the corresponding plaintext:

So, if $F(x)$ is the homomorphic evaluation of the decryption function parametrized by the ciphertext, $F(x) = \mathrm{Dec}_x(ct)$, we can write:

$$\mathrm{Dec}_{\mathrm{Enc}_{pk}(sk)}(ct) = \mathrm{Enc}_{pk}(\mathrm{Dec}_{sk}(ct)) \tag{2.21}$$

Knowing that $\mathrm{Dec}_{sk}(ct) = pt$, we obtain a fresh ciphertext $ct' = \mathrm{Enc}_{pk}(pt)$.The noise level of ct' has been significantly reduced back to a nominal baseline threshold and its level in the modulus chain has been restored to the original maximum capacity, allowing us to continue performing operations on it. It is also important to note that throughout this process, no private information—such as sk—is shared. Consequently, this process can be evaluated by an untrusted central server that receives from the user only the bootstrapping key $\mathrm{Enc}_{pk}(sk)$, which does not reveal any secret information.

Bootstrapping in CKKS must take into account the approximate nature of its data representation. Indeed, the homomorphic evaluation of the decryption circuit itself introduces additional noise.In addition, bootstrapping is one of the most resource-intensive operations in homomorphic encryption schemes.

For these reasons, the process is not implemented as a simple homomorphic evaluation of the decryption circuit, instead it is more complex and dependent on the specific implementation of the scheme [14].

We will never use bootstrapping in our FL workflow because, in the absence of homomorphic multiplications, as mentioned before, that are the operations that increase the most the noise in the ciphertexts, we will not need it. The description of bootstrapping has been included only for completeness in the description of the CKKS scheme.

2.2 The Federated Averaging Algorithm

In this section we will give a description of how the Federated Averaging (FedAvg) protocol works. This scheme was first introduced by the paper [15], which is also the paper that originally proposed the concept of FL.

We want to highlight that FL is a paradigm for training machine learning models across a network of clients. It defines a series of principles, such as the fact that raw data never leaves the clients and remains local, and that clients only communicate the update to the global model after a round of local training. It also defines roles, such as the clients, who perform local training, and the server, whose task is to aggregate the received updates.

However, the specific way in which this paradigm is put into practice depends on the specific algorithm that implements it. In this context, FedAvg is the most popular protocol used to implement FL: it is widely adopted in the literature and it was both the first and the simplest algorithm, these reasons are behind our choice to implement our simulator following the FedAvg protocol. Nevertheless, FedAvg is not the only algorithm for FL. Its limitations have often been identified, leading to new algorithms that explore different optimization strategies to address them. This is why several alternative algorithms exist besides FedAvg [16] [17] [18].

As we introduced before, in the FL training process two entities are involved: the clients, each of which has a local dataset D_i that it uses to train the model locally and never leaves the clients, and the server, which is responsible for storing the global model, distributing it at the beginning of each global training round and aggregating the updates sent by the clients in order to compute the new version of the global model.

Algorithm 1 FedAvg Algorithm

/ Server Initialization */* ▷ Run on Server
for each weight w_j **do**
    $w_j \leftarrow$ Random Initialization
end for
/ Global Training Rounds */*
for each round $t = 1, 2, \ldots,$ `num_rounds` **do**
    $S_t \leftarrow$ Sample a Random Subset of `round_clients` Clients
    for each client $i \in S_t$ **in parallel do**
        Download w^t from the Server
        / Local Training */*
        $w_i^{t+1} \leftarrow \text{LocalTraining}(w^t)$
        Upload w_i^{t+1} to the Server
    end for
    / Aggregation Phase */*
    $D_{tot} \leftarrow \sum_{i \in S_t} |D_i|$
    $w^{t+1} \leftarrow \sum_{i \in S_t} \frac{|D_i|}{D_{tot}} w_i^{t+1}$
end for

function LocalTraining(w) ▷ Run on Client i
    $\mathcal{B} \leftarrow$ (split D_i into batches of size `train_batch_size`)
    for each local step $ep = 1,2, \ldots, ls$ **do**
        for batch $b \in \mathcal{B}$ **do**
            $w \leftarrow w -$ `local_lr` $\cdot \nabla \ell(w; b)$
        end for
    end for
    return w
end function

As shown in Algorithm 1, at the beginning of the training process, the server initializes every weight of the model at random.

Then, for `num_rounds`, that is the total number of global rounds performed, a routine is executed. First of all, the server randomly selects `round_clients` clients among all the available clients. Indeed, only a subset of the clients participates in each global training round, a subset composed of `round_clients` nodes, not every client participates in every round. Each of the selected clients receives, over the communication channel, the current version of the global model from the server. Indeed, the server stores at every moment the current version of the model, with its current weights. At every global round this model is updated through the aggregation phase, which we will examine later.

After receiving the current version of the model, the clients selected for the round perform local training starting from it. The local dataset of each client is split into batches, each containing `train_batch_size` samples. The same routine is repeated ls times, this is a local step. Firstly, the batch is forwarded into the model, a prediction is made and this prediction is compared against the ground-truth label to compute the loss with respect to the loss function. With respect to every weight, the gradient of the loss function is computed and used to derive the new weights for that local step. `local_lr` is the local learning rate. This process is repeated ls times.

At the end of local training, each client holds a set of parameters that differs both from the other clients' and from the version communicated at the beginning of the global round by the server. This new version is the result of ls local training steps on the client's local dataset. The locally computed version of the model is then sent by each client to the server. The server subsequently performs an aggregation phase. The new version of the global model is computed weight by weight, by summing the local versions of each weight sent by every client. This sum is weighted: the weight assigned to each client is the ratio between the number of samples in its local dataset and the total number of samples across all clients participating in that round.

In this way, a new version of the global model is obtained. This process is repeated `num_rounds` times.

Chapter 3

Related Works

3.1 Alternative Fully Homomorphic Encryption Schemes

In this section we will give an overview of the most popular Fully Homomorphic Encryption (FHE) algorithms besides CKKS. For each scheme we will point out its main features, its pros and cons, and its possible applications. We will finally offer a comparison between the presented FHE algorithms, CKKS included.

First of all, we need to give a definition of FHE compared to the other types of HE that exist:

- Partially Homomorphic Encryption (PHE): allows only one type of homomorphic operation, which can be repeated an unlimited number of times.
- Somewhat Homomorphic Encryption (SHE): allows some types of operations, which can only be repeated a limited number of times.
- Fully Homomorphic Encryption (FHE): allows an unlimited number of operations, repeated an unlimited number of times.

All the presented FHE schemes are based on the hardness assumptions of the LWE/RLWE problems 2.1.1, just like CKKS.

The BGV and BFV Schemes The Brakerski-Gentry-Vaikuntanathan (BGV) encryption scheme was introduced in 2011 [19]. The Fan-Vercauteren variant of the Brakerski scheme (BFV) was introduced in 2012 [13]. The two algorithms are designed for the same purpose: they both implement exact arithmetic on integers. This is the main difference compared to CKKS, which is an approximate encryption scheme: BGV and BFV instead lead to exact results after encrypted computations, without introducing an error.

Moreover, BGV and BFV are designed to handle integers modulo t as input, unlike CKKS, which takes complex numbers as input. BGV and BFV both support batching, which allows several values to be encrypted within the same ciphertext and enables operations to be performed on it in parallel.

The main difference between the two schemes lies in the way they manage ciphertext noise. As already mentioned, homomorphic operations introduce noise into the ciphertext, especially homomorphic multiplication, which, if left to grow uncontrolled, can lead to incorrect decryption. For this reason, every FHE encryption scheme provides a method to refresh the ciphertext once the noise becomes too high, after a certain level of computation. BGV controls noise growth through modulus switching: the ciphertext is mapped to a progressively smaller modulus, following a predefined decreasing chain of moduli. Reducing the modulus proportionally reduces the noise as well. BFV, by contrast, operates under a single, fixed modulus q for the entire computation, without modulus switching. Noise control is instead achieved through scaling: when the plaintext is encrypted inside the ciphertext, it is multiplied by a scaling factor $\Delta = \lfloor \frac{q}{t} \rfloor$, keeping the plaintext in the high-order bits and thus preventing it from being corrupted by the noise.

BGV and BFV schemes, compared to CKKS, are particularly suited to fields where exact computation is required or where integer inputs are handled. For example, cases of the use of BGV or BFV can be found in the literature in applications such as e-voting, encrypted database queries and authentication through HE [20] [21] [22].

The TFHE Scheme The TFHE (Fast Fully Homomorphic Encryption over the Torus) algorithm is a FHE scheme introduced in 2016 [23]. This algorithm handles bits or small numbers represented as a small sequence of bits as input. It is therefore well suited for evaluating boolean circuits or computing bitwise operations in a homomorphic way. The main feature of TFHE is its fast bootstrapping, a routine that is often needed to refresh ciphertexts with too much noise accumulated after many consecutive homomorphic operations and is very computationally expensive. With TFHE, bootstrapping goes from requiring several seconds in other FHE schemes to requiring only a few milliseconds, making the difference significant [23]. However, TFHE does not support batching.

TFHE also introduces Programmable Bootstrapping. Normally, the bootstrapping operation simply returns a ciphertext equivalent to the one given as input, but refreshed, with the noise contained in it reduced to a low level. In TFHE instead it is possible to define a function through a Lookup Table, which is evaluated during the bootstrapping operation. In this way, when the bootstrapping is applied to a ciphertext, the value is not only refreshed from the accumulated noise, but the defined function is also applied to the plaintext contained inside it.

In practice, TFHE bootstrapping takes as input a ciphertext with high noise, $Enc(m, high\ noise)$, and returns a ciphertext with the function $f()$ applied and the noise level restored to a low level, $Enc(f(m), low\ noise)$.

This functionality is useful when evaluating non-linear functions homomorphically, such as activation functions in machine learning models. In fact, other FHE schemes, such as CKKS, can only perform additions and multiplications. Consequently, non-linear functions, such as ReLU, are approximated through a polynomial function, made of only additions and multiplications. This introduces an approximation error. With TFHE instead it is possible to evaluate non-linear functions with Programmable Bootstrapping, which allows defining an arbitrary function through a Lookup Table, including activation functions in deep learning models such as ReLU, which would otherwise need to be approximated in other FHE schemes because they are non-linear. This feature makes TFHE a good choice for homomorphic inference.

In particular, as proposed by OpenFHE [24], a well-known library implementing various HE schemes, it is possible to switch between ciphertexts encrypted with CKKS and ciphertexts encrypted with TFHE. In this way, CKKS can be used for the linear parts of the network, where floating-point arithmetic operations are required, and then switched to a TFHE representation when evaluating non-linear layers, such as activation functions.

TFHE finds applications in fields where its two main features are needed: where boolean values are involved, such as homomorphic circuit evaluation, or where programmable bootstrapping can be exploited, such as the evaluation of non-linear functions inside neural networks [25] [26].

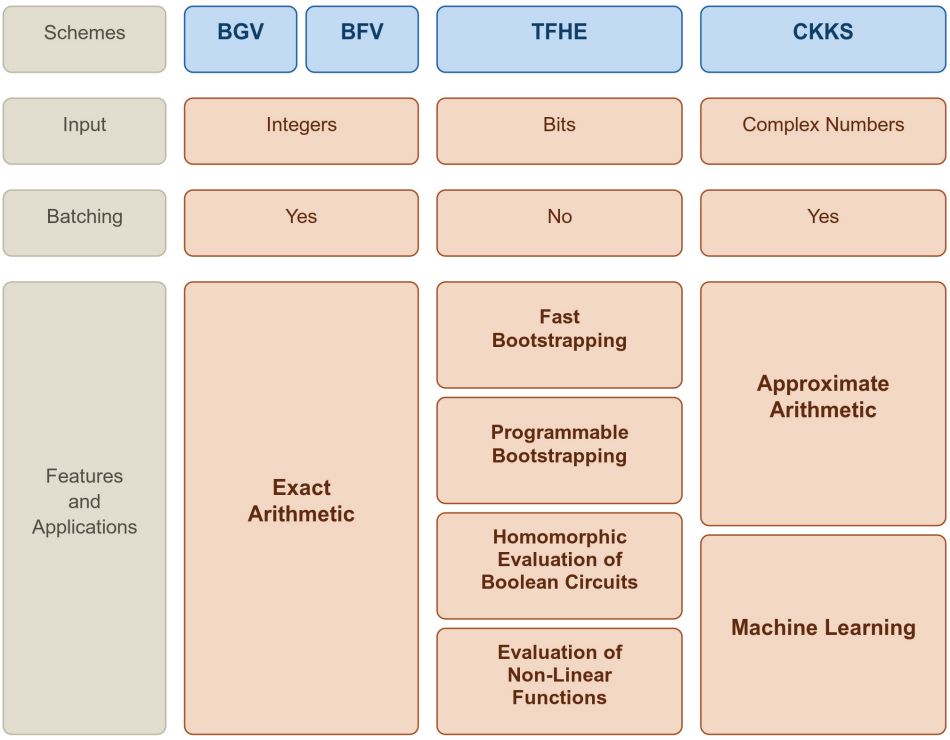


Figure 3.1: Overview of FHE Schemes and Their Properties

3.2 Fully Homomorphic Encryption Libraries

Table 3.1: Overview of FHE Libraries

Library	Language	FHE Schemes Implemented				Notes
		BGV	BFV	CKKS	TFHE	
Microsoft SEAL	C++	✓	✓	✓	✗	TenSEAL built on top of it.
PALISADE / OpenFHE	C++	✓	✓	✓	✓	PALISADE evolved into OpenFHE.
HElib	C++	✓	✗	✓	✗	One of the first FHE libraries.
TFHE	C/C++	✗	✗	✗	✓	Original TFHE Scheme implementation [23]
HEAAN	C++	✗	✗	✓	✗	Original CKKS Scheme implementation [9]
Lattigo	Go	✓	✓	✓	✗	
cuFHE	C++	✗	✗	✗	✓	GPU-accelerated
HEonGPU	C++	✗	✓	✓	✓	GPU-accelerated
NuFHE	Python	✗	✗	✗	✓	GPU-accelerated

Table 3.1 shows a selection of the most popular libraries that implement FHE.

We can observe that most of these libraries are designed for C++. This represents a limitation, since C++ is less common in modern applications and a Python wrapper is often missing, preventing less technical users, who would prefer a simpler interface, from adopting FHE. TenSEAL, on the other hand, provides a simple Python API and it is built on top of Microsoft SEAL.

We can also note that libraries other than HEonGPU attempt to accelerate FHE using GPUs (NuFHE and cuFHE), showing that this is an actively researched topic in the literature.

Chapter 4

Method

4.1 System Architecture and Threat Model

Our system adopts a classical federated learning architecture composed of two entities: a central server and a set of clients.

We already discussed the role of the actors in a FL system in 2.2, here we go into more detail regarding our implementation and how our simulator works.

The Server The server acts as the central coordinator of the training process. It is responsible for maintaining a global model, selecting the clients that will participate in the current training round and distributing to them the latest version of the global model. At the end of the local training phase, the server collects the updates to the global model, aggregates them and computes a new version of the global model.

In the case of the adoption of HE, the server stores the global model in encrypted form and never has access to the model weights. Moreover, in this scenario, the server cannot see the updates sent by the clients, since they are encrypted as well, and it performs the aggregation homomorphically.

An important clarification must be made: as will be discussed in the next sections, in our model the server performs an evaluation phase at the end of every training round on a test set in order to compute some monitoring metrics. This evaluation is performed in plaintext using an unencrypted version of the model. This represents a significant limitation of our simulator, since it implies that the server must have access to the private keys required to decrypt the global model before carrying out the evaluation. However, if the server possesses the private key, it can also access the weights of the global model or the updates sent by the clients, which, in a HE scenario, should remain private.

In real-world deployments, the server does not possess the private keys and does not perform model evaluation. However, our framework is intended as a simulator and we need to monitor the evolution of the training process through a set of metrics. Therefore, we consider the evaluation phase as the only moment in which the server is allowed to access the model weights. During all other phases, we assume that the server cannot decrypt the stored global model nor the updates sent by the clients.

The Clients Clients represent the entities that own local data and contribute to the training process without ever sharing their data. Each client maintains a local version of the model, which is updated with the latest version of the global model distributed by the central server whenever the client is selected to participate in a training round. After downloading the model, clients perform local training for a fixed number of steps using their local datasets. Once the training phase is completed, they compute the pseudogradients as the difference between the updated model parameters and the received ones and then send these pseudogradients to the server.

In the case of the adoption of HE, clients receive the latest version of the global model from the server encrypted with their public key. Therefore, before updating their local model, they must decrypt it using the corresponding private key and copy the decrypted parameters into the local version they maintain. Similarly, in this scenario, clients send the computed pseudogradients in encrypted form.

4.1.1 Threat Model

We now describe the threat model considered in our experiments.

The threat model defines the assumptions made about the capabilities and behaviours of the actors involved in the system. By identifying the potential adversaries and their goals, we can determine which assets must be protected and what security guarantees are provided by HE.

External Adversaries First, we consider external adversaries, that is, entities that do not participate in our framework. We aim to protect both the pseudogradients shared by the clients with the server and the global model distributed by the server to the clients at the beginning of each training round. We assume that an external adversary is able to eavesdrop on the communication channel between clients and server. For this reason, both the pseudogradients and the global model parameters are encrypted before being transmitted. In this way, even if an adversary successfully intercepts the communications between clients and server, they will not be able to extract meaningful information from the exchanged data, since all the protected information remains encrypted.

The Server We assume the server to be honest-but-curious. This means that the server correctly follows the training protocol: it selects the clients, distributes the model and performs the aggregation procedure as expected. However, it may still try to infer additional information from the data shared by the clients, in particular from the pseudogradients.

For this reason, pseudogradients are encrypted before being sent to the server. Consequently, the server cannot obtain any additional information by inspecting them, since it is not able to decrypt their content because it does not have access to the private keys. Similarly, by storing the global model in encrypted form, we prevent the server from accessing the model parameters. As for the pseudogradients, the server cannot extract information from the global model because its parameters remain encrypted and the server is not able to decrypt them. Therefore, HE provides confidentiality guarantees for both the exchanged updates and the stored global model.

