Training Encrypted Neural Networks on Encrypted Data with Fully Homomorphic Encryption

Luca Colombo
Politecnico di Milano
Milan, Italy
luca2.colombo@polimi.it

Alessandro Falcetta
Politecnico di Milano
Milan, Italy
alessandro.falcetta@polimi.it

Manuel Roveri
Politecnico di Milano
Milan, Italy
manuel.roveri@polimi.it

Abstract

Training machine and deep learning models on encrypted data is the next challenge in the field of privacy-preserving Machine and Deep Learning. The related literature in this field is very limited, since most of the solutions focus only on inference on encrypted data (leaving the training to be carried out on plain data). In this paper we introduce a multi-class and non-linear family of neural networks based on the Torus Fully Homomorphic Encryption (TFHE) scheme (named TFHE-NNs), which can be entirely trained on encrypted data. The proposed learning procedure, implementing a TFHE-compliant version of the Direct-Feedback-Alignment algorithm, is combined with a novel Cross-Validation procedure able to operate on encrypted models and encrypted accuracy. The experimental results demonstrate the feasibility of the proposed solution. The proposed models and algorithms are made available to the scientific community as a public repository.

CCS Concepts

• **Security and privacy** → **Privacy protections**; **Privacy-preserving protocols**; **Domain-specific security and privacy architectures**; **Cryptography**; • **Computing methodologies** → **Artificial intelligence**; **Machine learning**; **Neural networks**.

Keywords

Homomorphic Encryption, TFHE, Privacy-preserving, Machine Learning, Deep Learning

ACM Reference Format:
Luca Colombo, Alessandro Falcetta, and Manuel Roveri. 2024. Training Encrypted Neural Networks on Encrypted Data with Fully Homomorphic Encryption. In *Proceedings of the 12th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3689945.3694802

1 Introduction

Privacy-preserving Machine and Deep Learning is a novel emerging paradigm in the field of Artificial Intelligence [7]. Solutions in this field seek to design and develop Machine Learning (ML) and Deep Learning (DL) models able to guarantee the privacy of users, capable of operating on data characterized by strong constraints in terms of confidentiality and secrecy. The motivations for this important progress include, inter alia, the increasing fears of data-breaches [37], the appearance of restrictive data privacy regulations (e.g., [1]), and the growing demands for data control, visibility, and discretion of customers' personal data [7]. This last aspect is particularly relevant when ML and DL models are provided in a *as-a-service* manner, i.e., when user data are processed by a third-party cloud provider to provide services based on ML and DL [41].

One of the most promising approach in privacy-preserving ML and DL is Homomorphic Encryption (HE) [14], which is a special type of encryption that allows one to compute operations on encrypted data, hence guaranteeing the privacy of data *in-use* [7]. While the literature in this field explores works in which pre-trained models are applied on encrypted data to support the privacy-preserving inference, few studies have addressed the need to support also the training of general, multi-class, and deep models directly on encrypted data. This is mainly due to HE's computational overhead and functional limitations, which makes the execution of common training algorithms on encrypted data extremely difficult [32].

In this study we show that effective learning on encrypted data is possible. Specifically, we propose a novel HE-based approach to learning multi-class and non-linear models directly on encrypted data. Addressing this problem required us to deal with three different challenges.

The first concerns the fact that commonly employed HE schemes for encrypted inference allow only polynomial operations on encrypted data. Moreover, given that noise is injected into ciphertexts during encryption, only a finite number of operations can be performed on a ciphertext before it becomes unusable [15].

The second challenge is the overhead introduced by HE. An operation between two encrypted numbers could require up to five orders of magnitude more time compared to its plain version. This overhead makes training on encrypted data extremely time-consuming even with very small models, and required the introduction of mechanisms for the parallelization of the learning procedure.

The last challenge is cross-validation (CV), a very common technique used in ML and DL to identify the hyperparameter configuration guaranteeing the highest accuracy. Unfortunately, learning on encrypted data produces encrypted models and encrypted validation and test accuracies. In this scenario, the traditional CV algorithms cannot be used, hence new solutions have to be designed.

In summary, this study addresses the following three research questions (RQs):

RQ1. *How can we learn multi-class and non-linear models on encrypted data?*

WAHC '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1241-8/24/10
https://doi.org/10.1145/3689945.3694802

RQ2. *How can we reduce the overhead of training HE-enabled models by parallelizing the learning procedure?*
RQ3. *How can we choose the best model among many, even if the models and their validation accuracies are encrypted?*

To address these RQs, we present the following contributions and make the code used in this paper available to the scientific community as a public repository[1]:

C1. We designed a new family of HE-compliant Neural Networks (NN) called TFHE-NNs and proposed a novel learning framework for training these models on encrypted datasets, based on the Torus Fully HE (TFHE) scheme [10] and the Direct-Feedback-Alignment (DFA) [29] training algorithm.
C2. We designed and implemented the encrypted extension of Model Averaging (MA) [40], an ensemble DL technique, to allow the parallelization of the encrypted training procedure to reduce the overhead of the HE scheme.
C3. We propose a CV procedure capable of selecting the TFHE-NN that guarantees the highest validation accuracy. This is particularly useful from a practical point of view, as it allows a third party to train and evaluate models with different hyperparameters directly on an encrypted dataset.

The solution proposed in this paper represents an unprecedented attempt at end-to-end privacy-preserving learning and, potentially, a fundamental first step in this research direction in the field of machine and deep learning.

The paper is organized as follows. In Section 2, a motivating example is proposed. Section 3 reviews the related literature by considering studies on privacy-preserving inference and learning. Section 4 introduces the background on HE and DFA. Section 5 presents the proposed solution, while experimental results on encrypted datasets are shown in Section 6. Finally, conclusions are drawn in Section 7.

2 Motivating example

In this Section, we illustrate a motivating example in the area of ML and DL for healthcare, operating on encrypted data. As shown in Fig. 1, a hospital has collected a dataset TR of X-rays, taken on patients. While this data may be useful for internal medical purposes (e.g., classification into healthy/ill etc.), it represents particularly sensitive information and therefore cannot be transferred to a third-party (e.g., a cloud provider) due to strict privacy regulations. Unfortunately, the hospital lacks the resources and expertise to train ML/DL models on this data locally. This is where the proposed solution for privacy-preserving training on encrypted data comes into play.

The hospital can encrypt its dataset using a HE encryption function $E(\cdot)$ and a private key, obtaining the encrypted dataset $\widehat{TR}$. This encrypted dataset can be securely transferred to the third party server (e.g. a cloud provider) for training a ML/DL model. Privacy is not compromised as the cloud provider does not have access to the secret key and cannot decrypt the dataset. At this point, the cloud provider can train and select the best model for the classification task in question in an encrypted manner, leveraging the proposed solution. Finally, the encrypted model $\hat{m}$ can be returned to the hospital, which can decrypt it using the HE decryption function $D(\cdot)$ and the secret key to obtain the final plain model m.

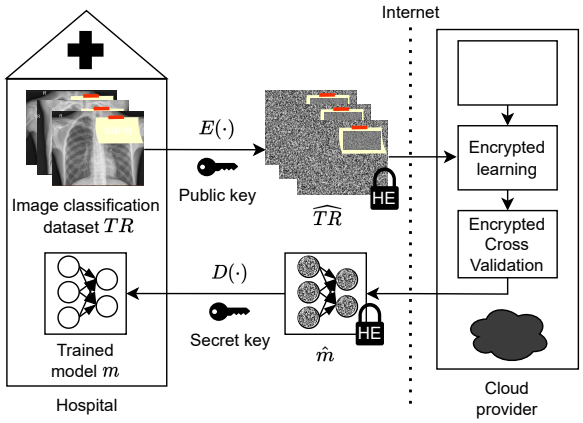

Figure 1: Motivating example: a hospital needs to train an image classification model to improve the quality of diagnoses by leveraging the expertise and computational resources of a cloud provider while ensuring the privacy of its data.

Furthermore, if the hospital does not have the computational resources to run m locally for inference on X-rays, the encrypted model $\hat{m}$ can be left in the cloud and executed in a fully encrypted inference *as-a-service* pipeline. In this case, the hospital uploads only encrypted images and the results, still encrypted, are sent back to the hospital. The prediction is decrypted by the hospital and used for medical purposes.

This example shows that our solution can be valuable where a strict privacy management is required. In the next section, we review the literature on this topic.

3 Related literature

The first problem addressed by the scientific community in this field is the inference of ML and DL models with HE. Notable contributions in this area concern CryptoNets [18], which introduces the application of a Convolutional Neural Network for image recognition on encrypted data, and [8], which presents some optimizations for the same task. A major limitation of these (and other similar) works in this context is the use of leveled HE schemes, which allow only a bounded number of linear operations to be computed. This means that commonly used activation functions (e.g., ReLU) cannot be exploited in the ML/DL model under consideration. To address this problem, such functions must be approximated, mainly using polynomial approximations [38]. However, this may degrade the accuracy of the NN. We also emphasize that privacy-preserving inference requires a plain (i.e., unencrypted) training set to train the model before using it on encrypted data.

With respect to training ML and DL models on encrypted data, [31] introduces a Support Vector Machine (SVM) model for binary classification, that can be trained on encrypted data. Although interesting, it cannot directly handle multi-class classification tasks. [26] presents Glyph, a framework that supports efficient training of NNs on encrypted data by switching between different HE schemes. This work, along with others [24, 30, 44], focuses on training only

[1] https://github.com/AI-Tech-Research-Lab/TFHE-NN

the final layers of an NN that was previously trained on plain data using transfer learning. However, this method is not suitable for scenarios where only an encrypted training dataset is provided. Hybrid approaches are presented by [42] and [36], which combine HE with differential privacy and federated learning techniques. Unfortunately, the former uses models with most of their parameters in plaintext, while the latter requires communication between multiple parties.

Most of the aforementioned works, along with others [21], utilize the Cheon–Kim–Kim–Song (CKKS) [9] scheme, a HE scheme that enables native encryption and floating-point computation. However, CKKS is mainly used in its leveled version, where only a limited number of HE operations are permitted. This is due to the bootstrapping procedure of the CKKS scheme, which, unlike other HE schemes such as Brakerski–Fan–Vercauteren (BFV) [5] or TFHE, does not reduce accumulated noise [13]. As a result, this limits NN models that can be trained to have a single hidden layer.