An important limitation, already discussed in the previous sections, is the fact that the server performs an evaluation phase at the end of each training round using the plaintext version of the model. Therefore, the server needs access to the private keys required to decrypt the global model before performing the evaluation. From a theoretical point of view, this means that the server could decrypt the global model at any time and access its parameters, or decrypt the pseudogradients sent by the clients. However, this scenario is not considered in our threat model. We assume that the server behaves honestly and uses the private keys only during the evaluation phase, to decrypt the model before computing the evaluation metrics and does not access the model parameters during any other phase of the training process.

The Clients Clients are also assumed to not deviate from the protocol and to behave honestly. This assumption excludes poisoning attacks from our threat model. Poisoning attacks represent a threat to the integrity, rather than the confidentiality, of the federated learning process. In these attacks, malicious clients deliberately manipulate their local data or model updates, either to indiscriminately degrade the performance of the global model or to induce a specific attacker-chosen behaviour, while potentially preserving the overall model accuracy [27]. Since these attacks are outside the scope of this work, they will not be considered in our threat model.

However, clients are still exposed to a serious privacy threat. In our scenario, all clients share the same pair of public and private keys (single-key homomorphic encryption). This means that clients are not protected from each other. If a client gains access to the communication between another client and the server, it is able to decrypt, for example, the pseudogradients shared by the other client. As previously discussed, some information about a client's local dataset can potentially be inferred from the knowledge of its pseudogradients [1] [2].

Therefore, while the communication of pseudogradients is protected from the server and external adversaries, it is not protected against other clients participating in the federated learning process.

The adoption of multi-key HE can mitigate this limitation, as each client owns its own pair of public and private keys . Consequently, a client is no longer able to decrypt the encrypted communications of other clients, providing stronger privacy guarantees among the participants.

4.2 Federated Learning Training Protocol

In this section, we explain in detail the workflow of the simulator used in our experiments. This tool simulates the training process in a federated learning environment. The simulator allows us to specify various parameters related to encryption (including the possibility of using no encryption at all), as well as parameters related to the training process. A more detailed discussion of the parameters that can be controlled is provided in the following chapters. Our simulator takes inspiration from the FedAvg protocol described in 2.2, but some aspects of our implementation differ from the original algorithm. In what follows, we go into detail on how our simulator works and we highlight the differences with respect to the original formulation.

4.2.1 Data Partitioning across Clients

In realistic scenarios, there is no single centralized dataset that gets partitioned among clients, rather, each client naturally possesses its own dataset, collected independently. However, for simulation purposes, we start from a single dataset that is then split among the clients, in order to reproduce the statistical properties we want to study.

First of all, the dataset is loaded into the simulator. The dataset is divided into a training set and a test set. More details about the dataset used in our experiments will be provided in the following chapters.

The dataset must then be distributed among the clients. To achieve this, a Dirichlet distribution is used to assign samples to each client.

Our goal is to simulate the heterogeneity of data typically found in realistic scenarios. In real-world settings, clients usually have unbalanced datasets, meaning that they do not necessarily possess the same number of samples and that the label distributions may differ across clients. Moreover, a client may have many samples belonging to a particular label, while another client may have only a few. We aim to reproduce this label imbalance, so that the majority of samples associated with certain labels may be concentrated in the dataset of a single client.

The Dirichlet distribution can simulate this behaviour, allowing us to create non-IID (Independent and Identically Distributed) partitions across clients. If our dataset contains k labels and there are N clients, the Dirichlet distribution generates k vectors of length N, $(p_1, p_2, \ldots, p_N)$, where $p_i \geq 0$ and $\sum p_i = 1$. Each vector represents the distribution of a specific label across the clients, meaning that a fraction p_i of the samples belonging to that label is assigned to client i.

The Dirichlet distribution is parameterized by α. Smaller values of α produce highly non-IID distributions, meaning that most samples of a given label are concentrated in the dataset of a small number of clients, often only one. On the other hand, larger values of α lead to distributions that are closer to IID, where each client tends to have a more balanced representation of all labels.

Another parameter that can be controlled in the simulator is the quantity skew. This parameter determines the distribution of the total number of samples across the local datasets of the clients. We consider three possible configurations:

- Every client has the same number of samples.
- The number of samples assigned to each client is determined a priori by a Log-Normal distribution.
- No constraints are imposed on the number of samples assigned to each client and the allocation is entirely determined by the Dirichlet distribution.

In our experiments, we will always consider the last option, as it represents the most realistic scenario. Finally, we impose a threshold of 10 on the minimum number of samples that a client must have in its local dataset, given that no constraints are imposed on the number of samples assigned to each client.

4.2.2 Local Training

At the beginning of the training process, the global model is initialized at the server side with randomly initialized weights. The server always maintains a copy of the global model, whose weights are updated after each round of training. When HE is employed, the global model weights are stored in encrypted form and the server never has access to them. However, this holds starting from the second round onward. Indeed, at initialization, the randomly generated weights are in plaintext and the global model for the first round is stored unencrypted, even when HE is employed. This does not represent a privacy concern, since these weights are only randomly generated and carry no sensitive information. Only after the aggregation phase of the first round the global model is in encrypted form and it remains encrypted for the rest of the training process. On the other hand, clients maintain a local version of the model, which is updated whenever they participate in a training round.

Another parameter of our simulator is the number of global rounds `num_rounds`. It indicates the total number of iterations performed during the federated learning process. From now on, we will refer to a single iteration of the global federated learning process as a round and to a single iteration of the local training process performed by clients as a local step. We will now describe a round of the protocol, which will be repeated `num_rounds` times.

At the beginning of each round, `round_clients` participants are randomly selected from the pool of clients to take part in local training. Therefore, not all clients participate in every round, but only a subset of them. `round_clients` is a configurable parameter of the simulator that can be tuned before launching the experiment.

At each round, the current global model maintained by the server is distributed to the selected clients. Here, two scenarios are possible. In the first one, the model weights are sent in plaintext and can therefore be directly copied into the clients' local models. In the second scenario, the model weights are encrypted and must be decrypted by the clients before they can be loaded into the local model. In both cases, the clients replace their local model, which was outdated with respect to the latest global model, with the received global version.

Clients selected for a round perform ls local training steps. We highlight the fact that local training is performed in plaintext. The training batch size is defined by the parameter `train_batch_size`. Both parameters can be configured before launching the experiment.

More specifically, during each local training step, a batch containing `train_batch_size` samples is forwarded through the model. Initially, the model corresponds to the version received from the server. A prediction is computed for each sample and the corresponding loss is evaluated with respect to the ground-truth labels. Next, the backward pass computes the gradients of the loss with respect to all trainable model parameters through backpropagation. Finally, the optimizer updates the model parameters using the computed gradients.

This process is repeated ls times for each client selected in the current round. As a result, the local model is progressively updated and gradually diverges from the global model originally received from the server.

It follows that each selected client performs local training on a total of $ls \cdot$ `train_batch_size` samples. These samples are drawn with replacement from the client's local dataset.

4.2.3 Model Aggregation

After ls local training steps, each selected client possesses a local model that differs from the version distributed by the server at the beginning of the round.

It is at this stage that our simulator deviates from the original FedAvg protocol described in 2.2. First, instead of transmitting the updated weights w_j^{new} directly, as done in the original algorithm, each client computes and sends the difference between the updated parameter at the end of local training and the initial parameter received from the server. Second, the aggregation weight assigned to each client is different: while in the original FedAvg protocol the weight of a client is given by the ratio between the number of samples in its local dataset and the total number of samples across all clients participating in the round, in our simulator every selected client is assigned the same weight, equal to $\frac{1}{\texttt{round_clients}}$.

At this point, for each model parameter w_j, the difference between the updated parameter at the end of local training w_j^{new} and the initial parameter received from the server w_j^{old} is computed. This difference is then weighted by the client's aggregation weight, which in our experiments is equal for all selected clients and corresponds to $\frac{1}{\texttt{round_clients}}$. We refer to the resulting quantity as a pseudogradient and denote it by u_j.

$$u_j = \frac{w_j^{\text{new}} - w_j^{\text{old}}}{\texttt{round_clients}} \tag{4.1}$$

Two scenarios are possible. In the first one, clients send the pseudogradients in plaintext. The server receives the weighted pseudogradients from all clients participating in the current round and aggregates them by summing them parameter by parameter. For each parameter w_i, the total update is obtained by summing the contributions of all selected clients. The resulting update is then added to the current global parameter w_i^t to obtain the new global parameter w_i^{t+1}.

$$\begin{aligned} u_j^{\text{tot}} &= \sum_{i=1}^{\texttt{round_clients}} u_{i,j} \\ w_j^{t+1} &= w_j^t + u_j^{\text{tot}} \end{aligned} \tag{4.2}$$

In the second scenario, clients send encrypted pseudogradients. The server performs the same aggregation procedure using homomorphic operations. Therefore, the pseudogradients can be summed directly in their encrypted form, producing an encrypted aggregate update equivalent to the one obtained in the plaintext case. The resulting encrypted update is then applied to the encrypted global model parameters.

It is worth noting that the aggregation phase of the first round constitutes a special case. Since the initial global model is still in plaintext at that point, as discussed in 4.2.2, the homomorphic addition performed during the first round combines a plaintext operand, the initial global model, with a ciphertext operand, the aggregated pseudogradient. From the second round onward, both operands of the homomorphic addition are ciphertexts, as the global model has by then been encrypted as a result of the first aggregation phase.

The training protocol does not involve any homomorphic multiplication. Therefore, when HE is employed, we have greater flexibility in selecting the modulus chain parameter, as a long modulus chain is not required. Moreover, our training protocol only relies on homomorphic additions, which introduce significantly less noise growth in the ciphertexts compared to homomorphic multiplications. Consequently, we do not need to account for a noise accumulation that would require additional techniques, such as bootstrapping, to refresh the ciphertexts.

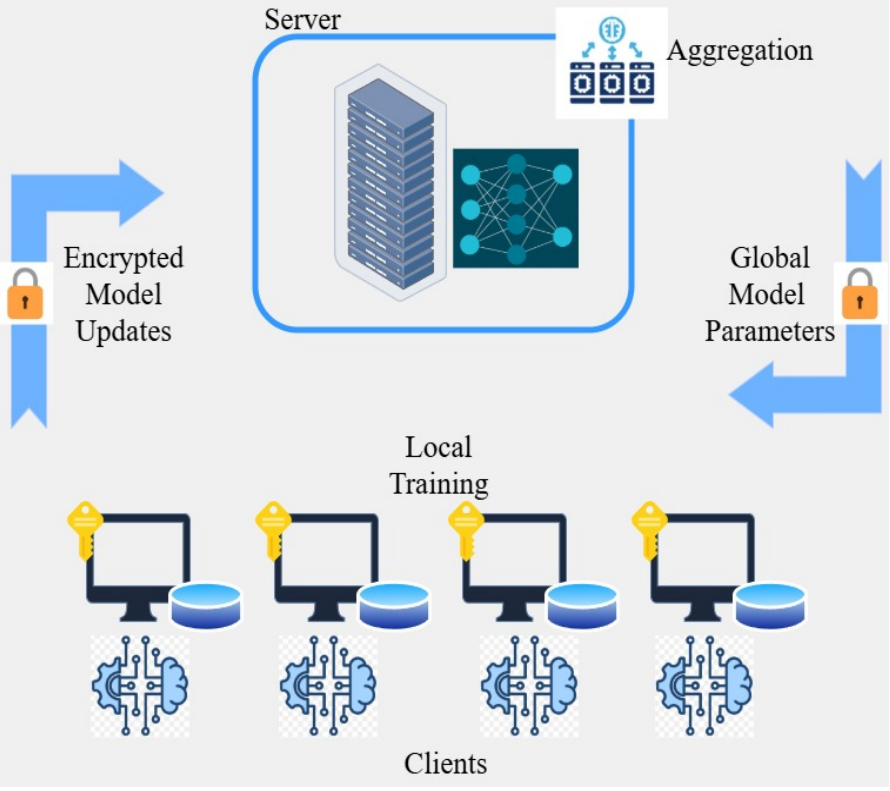


Figure 4.1: Workflow of Federated Learning with Homomorphic Encryption

4.2.4 Global Model Evaluation

At the end of each round, an evaluation phase is performed on the test dataset.As already discussed, even when HE is employed, the evaluation is performed in plaintext and not through encrypted inference using homomorphic operations. This represents a significant limitation of the simulator, as it requires the server to possess the secret key and be able to decrypt the encrypted global model before performing the evaluation. In real-world scenarios, the server would not possess the secret key required to decrypt the encrypted global model or the clients' updates.

The evaluation is performed in inference mode, without gradient computation, by processing the test samples in mini-batches. For each batch, the model predictions are compared with the ground-truth labels to compute the classification loss and accuracy.

Several other metrics are tracked throughout rounds, such as the aggregation time required by the server to compute the new global model, the encryption, decryption, and training time required by the clients, as well as the amount of data exchanged between clients and the server. More details about the tracked metrics and their computation will be provided in the following chapters.

This phase marks the end of the round. After the evaluation phase, a new round will begin.

Algorithm 2 Simulator Protocol with Homomorphic Encryption

$D \leftarrow$ Load Dataset ▷ Run on Server
$D_{\text{train}}, D_{\text{test}} \leftarrow$ Split Training and Test Set
/ Data Partitioning across Clients */*
$\{D_1, D_2, \ldots, D_N\} \leftarrow$ Partition Training Set across N clients via Dirichlet Distribution
/ Server Initialization */*
for each weight w_j **do**
    $w_j \leftarrow$ Random Initialization
end for
$w_{\text{global}} \leftarrow w^0$ Store Initial Global Model in Plaintext
/ Global Training Rounds */*
for each round $t = 1, 2, \ldots,$ `num_rounds` **do**
    $S_t \leftarrow$ Sample a Random Subset of `round_clients` Clients
    for each client $i \in S_t$ **in parallel do**
        Distribute w_{global} Global Model
        $\{\mathcal{B}_{i,1}, \mathcal{B}_{i,2} \ldots, \mathcal{B}_{i,ls}\} \leftarrow$ Sample ls batches of size `train_batch_size` from D_i
        / Local Training */*
        $u_i \leftarrow \textsc{LocalTraining}(w_{\text{global}}, t, \{\mathcal{B}_{i,1}, \mathcal{B}_{i,2} \ldots, \mathcal{B}_{i,ls}\})$
    end for
    / Aggregation Phase */*
    $u^{\text{tot}} \leftarrow \sum_{i \in S_t} u_i$ Homomorphic Addition
    if $t = 1$ **then**
        $w_{\text{global}} \leftarrow w_{\text{global}} + u^{\text{tot}}$ Homomorphic Addition Plaintext to Ciphertext
    else
        $w_{\text{global}} \leftarrow w_{\text{global}} + u^{\text{tot}}$ Homomorphic Addition Ciphertext to Ciphertext
    end if
    / Global Model Evaluation */*
    Evaluate w_{global} on the Test Set
end for

function $\textsc{LocalTraining}(w_{\text{global}}, t, \{\mathcal{B}_{i,1}, \mathcal{B}_{i,2} \ldots, \mathcal{B}_{i,ls}\})$ ▷ Run on Client i
    if $t = 1$ **then**
        $w_i^{\text{old}} \leftarrow w_{\text{global}}$
        $w_{\text{local}} \leftarrow w_{\text{global}}$ Store Distributed Model into Local Model
    else
        $w_{\text{local}} \leftarrow \text{Dec}(w_{\text{global}})$ Decrypt and Store Distributed Model into Local Model
        $w_i^{\text{old}} \leftarrow w_{\text{local}}$
    end if
    for each local step $ep = 1,2, \ldots, ls$ **do**
        $w_{\text{local}} \leftarrow w_{\text{local}} - \texttt{local_lr} \cdot \nabla \ell(w_{\text{local}}; \mathcal{B}_{i,ep})$
    end for
    / Compute Pseudogradients */*
    $u_i \leftarrow \frac{w_{\text{local}} - w_i^{\text{old}}}{\texttt{round_clients}}$
    return $\text{Enc}(u_i)$
end function

4.3 Research Questions and Experimental Scenarios

The choice of the encryption scenario and the parameter settings used in our experiments is driven by the research questions addressed in this thesis. By experimenting with different configurations and encryption libraries, we aim to quantify the computational and communication overhead introduced by HE in the FL workflow and investigate how this overhead can be reduced through the use of a new GPU-based library.

RQ1-Computational and Communication Overhead

- **RQ1.1** How much additional time does the adoption of a HE scheme introduce into the FL workflow?
  We aim to quantify the time overhead introduced by HE during the aggregation phase performed by the server. In fact, we expect the aggregation performed in a homomorphic manner to be more time-consuming. We will evaluate the time spent by each client encrypting and decrypting data during the training rounds. These times represent an additional overhead introduced by HE, since clients do not need to perform these operations in the absence of encryption.

- **RQ1.2** How much does the size of data exchanged between clients and the server increase when the data are encrypted?
  We will compare the size of the data uploaded by the clients - the local model updates shared with the server - and the size of the data downloaded by the clients - the global model sent by the server - in two different scenarios: one where the data are exchanged in plaintext form and one where the data are homomorphically encrypted.

- **RQ1.3** Does the performance of the model remain stable after the adoption of an HE scheme, or does encryption negatively affect the quality of the trained model? We will evaluate whether the two scenarios achieve comparable performance in terms of model accuracy, or whether the adoption of HE negatively affects the final model precision.

We will also investigate whether the time required by clients to train their local models remains stable in the HE scenario. Since local training is performed on plaintext data and does not involve homomorphic operations, we expect no significant difference in training time between the two scenarios.

To answer all these research questions, we evaluate the following two scenarios:

- **Plaintext Scenario:** The data exchanged between clients and the server are in plaintext form. Both the updates shared by the clients and the model sent by the server are not encrypted. The aggregation phase is performed on standard numerical values using conventional operations.