In contrast to the literature, the solution proposed in this paper provides two key aspects that are fundamental for privacy-preserving training of a NN. First, the proposed TFHE-NNs can directly deal with multi-layer models and multi-class classification problems. Second, TFHE-NNs are trained entirely on encrypted data, without using transfer learning from plain data.

4 Background

This section introduces the basic concepts of both the HE encryption scheme used in this study, and the learning procedure for training TFHE-NNs.

4.1 Homomorphic Encryption

A HE scheme is an encryption scheme that is able to compute a set of operations directly on encrypted data, while guaranteeing that the result of the operation is still encrypted. Formally, as defined in [3], an encryption function E and its corresponding decryption function D are homomorphic with respect to a class of functions $\mathcal{F}$ if, for any function $f \in \mathcal{F}$, it is possible to construct a corresponding function g_f such that $f(x) = D(g_f(E(x)))$ for a set of input x. In particular, the HE scheme used in this work is the Torus Fully Homomorphic Encryption (TFHE) scheme [10].

TFHE is a fully HE (FHE) scheme, which means that it can perform an unbounded number of operations on encrypted data. This distinguishes it from leveled HE schemes, which support only a limited number of operations on encrypted data. While suitable for encrypted inference, leveled HE schemes cannot handle the longer computational requirements of encrypted learning.

More specifically, TFHE is based on the Learning With Errors (LWE) problem [33] and its Ring variant (RLWE) [27], and it supports the evaluation of an unbounded number of logic gates (e.g., AND, OR, etc.) between encrypted bits. For this reason, the numerical operations (e.g., add, mul, etc.) required for the learning and inference procedures of the proposed solution have been redesigned and reimplemented using the bit-level operations offered by the TFHE scheme.

In the following, we detail the mathematical structures on which the TFHE scheme is based.

4.1.1 *Notations.*

Here we use the same notations adopted in [11], which is listed below:

- $\mathbb{B}$: The set {1,0} without any structure
- $\mathbb{T}$: The real *torus* $\mathbb{R}/\mathbb{Z}$, the set [0,1) of real numbers modulo 1.
- $\mathbb{T}_N[X]$: The $\mathbb{Z}$-module $\mathbb{R}[X]/(X^N + 1)$ modulo 1 of torus polynomials, where N (the number of coefficients) is a power of 2.
- $\mathcal{R}$: The ring of polynomials $\mathbb{Z}[X]/(X^N + 1)$.
- $\mathbb{B}_N[X]$: The sub-ring of polynomials in $\mathcal{R}$ with binary coefficients.
- $\leftarrow : x \leftarrow \mathcal{D}$ means that x is drawn from the distribution $\mathcal{D}$.
- $\xleftarrow{\$} : x \xleftarrow{\$} \mathbb{K}$ indicates that x is sampled uniformly at random from $\mathbb{K}$.
- $\xleftarrow{\mathcal{N}(\mu,\sigma^2)} : x \xleftarrow{\mathcal{N}(\mu,\sigma^2)} \mathbb{K}$ means that x is sampled according to a Gaussian distribution of mean μ and variance σ^2 from $\mathbb{K}$.

4.1.2 *Learning With Errors problem.*

The Learning With Errors is a computational problem commonly used in post-quantum cryptography, along with its Ring variant [4]. There are two different version of the problem: the *search* problem and the *decision* problem.

Definition 4.1 (LWE adaptation, Definition 2.5 [10]). Given:

- n integer, with $n \geq 1$
- ξ a distribution over $\mathbb{R}$ and $\mathcal{S}$ a distribution over $\mathbb{Z}^n$
- $\mathbf{s} \leftarrow \mathcal{S}$, $\mathbf{e} \leftarrow \xi$ and $\mathbf{a} \xleftarrow{\$} \mathbb{T}^n$
- $b = \mathbf{a} \cdot \mathbf{s} + e$
- $\mathrm{LWE}_{\mathbf{s},\xi}$ the distribution over $\mathbb{T}^n \times \mathbb{T}$ obtained by sampling a pair $(\mathbf{a}, b)$

The two problems are defined as:

(1) *Search* problem: given arbitrarily many independent samples $\mathrm{LWE}_{\mathbf{s},\xi}$, find the private polynomial $\mathbf{s}$.
(2) *Decision* problem: given arbitrarily many independent samples, distinguish between $\mathrm{LWE}_{\mathbf{s},\xi}$ samples and uniformly random samples from $\mathbb{T}^n \times \mathbb{T}$, for a fixed $\mathbf{s}$.

In particular, [6] demonstrated that LWE problems are at least as hard as standard worst-case lattice problems.

4.1.3 *TFHE Structures.*

The TFHE scheme is an adaptation of the LWE problem to $\mathbb{T}$. We describe the three main structures used to encrypt plaintexts, as formulated in [11]:

(1) TLWE is the torus version of the LWE problem. TLWE can be represented as a $(n+1)$-dimensional torus vector $(\mathbf{a}, b)$ with $\mathbf{a} \xleftarrow{\$} \mathbb{T}^n$ and $b = \mathbf{a} \cdot \mathbf{s} + e$, where $\mathbf{s} \xleftarrow{\$} \mathbb{B}^n$ is the secret key and $e \xleftarrow{\mathcal{N}(0,\sigma^2)} \mathbb{T}$ is the noise.
(2) TRLWE is the torus version of the RLWE problem. A pair $(\mathbf{a}, b) \in \mathbb{T}_N[X]^k \times \mathbb{T}_N[X]$ (i.e., a torus polynomial vector) is a valid TRLWE sample if $\mathbf{a} \xleftarrow{\$} \mathbb{T}_N[X]^k$ and $b = \mathbf{a} \cdot \mathbf{s} + e$, where $\mathbf{s} \xleftarrow{\$} \mathbb{B}_N[X]^k$ is a TRLWE secret key and $e \xleftarrow{\mathcal{N}(0,\sigma^2)} \mathbb{T}_N[X]$ is a noise polynomial.

Given $\mathcal{M}$ a discrete message space[2], in order to encrypt a message $m \in \mathcal{M} \subset \mathbb{T}$ (resp. $\mathcal{M} \subset \mathbb{T}_N[X]$), the trivial element $(\mathbf{0}, m) \in \mathbb{T}^{n+1}$ (resp. $(\mathbf{0}, m) \in \mathbb{T}_N[X]^k \times \mathbb{T}_N[X]$) is added to a TLWE (resp. TRLWE) sample. The encryption of m with the secret key $\mathbf{s}$ is indicated as the TLWE (resp. TRLWE) ciphertext $\mathbf{c} \in \text{TLWE}_{\mathbf{s}}(m)$ (resp. $\mathbf{c} \in \text{TRLWE}_{\mathbf{s}}(m)$).

The decryption of a ciphertext $\mathbf{c} \in \text{TLWE}_{\mathbf{s}}(m)$ (resp. $\mathbf{c} \in \text{TRLWE}_{\mathbf{s}}(m)$) is a function called *phase*. The phase of a sample $\mathbf{c}$ is defined as $\varphi_{\mathbf{s}}(\mathbf{c}) = b - \mathbf{a} \cdot \mathbf{s} = m + e$. Since the result of $\varphi_{\mathbf{s}}(\mathbf{c})$ is an approximation of m, it is rounded to the nearest element in $\mathcal{M}$. The error e has to be be small enough in order to be able to retrieve m, while high enough to guarantee the security of the scheme.

As stated in [10], TLWE (resp. TRLWE) samples can be linearly combined to obtain a new TLWE (resp. TRLWE) sample that encrypts the linear combination of the messages. However, in this case, the linear operations are not sufficient to achieve FHE. To solve this problem, another structure based on the GSW construction introduced in [17] is used.

(3) TRGSW is the torus version of the Ring-GSW problem and it is represented as a vector of l TRLWE ciphertexts. In order to encrypt a message m, it uses a gadget matrix[3]. The novelty introduced by [10] is the definition of an external product $\boxdot$ between a TRGSW sample A that encrypts $m_a \in \mathcal{R}$, and a TRLWE sample $\mathbf{b}$ that encrypts $m_b \in \mathbb{T}_N[X]$. This external product yields a TRLW sample $\mathbf{c}$ that encrypts $m_a \cdot m_b$.

4.1.4 TFHE Bootstrapping. Bootstrapping, proposed in [16], allows one to reduce the noise level of a ciphertext. This makes it possible to evaluate an unbounded number of operations on it, without corrupting the information. TFHE provides a *gate-bootstrapping* procedure that requires only 10 milliseconds [10] to run. It improves performance compared to other schemes, such as BGV, which has a bootstrap process that takes minutes with HElib [19].

As stated in [11], TFHE bootstrapping relies on three building blocks:

(1) BlindRotate: it rotates the coefficients of a polynomial encrypted in the input TRLWE ciphertext by multiplying it by an encrypted power of X. Its inputs are: a ciphertext $\mathbf{c} \in \text{TRLWE}_{\mathbf{s}}(m)$, a vector $(a_1, ..., a_p, a_{p+1} = b)$ where $\forall i$, $a_i \in \mathbb{Z}_{2N}$, and p TRGSW ciphertexts that encrypt $(s_1, ..., s_p)$ where $\forall i$, $s_i \in \mathbb{B}$. The output is a TRLWE sample $\mathbf{c'} \in \text{TRLWE}_{\mathbf{s}}(X^{\mathbf{a} \cdot \mathbf{s} - b} \cdot m)$.
(2) SampleExtract: its inputs are a ciphertext $\text{TRLWE}_{\mathbf{s}}(m)$ and a position $p \in [0, N[$, and returns a ciphertext $\mathbf{c'} \in \text{TRLWE}_{\mathbf{s}}(m_p)$ where m_p is the p^{th} coefficient of the polynomial m.
(3) KeySwitch: it transforms p ciphertexts $\mathbf{c}_i \in \text{TLWE}_{\mathbf{k}}(m_i)$ into a ciphertext $\mathbf{c'} \in \text{T(R)LWE}_{\mathbf{s}}(f(m_1, ..., m_p))$, where $f(\cdot)$ is a public linear morphism from $\mathbb{T}^p$ to $\mathbb{T}_N[X]$. Note that it can be used to perform TLWE-to-TLWE key switching and switch between scalar message spaces using $f(\cdot)$ equal to the identity function (id : $\mathbb{T} \rightarrow \mathbb{T}$).