- **TenSEAL Scenario:** The data exchanged between clients and the server are encrypted using the TenSEAL library [8], implementing CKKS encryption scheme. Both the updates shared by the clients and the model sent by the server are encrypted. Clients are responsible for encrypting and decrypting the data, while the aggregation phase is performed using homomorphic operations. Different sets of parameters, including the polynomial modulus degree n, ciphertext modulus q and scaling factor Δ, will be evaluated to analyze how these choices affect model
  performance, computational overhead and the security of the scheme. These parameters will be discussed in more detail in the following section.

The parameters of the FL process are kept identical in both scenarios to ensure a fair comparison between the two approaches.

RQ2-GPU Acceleration Given the parallel nature of homomorphic operations, can encryption, decryption and homomorphic operations be accelerated through the use of GPU?

To address this research question, we will use the HEonGPU library [10], implementing CKKS encryption scheme, which leverages GPU acceleration to speed up HE operations. Therefore, we introduce a new experimental scenario:

- **HEonGPU Scenario:** The data exchanged between clients and the server are encrypted using the HEonGPU library. The FL workflow remains the same as in the TenSEAL scenario, with encrypted client updates and an encrypted global model shared by the server. The aggregation phase is performed in a homomorphic fashion.

We will compare the metrics obtained from the experiments in this scenario with those obtained from the TenSEAL scenario. The parameters of the HE scheme (n, q, Δ) are kept identical in both scenarios to ensure a fair comparison between the two approaches.

Does the size of the ciphertexts remain stable compared to the TenSEAL scenario, or does it increase? Does the model performance remain stable even in the HEonGPU scenario?

To answer these questions, we will measure the aggregation, encryption and decryption times and compare them with those obtained from the TenSEAL scenario.

Moreover, we will measure the ciphertext size generated by the HEonGPU library and compare it with the one generated by the TenSEAL library. Finally, we will verify that the adoption of the new library does not negatively affect the local training time or the final accuracy of the model.

The metrics used in the experiments and the methodology adopted to measure them will be described in more detail in the following sections.

4.4 CKKS Parameter Configuration

The three main parameters of the CKKS encryption scheme are the Polynomial Modulus Degree n, the Coefficient Modulus Chain q_{chain} and the Scaling Factor Δ.

The selection of these parameters has significant implications for both the security level of the scheme and the temporal and computational overhead introduced by HE in the FL workflow. We will evaluate different scenarios defined by combinations of these parameters in order to analyze the overhead introduced and the security levels achieved by each configuration, with the aim of finding a trade-off that ensures acceptable security levels while maintaining good performance.

4.4.1 Polynomial Modulus Degree

The Polynomial Modulus Degree is denoted by n and represents the degree of the cyclotomic polynomial $X^n + 1$, which is used as the modulus in the polynomial ring $\mathbb{Z}_q[X]/(X^n + 1)$. The parameter n is usually chosen as a power of two for efficiency reasons in the CKKS protocol.

A higher value of n increases the security level of the scheme.

A larger n allows for a longer modulus chain without negatively impacting security. In fact, for the most common values of n, standard tables exist that define the maximum bit length for the coefficient modulus q. [28] [29]

However, increasing n makes the scheme computationally more expensive. We recall that each element of the polynomial ring $Z_q[X]/(X^n + 1)$ has at most $n - 1$ coefficients, thus, when n increases, the elements of the polynomial ring have more coefficients. Consequently, increasing n also increases the size of the ciphertexts, since more coefficients must be stored, and therefore the volume of data exchanged between clients and the server. Moreover, the time required for encryption, decryption and homomorphic operations increases with larger values of n.

On the other hand, a higher n allows more values to be encrypted within a single ciphertext. In fact, the size of the input vector is $\frac{n}{2}$. Therefore, if we increase n, we increase the size of a ciphertext and the time required to perform operations on it, but at the same time we need fewer ciphertexts to represent the same amount of data. Consequently, we need to find a trade-off between increased costs and the reduction in the number of ciphertexts to be processed.

4.4.2 Coefficient Modulus Chain

The coefficient modulus q, used to reduce the coefficients of the polynomial ring $Z_q[X]/(X^n + 1)$, is defined as the product of several moduli $q = q_0 \cdot q_1 \cdot \cdots \cdot q_L$. In CKKS, the modulus chain q_{chain} is specified as a list of bit-lengths ql_i. Each modulus q_i is then generated as a random prime number of bit-length ql_i, and is therefore of the order of magnitude of 2^{ql_i}.

Example For example, if we choose the modulus chain $q_{\text{chain}} = [60, 40, 60]$, then the resulting coefficient modulus q is approximately $q \approx 2^{60} \cdot 2^{40} \cdot 2^{60}$ and thus has an overall bit-length of $60 + 40 + 60 = 160$ bits.

Longer modulus chains reduce the security level. For different choices of n, there exist maximum recommended bit-lengths for q.

Each rescaling operation performed after a homomorphic multiplication consumes one modulus from the modulus chain, so a longer chain allows for higher multiplicative depth and more consecutive operations. As we saw in the description of the training protocol, our simulator and our FL workflow do not use homomorphic multiplication, therefore, we do not need to consider the length of the modulus chain. This will not represent a limitation for our purposes.

However, longer modulus chains make the scheme computationally more expensive. Each coefficient of the polynomials in the polynomial ring $Z_q[X]/(X^n+1)$ is represented in the Residue Number System (RNS), meaning it is stored as a vector of its remainders modulo each element of the modulus chain. Therefore, if the modulus chain is longer, each coefficient requires a longer remainder vector for its representation. This is why longer modulus chains lead to larger ciphertexts, increasing communication cost. In addition, in a setting with a longer modulus chain, all operations such as encryption, decryption, and homomorphic arithmetic become more time-consuming.

The central elements of the modulus chain are chosen to match the exponent of the scaling factor Δ. This is done to ensure the correctness of the rescaling operation after each homomorphic multiplication. In fact, after every homomorphic multiplication, rescaling is performed, removing one level of the modulus chain and representing the coefficients of the polynomials in the polynomial ring with a smaller modulus. This operation can be seen as a division by q_i, that is, the modulus that is removed at the current level. If this modulus is close to the value of Δ it effectively behaves as a division by Δ, which is exactly the goal of rescaling, allowing one to return from a scale of Δ^2 to a scale of Δ.

4.4.3 Scaling Factor

The scaling factor Δ is used to transform vectors of real numbers, which represent the input data, into integer values that will form the coefficients of the plaintext polynomials used by the CKKS encryption scheme. In fact, during the encoding phase, every value is multiplied by Δ and then rounded to the nearest integer:

$$\Delta m_i = \lfloor \Delta \cdot m_i \rceil \tag{4.3}$$

This corresponds to a fixed-point representation, where every value has the same precision for the fractional part and the same number of bits is used to represent it. Consequently, higher values of Δ allow more significant digits to be preserved and provide a more precise representation, as more bits of precision are available for the fractional part. However, a larger Δ also requires a larger modulus q. This increases the computational cost and the ciphertext size.

Typically, Δ is chosen as a power of 2, $\Delta = 2^{\delta}$,where δ determines the number of bits of precision reserved for the fractional part.

The scaling factor Δ does not directly affect the security of the scheme.

Chapter 5

Experimental Setup

5.1 Dataset: CIFAR-10

We will use the CIFAR-10 dataset [30] to train and evaluate our model built in a FL framework. The data samples distributed to clients at the beginning of the training process will be the samples from the CIFAR-10 dataset. The data will be used both to train the local model on the clients and to evaluate the global model round by round with various metrics on the server.

CIFAR-10 is a collection of images representing various objects. It is composed of 60,000 low-resolution color images, 32×32 pixels, with 3-channel RGB, meaning each pixel is encoded by three numerical values between 0 and 255 representing the intensity of red, green and blue, which together produce the full range of possible colors. Each image is labeled with a unique class. There are 10 different classes (airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships and trucks). Each class is represented by 6,000 samples, so each label represents the same percentage of samples in the whole dataset.

The dataset is split into a training set, used to locally train the model on the clients, composed of 50,000 samples, and a test set, used during the evaluation phase at the end of each round to compute some monitoring metrics, composed of 10,000 samples.

It was developed by the Canadian Institute for Advanced Research, from which it takes its name. It is a subset of 80 Million Tiny Images [31], another image dataset.

CIFAR-10 is a standard benchmark in machine learning research. It is often used to train and validate models in the literature, making the results comparable. Its small size and simplicity make it a good choice to validate the behavior of models with limited resources, as an initial test.

5.1.1 Preprocessing

For both the training and test sets, the values of all three channels are scaled to the range [0,1] through Min-Max scaling. Both datasets are then standardized ($x' = \frac{x-\mu}{\sigma}$) using the mean and standard deviation of the training set, as good practice.

Additionally, only the training set images are randomly cropped and, with a probability of 0.5, horizontally flipped on the fly. As a result, each batch can see a different version of the same training image when it is selected, with a different crop and possibly horizontally flipped. These last two transformations are obviously not applied to the test dataset because they are useful only for training and could lead to incorrect metrics during evaluation.

5.2 Models

In this section, we describe the architecture of the models used in the federated training process. This is the model stored on the server and distributed to the clients at the beginning of each training round, and the model locally trained by the clients during each round.

At the beginning of our research, we focused on two types of architectures: Convolutional Neural Networks (CNNs) and Residual Networks (ResNets). These two types of architectures are widely used in FL literature [3] [15] [18] and perform well on image recognition, which is the task we will address in this work. In the first experiments, we compared these two models across several metrics related to performance, training time, aggregation time and resource usage in order to decide which model to use in future research. For example, we considered statistics such as the accuracy achieved at the end of training, the average local training time per client across all rounds, the time used by the server to aggregate together the clients'updates to the model and compute the new version of the global model and the amount of exchanged data between clients and server, which depends directly on the model size, i.e. the number of parameters.

We quickly found that the CNN architecture is more suitable for classifying images from the CIFAR-10 dataset, achieving higher accuracy, approximately +15 percentage points, under the same training settings. Training time was also lower: CNN is faster to train and the server aggregates updates more quickly compared to ResNet, which is approximately $\times 3$ slower on average for both metrics. Therefore, in the following experiments, we only consider CNN as the main architecture of our study.

Being a deeper network, we believe that ResNet could perform better with more training rounds or more extensive hyperparameter tuning. However, given the simulation purpose of our study, the parameters are smaller compared to real-world scenarios and are set at this scale to accelerate experiments and data collection. In real-world scenarios, these parameters would of course need to be increased.

5.2.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of deep learning models especially effective for computer vision tasks. They use convolutional filters to automatically extract spatial features from images, such as edges, textures and shapes. CNNs are particularly powerful for image classification, making them well-suited for our dataset and task.

A convolutional filter is a small matrix of learnable weights that slides across the input image to detect specific patterns of the images. During this sliding process, the filter performs an element-wise multiplication between its weights and the corresponding portion of the input, summing the results to produce a single output value. This operation is repeated across the entire spatial dimension of the input, generating a new matrix called a feature map.

A convolutional filter is characterized by a few key elements that determine how it processes the input. The kernel is the filter itself, typically represented as a small square matrix. However, given that the input has multiple channels, the kernel is not simply a 2D matrix, but a 3D one with the same depth as the number of input channels: each input channel has its own kernel and the results obtained from all channels are summed together to produce a single output value at each spatial position. Each convolutional layer operates on a certain number of input channels, which refers to the number of feature maps that the layer receives as input. In addition, the layer produces a certain number of output channels, which indicates how many distinct filters are applied and, consequently, how many feature maps are generated as output. Each filter is trained to detect a different pattern like vertical edges, horizontal edges or other specific textures. Finally, the stride defines how many pixels the filter moves at each step while sliding across the input.

Another important operation in CNNs is pooling. Pooling operations are used to progressively reduce the spatial dimensions of feature maps. Similarly to convolutional filters, a pooling operation slides a small window across the input with a given stride, but instead of applying learnable weights, it summarizes the values within each window using a fixed function. The most common variants are, for example, max pooling and average pooling, which select the maximum and the average value within each window, respectively. Unlike convolutional layers, pooling operations have no learnable parameters.

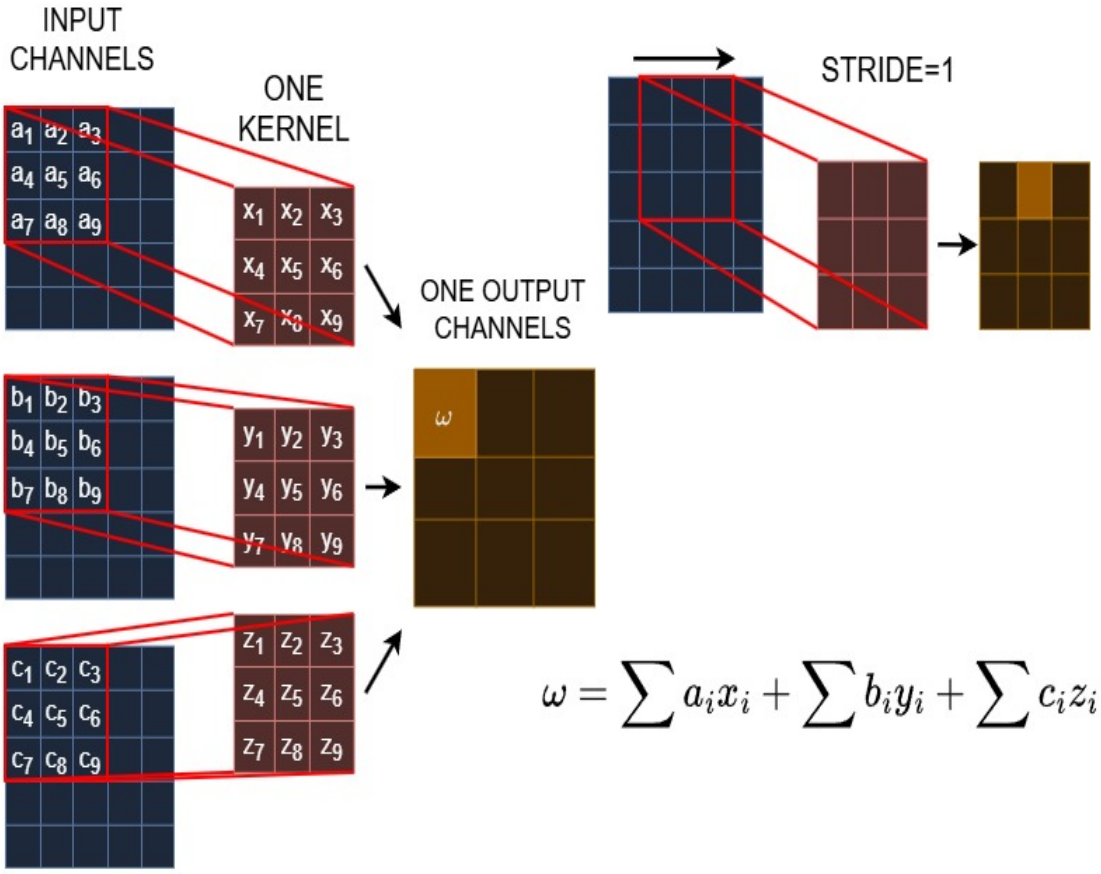


Figure 5.1: Illustration of a convolution filter applied to a multi-channel input

In particular, the architecture used in this work will be referred to as CNN5, since it consists of two convolutional layers and three fully connected layers [1]

The architecture of the model is as follows:

Table 5.1: Architecture of the CNN5 model

	In/Out Channels	Kernel Size (Stride)	
Conv1	3 (RGB) / 64	5×5 (1)	Convolutional Layer
ReLU			$f(x) = \max(0, x)$
MaxPool1		2×2 (2)	Max Pooling
Conv2	64 / 64	5×5 (1)	Convolutional Layer
ReLU			$f(x) = \max(0, x)$
MaxPool2		2×2 (2)	Max Pooling
FC1	1600 (flattened) / 384		Fully Connected Layer
ReLU			$f(x) = \max(0, x)$
FC2	384 / 192		Fully Connected Layer
ReLU			$f(x) = \max(0, x)$
FC3	192 / 10		Fully Connected Layer, 10 is the number of classes in the CIFAR-10 dataset

This architecture has approximately 800,000 trainable parameters. This metric directly influences the amount of data exchanged between clients and server: the larger the model, the higher the communication cost.

5.2.2 Residual Networks

Residual Networks (ResNets) [32] are a class of deep convolutional neural networks designed to improve the training of very deep architectures. They introduce residual learning. Thanks to them, this type of architecture helps mitigate the vanishing gradient problem [33] and makes it possible to train much deeper networks effectively.

[1] A fully connected layer is the classical architecture of a neural network, in which each output neuron is computed as a weighted sum of all the input neurons. The associated weights are learnable.

Residual learning reformulates the learning objective: instead of directly learning a function $H(x)$, the network learns a residual function defined as $F(x) = H(x) - x$. The final output is then computed as $F(x) + x$, recovering the original mapping $H(x)$. This formulation is effective because, in very deep networks, later layers often behave close to identity mappings, since most meaningful features are already extracted in earlier layers. With residual learning, instead of forcing layers to learn the identity function directly, they only need to learn a residual close to zero, which simplifies optimization.

The network architecture we will use is referred to as ResNet20, because it has 20 layers with learnable weights: 1 initial convolutional layer, 18 convolutional layers with residual connections (3 stages, each with 3 `BasicBlocks`, each containing 2 convolutions) and 1 final fully connected layer.

First Convolutional Layer The initial convolutional layer consists of a 3×3 convolution with 3 input channels (RGB), 16 output channels, stride 1 and padding 1, that means that each side of the image is padded with a 1-pixel border of zeros. It follows a normalization layer based on Group Normalization [34] with 2 groups and finally a ReLU activation function.

The network consists then of three stages, each composed of 3 `BasicBlock` modules, that implement residual learning.