[2]It means that the torus is discretized w.r.t. the plaintext modulus [11]. For example, if $m \in \mathbb{Z}_3 = \{0, 1, 2\}$ is the message to encrypt, it can be encoded in $\mathbb{T}$ as one of the following values $\{0, \frac{1}{3}, \frac{2}{3}\}$.
[3]Refer to Definition 3.6 and Lemma 3.7 of the TFHE paper [10] for more details.

4.1.5 Homomorphic Gates. Homomorphic evaluation of binary logic gates is achieved by performing two additions and a gate bootstrapping on TLWE samples. [10] lists all the basic gates available on TFHE and how they are computed (given $m = \frac{1}{4}$):

- $\text{HomAND}(\mathbf{c}_1, \mathbf{c}_2) = \text{Bootstrap}((\mathbf{0}, -\frac{1}{8}) + \mathbf{c}_1 + \mathbf{c}_2)$
- $\text{HomNAND}(\mathbf{c}_1, \mathbf{c}_2) = \text{Bootstrap}((\mathbf{0}, \frac{5}{8}) - \mathbf{c}_1 - \mathbf{c}_2)$
- $\text{HomOR}(\mathbf{c}_1, \mathbf{c}_2) = \text{Bootstrap}((\mathbf{0}, \frac{1}{8}) + \mathbf{c}_1 + \mathbf{c}_2)$
- $\text{HomXOR}(\mathbf{c}_1, \mathbf{c}_2) = \text{Bootstrap}(2 \cdot (\mathbf{c}_1 + \mathbf{c}_2))$

The NOT gate consists of an addition without bootstrapping since it only negates the coefficients of the input:

- $\text{HomNOT}(\mathbf{c}) = (\mathbf{0}, \frac{1}{4}) - \mathbf{c}$

4.2 Direct Feedback Alignment

DFA is a learning procedure for DL models based on the Feedback Alignment (FA) [25] principle. The main difference with respect to other learning solutions, such as Backpropagation (BP) [23], is that the weights used during the forward pass of the NN are not the same as those used during the backward pass. Indeed, the error is propagated directly to each hidden layer through fixed and random feedback weights extracted from a uniform distribution.

Let H be the number of hidden layers of a feed-forward NN. Formally, the layer's update directions for DFA – denoted by δ^h_{DFA}, $h = 1, \ldots, H$ – are calculated as:

$$\delta^h_{DFA} = \begin{cases} L' \odot \text{act}'^{\,h}(\mathbf{a}^h) & \text{for } h = H \\ L' \cdot B^h \odot \text{act}'^{\,h}(\mathbf{a}^h) & \text{for } 1 \leq h \leq H-1 \end{cases} \tag{1}$$

where L' is the derivative of the Loss Function, $\odot$ is the element-wise multiplication operator, and $\text{act}'^{\,h}(\cdot)$, $\mathbf{a}^h$, and B^h are the derivative of the activation function, the output of the layer before the application of the non-linearity, and the fixed random weight matrix of layer h, respectively. Once all the δ^h_{DFA}-s have been computed, the weights' updates, for each layer $h \in [1, H]$, are calculated as:

$$\delta W^h = -\,\mathbf{o}^{h-1\,\intercal} \cdot \delta^h_{DFA} \tag{2}$$

$$\delta \mathbf{b}^h = -\,\delta^h_{DFA} \tag{3}$$

where $\mathbf{o}^{h-1}$ is the output of the previous layer in the network.

Interestingly, DFA has been successfully used in PocketNN [39], a NN tailored for Internet-Of-Things (IoT) devices. The use of DFA in our solution is motivated by the need to have only integer values in the entire architecture, due to time constraints (as detailed in Section 5). However, using BP on integer values increases the chances of overflow during the learning procedure, given that the error passes through each layer of the network. Conversely, DFA overcomes this issue by directly propagating the error from the output to each single layer, which is trained independently, limiting the growth of the weights' values.

5 The proposed solution

Section 5.1 provides an overview of the proposed solution. Then, Section 5.2 presents details on the design of TFHE-NNs layers capable of operating on encrypted data. The learning procedure to learn a TFHE-NN on encrypted data is explained in Section 5.3, while the cross-validation algorithm able to compare encrypted TFHE-NN models is shown in Section 5.4.

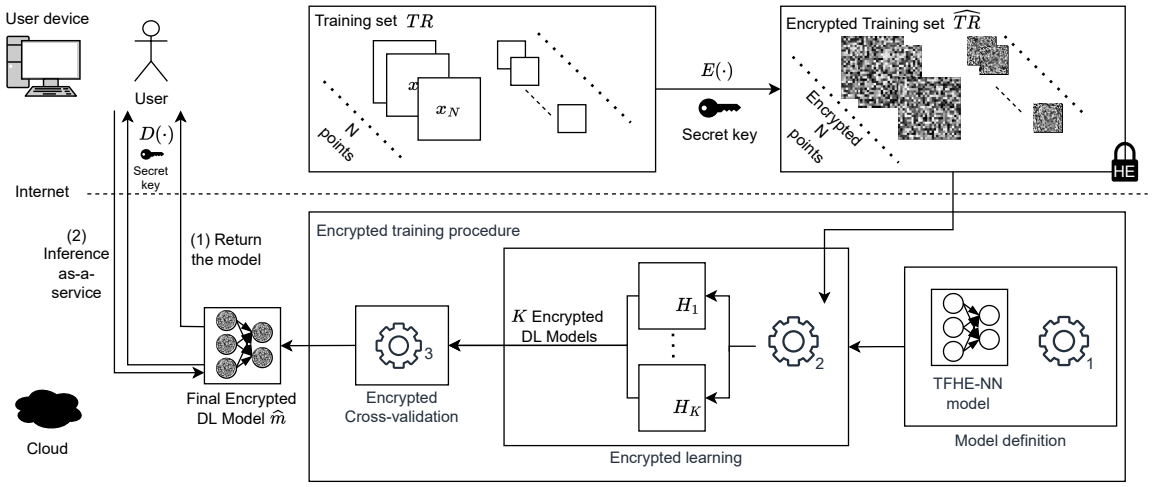

Figure 2: The proposed privacy-preserving architecture.

5.1 An overview of the proposed solution

An overview of the proposed solution is shown in Fig. 2. Without any loss of generality, in the followings a supervised learning image classification task is considered as an example.

Consider a user U, whose goal is to obtain a model trained on its dataset TR, which must be kept private. To achieve this goal, U locally encrypts the dataset, which becomes $\widehat{TR} = E(k_s, TR)$, with $E(\cdot)$ being the encryption function and k_s being the secret key generated by U. Note that $\widehat{TR}$ includes both the encrypted features and the encrypted labels. $\widehat{TR}$ is then sent to the third party service for the privacy-preserving training. Then, the proposed encrypted learning procedure operates as follows:

(i) The TFHE-NN model, represented by $f_{\blacktriangledown}(\hat{X}, \hat{\theta})$ and suitably designed and implemented to be trained on encrypted data, is defined.

(ii) The Cloud provider trains the model $f_{\blacktriangledown}(\hat{X}, \hat{\theta})$ on the encrypted training set $\widehat{TR}$ by considering different configurations of hyperparameters $\mathcal{H}_k, k = 1 \dots K$, being K a tunable parameter. We emphasize that the trained TFHE-NNs are characterized by weights and biases that are encrypted.

(iii) The encrypted Cross-Validation procedure is used to identify the best TFHE-NN of those trained with the K different hyperparameter configurations. Such model, denoted by $\hat{m}$, is made available to the user U in two different ways: the first way consists in returning the encrypted model directly to the user. In this case, U will decrypt $\hat{m}$ using the decryption function $D(\hat{m}, k_s)$, with k_s being the secret key. The other way consists in saving the encrypted model $\hat{m}$ on the Cloud, and making it available to operate in a privacy-preserving inference *as-a-service* manner.

5.2 Defining TFHE-NN models for encrypted learning

This subsection presents the building blocks needed to create the TFHE-NN models that can be trained and operate on encrypted data, thus addressing RQ1. Specifically, Section 5.2.1 details the data types and the operators, Section 5.2.2 explains how the TFHE-NN layers are modified to make them TFHE-compliant, while Section 5.2.3 describes the definition of the TFHE-NNs.

In this study, integer-only TFHE-NNs are considered. This has two main advantages. The first is that the design of arithmetic operators using TFHE logic gates for integers is much simpler than that for floating-point numbers. The second is that integers can be represented with fewer bits than floating-point values, thus reducing the time complexity of the TFHE-NN layers, which depends on the number of bits involved in the HE operations.

5.2.1 Sets and operators for TFHE-NN layers. First of all, the sets for all the data types used by TFHE-NNs were defined. A graphical representation of these sets is shown in Fig. 3. More specifically, the TFHE-NN weights, inputs, and outputs consist in the encryption of signed integers represented by a binary decomposition of length $N \in \mathbb{N}$. Formally, we *restrict* the set of integers $\mathbb{Z}$ by defining $\mathbb{Z}_N = \left[-\frac{2^N}{2}, +\frac{2^N}{2} - 1\right]$, being N the number of bits.

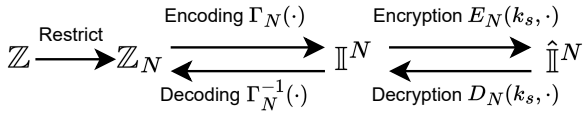

Figure 3: The sets considered in this study.

The next step consists in *encoding* the elements of $\mathbb{Z}_N$ with their two's complement binary decomposition. Let $\mathbb{I} = \{0, 1\}$, and let $\Gamma_N(\cdot) : \mathbb{Z}_N \rightarrow \mathbb{I}^N$ be the encoding function that transforms a member of $\mathbb{Z}_N$ into its two's complement binary decomposition over N bits:

$$\Gamma_N(z) = [i_{N-1}, \dots, i_0]$$

where $z \in \mathbb{Z}_N$ and $\{i_0, \dots, i_{N-1}\} \in \mathbb{I}$. For instance, the binary decomposition of $z = 7 \in \mathbb{Z}_N$ over $N = 16$ bits is: [0000000000000111].