- Stage 1: 3 `BasicBlock`, $C_{in} = 16$ input channels and $C_{out} = 16$ output channels, stride 1 for every convolution
- Stage 2: 3 `BasicBlock`, $C_{in} = 16$ input channels and $C_{out} = 32$ output channels, first convolution of first block stride 2, others stride 1
- Stage 3: 3 `BasicBlock`, $C_{in} = 32$ input channels and $C_{out} = 64$ output channels, first convolution of first block stride 2, others stride 1

Table 5.2: Architecture of a `BasicBlock`.

	In/Out Channels	Kernel Size (Stride)	
Main path			
ConvA	C_{in} / C_{out} (1st Block) C_{out} / C_{out} (2nd and 3rd Block)	3×3 (2 or 1)	Each side of the image is padded with a 1-pixel border of zeros
GroupNormA			Normalization layer [34]
ReLU			$f(x) = \max(0, x)$
ConvB	C_{out} / C_{out}	3×3 (1)	Each side of the image is padded with a 1-pixel border of zeros
GroupNormB			Normalization layer [34]
Shortcut path			
Shortcut			Identity or Option A [32]
Merge			
Element-wise addition			Sum of the main and shortcut paths
ReLU			$f(x) = \max(0, x)$

In summary, the `BasicBlock` is composed of two parallel paths: the main path (ConvA → GroupNormA → ReLU → ConvB → GroupNormB) and the shortcut path (Identity or Option A [32]). The outputs of these two paths are summed element-wise and then passed through a final ReLU activation. This represents how the residual connection, typical of ResNet architectures, is implemented in our design.

The shortcut connection follows the Option A design proposed in [32]. When the input and output feature maps have the same dimension and number of channels, the shortcut is implemented as an identity mapping. Otherwise, no additional learnable parameters are introduced. Instead, the input feature map is spatially downsampled by selecting one pixel every two along both spatial dimensions , equivalent to a stride of 2, reducing the height and width by a factor of two. To match the increased number of output channels, zero-valued feature maps are padded before and after the original channels. This parameter-free transformation produces a tensor with the same dimensions as the output of the main branch, allowing the two paths to be added element-wise.

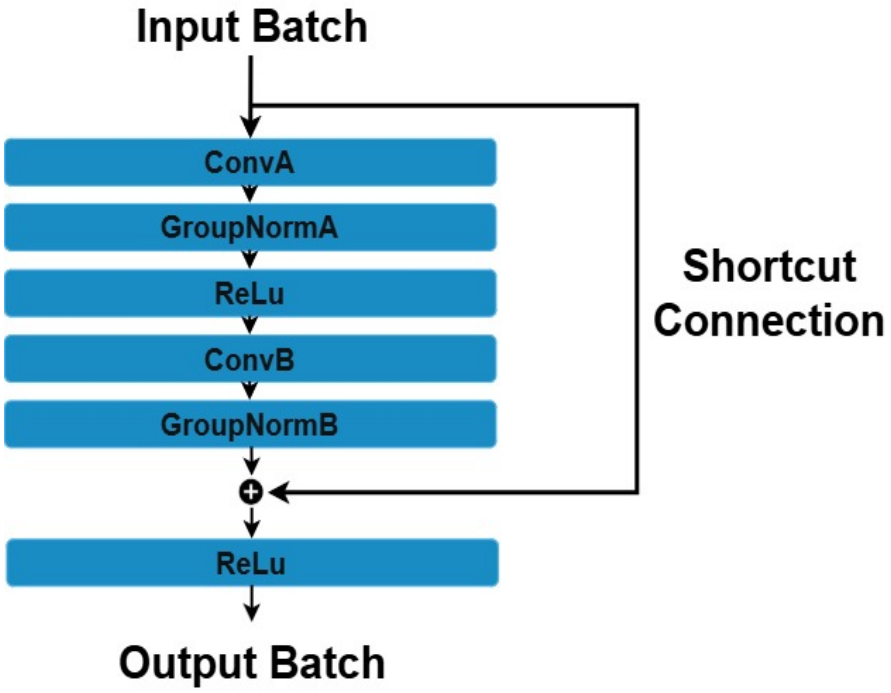


Figure 5.2: Residual Block with Skip Connection

Final Fully Connected Layer The final fully connected layer consists of a global average pooling layer, collapsing each channel into a single value. The resulting tensor is then flattened into a 1D vector of size 64. Finally, a fully connected linear layer maps the 64-dimensional representation to 10 output units, corresponding to the number of classes in the CIFAR-10 dataset.

This architecture has approximately 270,000 trainable parameters. Although ResNet20 is deeper, it uses fewer parameters than CNN5.

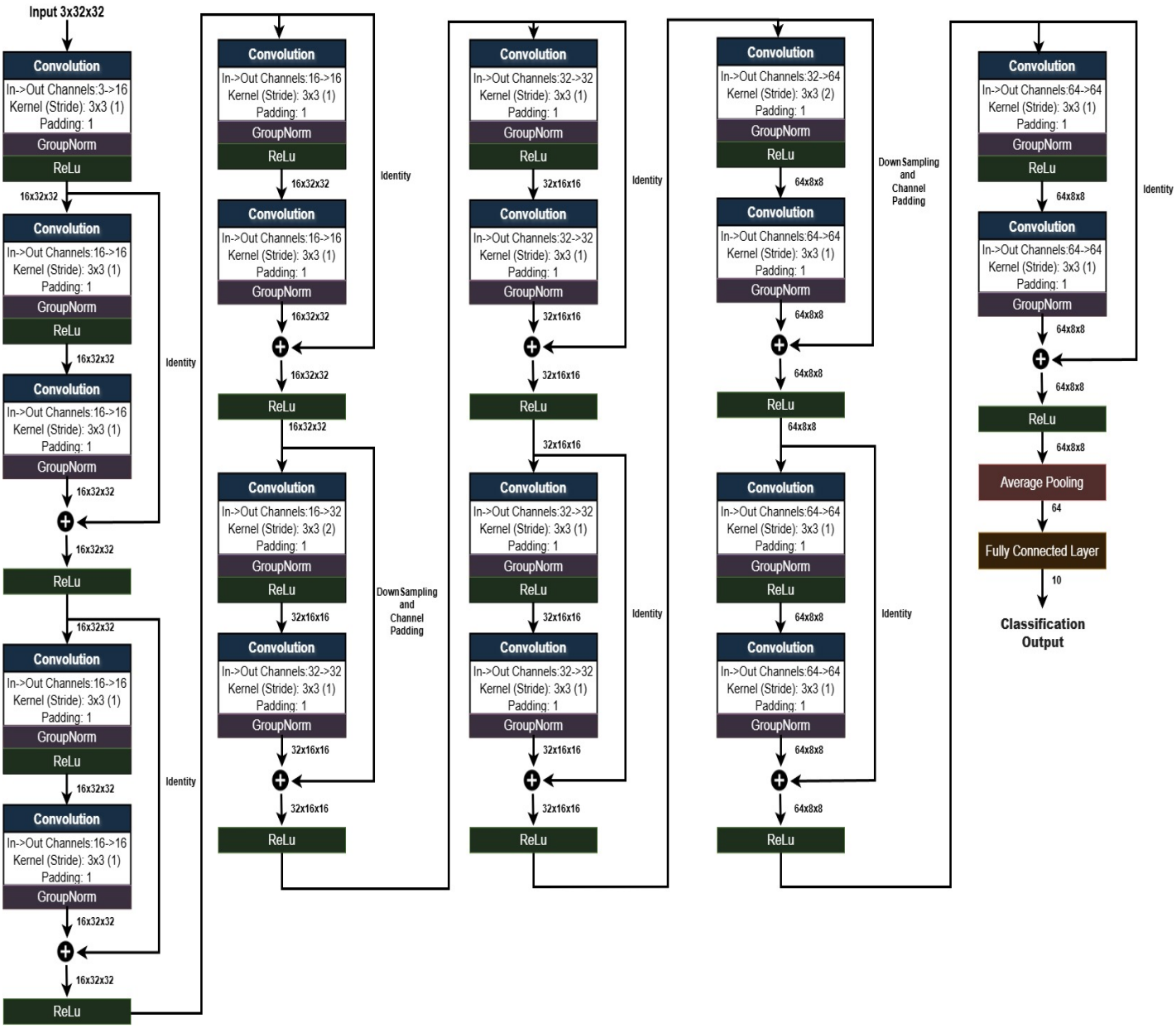


Figure 5.3: Architecture of the ResNet20 model

5.3 Evaluation Metrics

In this section, we will take a look at the metrics that guided our research throughout this thesis. These statistics helped us comment on each experiment, understand the behaviour of the simulator under different training and encryption parameter settings and make decisions for our future research.

We divide our metrics into three macro-categories:

- statistics related to the performance of the model, which indicate how well the model performs the task it is designed for
- statistics related to time, namely how much time our tool takes to perform different types of operations (aggregation by the server, training, encryption and decryption by the clients). These are particularly important to quantify the overhead introduced by HE in the FL process and to understand whether our proposed solution, the GPU-based HE library, can help mitigate this problem
- statistics related to the amount of data exchanged between the clients and the server

Accuracy The accuracy is computed during the evaluation phase as the fraction of correct predictions over the total number of predictions on the test set. During the evaluation phase, our model enters inference mode and passes the test dataset through the network, making predictions about the labels of the samples without updating the weights anymore. If these predictions correspond to the original ground-truth labels of the samples in the test dataset, they are considered correct.

Accuracy is measured at the end of every global training round. In our work, this metric is computed as a moving average over the last 30 rounds. This means that, at every round, we take into account the accuracy values from the last 30 rounds and compute their average. This is done because the accuracy can fluctuate significantly between rounds and we want to avoid being misled by temporary spikes. Our objective is to reduce the noise.

Before introducing the metrics related to time, we want to provide a small clarification about how our simulator works and how these metrics are measured. In our tool, the clients work sequentially: at every round, the selected clients perform the local fine-tuning one at a time, one after another. On the contrary, in a real-world scenario, the clients would work in parallel, performing the local training simultaneously.

The client-side time metrics (training time, encryption time and decryption time) are measured for all clients at every round. Every time a client enters one of these phases (training, encryption or decryption), the time spent by that client on that activity is measured. At the end, the times measured for every client in every round are summed together to obtain the total time spent by all clients over all rounds. Finally, the results reported in the tables are obtained by dividing this total time by the total number of clients, N. In this way, we report the average time that a client spends performing these activities (training, encryption and decryption) over all the global rounds, `num_rounds`.

Training Time The training time measures the time spent by each client during the local training phase. In particular, it includes the time required to perform the forward pass, the loss computation, the gradient computation and the weight update. We recall that local training is performed in plaintext, even when HE is employed. For this reason, we expect this time to be stable across every scenario and every encryption parameter setting, both in the plaintext case, in the TenSEAL case and in the HEonGPU case, because the local training phase is not affected by different HE settings.

Encryption Time The encryption time measures the time spent by each client encrypting the model updates after local training before communicating them to the server. Obviously, in the plaintext scenario, where HE is not used, this time is 0. As for the training time and the decryption time, this is a client-side time metric: encryption is a routine performed by the clients and it is measured on the client side.

Decryption Time The decryption time measures the time spent by the clients decrypting the weights of the global model that the server shares with them at the beginning of the local training phase, if they are selected for the round. As for the encryption time, in the plaintext scenario this time is clearly 0. The decryption time does not include the time that the server uses to decrypt the model before the evaluation phase if the global model is stored encrypted. As mentioned in previous chapters, this is a limitation of our tool that deviates from the usual real-world behaviour. Therefore, we do not want to affect this metric with this misleading time, as it would not represent a realistic scenario.

Aggregation Time The aggregation time measures the time spent by the server to aggregate the pseudogradients received from the selected clients and update the global model. In particular, it includes the time required to sum, parameter by parameter, all the pseudogradients from the clients and then add this quantity to the global model to compute the new version. This is a server-side time metric, as the aggregation phase is performed entirely by the server. Differently from the client-side time metrics, this time is obviously not averaged and represents the total time across all global rounds that the server needs to perform the aggregation phase.

We want to highlight that all experiments were performed on a remote server with shared computational resources, including shared CPU resources and two GPUs. Since the machine was not dedicated exclusively to our experiments, the workload generated by other users could affect the execution time of some operations and consequently introduce small variations in the collected metrics. For example, when the server was under heavy workload, some routines could require more execution time due to limited resource availability, while during periods of low activity the same operations could be completed faster. To minimize the impact of these external factors, we repeated the experiments during periods in which the server workload was low, aiming to improve the comparability of the results across different experiments. It is important to note that this external variability only affects time-related metrics, as the execution time of the operations can change depending on the server workload. On the other hand, metrics such as the exchanged data and the model accuracy are not influenced by the workload of the machine.

Exchanged Data We measure the amount of data exchanged between the clients and the server during the two communication phases of the federated learning process: when the server sends the global model to the selected clients at the beginning of the local training phase and when the clients send their model updates back to the server. We expect the amount of exchanged data to be approximately the same in both directions. This is because the global model sent by the server and the pseudogradients sent by the clients have the same number of elements: the clients compute one pseudogradient value for each weight of the model. Moreover, in the plaintext scenario, both the global model and the pseudogradients are transmitted in plaintext, while when HE is employed, both are transmitted in encrypted form. Therefore, we expect the size of the data exchanged from the server to the clients and from the clients to the server to be comparable. For this reason, to separately estimate the uploaded traffic from the clients to the server and the downloaded traffic from the server to the clients, we divide the total exchanged data in half. Every object transmitted through the communication channel is measured and converted into bytes in order to estimate its actual size.

5.4 Parameters

In this section we present the parameters that can be tuned in our simulator. We have two classes of parameters: the first type of parameters is related to the training process, the second one to the encryption applied in the FL process. There are some parameters that will be fixed for every experiment and their values will not change throughout this work. Others, on the other hand, that in Table 5.3 are indicated with $*$, will change across experiments and their value will be specified before every scenario. This variation of parameters guides our research: observing the tool under various parameter settings offers us the possibility to inspect how these parameters influence the behaviour of the simulator and to find a compromise between performance, security and computational and communication costs.

Table 5.3: Federated Learning and Encryption Hyperparameters used in the Experiments

Parameter	**Value**
Encryption Library and Scheme	$*$
Model	$*$
`num_rounds`	$*$
N	100
`round_clients`	$*$
`local_lr`	0.01
ls	20
`train_batch_size`	50
`test_batch_size`	100
α	$*$
n	$*$
q_{chain}	$*$
Δ	$*$

The two library options that we will present in this work are TenSEAL and HEonGPU. TenSEAL is chosen as baseline because it is a popular library
implementing HE and it will be used to conduct a first assessment of the costs introduced by HE in FL. HEonGPU, on the other hand, will be our proposed solution to speed up the HE operations and reduce the time needed in the scenarios that use HE, through the use of the GPU during homomorphic operations. Both libraries implement the CKKS scheme, which is the scheme that we will use because it suits a machine learning scenario perfectly, being able to handle real values such as model weights. There is also the possibility of no encryption at all in the FL process. This scenario will work as a baseline for the comparison, inspecting the computational and communication costs of HE applied to FL.

In the previous section 5.2 we presented the model architectures that we will consider in our work: ResNet20 and CNN5. However, for most of our experiments we will use CNN5 because, after some initial analysis, we found out that this architecture fits our objectives better and is less time-consuming.

`num_rounds` is the total number of global rounds of federated training. The FL pipeline will be repeated `num_rounds` times. N is the total number of clients and `round_clients` is the number of clients that participate in the local training each round, because not all N clients participate in every round of training, but only a random subset of them.

`local_lr` is the learning rate used by the clients in the local training. ls is the number of local training steps performed by each client when it is selected for a round. For each local step, `train_batch_size` samples are passed through the network and a label is predicted for each sample, the loss is computed and, consequently, the gradients with respect to the weights of the model, then the weights are updated.

During the evaluation phase, the entire test dataset is not passed through the network at once, instead, the test dataset is sliced into sets containing `test_batch_size` samples and each set of `test_batch_size` samples is entirely passed through the network to compute the accuracy metric. This is done for memory reasons, because passing the entire test dataset at one time could become too heavy. `test_batch_size` must be a divisor of 10,000, which is the length of the total test dataset, because we need to pass through the entire test dataset before computing the accuracy.

α is the parameter of the Dirichlet distribution used to assign samples from the entire dataset to the clients at the beginning of the FL training process. Low values of α lead to very non-IID partitions, while higher values lead to clients that have quite the same percentage of representation of labels in their dataset.

Finally, n, q_{chain} and Δ are the parameters for the CKKS scheme when HE is employed 4.4. Obviously, in the plaintext scenario these parameters do not need to be specified.

Chapter 6

Results

6.1 Plaintext Scenario

The first case that we will inspect is the Plaintext Scenario. In this setting, HE will not be employed. At the end of the local training performed by clients, the nodes share the computed updates to the global model in plain form. Similarly, at the beginning of every global training round, the server shares the weights of the global model to be fine-tuned with the clients in plain form. Both of these practices pose a serious threat to the security of our system: as previously discussed, an external adversary that eavesdrops on the communication channel or even an untrusted server can have access to the global model specifications. Moreover, both the untrusted server and external adversaries that sniff the communication channel can reconstruct some information about the local datasets of clients starting from the pseudogradients shared by the clients [1] [2].

The experiments from this scenario will serve as a baseline. We will compare the results achieved in these runs with those obtained from runs that use HE to encrypt the global model specifications and the local updates from clients under the same training parameters, ensuring a fair comparison. In this way, we will provide an initial evaluation of the overhead introduced by the use of HE in the FL training process, considering how much additional time is introduced for both clients and the server and how much the amount of data exchanged over the communication channel increases.

Initially, we decided to perform a parameter sweep on two training hyperparameters: the architecture of the trained model and `round_clients`, the number of clients that participate in each training round.

As discussed in 5.2, we will inspect the two presented architectures, CNN5 and ResNet20. These architectures have been considered because they provide good performance in image recognition, which is the task for which we train the model using the CIFAR-10 dataset, and because they are widely used in machine learning research. In this way, we can compare our results with those presented in the machine learning research literature. We want to compare these two architectures in order to make a choice for future experiments. We will select one of the two options to use in the following experiments, while the other architecture will no longer be considered. The choice will be based on the accuracy achieved by the models at the end of the global training, the delays accumulated during the training process, considering both the training time and the aggregation time, and the amount of data that the clients and the server need to exchange during the training process, which is directly dependent on the number of parameters of the models.

Secondly, we will also analyze how the `round_clients` parameter influences the collected metrics. We expect that, with a higher number of clients participating in the local training at each round, therefore with more updates to the global model in every round, we will achieve a higher level of accuracy. At the same time, we expect that the average time spent by clients training the model locally across all rounds, as well as the global aggregation time spent by the server across all rounds, will increase.

Table 6.1: Plaintext Scenario: Sweeping on `round_clients` and Model Architecture, `num_rounds` = 100, $\alpha = 0.3$

Round Clients	**Model Architecture**	**Accuracy (%)**	**Training Time (s)**	**Aggregation Time (s)**	**Exchanged Data (GB)**
10	CNN5	35.82	0.84	0.20	5.95
10	ResNet20	20.64	2.40	0.57	2.01
20	CNN5	40.02	1.60	0.31	11.89
20	ResNet20	26.15	4.79	1.00	4.02

We can see that the CNN5 model achieves better accuracy than the ResNet20 model. We believe that ResNet20 needs more training rounds to achieve better performance because it is a deeper network. In other words, in a short-round setting such as ours, CNN5 is favored due to its faster convergence, while ResNet20 is expected to outperform CNN5 over a longer training session, once it has had enough rounds to fully exploit its greater depth.

In general, we can see that the accuracy is low for every parameter setting. This is expected because the total number of global training rounds is limited to 100. In practical FL scenarios, a significantly higher number of global training rounds is usually required to achieve better model performance.

However, in this work, we intentionally use a limited number of global training rounds to speed up the collection of experimental results, since longer training sessions require more time. At this stage, our focus is on evaluating other aspects of privacy-preserving FL, such as the overhead introduced by HE, how it can be reduced through the use of GPU libraries and to verify that neither the use of HE nor its GPU-accelerated implementation negatively affects the learning performance of the model, therefore does not reduce the achieved accuracy. For this reason, the relatively low accuracy observed in these experiments does not represent a limitation for our analysis. At the end of this work, we will also perform experiments with a larger number of global training rounds to demonstrate that higher accuracy can be achieved.

Additionaly, ResNet20 is slower to train: it requires more local training time on the nodes and more aggregation time on the server side.

Regarding the local training time, ResNet20 is a deeper model, consequentially during each local training step, both the forward and backward passes require more computational operations due to the higher number of layers. In particular, the gradient computation and propagation through the network, together with the subsequent weight updates, require more time compared to CNN5. This results in a training time approximately three times higher for ResNet20 compared to CNN5, for both configurations, `round_clients` = 10 and `round_clients` = 20.

A similar behavior can be observed for the aggregation time. Despite having a lower number of parameters compared to CNN5 (approximately 270,000 parameters for ResNet20 against approximately 800,000 for CNN5), ResNet20 requires more time during the aggregation phase. This can be explained by the different organization of the model parameters. ResNet20 is composed of a higher number of smaller layers, while CNN5 contains fewer layers with larger tensors. Since the aggregation procedure iterates over the layers during the update of the global model, the higher number of components in ResNet20 introduces additional overhead. As a result, the aggregation time of ResNet20 is also approximately three times higher than the one of CNN5, for both configurations, `round_clients` = 10 and `round_clients` = 20.

Overall, CNN5 provides a faster training, despite having a significantly higher number of parameters.

By contrast, the exchanged data metric is directly dependent on the number of model parameters. We can observe that the runs training ResNet20, which is a lighter model with a lower number of parameters than CNN5, require a smaller amount of data to be exchanged between the clients and the server, for both configurations, `round_clients` = 10 and `round_clients` = 20.

When considering the effect of the sweep on `round_clients` on the behavior of the simulator, we can note that, as expected, the accuracy increases with the number of clients selected to participate in a local training round.

With a higher `round_clients` value, the global model at each round receives more information through a higher number of pseudogradients shared by the clients, speeding up the convergence process and leading to better accuracy.

We can see that the Training Time and Exchanged Data metrics are directly related to `round_clients`: when we double it, both the statistics double. This is because increasing the number of participants in each round increases the probability of being selected to participate in a round. Therefore, by doubling this value, we also double the average time that a client spends fine-tuning the global model locally, as well as the number of clients that share information with the server at each round.

Finally, we can observe that the aggregation phase performed by the server is also longer for runs with a greater `round_clients` value, but the correlation is less strong and the values do not exactly double. However, performing aggregation with a larger number of updates is naturally more computationally expensive, as a larger amount of data has to be processed.

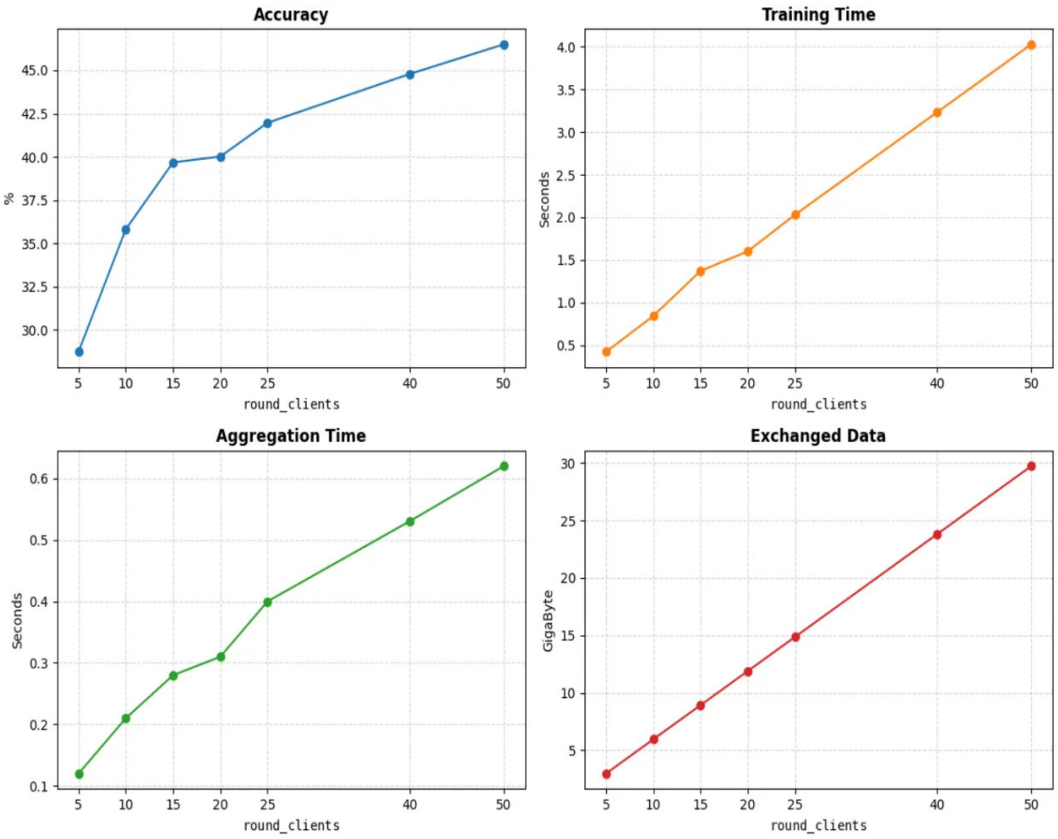


Figure 6.1: Plaintext Scenario: Further `round_clients` sweep with fixed model architecture. Model Architecture = CNN5, `num_rounds` = 100, $\alpha = 0.3$

Figure 6.1 shows the metrics of additional runs with a larger sweep on the `round_clients` parameter, keeping the model architecture fixed to CNN5. For the reasons mentioned previously, from now on we will only consider the CNN5 model, as it achieves higher accuracy and is faster to train, and we will no longer inspect the ResNet20 model.

From figure 6.1 we can confirm our previous observations. The accuracy achieved at the end of the FL training process grows with the number of clients that participate in each round. However, beyond a certain threshold the curve flattens out and the gain becomes smaller, meaning that the extra information obtained from more clients participating in local training yields only marginal benefits.

For the other metrics, we can clearly see that they are quite linear. As `round_clients` increases, they increase linearly as well. The aggregation time is not perfectly linear, the training time is more linear and the exchanged data is perfectly linear.

We recall that we are working on a shared server with shared resources, so the workload of the server can sometimes affect the collected metrics, but only the time-related ones, not the communication cost. Even though we tried to repeat each run under the same best-case workload conditions, the noise observed in the time-related metrics can be attributed to this limitation.

These observations match our analysis: more clients participating in each round means more pseudogradients to aggregate and process, which makes the aggregation phase longer. If more clients participate in a round, each client has a higher chance of being selected, so the average time spent training the model locally, across all global rounds, increases. Finally, more clients communicating every round on the communication channel inevitably increases the communication cost.

6.2 TenSEAL Scenario

In this setting we will employ the TenSEAL library [8], implementing the CKKS homomorphic encryption scheme, to protect our communication channel. TenSEAL is a widespread library, commonly used in the literature [35][36][37]. It is a popular choice and it will serve two objectives: first, to quantify the computational and communication overhead introduced by HE in the FL training process, by comparing the runs from the previous Plaintext Scenario 6.1 with runs that use the TenSEAL library to encrypt the communication channel under the same parameter settings, and second, it will be used as a baseline to compare with the HEonGPU Scenario, in order to understand whether the GPU-accelerated library can speed up the homomorphic operations without affecting the performance of the model and the communication cost.

In addition, TenSEAL automatically handles tensors, which are the basic building block on which deep learning libraries such as PyTorch [38] or NumPy [39], used in our simulator, operate. This makes TenSEAL a natural choice to first assess the impact of HE on our FL framework, together with the popularity of the library.

In this study case, the clients encrypt the model updates computed after the local training before sending them to the server. On the other hand, the server stores and communicates the model specifications in an encrypted form without ever having access to them. Additionally, it performs the aggregation phase using homomorphic operations. Also, the clients need to decrypt the global model sent by the server before fine-tuning it.

In these experiments we will perform a parameter sweep on the arguments of the CKKS encryption scheme, n, q_{chain} and Δ. As we described in 4.4, we will try to find a tradeoff between the security of the scheme, the performance achieved by the model in terms of accuracy, the computational overhead introduced by HE and the communication cost in terms of exchanged data. We will then compare these runs with the previous ones in the Plaintext Scenario, keeping the training parameters fixed in order to ensure a fair comparison.

The values for the arguments of the CKKS encryption scheme, n and q_{chain}, subject to this sweep, were selected in order to respect, and in some cases exceed, the 128-bit security parameter, considered as the baseline security level, which limits the bit-length of the coefficient modulus q based on the value of n [28] [29]. The values for n were chosen among those typically adopted in the literature [35] [36]. For each value of n, different combinations of q_{chain} were tested in order to explore the computational and communication cost derived from them. We recall that our system does not require a particular multiplicative depth guaranteed by q_{chain}, since the aggregation phase is performed in an additive manner, consequently, the combinations of the coefficient modulus chain were chosen for research purposes only. We also tested different values of Δ, which does not affect the security of the scheme, in order to assess whether this parameter influences the performance of the model in terms of accuracy, given that Δ controls the precision of the encrypted numbers. The values for Δ were chosen among those typically used in the literature [35] [36].

Table 6.2: TenSEAL Scenario : n and q_{chain} sweep, $\Delta = 2^{30}$, `round_clients` = 15, Model Architecture = CNN5, `num_rounds` = 100, $\alpha = 0.3$

n	q_{chain}	**Accuracy (%)**	**Training Time (s)**	**Aggregation Time (s)**	**Encryption Time (s)**	**Decryption Time (s)**	**Exchanged Data (GB)**
8192	[60,40]	39.67	1.26	11.17	12.93	1.73	73.36
8192	[60,40,40]	39.66	1.23	20.87	16.65	3.70	131.55
8192	[60,40,40,40]	39.66	1.22	29.75	20.24	5.32	186.83
16384	[60,40]	39.66	1.22	9.88	13.62	1.86	75.88
16384	[60,40,40]	39.65	1.25	20.98	17.60	3.86	132.95
16384	[60,40,40,40]	39.67	1.25	33.47	21.89	5.72	190.22

Table 6.3: TenSEAL Scenario : n and q_{chain} sweep, $\Delta = 2^{40}$, `round_clients` = 15, Model Architecture = CNN5, `num_rounds` = 100, $\alpha = 0.3$

n	q_{chain}	**Accuracy (%)**	**Training Time (s)**	**Aggregation Time (s)**	**Encryption Time (s)**	**Decryption Time (s)**	**Exchanged Data (GB)**
8192	[60,40]	39.67	1.23	11.55	12.91	1.72	73.36
8192	[60,40,40]	39.66	1.25	20.91	16.69	3.69	131.55
8192	[60,40,40,40]	39.66	1.24	31.50	20.46	5.34	186.83
16384	[60,40]	39.67	1.23	9.95	13.56	1.85	75.88
16384	[60,40,40]	39.67	1.24	22.10	17.67	3.83	132.95
16384	[60,40,40,40]	39.68	1.22	32.59	21.87	5.64	190.22

Training Time From Tables 6.2 and 6.3 we can observe that the training time remains stable across all parameter settings. This confirms our predictions, as we expected the different configurations of the CKKS parameters not to impact the time spent by clients during local model training, since the local fine-tuning phase is performed in plaintext. For the same reason, it is also a positive result that the training times are comparable to the one measured in the plaintext scenario with the same number of `round_clients` ($1.37s$) considering the measurement noise caused by the variable workload on the server during the experiments. Therefore, as expected, neither the different CKKS parameter configurations nor the use of HE itself have an impact on the training time.

In general, we can note that encryption is a more time-consuming routine than decryption. This is due to the fact that the encryption operation involves random number generation processes, which are resource-intensive, while decryption consists of a standard circuit evaluation.

Accuracy From Tables 6.2 and 6.3 we can observe that the accuracy metrics are comparable to the one obtained in the Plaintext Scenario using the same number of clients participating in each round and the same training hyperparameters, which was 39.68%. This is a positive result, as it indicates that the applied encryption scheme does not negatively affect the model performance, even considering that the CKKS encryption scheme is an approximate algorithm. It is also noticeable that, as expected, n and q_{chain} do not impact the accuracy metric and all the runs achieve the same level of precision. Contrary to our expectations, increasing the Δ value does not lead to higher accuracy levels.

Impact of Δ We can see that the accuracy values from Tables 6.2 and 6.3 are basically the same, meaning that increasing Δ does not lead to better performance. This was not expected because, with a greater Δ, more information about the fractional part of the numbers is preserved during the encoding phase, therefore, using more precise values, we expected more accurate computations and, consequently, a model more closely fitted to the data and with better performance on the test dataset. We believe that this did not happen because $\Delta = 2^{30}$ is already a good value, allowing enough information to be preserved to achieve this level of accuracy. Increasing the value of Δ to 2^{40} does not provide any additional benefit because it apparently does not introduce enough extra information to achieve better performance. Instead, improving the model performance likely requires increasing the training parameters, such as the number of global rounds `num_rounds`. Finally, we can also note that the other metrics, such as execution times and data costs, are not affected by the change of Δ and remain similar across the different parameter configurations. This was expected because Δ does not influence operations such as encryption, decryption or the addition between two encrypted vectors, nor does it affect the ciphertext size.

Impact of n We can observe from the tables above that runs with the same q_{chain} but different n values, namely 8192 and 16384, show comparable time metrics (aggregation time, encryption time and decryption time) as well as communication costs.

In general, larger values of n increase the computation time required for encryption, decryption and homomorphic operations. This is because a larger n corresponds to polynomials with a higher degree, which are the fundamental building blocks of the CKKS scheme, and are therefore more computationally expensive to process. The time required to encrypt a single vector, decrypt a single vector or homomorphically add two encrypted vectors does not exactly double when n is doubled, but it is close to this behavior. A similar consideration applies to the ciphertext size, which directly determines the amount of data exchanged throughout the training process. In this case, doubling n doubles the number of coefficients that must be stored in each ciphertext, making the ciphertext size twice larger.

However, another important aspect must be taken into account. A larger n allows more values to be packed into the same ciphertext, reducing the total number of ciphertexts required to encrypt the model. In CKKS, each encrypted vector stores $\frac{n}{2}$ values at the same time. Therefore, doubling n doubles the number of values that can be packed into a single ciphertext, reducing by approximately half the number of ciphertexts that need to be processed for the same model. We measured that encrypting our CNN5 model requires 199 encrypted vectors when using $n = 8192$, instead only 101 encrypted vectors are needed when using $n = 16384$.

These two factors explain why runs with the same parameters but different values of n (8192 and 16384) show similar computation times and communication costs. With $n = 8192$, individual operations are approximately twice faster, but twice as many ciphertexts must be processed whenever the model is encrypted, decrypted or aggregated through homomorphic addition. On the contrary, with $n = 16384$, each operation is approximately twice more expensive, but only half of the ciphertexts need to be processed when encrypting, decrypting or homomorphically aggregating the entire model. A similar trade-off applies to the communication cost: with $n = 8192$, ciphertexts are half the size, but twice as many must be transmitted over the communication channel, with $n = 16384$, each ciphertext is twice larger, but only half of the ciphertexts need to be be transmitted.

Table 6.4: TenSEAL Benchmark: Encryption and Decryption Time of a single vector of length $\frac{n}{2}$ and homomorphic Addition Time for two encrypted vectors of length $\frac{n}{2}$, measured on our server for different configurations of n and q_{chain}.