Let $\hat{\mathbb{I}}$ be the set of TFHE ciphertexts, introduced in Section 4, and let $E(k_s, \cdot) : \mathbb{I} \rightarrow \hat{\mathbb{I}}$ be the encryption function that encrypts a single bit into a TFHE ciphertext. To *encrypt* a member of $\mathbb{Z}_N$, it is sufficient to encrypt each bit composing its binary decomposition. This function, operating on $\mathbb{I}^N \rightarrow \hat{\mathbb{I}}^N$, is defined as:

$$E_N(k_s, \Gamma(z)) = [E(k_s, i_{N-1}), \ldots, E(k_s, i_0)]$$

where z is an element of $\mathbb{Z}_N$, $\{i_0, \ldots, i_{N-1}\}$ are the elements composing its binary decomposition $\Gamma_N(z)$, and $\hat{\mathbb{I}}^N$ is the set of encrypted binary decompositions. The inverse of both the encryption and the encoding functions are the *decryption* function $D_N(k_s, \cdot)$ and the *decoding* function $\Gamma_N^{-1}(\cdot)$, respectively, with k_s being the secret key of user U.

We can now build a set of arithmetic operators over $\hat{\mathbb{I}}^N$ that represent the building blocks of the TFHE-NN layers. Formally, the following HE arithmetic operators have been designed and implemented:

$$\begin{aligned} \{+, -, *\}_\heartsuit &: \{\hat{\mathbb{I}}^N, \hat{\mathbb{I}}^N\} \rightarrow \hat{\mathbb{I}}^N, \\ \{<<, >>\}_\heartsuit &: \{\hat{\mathbb{I}}^N, \mathbb{N}\} \rightarrow \hat{\mathbb{I}}^N, \\ \{>, <, ==\}_\heartsuit &: \{\hat{\mathbb{I}}^N, \hat{\mathbb{I}}^N\} \rightarrow \hat{\mathbb{I}}, \\ \{\max(\cdot), \arg\max(\cdot)\}_\heartsuit &: \{\hat{\mathbb{I}}^N\}^M \rightarrow \hat{\mathbb{I}}^N. \end{aligned} \tag{4}$$

These operators have been designed relying on the TFHE Boolean gates and, from now on, the symbol ♥ will be used to denote operators and layers capable of processing TFHE encrypted values in $\hat{\mathbb{I}}^N$. In the following, we detail three HE arithmetic operators that represent the building blocks of the TFHE-NN layers of the model $f_\heartsuit(\hat{X}, \hat{\theta})$, leaving for the Appendix the complete description of the remaining. Let $\hat{a} = [\hat{A}_{N-1} \ldots \hat{A}_i \ldots \hat{A}_0]$, $\hat{b} = [\hat{B}_{N-1} \ldots \hat{B}_i \ldots \hat{B}_0]$, and $\hat{c} = [\hat{C}_{N-1} \ldots \hat{C}_i \ldots \hat{C}_0]$ be three encrypted values obtained after the application of the encryption function $E_N(k_s, \Gamma(\cdot))$.

Encrypted Adder. The $+_\heartsuit$ operator is implemented through an encrypted adder circuit, solely designed to receive in input two encrypted elements of $\hat{\mathbb{I}}^N$ and to produce as output an element of $\hat{\mathbb{I}}^N$:

$$\hat{a} +_\heartsuit \hat{b} = \hat{c} \;\text{ with } \hat{a}, \hat{b}, \hat{c} \in \hat{\mathbb{I}}^N$$

The basic block of $+_\heartsuit$ is a HE full adder, which is able to sum two individual TFHE bits, i.e., $\hat{A}_i \in \hat{a}$ and $\hat{B}_i \in \hat{b}$, $i = 0, \ldots, N-1$. In particular, the HE full adder computes a sum bit, $\hat{S}_i$, and a carry bit, $\hat{C}_i$, which are passed to the next full adder in the chain. By combining N full adders, it is possible to sum two encrypted binary values in $\hat{\mathbb{I}}^N$ as follows:

$$\begin{aligned} \hat{S}_i &= (\hat{A}_i \oplus_\heartsuit \hat{B}_i) \oplus_\heartsuit \hat{C}_{i-1} \\ \hat{C}_i &= (\hat{A}_i \bullet_\heartsuit \hat{B}_i) \underline{+}_\heartsuit (\hat{C}_{i-1} \bullet_\heartsuit (\hat{A}_i \oplus_\heartsuit \hat{B}_i)) \end{aligned}$$

where $\oplus_\heartsuit$ is the TFHE *XOR* gate, $\bullet_\heartsuit$ is the TFHE *AND* gate, and $\underline{+}_\heartsuit$ is the TFHE *OR* gate, while a graphical representation of $+_\heartsuit$ is shown in Fig. 4.

Encrypted Multiplier. The encrypted multiplier $*_\heartsuit$,

$$\hat{a} *_\heartsuit \hat{b} = \hat{c} \;\text{ with } \hat{a}, \hat{b}, \hat{c} \in \hat{\mathbb{I}}^N,$$

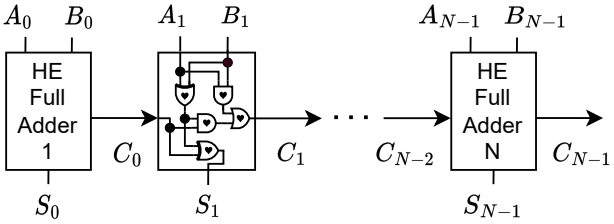

Figure 4: The encrypted adder $+_\heartsuit$.

is defined as $*_\heartsuit : \{\hat{\mathbb{I}}^N, \hat{\mathbb{I}}^N\} \rightarrow \hat{\mathbb{I}}^N$. The $*_\heartsuit$ operator is implemented through the shift-and-add multiplier circuit, which works by shifting the multiplicand $\hat{a}$ to the left by i bits, with $i = 0, \ldots, N-1$, and adding to the product $\hat{c}$ the result of a $MUX_\heartsuit$ gate which, based on the value of $\hat{B}_i$, selects between the shifted multiplicand $\hat{a}$ and an encrypted zero $\hat{0}$. Specifically, the $MUX_\heartsuit$ gate outputs an encrypted zero $\hat{0}$ if $\hat{B}_i$ corresponds to $\hat{0}$, the left-shifted multiplicand $\hat{a}$ if $\hat{B}_i$ is equal to an encrypted one $\hat{1}$. Formally:

$$\hat{c} = \sum_{i=0}^{N-1}{}_\heartsuit \, MUX_\heartsuit \left(\hat{B}_i,\ \hat{a} <<_\heartsuit i,\ \hat{0}\right)$$

More in detail, the HE $MUX_\heartsuit$ gate receives in input two encrypted values, i.e., $\hat{a} \in \hat{\mathbb{I}}^N$ and $\hat{b} \in \hat{\mathbb{I}}^N$, and an encrypted control signal bit $\hat{S} \in \hat{\mathbb{I}}$ (i.e., the value used to select the output value from the input values), and produces an encrypted output in $\hat{\mathbb{I}}^N$ as follow:

$$MUX_\heartsuit(\hat{S}, \hat{a}, \hat{b}) = (\hat{S} \bullet_\heartsuit \hat{a}) \underline{+}_\heartsuit ((\neg_\heartsuit \hat{S}) \bullet_\heartsuit \hat{b})$$

where $\bullet_\heartsuit$ is the TFHE *AND* gate, $\underline{+}_\heartsuit$ is the TFHE *OR* gate, and $\neg_\heartsuit$ is the TFHE *NOT* gate.

Encrypted Comparisons. The encrypted comparisons $>_\heartsuit$ ($<_\heartsuit$) and $==_\heartsuit$,

$$\begin{aligned} \hat{a} >_\heartsuit (<_\heartsuit) \hat{b} = \hat{C} \;&\text{ with } \hat{a}, \hat{b} \in \hat{\mathbb{I}}^N \text{ and } \hat{C} \in \hat{\mathbb{I}}, \\ \hat{a} ==_\heartsuit \hat{b} = \hat{C} \;&\text{ with } \hat{a}, \hat{b} \in \hat{\mathbb{I}}^N \text{ and } \hat{C} \in \hat{\mathbb{I}}, \end{aligned}$$

are defined as $>_\heartsuit$ ($<_\heartsuit$) : $\{\hat{\mathbb{I}}^N, \hat{\mathbb{I}}^N\} \rightarrow \hat{\mathbb{I}}$ and $==_\heartsuit$: $\{\hat{\mathbb{I}}^N, \hat{\mathbb{I}}^N\} \rightarrow \hat{\mathbb{I}}$. They output a single encrypted bit $\hat{C}$, initialized to zero $\hat{0}$, which is equal to an encrypted one $\hat{1}$ if the comparison is *True*, or an encrypted zero $\hat{0}$ if the comparison is *False*. For example, the $>_\heartsuit$ operator outputs $\hat{C}$ equals to an encrypted one $\hat{1}$ if $\hat{a}$ is greater than $\hat{b}$, an encrypted zero $\hat{0}$ otherwise. The update rule of $\hat{C}$, for each bit $i = 0, \ldots, N-1$, is as follows:

$$\hat{C} = MUX_\heartsuit \left((\hat{a} \odot_\heartsuit \hat{b})_i, \hat{C}, \hat{A}_i\right)$$

To handle negative values as well, a final $MUX_\heartsuit$ is required:

$$\hat{C} = MUX_\heartsuit \left((\hat{a} \odot_\heartsuit \hat{b})_{N-1}, \neg_\heartsuit \hat{C}, \hat{C}\right)$$

where $\odot_\heartsuit$ is the TFHE *XNOR* gate, and $\neg_\heartsuit$ is the TFHE *NOT* gate. For the $==_\heartsuit$ operator, instead, $\hat{C}$, for $i = 0, \ldots, N-1$, is updated as:

$$\hat{C} = (\hat{a} \oplus_\heartsuit \hat{b})_i \underline{+}_\heartsuit \hat{C}$$

where $\oplus_\heartsuit$ is the TFHE *XOR* gate, and $\underline{+}_\heartsuit$ is the TFHE *OR* gate. The resulting value of $\hat{C}$ is negated through the TFHE *NOT* gate $\neg_\heartsuit$ before being produced as output.

5.2.2 TFHE-NN layers. Once the arithmetic operators have been defined, the common layers used in NNs are re-designed to process encrypted data belonging to $\hat{\mathbb{I}}^N$. In particular, the following layers have been defined: encrypted Fully Connected layer ($FC_\blacktriangledown$), encrypted Max Pooling layer ($MAX_\blacktriangledown$), and the encrypted *tanh* activation function ($tanh_\blacktriangledown$).

Encrypted Fully Connected layer. The $FC_\blacktriangledown$ layer has been designed to use the $*_\blacktriangledown$ operator to multiply the encrypted inputs $\hat{x}_j$ with the encrypted weights $\hat{W}_{ij}$, and the $+_\blacktriangledown$ operator to compute the sum over the multiplication output and the encrypted biases, as expressed in Equation 5:

$$\hat{y}_i = \sum_{j=1}^{n}{}_\blacktriangledown \; \hat{w}_{ij} *_\blacktriangledown \hat{x}_j +_\blacktriangledown \hat{b}_i \tag{5}$$

where $\hat{y}_i \in \hat{\mathbb{I}}^N$ is the encrypted output of the i-th neuron of the current layer, $\hat{x}_j \in \hat{\mathbb{I}}^N$ is the encrypted output of the j-th neuron of the previous layer, $\hat{w}_{ij} \in \hat{\mathbb{I}}^N$ is the encrypted weight of the connection between the i-th neuron in the current layer and the j-th neuron of the previous layer, $\hat{b}_i \in \hat{\mathbb{I}}^N$ is the encrypted bias of the i-th neuron in the current layer, and n is the total number of neurons in the previous layer.

Encrypted Max Pooling layer. The $MAX_\blacktriangledown$ layer relies on the $\max_\blacktriangledown$ function designed to operate on tensors of elements in $\hat{I}^N$ of any size. Let (h, w) be the window size and (s, s) the stride. The $MAX_\blacktriangledown$ layer can be expressed as:

$$\hat{y}_{i,j} = \max_{h,w}{}_\blacktriangledown(\hat{x}_{is:is+h, js:js+w}) \tag{6}$$

where $\hat{x} \in \{\hat{\mathbb{I}}^N\}^{H\times W}$ is the encrypted input tensor of size $H \times W$, and $\hat{y} \in \{\hat{\mathbb{I}}^N\}^{K\times L}$ is the encrypted output tensor of size $K \times L$, with

$$(K, L) = \left(\frac{(H-h)}{s} + 1, \frac{(W-w)}{s} + 1\right).$$

The implementation details of the encrypted circuit that performs the $\max_\blacktriangledown$ function are provided in the Appendix.

Encrypted Activation Function. The activation function of DL models requires special attention, as it usually operates with real values that cannot be directly computed using integer-only arithmetic. Inspired by [28] and [39], this work implements the encrypted version of the rescaled Piecewise Linear Approximation (PLA) of the *tanh* function, which is considered the best activation function to use in conjunction with the DFA learning algorithm [29]. PLA consists in the segment approximation of non-linear activation functions [2] to make them effective in integer-only NNs. The encrypted circuit of the $tanh_\blacktriangledown$ relies on a HE lookup table ($LUT_\blacktriangledown$). This table is designed by combining together several multiplexer gates of the TFHE scheme. Depending on the value of the input, the output of the $LUT_\blacktriangledown$ is one of the segments into which the $tanh_\blacktriangledown$ is partitioned. Interestingly, the activation gradients $tanh'_\blacktriangledown$ are also calculated through a $LUT_\blacktriangledown$ in which the outputs are the slopes of the segments used to linearize the original hyperbolic tangent. More precisely, the outputs are computed as their rescaled inverse, since the original slopes are real values in $[0, 2]$.

5.2.3 Designing TFHE-NN models. Having defined all the necessary operators and layers, we can design the processing of an entire TFHE-NN that operates on encrypted inputs. Let $Y = f(X; \theta)$ be the model, where X is the input, Y the output, and

$$\theta = \left\{[W^1, \boldsymbol{b}^1], \ \ldots \ , [W^H, \boldsymbol{b}^H]\right\}$$

the set of weights and biases, where H is the number of hidden layers in f. To convert f to $f_\blacktriangledown$ (i.e., its corresponding TFHE version), it is necessary to convert all its layers to their corresponding HE version, as well as to encode and encrypt their weights and biases. The final model, capable of processing encrypted inputs, is defined as follows:

$$f_\blacktriangledown(\hat{X}; \hat{\theta}) = g_{\blacktriangledown_H}(g_{\blacktriangledown_{H-1}}(\ldots(g_{\blacktriangledown_1}(\hat{X}; \hat{W}^1, \hat{\boldsymbol{b}}^1); \ldots); \hat{W}^H, \hat{\boldsymbol{b}}^H)$$

where $g_{\blacktriangledown_h}$ is the non-linear function applied in the h-th layer.

The shape of $\hat{X}$ depends on the task under consideration. For instance, in an image classification scenario, $\hat{X}$ is a three-dimensional matrix of size $h \times w \times c$, with h being the height, w the width, and c the number of channels.

5.3 Training TFHE-NNs

This section introduces the encrypted loss function, the training algorithm and the parallel learning of TFHE-NNs as well as the encrypted cross-validation circuit able to select the best TFHE-NN in a privacy-preserving manner.

5.3.1 The Encrypted Loss Function. During the encrypted learning procedure, both the loss and the accuracy of the TFHE-NN remain encrypted after the forward pass. Nonetheless, it is essential to be able to compute the derivative of the loss because it is used to calculate the update of the network weights in the DFA learning algorithm, as shown in Equation 1. The loss function that is considered in this study is the HE version of the *sum of squared errors*. Indeed, its derivative $L'_\blacktriangledown$ is a simple subtraction between the TFHE-NN prediction $f_\blacktriangledown(\hat{x})$ and the target output $\hat{y}$, an operation that can be calculated on encrypted values by using the HE operator defined in Eq. 4. Formally, with bs being the mini-batch size, we can define $L'_\blacktriangledown$ as:

$$L'_\blacktriangledown = \sum_{i=1}^{bs}{}_\blacktriangledown \; f_\blacktriangledown(\hat{x}_i) -_\blacktriangledown \hat{y}_i.$$

5.3.2 The Training Algorithm. Training a TFHE-NN model over encrypted data has to be performed by considering that only integer values can be used during the entire training procedure. This differentiates this study from other works, e.g., in the TinyML [45] area, where DL models are initially trained by using floating-point values which are then quantized at the end of the procedure.

In this work, the learning algorithm DFA has been redesigned into $\text{DFA}_\blacktriangledown$, which is its HE-compliant version that uses the arithmetic operators defined in Eq. 4. With H being the number of hidden layers of the TFHE-NN, we can rewrite the layers update directions defined in Eq. 1 into $\hat{\delta}^h_{DFA_\blacktriangledown}$, $h = 1, \ldots, H$, as:

$$\hat{\delta}^h_{DFA_\blacktriangledown} = \begin{cases} L'_\blacktriangledown \odot_\blacktriangledown \text{act}'^h_\blacktriangledown(\hat{\mathbf{a}}^h) & \text{for } h = H \\ L'_\blacktriangledown \cdot_\blacktriangledown \hat{B}^h \odot_\blacktriangledown \text{act}'^h_\blacktriangledown(\hat{\mathbf{a}}^h) & \text{for } 1 \le h \le H-1 \end{cases}$$

where $\cdot_\blacktriangledown$ is the HE matrix multiplication operator, $\odot_\blacktriangledown$ is the HE element-wise multiplication operator, and $\hat{\mathbf{a}}^h$ and $\hat{B}^h$ are the encrypted output of the layer before the application of the non-linearity, and the encrypted fixed random weight matrix of layer h, respectively. In our TFHE-NN, we consider $\text{act}'^h(\cdot) = tanh'^h_\blacktriangledown$. Consequently, the weights and bias updates defined in Eq. 2 and Eq. 3, for each layer $h = 1, \dots, H$, become:

$$\delta\hat{W}^h = -_\blacktriangledown\ \hat{\mathbf{o}}^{h-1\intercal} \cdot_\blacktriangledown \hat{\delta}^h_{DFA_\blacktriangledown}$$
$$\delta\hat{\mathbf{b}}^h = -_\blacktriangledown\ \hat{\delta}^h_{DFA_\blacktriangledown}$$

where $\hat{\mathbf{o}}^{h-1}$ is the encrypted output of the previous layer in the TFHE-NN.

More specifically, $\text{DFA}_\blacktriangledown$ receives in input the TFHE-NN to be trained $f_\blacktriangledown(\hat{X}, \hat{\theta})$, the encrypted training set $\widehat{TR}$, and a configuration of hyperparameters $\mathcal{H} = \{e, bs, \eta\}$, where e is the number of epochs, bs the mini-batch size, and η the learning rate. As only integer values are allowed in the proposed solution, we consider values of the learning rate η that are negative powers of two (i.e., $\eta = \frac{1}{128}, \frac{1}{256}, \dots$). In this way, the gradient of the activation function simply needs to be shifted by a fixed amount $\alpha = \log_2 \eta^{-1}$ before being used to update the weights.

5.3.3 Parallel training. To answer research question RQ2, we designed and implemented the encrypted version of the Model Averaging mechanism [40] to parallelize the learning procedure on M different computing units, being M a power of two. Specifically, the encrypted training dataset $\widehat{TR}$ is randomly split into M different batches $\hat{X}_1, \dots, \hat{X}_M$ of equal size, being M a parameter that is bound by the technological solution operating on the third-party service. Here we assume, without any loss of generality, that the size of TR is a multiple of M. Then, each machine trains the model $f_\blacktriangledown(\hat{X}_m; \hat{\theta})$, $m = 1, \dots, M$, with the same configuration of hyperparameters $\mathcal{H}$, obtaining $\hat{\theta}_m$. The choice of M is a trade-off between the final accuracy loss and the reduction in the total training time, which is exactly proportional to M. This aspect is further explored in Section 6.2. At the end of the parallel training procedure, M different trained TFHE-NN models are collected. In order to obtain the final set of network parameters $\hat{\theta}$, we average the encrypted weights and biases for each layer, by using the HE operators defined in Eq. 4:

$$\hat{\theta} = \frac{1}{M} \sum_{m=1}^{M}{}_\blacktriangledown\ \hat{\theta}_m$$

As with the learning rate η, setting M to a power of two allows us to keep integer values by simply shifting the result of the summation by a fixed amount $\alpha = \log_2 M$.