Encryption Time (ms)

q_{chain}	$n = 8192$	$n = 16384$
[60,40]	5.47	11.13
[60,40,40]	7.02	13.78
[60,40,40,40]	8.50	17.16

Decryption Time (ms)

q_{chain}	$n = 8192$	$n = 16384$
[60,40]	0.69	1.43
[60,40,40]	1.49	3.15
[60,40,40,40]	2.19	4.67

Addition Time (μs)

q_{chain}	$n = 8192$	$n = 16384$
[60,40]	37.35	71.48
[60,40,40]	74.00	163.22
[60,40,40,40]	121.25	241.50

Impact of q_{chain} q_{chain} is clearly the most determinant factor influencing the increase of encryption, decryption and aggregation times, as well as the communication cost. When the length of the q_{chain} is increased, all the time related metrics increase for both $n = 8192$ and $n = 16384$. The amount of exchanged data between clients and the server also increases because each coefficient of the polynomials used to represent data in the CKKS scheme is represented through the Residue Number System (RNS), meaning that each coefficient is stored as a vector containing its remainders modulo each element of the modulus chain, therefore, when the modulus chain is longer, each coefficient requires a longer remainder vector for its representation. This results in larger ciphertexts and, consequently, higher communication costs.

Additionally, we can observe that the transition from [60,40] to [60,40,40] has a stronger impact on time related metrics and on the amount of exchanged data than the transition from [60,40,40] to [60,40,40,40].

The Encryption Time is the most stable metric, showing an increase of $\times 1.29$ for both $n = 8192$ and $n = 16384$ when moving from [60,40] to [60,40,40], while when increasing the modulus chain further, from [60,40,40] to [60,40,40,40], the increase is $\times 1.22$ and $\times 1.24$ for $n = 8192$ and $n = 16384$ respectively.

The Decryption Time is the metric most affected by the first increase in the modulus chain. Moving from [60,40] to [60,40,40], the decryption time more than doubles for both $n = 8192$ and $n = 16384$. The additional increase from [60,40,40] to [60,40,40,40] leads to a smaller growth, with factors of $\times 1.44$ and $\times 1.48$ for $n = 8192$ and $n = 16384$ respectively.

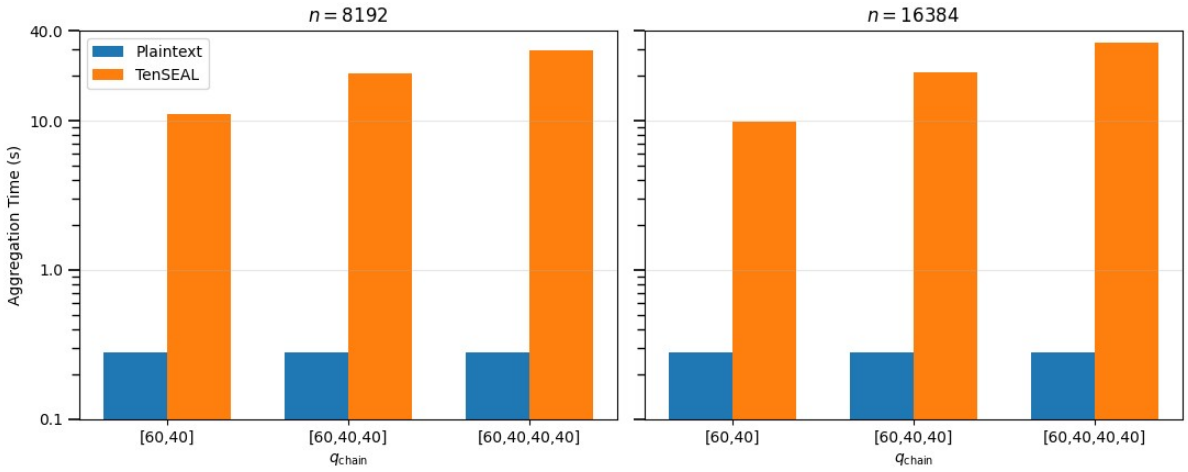
Finally, Aggregation Time and Exchanged Data follow a similar trend, with a larger increase occurring when adding the third modulus level rather than the fourth one. For Aggregation Time, the transition from [60,40] to [60,40,40] results in increases of $\times 1.87$ and $\times 2.12$ for $n = 8192$ and $n = 16384$ respectively. On the other hand, the transition from [60,40,40] to [60,40,40,40] produces increases of $\times 1.43$ and $\times 1.60$ for $n = 8192$ and $n = 16384$ respectively.

Regarding Exchanged Data, the increase from [60,40] to [60,40,40] is $\times 1.79$ and $\times 1.75$ for $n = 8192$ and $n = 16384$ respectively. The additional increase from [60,40,40] to [60,40,40,40] produces a smaller growth, with factors of $\times 1.42$ and $\times 1.43$ for $n = 8192$ and $n = 16384$ respectively.

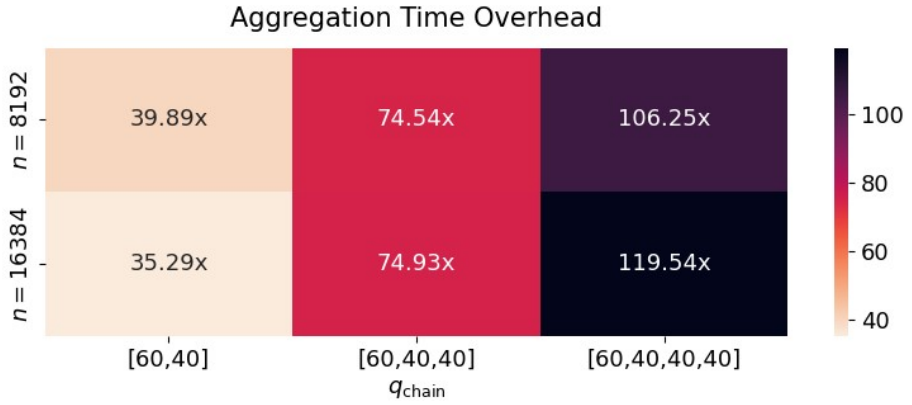
It is important to recall that in our framework homomorphic multiplication is not used during the aggregation phase. This is the reason why extending the coefficient modulus chain is not necessary. In fact, increasing the modulus chain is generally required to support a larger multiplicative depth. However, in our scenario, increasing q_{chain} only introduces higher computational and communication costs and reduces the security level of the scheme.

6.2.1 Evaluation of the Temporal and Communication Overhead Introduced by TenSEAL

We can now compare the runs from the Plaintext Scenario with those from the TenSEAL Scenario in order to quantify the temporal and communication overhead introduced by HE in our FL framework. The comparison is performed under the same training hyperparameters to ensure a fair evaluation, while sweeping different encryption parameter settings to analyze how their values influence the experimental results.

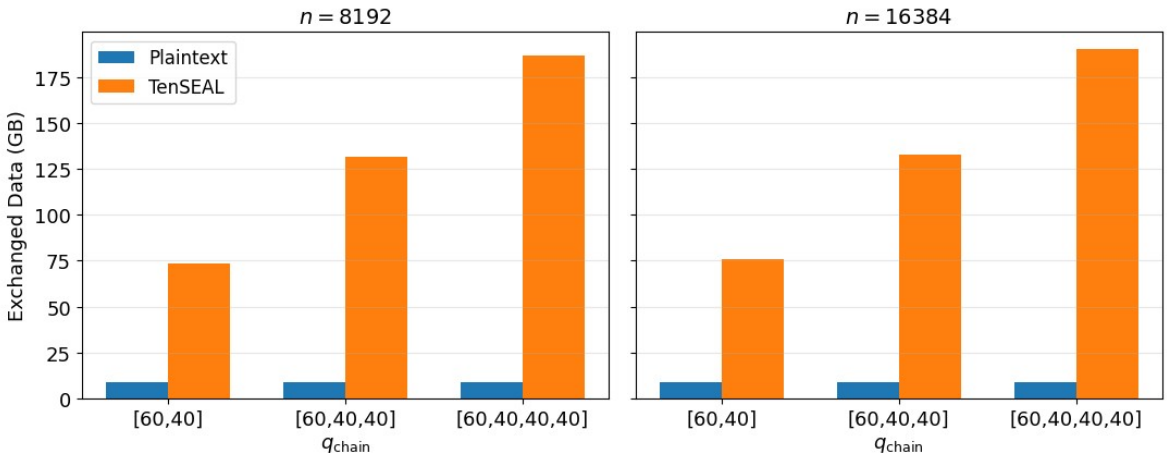


(a) Aggregation Time comparison between Plaintext and TenSEAL Scenarios

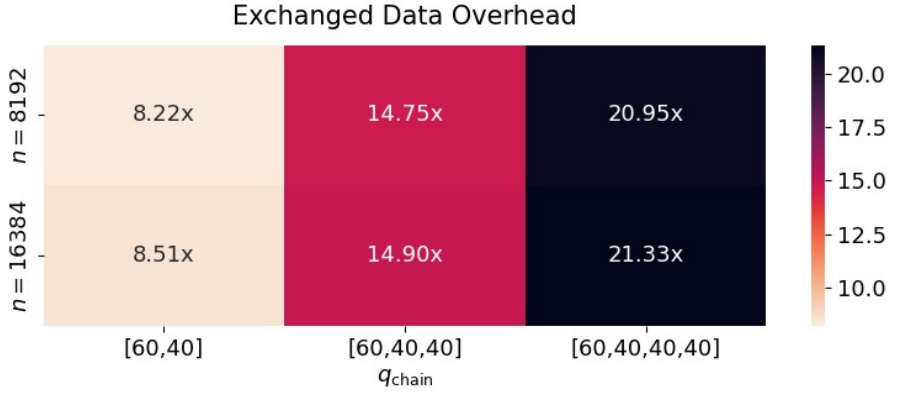


(b) Aggregation Time Overhead introduced by Homomorphic Encryption

Figure 6.2: Impact of Homomorphic Encryption on Aggregation Time, $\Delta = 2^{30}$



(a) Exchanged Data comparison between Plaintext and TenSEAL Scenarios



(b) Aggregation Time Overhead introduced by Homomorphic Encryption

Figure 6.3: Impact of Homomorphic Encryption on Exchanged Data, $\Delta = 2^{30}$

In the figures 6.2 and 6.3, the statistics regarding the final reached Accuracy and the average training time spent by each client across all training rounds are not reported because, as already discussed, they remain stable between the Plaintext and the TenSEAL Scenarios. This confirms, as expected, that these metrics are not affected by the use of HE in our framework.

Furthermore, the Encryption and Decryption Times are not reported in 6.2 and 6.3 because, obviously, in the Plaintext Scenario these values are always equal to zero, making a direct comparison meaningless. It should be noted that, even if these values may appear low in tables 6.2 and 6.3, the considered runs use experimental parameters with intentionally low values to accelerate the data collection process.

In real-world scenarios, training hyperparameters may increase significantly and consequently these times can grow considerably.

We consider the encryption time to be the real bottleneck in our workflow, representing the most computationally expensive routine and the main factor that makes the adoption of HE in FL scenarios challenging.

Speaking about the Aggregation Time and the Exchanged Data, the two metrics that offer us the best view for a comparison between the Plaintext Scenario and the TenSEAL Scenario, we can observe that the overhead on the Aggregation Time is generally more significant than the one on the Exchanged Data. Therefore, from this first glance, the bottleneck seems to be more related to the computations performed by the server (Aggregation Time) and by the clients (Encryption and Decryption Time) rather than to the communication cost, which obviously increases, but its growth is less significant compared to the one experienced by the routines that the participants in the federated training process must perform in a scenario where HE is used.

6.3 HEonGPU Scenario

In this setting, as in the previous TenSEAL Scenario, the communication channel is again protected with HE in both directions, from the clients to the server and from the server to the clients. As in the TenSEAL Scenario, both the pseudogradients shared by the clients at the end of the local training phase and the model parameters sent to the clients at the beginning of the local training phase are encrypted using the CKKS encryption scheme. The main difference between the two scenarios is that this time the library used to implement the CKKS encryption scheme is no longer TenSEAL, but HEonGPU [10].

HEonGPU is a library that implements various HE schemes with the aim of reducing the overhead classically associated with HE operations through the parallelization offered by GPUs. As already discussed, homomorphic operations are often independent from each other and repetitive, making them particularly suitable for GPU acceleration. For example, the same computations on different data can be parallelized through the use of GPUs. The aim of this section is to verify whether this GPU acceleration can be effectively applied to our FL framework with HE.

Firstly, as in the previous TenSEAL Scenario, we will perform a parameter sweep on the inputs of the CKKS encryption scheme, n, q_{chain} and Δ, to inspect how they influence this new library, as already done before with TenSEAL. We will also verify that HEonGPU, just like TenSEAL, does not affect the performance of the model in terms of the final achieved accuracy and that the training time remains comparable to both the TenSEAL and the Plaintext Scenarios, since the training is performed in plaintext and therefore should not change across different settings.

Furthermore, we will analyze whether the GPU acceleration actually benefits the homomorphic operations by observing whether the metrics related to the execution time of homomorphic operations, such as Encryption, Decryption, and Aggregation Time, decrease compared to the TenSEAL Scenario. Finally, we also want to verify that the communication cost remains stable. We will therefore inspect the amount of data exchanged between the clients and the server in this setting, which is directly dependent on the size of the HEonGPU ciphertexts, to determine whether the communication cost increases or can even be reduced.

To carry out these analyses, we need to compare the runs from the two different settings, while ensuring a fair comparison by selecting the same encryption parameters. For this purpose, before running the simulator with the HEonGPU library, additional experiments were conducted using the TenSEAL setting with different parameters from those of the runs analyzed previously. This was done because the HEonGPU library provides some recommendations. In particular, it recommends using the same initial and final value for the q_{chain}, which in our case is always 60, while the intermediate values should all be equal and also equal to the exponent δ of the scaling factor $\Delta = 2^{\delta}$ [10]. For these reasons, the runs in the HEonGPU setting were conducted following these recommendations and, to ensure a fair comparison, they are compared with TenSEAL runs performed using the same parameter choices.

6.3.1 Accuracy and Training Time

Table 6.5: HEonGPU Scenario: n, q_{chain} and Δ sweep, Accuracy and Training Time. `round_clients` = 15, Model Architecture = CNN5, `num_rounds` = 100, α = 0.3

n	$q_{\textbf{chain}}$	Δ	**Accuracy (%)**	**Training Time (s)**
8192	[60,30,60]	2^{30}	39.66	1.15
8192	[60,30,30,60]	2^{30}	39.68	1.16
16384	[60,30,60]	2^{30}	39.67	1.15
16384	[60,30,30,60]	2^{30}	39.67	1.16
8192	[60,40,60]	2^{40}	39.66	1.18
8192	[60,40,40,60]	2^{40}	39.67	1.19
16384	[60,40,60]	2^{40}	39.68	1.18
16384	[60,40,40,60]	2^{40}	39.66	1.17

From Table 6.5 we can confirm our expectations: neither the Accuracy nor the Training Time are affected by the introduction of the new library. This is a good sign because it means that CKKS, despite being an approximate encryption scheme, does not influence the final accuracy achieved by the model at the end of the federated training. Moreover, as expected since the local training is performed in plaintext, the training time is comparable across all three scenarios examined.

In particular, the final accuracy remains around $39.66\% - 39.68\%$ in all the runs of the HEonGPU scenario, as well as in all the runs of the other two scenarios. This indicates that the encryption parameters of the CKKS scheme, including the scaling factor Δ, do not affect the model performance. In fact, as already observed for TenSEAL, a scaling factor equal to 2^{30} is probably already sufficient, and increasing Δ to 2^{40} does not provide enough additional numerical precision to improve the model performance. These results are also consistent with those reported in Sections 6.2 and 6.1.

On the other hand, the training times are also consistent with those reported previously, further confirming the consistency of our results. In this case, the training time is slightly lower, ranging from 1.15 to $1.19s$, compared to $1.22 - 1.26s$ in the TenSEAL Scenario and $1.37s$ in the Plaintext Scenario. We believe that these differences are small and can be considered normal. They are likely due to the dynamic workload of the shared server on which the experiments were executed. Since the server load varies over time, it can slightly affect the efficiency with which the training tasks are completed.

6.3.2 Communication Cost

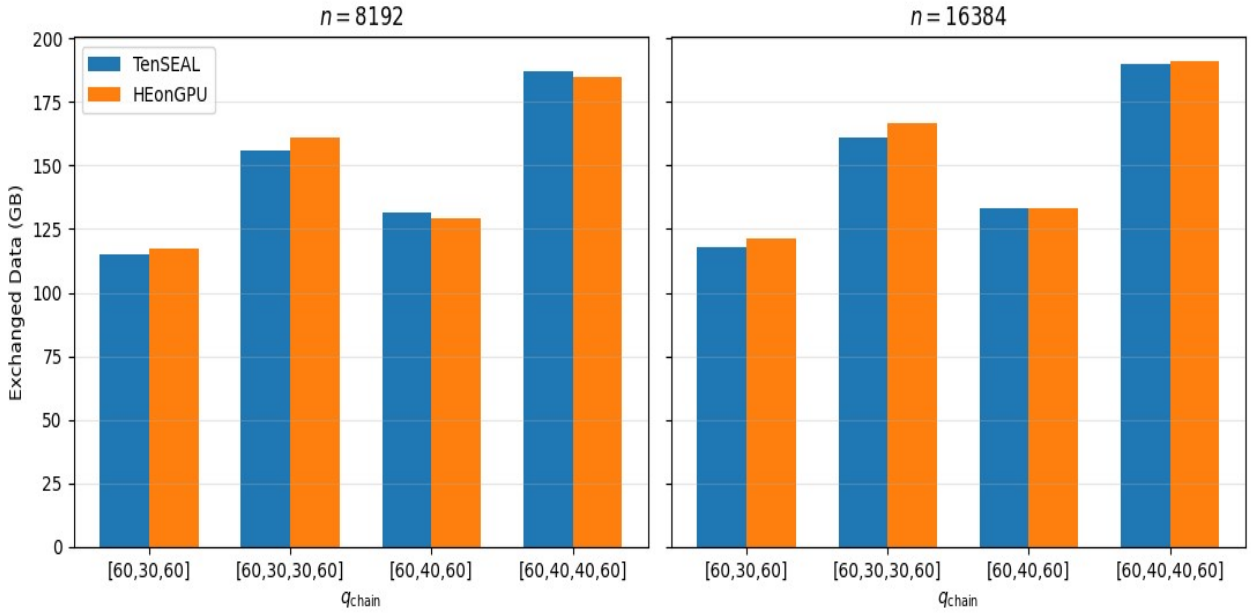


Figure 6.4: HEonGPU vs TenSEAL: Total Exchanged Data over All Training Rounds

The first thing that we can notice from Figure 6.4 is that the total amount of exchanged data across the entire federated training process is comparable between the two scenarios. The HEonGPU library produces ciphertexts with similar sizes compared to TenSEAL, resulting in a similar communication cost for both settings.

More in detail, we can observe some small differences across the different parameter configurations. For example, for both values of n, 8192 and 16384, when $\Delta = 2^{30}$, meaning that the central values of q_{chain} are equal to 30, the exchanged data in the HEonGPU runs is slightly higher than in the TenSEAL runs ($1.89\% - 3.54\%$ higher). On the contrary, for $n = 8192$ and $q_{\text{chain}} = [60,40,60]$ and [60,40,40,60], the HEonGPU experiments show a slightly lower communication cost (-1.97% and -1.21%, respectively). Finally, for $n = 16384$ and $q_{\text{chain}} = [60,40,60]$ and [60,40,40,60], the results are almost identical.

This analysis highlights that the communication cost is essentially the same for both libraries, with only minor differences that do not appear to be significant.

Furthermore, we can observe that the two figures inside 6.4 are basically the same. This means that, as already discussed before, doubling n while keeping all the other parameters fixed leads to substantially the same communication cost.

As already explained, this is because when we double n, we also double the size of a ciphertext, that is directly dependent on n, since n determines the degree of the polynomials used to encode the numbers in the CKKS encryption scheme. However, at the same time, a larger n allows us to pack more values into the same ciphertext, because n also determines the length of the input vector supported by CKKS ($\frac{n}{2}$).

This trade-off leads to similar communication costs for runs with the same encryption parameters but with a doubled value of n. This behavior is observed both in HEonGPU and TenSEAL.

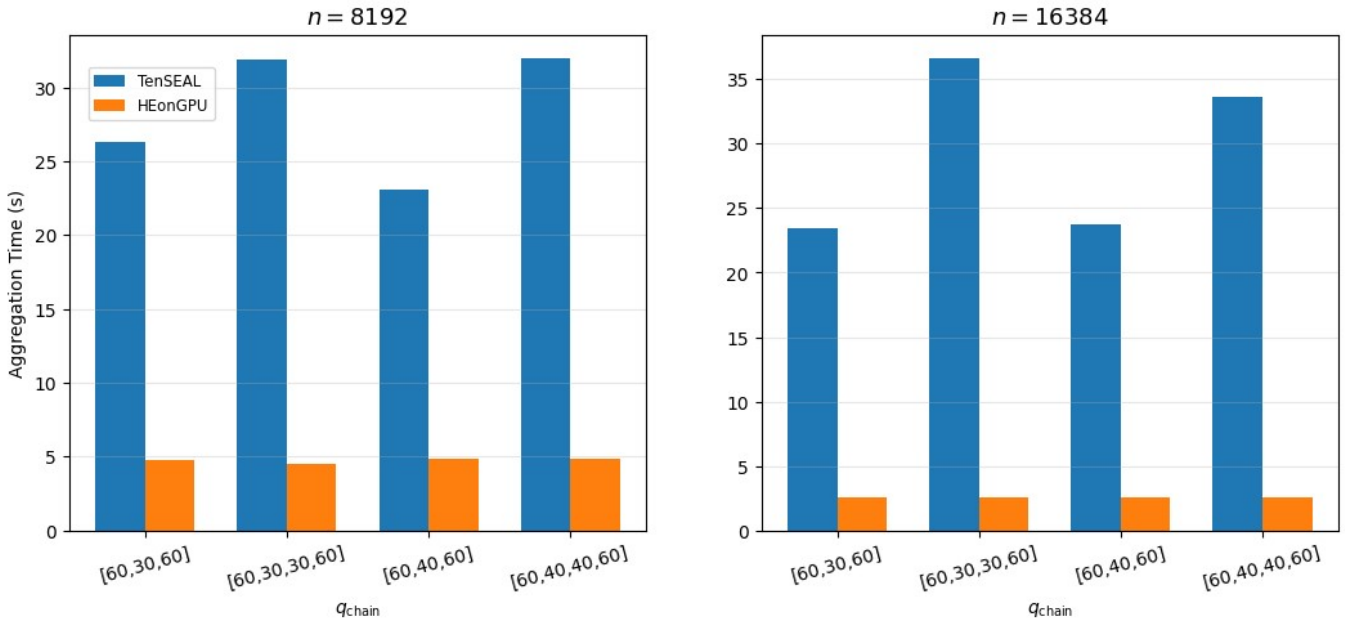
Finally, we can observe two additional behaviors that were expected: the increase in communication costs when adding a level to the q_{chain} and when comparing the runs with $\Delta = 2^{30}$ and central values of 30 in the q_{chain} with those using $\Delta = 2^{40}$ and central values of 40 in the q_{chain}. We expected this increase because the ciphertext size and consequently the total amount of exchanged data, is directly dependent on both the length of the q_{chain} and the size of the moduli contained in it.

In fact, each coefficient of the polynomials used in CKKS is represented through the RNS (Residue Number System) as a vector of remainders corresponding to each modulus in the modulus chain. Therefore, larger moduli imply larger values to be represented, while adding a new level requires adding one additional element to the remainder vector for each ciphertext.

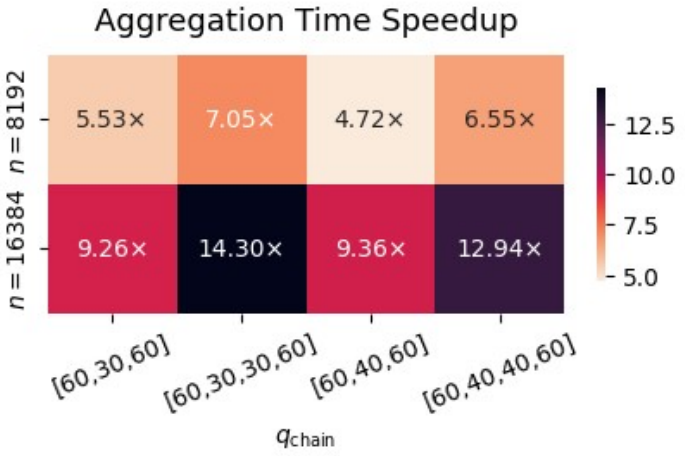
We can notice that the effect is more significant when adding a new level to the q_{chain} rather than increasing the modulus size from [60,30,60] to [60,40,60] or from [60,30,30,60] to [60,40,40,60]. In particular, HEonGPU is slightly less affected by the increase from 30 to 40 in the moduli of the q_{chain} compared to TenSEAL. For the transition from [60,30,60] to [60,40,60], TenSEAL shows an increase of approximately 14.48% and 12.61% for $n = 8192$ and $n = 16384$, respectively, while HEonGPU shows an increase of 10.15% and 10.01% for the same values of n. Similarly, when moving from [60,30,30,60] to [60,40,40,60], the increase is approximately 19.86% and 18.30% for TenSEAL, compared to 14.54% and 14.72% for HEonGPU, for $n = 8192$ and $n = 16384$ respectively.

On the other hand, the increase caused by adding an additional level (from [60,30,60] to [60,30,30,60] and from [60,40,60] to [60,40,40,60]) is comparable for both libraries. In the first case, both libraries show an increase of approximately 36% for both values of n. In the second case, the increase is around 43% for both HEonGPU and TenSEAL, independently of the value of n.

6.3.3 Computational Costs

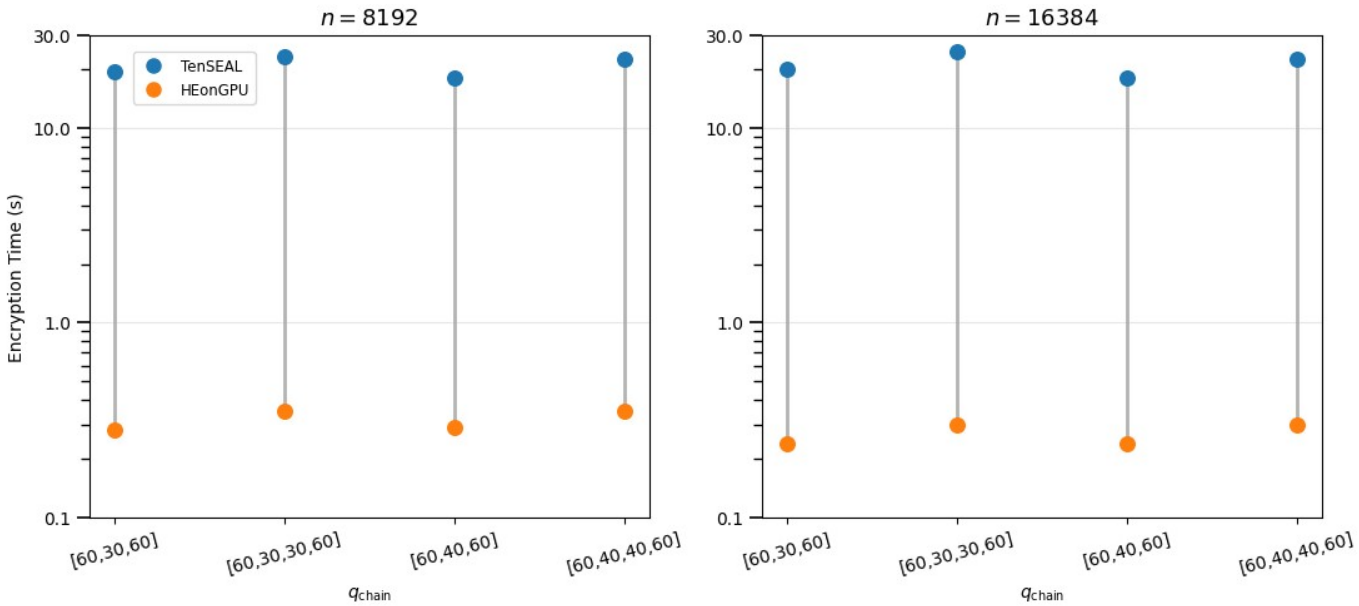


(a) HEonGPU vs TenSEAL: Aggregation Time Comparison

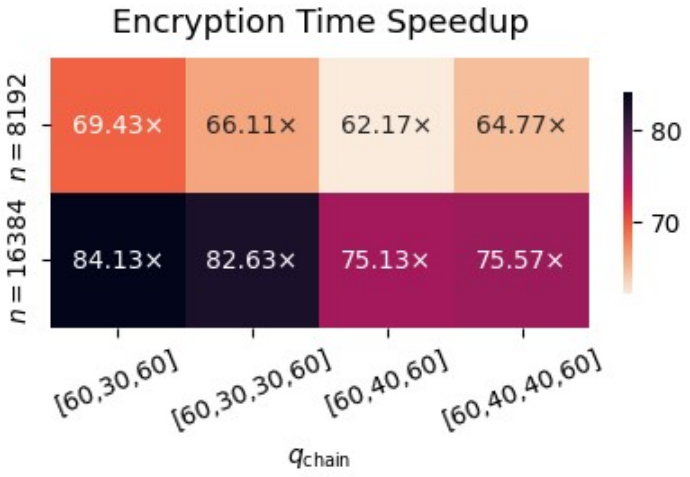


(b) HEonGPU vs TenSEAL: Aggregation Time Acceleration achieved by HEonGPU

Figure 6.5: HEonGPU vs TenSEAL: Aggregation Time

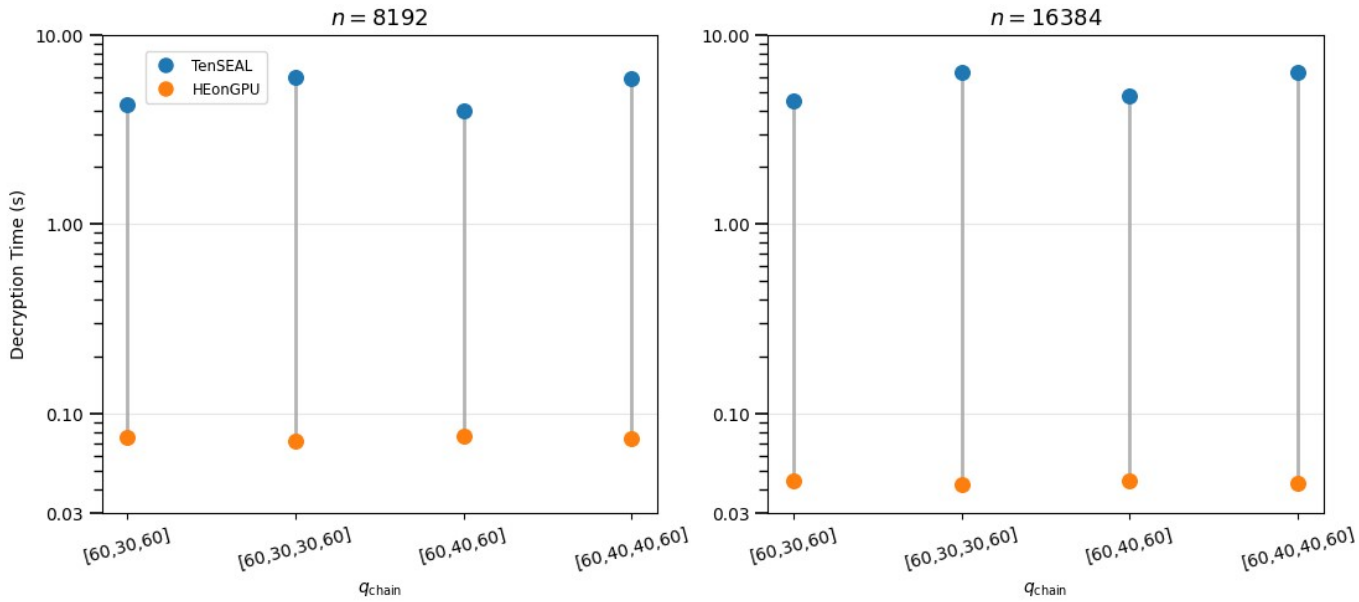


(a) HEonGPU vs TenSEAL: Encryption Time Comparison

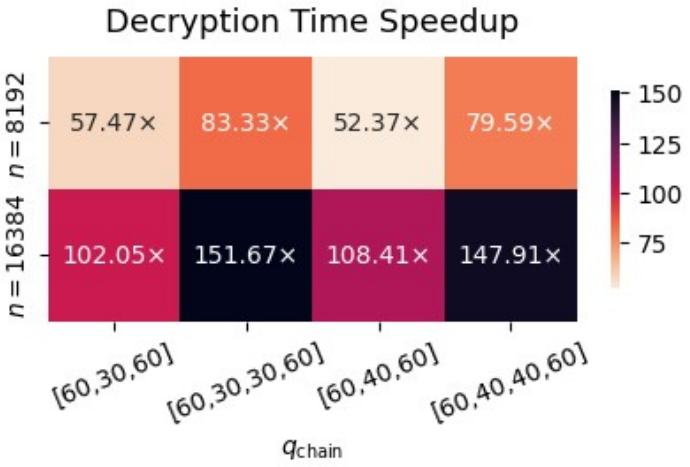


(b) HEonGPU vs TenSEAL: Encryption Time Acceleration achieved by HEonGPU

Figure 6.6: HEonGPU vs TenSEAL: Encryption Time



(a) HEonGPU vs TenSEAL: Decryption Time Comparison



(b) HEonGPU vs TenSEAL: Decryption Time Acceleration achieved by HEonGPU

Figure 6.7: HEonGPU vs TenSEAL: Decryption Time

In general, from all three figures 6.5, 6.6 and 6.7, we can notice that the performance gain introduced by the HEonGPU library is significant, with all three metrics showing a large drop. For all three tracked stats, we can see that the best improvements happen in the runs with a bigger n, meaning that the HEonGPU library scales better than the TenSEAL one when doubling n and the computational complexity, with our system remaining stable when using HEonGPU, unlike when using TenSEAL. We can notice that the best performance gains are recorded on the Decryption Time, with gains from 52.37× to 151.67× depending on the parameter configuration, which is also the most sensitive metric to changes in n and q_{chain}, in the sense that the speedups vary considerably as a function of these two parameters. Then we have the Encryption Time in order of acceleration delivered by the new library, with a speedup between 62.17× and 84.13× based on the different parameter settings, and finally the metric that benefits the least, while still showing significant improvements, is the Aggregation Time, ranging from 4.72× to 14.30×.

In the HEonGPU runs, the Aggregation and Decryption Times halve when passing from $n = 8192$ to $n = 16384$, keeping q_{chain} fixed. The Encryption Times for the runs with $n = 16384$ are also lower compared to the ones with the same q_{chain} but $n = 8192$, even if the difference is smaller, around -15%, so a more contained decrease. Instead, as we already discussed in 6.2, runs that use TenSEAL show comparable times in experiments with different n but the same q_{chain}. This leads to better speedups when comparing runs from the two scenarios with $n = 16384$, as we were mentioning before. This is because the HEonGPU library guarantees the same timing to decrypt a vector or to homomorphically add two encrypted vectors even if the n parameter is doubled. Therefore, in experiments where n is doubled, so the input vector can contain double the values then encrypted in the same ciphertext, we need half of the ciphertexts to process the entire model each time at the same cost, so the times are halved. The same concept applies to the Encryption Times, with the difference that the time to encrypt a single vector is not exactly the same but is a little higher. On the other hand, as we already showed in 6.4, TenSEAL shows double times when encrypting a single vector, decrypting a single vector, or homomorphically adding two encrypted vectors, when n is doubled, so with double the times but half of the ciphertexts to be processed, the statistics from runs with $n = 8192$ and $n = 16384$ remain comparable.