We emphasize that we are not proposing a new model averaging method, but we have redesigned the well-known MA [40] to make it HE-compliant and to enable it to work with encrypted data and encrypted weights in order to reduce the computational overhead added by the TFHE scheme.

5.4 Encrypted Cross-Validation

In order to evaluate different configurations of hyperparameters and select the one that provides the highest accuracy, we defined K different sets of hyperparameters $\mathcal{H} = [\mathcal{H}_1, \mathcal{H}_2, \dots, \mathcal{H}_K]$, with K being a tunable parameter, on which K different TFHE-NNs have been trained, each of which with its own configuration $\mathcal{H}_k$. Once the training is completed, the K different TFHE-NNs, i.e., $f_\blacktriangledown(\hat{X}, \hat{\theta}_k)$ with their trained encrypted weights and biases $\hat{\theta}_k$, $k = 1, \dots, K$, are provided to the HE Cross-Validation ($CV_\blacktriangledown$) circuit.

Algorithm 1 Encrypted Learning with $CV_\blacktriangledown$

1: Define $f_\blacktriangledown(\hat{X}, \hat{\theta})$
2: Let $\mathcal{H}_k = \{e_k, bs_k, \eta_k\}$ **for each** $k \in [1, K]$
3: **for** $k = 1$ **to** K **do**
4: Train model k with $\text{DFA}_\blacktriangledown(f_\blacktriangledown(\hat{X}, \hat{\theta}), \widehat{TR}, \mathcal{H}_k)$
5: Compute $\hat{V}_k$
6: **end for**
7: $\hat{\beta} = \underset{k\in[1,K]}{\arg\max}{}_\blacktriangledown(\hat{V}_k)$
8: $\hat{\theta}_{\hat{\beta}} = CV_\blacktriangledown\left(\hat{\beta}, \left[f_\blacktriangledown(\hat{X}, \hat{\theta}_1), \dots, f_\blacktriangledown(\hat{X}, \hat{\theta}_K)\right]\right)$
9: return $f_\blacktriangledown(\hat{X}, \hat{\theta}_{\hat{\beta}})$

The goal of the cross validation procedure is to select, among these K TFHE-NNs, the one with the highest validation accuracy. However, given that the TFHE-NNs are encrypted, the corresponding K accuracies on the validation part of the dataset are also encrypted. This means that even this phase must be performed in an encrypted manner by using the HE operators defined in Eq. 4. More specifically, let $\hat{V}_k$ be the encrypted validation accuracy of the model $f_\blacktriangledown(\hat{X}, \hat{\theta}_k)$, $k = 1, \dots, K$. We can define the following $\arg\max_\blacktriangledown$ operator, detailed in the Appendix, that works on encrypted values:

$$\hat{\beta} = \underset{k\in[1,K]}{\arg\max}{}_\blacktriangledown(\hat{V}_k)$$

In order to design the $CV_\blacktriangledown$ circuit, which is able to extract $\hat{\theta}_{\hat{\beta}}$ (i.e., the set of encrypted weights and biases of the best model), we combined several $MUX_\blacktriangledown$ gates, as detailed in Fig. 5. In particular, each element of $\hat{W}^l_k$ and $\hat{\boldsymbol{b}}^l_k$, for all K models and for all L layers, passes through a $MUX_\blacktriangledown$ gate. After the $MUX_\blacktriangledown$ chain, for each layer L, we obtained $\hat{W}^l_{\hat{\beta}}$ and $\hat{\boldsymbol{b}}^l_{\hat{\beta}}$, which represent the weights matrix and the biases vector whose elements are extracted from the best model $\hat{\beta}$.

Before this, we need to compute the vector of control signals $\hat{s} = [\hat{S}_1 \dots \hat{S}_K]$, which represents the encrypted one-hot encoding of $\hat{\beta}$ (i.e., a vector of $K-1$ encrypted 0's and only one encrypted 1 in position $\hat{\beta}$). Since $\hat{\beta}$ is also encrypted, to place the encrypted 1 in the correct position of $\hat{s}$, we simply run a *for* loop and check the index, thanks to the $==_\blacktriangledown$ circuit defined in Eq. 4.

The $CV_\blacktriangledown$ circuit can be summarized as follows:

$$\hat{\theta}_{\hat{\beta}} = CV_\blacktriangledown\left(\hat{\beta}, \left[f_\blacktriangledown(\hat{X}, \hat{\theta}_1), \dots, f_\blacktriangledown(\hat{X}, \hat{\theta}_K)\right]\right),$$

and the final model $f_\blacktriangledown(\hat{X}, \hat{\theta}_{\hat{\beta}})$, obtained as specified in Algorithm 1, is then made available to the user U according to the modalities described in Section 5.1.

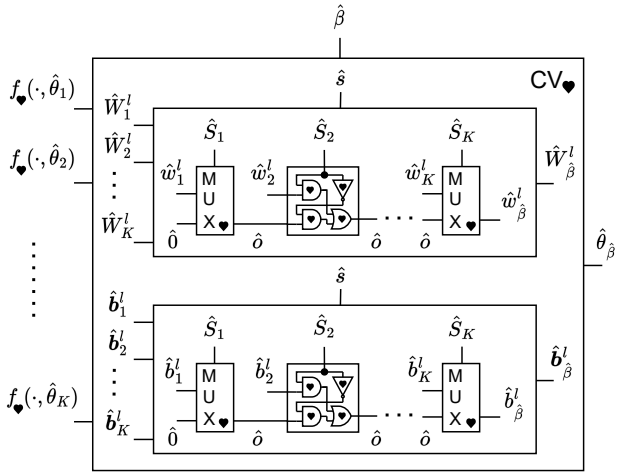

Figure 5: The encrypted Cross-Validation circuit $CV_{\heartsuit}$.

6 Experimental results

In this section, the proposed solution is evaluated from different perspectives. The TFHE-NNs designed and datasets considered in this experimental campaign are detailed in Tab. 1 and Tab. 2, respectively. The code of the following experiments, written in Python, is available as a public repository.

6.1 Training on encrypted data

The goal of the first experiment is to show that the proposed TFHE-NN can be trained on encrypted data by using the proposed DFA$_{\heartsuit}$ learning algorithm. To this end, we trained TFHE-NN-1 (see details in Tab. 1) by using $N = 22$ bits long binary decomposition on the T-MNIST dataset, with the following set of hyperparameters: $e = 3$, $bs = 5$, and $\eta = \frac{1}{256}$. We evaluated the validation and test accuracies and compared them to those obtained by the same model trained on plain data.

As we can see from Tab. 3, there are no differences in terms of validation and test accuracy between training the model on plain or encrypted data. This is not surprising, as the model obtained with the encrypted learning is exactly the same as that obtained with the plain learning. To expand on this point, we computed $\left\|\theta - D(k_s, \hat{\theta})\right\|_2$, i.e., the L_2 norm of the difference between θ and $D(k_s, \hat{\theta})$, where θ is the vector of weights and biases of the model obtained by training on plain data and $\hat{\theta}$ is the vector of weights and biases of the model obtained by training on encrypted data, and we have verified that it is exactly equal to 0. We emphasize that the training on encrypted and plain data required 13140 minutes and 2.75 seconds, respectively. This demonstrates the significant computational overhead when dealing with encrypted data through HE.

6.2 Parallel training

In this section we show that the encrypted model averaging presented in Section 5.3.3, which is used to parallelize the learning procedure, allows to reduce the computational demand when training with DFA$_{\heartsuit}$.

Table 1: TFHE-NNs considered in this experimental section. The *tanh*$_{\heartsuit}$ activation function has been used after each $FC_{\heartsuit}$ layer.

Model	Layers
TFHE-NN-1	$MAX_{\heartsuit}(8 \times 8) \rightarrow FC_{\heartsuit}(4) \rightarrow FC_{\heartsuit}(2) \rightarrow FC_{\heartsuit}(3)$
TFHE-NN-2	$FC_{\heartsuit}(200) \rightarrow FC_{\heartsuit}(10)$
TFHE-NN-3	$FC_{\heartsuit}(2) \rightarrow FC_{\heartsuit}(3)$

Table 2: Datasets for the classification tasks considered in this experimental campaign. T-MNIST is a subset of MNIST [22] which comprises 50 randomly sampled images representing three digits (i.e., 0, 1, and 2), cropped in the center. Fashion-MNIST [46] contains images of Zalando's articles. For the Penguins [20] dataset, only the four out-of-six most significant features were used.

	T-MNIST	FashionMNIST	Penguins
Type	Images	Images	Numerical
Input size	16×16	28×28	4
Classes	3	10	3
Train size	50	50000	150
Validation size	5000	10000	64
Test size	10000	10000	130

First, we reconsidered the experiment presented in Section 6.1 by parallelizing the training on $M = 2$ computing units. As the results in Tab. 4 show, the total training time is halved. This comes at the expense of a loss in the validation and test accuracy between the model trained on the entire dataset (i.e., $M = 1$) and the model trained in a parallelized manner (i.e., $M = 2$), which are -5.38% and -5.18%, respectively. It is important to consider that the time overhead introduced by the weights' aggregation is only about 29 minutes.

Then, we trained TFHE-NN-2 on the FashionMNIST dataset. It should be noted that this experiment was carried out only on plain data, over 100 runs. The reason is that, given the current computational overhead of TFHE, this experiment would have taken about 1.5×10^{11} minutes to complete, for $M = 1$. Nonetheless, the trained models are exactly the same as those that would be obtained in an encrypted scenario, as shown in Section 6.1. The results, for different values of M (i.e., $M = 1, 2, 4, \ldots$), are collected in Tab. 5. These results show that the parallel training allows to reduce the total training time by a factor M. Nonetheless, this advantage comes at the expense of a reduction in the validation accuracy of the models, thus highlighting a trade-off between training time reduction and accuracy loss.

6.3 Cross-validation

Lastly, an experiment to evaluate the encrypted CV procedure $CV_{\heartsuit}$, introduced in Section 5.4, is proposed. We performed encrypted training of TFHE-NN-3 (with $N = 16$ bits) on the encrypted Penguin dataset, by considering 4 different hyperparameter configurations. As shown in Tab. 6, the four different sets of hyperparameters resulted in four different encrypted model weights $\hat{\theta}_k, k = 1, \ldots, 4$.