Table 6.6: HEonGPU Benchmark: Encryption and Decryption Time of a single vector of length $\frac{n}{2}$ and homomorphic Addition Time for two encrypted vectors of length $\frac{n}{2}$, measured on our server for different configurations of n and q_{chain}.

Encryption Time (μs)

q_{chain}	$n = 8192$	$n = 16384$
[60,30,60]	84.78	141.80
[60,30,30,60]	108.90	177.03
[60,40,60]	81.57	141.53
[60,40,40,60]	102.07	176.90

Decryption Time (μs)

q_{chain}	$n = 8192$	$n = 16384$
[60,30,60]	19.94	18.97
[60,30,30,60]	19.39	19.14
[60,40,60]	18.97	19.58
[60,40,40,60]	15.98	18.93

Addition Time (μs)

q_{chain}	$n = 8192$	$n = 16384$
[60,30,60]	13.04	15.43
[60,30,30,60]	15.48	15.43
[60,40,60]	14.26	14.97
[60,40,40,60]	15.71	15.49

Speaking about the effect of the q_{chain}, we can see that for the Decryption and Aggregation Times this parameter has a limited impact when using HEonGPU. Looking at the HEonGPU runs, we can note that, keeping n fixed and varying q_{chain} among our four configurations ([60,30,60], [60,30,30,60], [60,40,60], and [60,40,40,60]), both the Decryption and Aggregation Times remain almost the same. In particular, both adding an additional level to the modulus chain (from [60,30,60] to [60,30,30,60] and from [60,40,60] to [60,40,40,60]) or increasing the scale factor (from [60,30,60] with $\Delta = 2^{30}$ to [60,40,60] with $\Delta = 2^{40}$ and from [60,30,30,60] with $\Delta = 2^{30}$ to [60,40,40,60] with $\Delta = 2^{40}$) have only a marginal effect on these two metrics.

Instead, in the runs using TenSEAL, these changes have a much more noticeable impact. As expected, increasing the length of q_{chain} results in an increase in both Decryption and Aggregation Times. Moreover, moving from the configurations with intermediate modulus 30 and $\Delta = 2^{30}$ to those with intermediate modulus 40 and $\Delta = 2^{40}$ also leads to higher execution times.

Finally, we can note that the Encryption Times scale similarly for HEonGPU and TenSEAL. In both libraries, increasing the length of q_{chain} or changing the intermediate moduli together with the corresponding value of Δ leads to comparable increases in the encryption time.

6.3.4 Impact of Number of Global Rounds and Data Distribution on Model Performance

In the last tests we tried to achieve better performance of the model in terms of the level of accuracy reached at the end of the training. To accomplish this goal, we increased the number of global rounds. In fact, in all other experiments we used 100, which for a real-life scenario can appear low and can influence the final performance of the model. In these final runs, we set this value to 2000 to allow the model more time to converge, so extending the training time.

We also tested two different configurations of α to inspect how this parameter influences the learning curve. In fact, we remark that this value controls the Dirichlet distribution that determines the partitions of the dataset across clients. A lower value of α leads to more non-IID partitions, with nodes having more heterogeneous datasets, so the learning process will be more difficult. On the other hand, high values of α lead to clients having quite similar percentages of label representation in their dataset, so in theory the learning curve should be faster.

In these experiments the HEonGPU library was used.

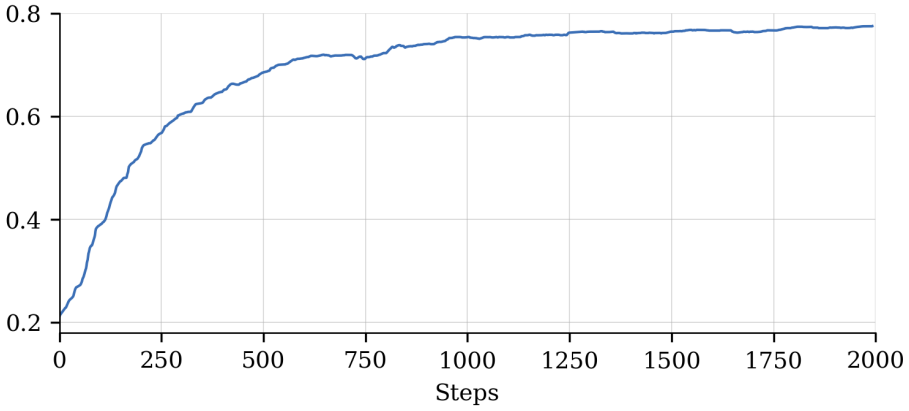


Figure 6.8: $\alpha = 0.3$, `round_clients` $= 10$, `num_rounds` $= 2000$, $n = 8192$, $q_{\text{chain}} = [60,30,60]$, $\Delta = 2^{30}$

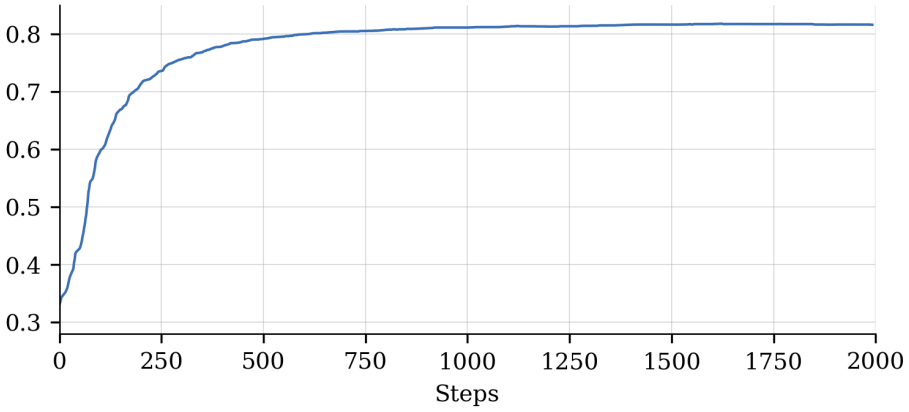


Figure 6.8: $\alpha = 10$, `round_clients` $= 10$, `num_rounds` $= 2000$, $n = 8192$, $q_{\text{chain}} = [60,30,60]$, $\Delta = 2^{30}$

The run with $\alpha = 0.3$ reaches a final accuracy of 77.81% on the test set, while the one with $\alpha = 10$ reaches 81.82%. Both configurations achieve a performance substantially better than all the other experiments, showing that the total number of global rounds, `num_rounds`, is a fundamental factor for the model's learning capability. With more rounds, we have more time to better refine the weights of our global model on the training dataset. More specifically, it can be noted that in the scenario with $\alpha = 0.3$, the learning curve has still not definitely converged, showing that a few more percentage points of accuracy could be achieved with some more rounds.

The main difference between the two tests is the steepness of the learning curve. As expected, the convergence when α is equal to 10 is significantly faster, reaching convergence at around round 600. In the other case instead, as already mentioned, convergence has almost not been reached yet. This is because with higher values of α clients tend to have more similar datasets, so it is easier for the model to learn and fewer rounds are needed to further refine its weights.

Chapter 7

Conclusions

Federated Learning represents a significant improvement in user privacy thanks to its distinctive feature of keeping clients' data local, without ever sharing it, sharing instead only the updates to the global model computed after a local training phase. However, some threats still exist: some information about a node's data can be inferred only from its shared updates. Homomorphic Encryption can represent a solution to this concern, protecting the communication channel between the clients and the server while still allowing computations on the encrypted data to be performed by the central entity. However, Homomorphic Encryption implies a significant overhead in terms of time and communication cost.

In this thesis, we quantified this temporal and communication overhead through the use of specifically defined metrics, such as the aggregation time, the encryption time, the decryption time and the amount of exchanged data. We compared a setting utilizing the TenSEAL library for the CKKS encryption scheme to protect the communication channel with a baseline in which clients transmit updates in plaintext. We performed a parameter sweep over the main encryption parameters of the CKKS scheme in order to examine how these parameters influence the monitored metrics and the performance of the model. Thanks to the findings of this analysis and taking into account the influence of these parameters on the security of our model, we were allowed to evaluate the trade-off between security, performance and latency.

In order to address the overhead posed by HE in the FL workflow, we tried to exploit the parallelism offered by GPUs, which suits well the repetitive, non-sequential nature of HE operations. We implemented HEonGPU in our simulator, a library that leverages GPUs to accelerate the heavy operations related to HE. In this way, we demonstrated that every monitored metric showed a considerable speedup, the aggregation time, the encryption time and the decryption time, thanks to the use of the GPU. All of this while keeping the communication cost stable, without exchanging more data than in the TenSEAL scenario.

In all the inspected scenarios, we demonstrated that HE does not negatively impact the performance of the trained model in terms of the final accuracy reached.

We acknowledge that our work has some limitations, of which we are aware and which can drive future research. In particular, we mention the fact that FL workflows often involve devices with limited computational resources, such as IoT devices. Especially when working with this kind of clients, the implementation of HE in the FL training process is challenging, considering the limited computational capacity of these devices. For this reason, one could argue that a GPU-accelerated library is not a suitable solution for devices that, in most cases, do not even have a GPU. Therefore, we think that our solution can be taken into account only in certain fields of FL research. For example, cross-silo federated learning is a particular type of FL in which the clients are a limited number of large organizations, such as hospitals, banks or research institutions, that typically possess greater computational resources. In this setting, participants are more likely to have access to efficient GPUs, making the adoption of GPU-accelerated HE libraries a practical option. This is instead in contrast with cross-device federated learning, where the clients consist of a much larger population of resource-constrained devices, such as smartphones or IoT sensors, which are typically characterized by limited computational power, intermittent connectivity and a higher likelihood of lacking GPUs.

Additionally, in our work, the clients are simulated on our server, they are not truly independent devices working in parallel. In our simulator, the clients operate one after another. We must also remember that our server is shared, so its workload is variable and can affect our measurements. In every experiment, we tried to maintain the highest possible efficiency and to always work under the same conditions, in order to facilitate reproducibility among the experiments being compared, but this is certainly a limitation. Future work could explore the implementation of HEonGPU in an FL framework running on real, isolated devices that communicate with a central entity.

Finally, emerging techniques for multi-key HE can represent an interesting field for application to FL. Multi-key HE allows computations to be performed on data encrypted with different keys. This can be an improvement for user privacy because, in this scenario, every client owns its own pair of keys, while still allowing the aggregation phase to be performed on updates encrypted with different keys. In this scenario, a client cannot read the updates shared by another client. In our work, instead, all the clients share the same pair of keys, so every malicious client can read the updates shared by another client, since it possesses the keys needed to decrypt them.

Bibliography

[1] Jonas Geiping, Hartmut Bauermeister, Hannah Dröge, and Michael Moeller. «Inverting Gradients - How easy is it to break privacy in federated learning?» In: *Advances in Neural Information Processing Systems (NeurIPS) 2020*. 2020.

[2] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. «Exploiting Unintended Feature Leakage in Collaborative Learning». In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019.

[3] Febrianti Wibawa, Ferhat Ozgur Catak, Murat Kuzlu, Salih Sarp, and Umit Cali. «Homomorphic Encryption and Federated Learning based Privacy-Preserving CNN Training: COVID-19 Detection Use-Case». In: *European Interdisciplinary Cybersecurity Conference (EICC) 2022*. 2022.

[4] Abdullah Ghanim Jaber. «Privacy-Preserving and Secure AI-based Stress Monitoring Federated Learning on Wearable Edge Devices Using Homomorphic Encryption». In: *International Journal of Research and Innovation in Applied Science* (2026).

[5] Yunlong He, Lingtao Wei, Fei Chen, Hanlin Zhang, Jia Yu, and Haiyang Wang. «Fedai: Federated recommendation system with anonymized interactions». In: *Expert Systems with Applications* (2025).

[6] Haolin Wang, Yun Zhao, Shan He, Yong Xiao, Jianlin Tang, and Ziwen Cai. «Federated learning-based privacy-preserving electricity load forecasting scheme in edge computing scenario». In: *International Journal of Communication Systems* (2023).

[7] Yange Chen, Lei Liu, Yuan Ping, Mohammed Atiquzzaman, Shahid Mumtaz, Zhili Zhang, Mohsen Guizani, and Zhihong Tian. «A Privacy-Preserving Federated Learning Framework With Lightweight and Fair in IoT». In: *IEEE Transactions on Network and Service Management* (2024).

[8] Ayoub Benaissa, Bilal Retiat, Bogdan Cebere, and Alaa Eddine Belfedhal. «TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption». In: *ICLR 2021 Workshop on Distributed and Private Machine Learning*. 2021.

[9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. «Homomorphic Encryption for Arithmetic of Approximate Numbers». In: *ASIACRYPT 2017*. 2017.

[10] Ali Şah Özcan, Can Ayduman, Enes Recep Türkoğlu, and Erkay Savaş. «Homomorphic Encryption on GPU». In: *IEEE Access* (2023).

[11] Perrin Derek. «A Summary of the FV Homomorphic Encryption Scheme and the Average-Case Noise Growth». PhD thesis. University of Saskatchewan, 2021.

[12] Zhichuang Liang and Yunlei Zhao. *Number Theoretic Transform and Its Applications in Lattice-based Cryptosystems: A Survey*. arXiv:2211.13546. 2022.

[13] Junfeng Fan and Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. IACR Cryptology ePrint Archive. 2012.

[14] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. «Bootstrapping for Approximate Homomorphic Encryption». In: *EUROCRYPT 2018*. 2018.

[15] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. «Communication-Efficient Learning of Deep Networks from Decentralized Data». In: *AISTATS 2017*. 2017.

[16] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. «Federated Optimization in Heterogeneous Networks». In: *Machine Learning and Systems (MLSys) 2020*. 2020.

[17] Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank J. Reddi, Sebastian U. Stich, and Ananda Theertha Suresh. «SCAFFOLD: Stochastic Controlled Averaging for Federated Learning». In: *International Conference on Machine Learning (ICML) 2020*. 2020.

[18] Sashank J. Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečný, Sanjiv Kumar, and H. Brendan McMahan. «Adaptive Federated Optimization». In: *International Conference on Learning Representations (ICLR) 2021*. 2021.

[19] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. *Fully Homomorphic Encryption without Bootstrapping*. IACR Cryptology ePrint Archive. 2011.

[20] Yuki Kamada, Kurt Rohloff, and Masato Oguchi. «Homomorphic Encryption for Privacy-Preserving Genome Sequences Search». In: *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*. 2019.

[21] Jie Li, Yamin Liu, and Shuang Wu. «Pipa: Privacy-preserving Password Checkup via Homomorphic Encryption». In: *ACM ASIA Conference on Computer and Communication Security 2021*. 2021.

[22] Santosh Kumar Henge et al. «EHC-BGV hybrid homomorphic encryption algorithmic sequences for processing authenticated role based secure keys». In: *Journal of Discrete Mathematical Sciences and Cryptography* (2025).

[23] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. *TFHE: Fast Fully Homomorphic Encryption over the Torus*. IACR Cryptology ePrint Archive. 2018.

[24] Ahmad Al Badawi et al. *OpenFHE: Open-Source Fully Homomorphic Encryption Library*. IACR Cryptology ePrint Archive. 2022.

[25] Sadhana Selvakumar and B. Senthilkumar. «A privacy preserving machine learning framework for medical image analysis using quantized fully connected neural networks with TFHE based inference». In: *Scientific Reports* (2025).

[26] Daphné Trama, Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. «A Homomorphic AES Evaluation in Less than 30 Seconds by Means of TFHE». In: *WAHC 2023 – 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2023.

[27] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. «How To Backdoor Federated Learning». In: *AISTATS 2020*. 2020.

[28] Kim Laine. *Simple Encrypted Arithmetic Library 2.3.1*. Microsoft Research. 2017.

[29] Jean-Philippe Bossuat et al. «Security Guidelines for Implementing Homomorphic Encryption». In: *IACR Communications in Cryptology* (2025).

[30] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. University of Toronto, 2009.

[31] Antonio Torralba, Rob Fergus, and William T Freeman. «80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2008).

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep Residual Learning for Image Recognition». In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[33] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. «Learning Long-Term Dependencies with Gradient Descent is Difficult». In: *IEEE Transactions on Neural Networks* (1994).

[34] Yuxin Wu and Kaiming He. «Group Normalization». In: *European Conference on Computer Vision (ECCV) 2018*. 2018.

[35] Xavier Lessage, Leandro Collier, Charles-Henry Bertrand Van Ouytsel, Axel Legay, Saïd Mahmoudi, and Philippe Massonet. «Secure federated learning applied to medical imaging with fully homomorphic encryption». In: *2024 IEEE 3rd International Conference on AI in Cybersecurity (ICAIC)*. 2024.

[36] M. Gayathri Hegde, B. Ruthvika, Ruthu B. Jain, P. Deepa Shenoy, K. R. Venugopal, and Arvind Canchi. «A Privacy-Preserving Federated Learning Method with Homomorphic Encryption for Chronic Kidney Disease Stage Prediction». In: *Engineering, Technology & Applied Science Research* (2025).

[37] Mahesh Reddy Konatham, Dharmateja Priyadarshi Uddandarao, Ravi Kiran Vadlamani, and Sai Kishore Reddy Konatham. «Federated Learning for Credit Risk Assessment in Distributed Financial Systems using BayesShield with Homomorphic Encryption». In: *2025 International Conference on Computing Technologies & Data Communication (ICCTDC)*. 2025.

[38] Adam Paszke et al. «PyTorch: An Imperative Style, High-Performance Deep Learning Library». In: *Advances in Neural Information Processing Systems (NeurIPS) 2019*. 2019.

[39] Charles R. Harris et al. «Array programming with NumPy». In: *Nature* (2020).