Table 3: Validation and test accuracy of TFHE-NN-1 trained on the T-MNIST dataset, with training times (in minutes). Two cases are shown: *Plain*, in which the model is trained on the plain training set, and *Encrypted*, in which the model is trained on the encrypted training set.

Dataset	Validation Acc.	Test Acc.	Training time [min]	$\left\|\theta - D(k_s, \hat{\theta})\right\|_2$
Plain	0.920	0.919	0.05	/
Encrypted	0.920	0.919	13140.0	0

Table 4: Results of the parallel encrypted learning of TFHE-NN-1 on the T-MNIST dataset, with training times (in minutes).

M	Images per model	Validation Acc. Aggregation	Test Acc. Aggregation	Training time [min]
1	50	0.920	0.919	13140.0
2	25	0.865	0.867	6570.0

The weights that provided the highest accuracy, i.e., $\hat{\theta}_{\hat{\beta}}$, were selected by $CV_{\heartsuit}$ and then decrypted in order to check the testing accuracy of the final model. Given that $CV_{\heartsuit}$ operates by selecting the TFHE-NN that provides the highest validation accuracy without decrypting either the models or the validation accuracies, the output of $CV_{\heartsuit}$ is the TFHE-NN with the same weights as the model trained with configuration $K = 1$. This is confirmed by the fact that the L_2 norm of the difference between $\hat{\theta}_1$ and $\hat{\theta}_{\hat{\beta}}$, after decryption, is equal to 0. The final test accuracy of the best model $f_{\heartsuit}(\hat{X}, \hat{\theta}_1)$ is equal to 0.933.

Concerning the training times, we can see that training a single model took more than three days, while the encrypted validation accuracy computation took about 5 hours. The overhead introduced by the $CV_{\heartsuit}$ circuit only took about 20 seconds.

6.4 Limits and Advantages

While the proposed solution allows us to train TFHE-NNs on encrypted data, it suffers from two main limitations. First, the choice of DFA as the learning algorithm, as well as the choice of an integer-only arithmetic, might lead to a sub-optimal solution compared to the same NN trained with BP and floating-point values. Second, the overhead added by TFHE increases the total training time of the TFHE-NN on encrypted data with respect to the same network trained on plain data.

On the other hand, the proposed solution is the first attempt to provide end-to-end privacy-preserving learning on encrypted data in a *as-a-service* scenario. Our solution is perfectly capable of training an artificial neural network and selecting the best model thanks to the encrypted cross-validation procedure, without ever seeing neither the user-supplied data or the models obtained. Our work is paving the way in this area of research, where advances are expected to significantly improve the efficiency of HE, leading to progressively faster processing and computation on encrypted data [12, 34, 35, 43].

Table 5: Results of the parallel encrypted learning on FashionMNIST dataset, over 100 runs.

M	Images per model	Validation Acc. Aggregation	Training time reduction factor
1	60000	$0.860 \pm 4.38 \times 10^{-6}$	1
2	30000	$0.834 \pm 2.65 \times 10^{-4}$	2
4	15000	$0.839 \pm 1.69 \times 10^{-5}$	4
8	7500	$0.832 \pm 3.97 \times 10^{-6}$	8
16	3750	$0.818 \pm 3.47 \times 10^{-6}$	16
32	1875	$0.805 \pm 3.75 \times 10^{-6}$	32
64	937	$0.774 \pm 4.73 \times 10^{-6}$	64

Table 6: Cross-validation results, for 4 different sets of hyperparameters configurations $\mathcal{H}_k = \{\eta, bs\}$. $\hat{V}_k$ is the encrypted validation accuracy. The times for training (t_t), validation (t_v), and $CV_{\heartsuit}$ ($t_{CV_{\heartsuit}}$) are expressed in minutes. $\Delta(\hat{\theta}_k, \hat{\theta}_{\hat{\beta}}) = \left\|D(k_s, \hat{\theta}_k) - D(k_s, \hat{\theta}_{\hat{\beta}})\right\|_2$.

K	η^{-1}	bs	$\hat{V}_k$	$\Delta(\hat{\theta}_k, \hat{\theta}_{\hat{\beta}})$	t_t	t_v	$t_{CV_{\heartsuit}}$
1	256	25	0.922	0	4960.0	600.0	0.3
2	256	50	0.875	-414.5	4960.0	600.0	/
3	512	25	0.859	-1112.5	4960.0	600.0	/
4	512	50	0.844	-2595.9	4960.0	600.0	/

7 Conclusions

The aim of this study was to introduce a novel approach for training privacy-preserving NNs on encrypted data, leveraging HE. Specifically, we introduced: TFHE-NNs, which are HE-compliant neural networks specifically designed to work with encrypted data, a special learning algorithm called DFA$_{\heartsuit}$, able to train TFHE-NNs on encrypted data, and a CV procedure adapted to work on encrypted TFHE-NNs.

While the current computational overhead is relevant (as shown in Section 6.2), our proposed solution paves the way for privacy-preserving training on encrypted data. The proposed TFHE-NN and DFA-based encrypted learning algorithm are a first step in this research direction. Future works will investigate new solutions specifically designed to meet the requirements of the TFHE scheme, as well as the introduction of new layers in the TFHE-NN family and further optimizations in the learning procedure.

Acknowledgments

This paper is supported by Dhiria s.r.l. and by PNRR-PE-AI FAIR project funded by the NextGeneration EU program.

References

[1] European Commission [n. d.]. *2018 reform of EU data protection rules.* European Commission. https://commission.europa.eu/law/law-topic/data-protection/data-protection-eu_en

[2] Koldo Basterretxea, Jose Manuel Tarela, and Inés del Campo. 2004. Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons. *IEE Proceedings-Circuits, Devices and Systems* 151, 1 (2004), 18–24.

[3] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. 2019. nGraph-HE2: A high-throughput framework for neural network inference on encrypted data. In *Proceedings of the 7th ACM Workshop on Encrypted*

Computing & Applied Homomorphic Cryptography. 45–56.
[4] Jean-Philippe Bossuat, Rosario Cammarota, Jung Hee Cheon, Ilaria Chillotti, Benjamin R Curtis, Wei Dai, Huijing Gong, Erin Hales, Duhyeong Kim, Bryan Kumara, et al. 2024. Security Guidelines for Implementing Homomorphic Encryption. *Cryptology ePrint Archive* (2024).
[5] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.
[6] Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. 2013. Classical hardness of learning with errors. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. 575–584.
[7] Mark Campbell. 2022. Privacy-Preserving Computation: Doomed to Succeed. *Computer* 55, 8 (2022), 95–99.
[8] Ke Cheng, Ning Xi, Ximeng Liu, Xinghui Zhu, Haichang Gao, Zhiwei Zhang, and Yulong Shen. 2023. Private inference for deep neural networks: a secure, adaptive, and efficient realization. *IEEE Trans. Comput.* (2023).
[9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International conference on the theory and application of cryptology and information security*. Springer, 409–437.
[10] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91.
[11] Pierre-Emmanuel Clet, Martin Zuber, Aymen Boudguiga, Renaud Sirdey, and Cédric Gouy-Pailler. 2022. Putting up the swiss army knife of homomorphic calculations by means of TFHE functional bootstrapping. *Cryptology ePrint Archive* (2022).
[12] Yarkın Doröz, Erdinç Öztürk, and Berk Sunar. 2014. Accelerating fully homomorphic encryption in hardware. *IEEE Trans. Comput.* 64, 6 (2014), 1509–1521.
[13] Nir Drucker, Guy Moshkowich, Tomer Pelleg, and Hayim Shaul. 2024. BLEACH: cleaning errors in discrete computations over CKKS. *Journal of Cryptology* 37, 1 (2024), 3.
[14] Alessandro Falcetta and Manuel Roveri. 2022. Privacy-preserving deep learning with homomorphic encryption: An introduction. *IEEE Computational Intelligence Magazine* 17, 3 (2022), 14–25.
[15] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive* (2012).
[16] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Stanford university.
[17] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Annual Cryptology Conference*. Springer, 75–92.
[18] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*. PMLR, 201–210.
[19] Shai Halevi and Victor Shoup. 2021. Bootstrapping for helib. *Journal of Cryptology* 34, 1 (2021), 1–44.
[20] Allison Marie Horst, Alison Presmanes Hill, and Kristen B Gorman. 2020. *palmerpenguins: Palmer Archipelago (Antarctica) penguin data*. https://doi.org/10.5281/zenodo.3960218 R package version 0.1.0.
[21] Tanveer Khan, Mindaugas Budzys, Khoa Nguyen, and Antonis Michalas. 2024. Wildest Dreams: Reproducible Research in Privacy-preserving Neural Network Training. *arXiv preprint arXiv:2403.03592* (2024).
[22] Yann LeCun. 2018. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/
[23] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation* 1, 4 (1989), 541–551.
[24] Seewoo Lee, Garam Lee, Jung Woo Kim, Junbum Shin, and Mun-Kyu Lee. 2023. HETAL: efficient privacy-preserving transfer learning with homomorphic encryption. In *International Conference on Machine Learning*. PMLR, 19010–19035.
[25] Timothy P Lillicrap, Daniel Cownden, Douglas B Tweed, and Colin J Akerman. 2014. Random feedback weights support learning in deep neural networks. *arXiv preprint arXiv:1411.0247* (2014).
[26] Qian Lou, Bo Feng, Geoffrey Charles Fox, and Lei Jiang. 2020. Glyph: Fast and accurately training deep neural networks on encrypted data. *Advances in Neural Information Processing Systems* 33 (2020), 9193–9202.
[27] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On ideal lattices and learning with errors over rings. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 1–23.
[28] Ashkan Hosseinzadeh Namin, Karl Leboeuf, Huapeng Wu, and Majid Ahmadi. 2009. Artificial neural networks activation function HDL coder. In *2009 IEEE international conference on electro/information technology*. IEEE, 389–392.
[29] Arild Nøkland. 2016. Direct feedback alignment provides learning in deep neural networks. *Advances in neural information processing systems* 29 (2016).
[30] Prajwal Panzade, Daniel Takabi, and Zhipeng Cai. 2024. I can't see it but I can Fine-tune it: On Encrypted Fine-tuning of Transformers using Fully Homomorphic Encryption. *arXiv preprint arXiv:2402.09059* (2024).
[31] Saerom Park, Junyoung Byun, Joohee Lee, Jung Hee Cheon, and Jaewook Lee. 2020. HE-friendly algorithm for privacy-preserving SVM training. *IEEE Access* 8 (2020), 57414–57425.
[32] Robert Podschwadt, Daniel Takabi, Peizhao Hu, Mohammad H Rafiei, and Zhipeng Cai. 2022. A Survey of Deep Learning Architectures for Privacy-Preserving Machine Learning with Fully Homomorphic Encryption. *IEEE Access* (2022).
[33] Oded Regev. 2009. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)* 56, 6 (2009), 1–40.
[34] Dayane Reis, Jonathan Takeshita, Taeho Jung, Michael Niemier, and Xiaobo Sharon Hu. 2020. Computing-in-memory for performance and energy-efficient homomorphic encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 11 (2020), 2300–2313.
[35] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 387–398.
[36] Sinem Sav, Apostolos Pyrgelis, Juan R Troncoso-Pastoriza, David Froelicher, Jean-Philippe Bossuat, Joao Sa Sousa, and Jean-Pierre Hubaux. 2020. POSEIDON: Privacy-preserving federated neural network learning. *arXiv preprint arXiv:2009.00349* (2020).
[37] Frederic Schlackl, Nico Link, and Hartmut Hoehle. 2022. Antecedents and Consequences of Data Breaches: A Systematic Review. *Information & Management* (2022), 103638.
[38] Gordon K Smyth. 1998. Polynomial approximation. *Encyclopedia of Biostatistics* 13 (1998).
[39] Jaewoo Song and Fangzhen Lin. 2022. PocketNN: Integer-only Training and Inference of Neural Networks via Direct Feedback Alignment and Pocket Activations in Pure C++. *arXiv preprint arXiv:2201.02863* (2022).
[40] Hang Su and Haoyu Chen. 2015. Experiments on parallel training of deep neural network using model averaging. *arXiv preprint arXiv:1507.01239* (2015).
[41] Harry Chandra Tanuwidjaja, Rakyong Choi, Seunggeun Baek, and Kwangjo Kim. 2020. Privacy-preserving deep learning on machine learning as a service—a comprehensive survey. *IEEE Access* 8 (2020), 167425–167447.
[42] Han Tian, Chaoliang Zeng, Zhenghang Ren, Di Chai, Junxue Zhang, Kai Chen, and Qiang Yang. 2022. Sphinx: Enabling privacy-preserving online learning over the cloud. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2487–2501.
[43] Furkan Turan, Sujoy Sinha Roy, and Ingrid Verbauwhede. 2020. HEAWS: An accelerator for homomorphic encryption on the Amazon AWS FPGA. *IEEE Trans. Comput.* 69, 8 (2020), 1185–1196.
[44] Roman Walch, Samuel Sousa, Lukas Helminger, Stefanie Lindstaedt, Christian Rechberger, and Andreas Trügler. 2022. Cryptotl: Private, efficient and secure transfer learning. *arXiv preprint arXiv:2205.11935* (2022).
[45] Pete Warden and Daniel Situnayake. 2019. *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media.
[46] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747* (2017).

A HE Operators

In the following, we detail all the HE arithmetic operators defined in Section 5.2.1, which represent the building blocks of the TFHE-NN layers of the model $f_{\blacktriangledown}(\hat{X}, \hat{\theta})$. Let $\hat{a} = [\hat{A}_{N-1} \ \dots \ \hat{A}_i \ \dots \ \hat{A}_0]$, $\hat{b} = [\hat{B}_{N-1} \ \dots \ \hat{B}_i \ \dots \ \hat{B}_0]$, and $\hat{c} = [\hat{C}_{N-1} \ \dots \ \hat{C}_i \ \dots \ \hat{C}_0]$ be three encrypted values obtained after the application of the encryption function $E_N(k_s, \Gamma(\cdot))$ as explained in Section 5.2.1.

A.1 Encrypted Subtractor

The encrypted subtractor $-_{\blacktriangledown}$,

$$\hat{a} -_{\blacktriangledown} \hat{b} = \hat{c} \ \text{ with } \hat{a}, \hat{b}, \hat{c} \in \hat{\mathbb{I}}^N,$$

is defined as $-_{\blacktriangledown} : \{\hat{\mathbb{I}}^N, \hat{\mathbb{I}}^N\} \rightarrow \hat{\mathbb{I}}^N$. The basic block of $-_{\blacktriangledown}$ is a HE full subtractor, which is able to subtract two individual TFHE bits, i.e., $\hat{A}_i \in \hat{a}$ and $\hat{B}_i \in \hat{b}$, $i = 0, \dots, N-1$. In particular, the HE full subtractor computes a difference bit, $\hat{D}_i$, and a borrow bit, $\widehat{BR}_i$, which are passed to the next full subtractor in the chain. By combining N full subtractors, it is possible to subtract two

encrypted binary values in $\hat{\mathbb{I}}^N$ as follows:

$$\hat{D}_i = (\hat{A}_i \oplus_\heartsuit \hat{B}_i) \oplus_\heartsuit \widehat{BR}_{i-1}$$

$$\widehat{BR}_i = (\neg_\heartsuit \hat{A}_i \bullet_\heartsuit \hat{B}_i) \underline{+}_\heartsuit (\neg_\heartsuit \hat{A}_i \bullet_\heartsuit \widehat{BR}_{i-1}) \underline{+}_\heartsuit (\hat{B}_i \bullet_\heartsuit \widehat{BR}_{i-1})$$

where $\oplus_\heartsuit$ is the TFHE *XOR* gate, $\neg_\heartsuit$ is the TFHE *NOT* gate, $\bullet_\heartsuit$ is the TFHE *AND* gate, and $\underline{+}_\heartsuit$ is the TFHE *OR* gate.

A.2 Encrypted Shifts

The encrypted left shift $<<_\heartsuit$,

$$\hat{a} <<_\heartsuit n \text{ with } \hat{a} \in \hat{\mathbb{I}}^N \text{ and } n \in \mathbb{N},$$

is defined as $<<_\heartsuit \colon \{\hat{\mathbb{I}}^N, \mathbb{N}\} \rightarrow \hat{\mathbb{I}}^N$. Exactly as with plain values, the $<<_\heartsuit$ operator shifts the bits of $\hat{a}$ to the left by n positions, inserting n encrypted zeros $\hat{0}$ into the least significant bits of $\hat{a}$. For example, $\hat{a} <<_\heartsuit 2$ becomes $\hat{a} = [\hat{A}_{N-3} \ \dots \ \hat{A}_0 \ \hat{0} \ \hat{0}]$. Symmetrically, the encrypted right shift, defined as $>>_\heartsuit \colon \{\hat{\mathbb{I}}^N, \mathbb{N}\} \rightarrow \hat{\mathbb{I}}^N$, shifts the bits of $\hat{a}$ to the right by n positions. Since integer values are represented in two's complement notation, the $>>_\heartsuit$ operator inserts n copies of $\hat{A}_{N-1}$ (i.e., the previous most significant bit) into the most significant bits of $\hat{a}$. For example, $\hat{a} >>_\heartsuit 2$ becomes $\hat{a} = [\hat{A}_{N-1} \ \hat{A}_{N-1} \ \hat{A}_{N-1} \ \dots \ \hat{A}_2]$.

A.3 Encrypted Max

Let $\hat{x} = [\hat{a}, \hat{b}, \hat{c}, \dots]$ a vector of encrypted values in $\hat{\mathbb{I}}^N$ of length M. The encrypted function $\max_\heartsuit(\cdot)$,

$$\max{}_\heartsuit(\hat{x}) = \hat{\alpha} \text{ with } \hat{x} \in \{\hat{\mathbb{I}}^N\}^M \text{ and } \hat{\alpha} \in \hat{\mathbb{I}}^N,$$

is defined as $\max_\heartsuit(\cdot) : \{\hat{\mathbb{I}}^N\}^M \rightarrow \hat{\mathbb{I}}^N$. To compute the output $\hat{\alpha}$ (i.e., the encrypted maximum value among those of the vector x), the $\max_\heartsuit(\cdot)$ operator relies on the encrypted comparison $<_\heartsuit$ and the encrypted $MUX_\heartsuit$ gate. At the beginning, $\hat{\alpha}$ is initialized with the first element of the array $\hat{x}$ (i.e., $\hat{x}_0 = \hat{a}$). Then, for each element $\hat{x}_m$, with $m = 1, \dots, M-1$, the output $\hat{\alpha}$ is updated as:

$$\hat{\alpha} = MUX_\heartsuit\Big((\hat{\alpha} <_\heartsuit \hat{x}_m), \hat{x}_m, \hat{\alpha}\Big)$$

A.4 Encrypted Argmax

Let $\hat{x} = [\hat{a}, \hat{b}, \hat{c}, \dots]$ a vector of encrypted values in $\hat{\mathbb{I}}^N$ of length M. The encrypted function $\arg\max_\heartsuit(\cdot)$,

$$\underset{m \in [0, M-1]}{\arg\max}{}_\heartsuit(\hat{x}) = \hat{\beta} \text{ with } \hat{x} \in \{\hat{\mathbb{I}}^N\}^M \text{ and } \hat{\beta} \in \hat{\mathbb{I}}^N,$$

is defined as $\arg\max_\heartsuit(\cdot) : \{\hat{\mathbb{I}}^N\}^M \rightarrow \hat{\mathbb{I}}^N$. To compute the output $\hat{\beta}$ (i.e., the encrypted position $\hat{m}$ of the maximum value among those of the vector x), the $\arg\max_\heartsuit(\cdot)$ operator relies on the encrypted $\max_\heartsuit(\cdot)$ function, the encrypted comparison $==_\heartsuit$, and the encrypted $MUX_\heartsuit$ gate. At the beginning, the max value $\hat{\alpha} = \max_\heartsuit(\hat{x})$ is computed, and $\hat{\beta}$ is initialized to an encrypted zero $\hat{0}$. Then, for each element $\hat{x}_m$, with $m = 0, \dots, M-1$, the output $\hat{\beta}$ is updated as:

$$\hat{\beta} = MUX_\heartsuit\Big((\hat{\alpha} ==_\heartsuit \hat{x}_m), \hat{m}, \hat{\beta}\Big)$$