POSEIDON:
Privacy-Preserving Federated Neural Network Learning
Sinem Sav, Apostolos Pyrgelis, Juan R. Troncoso-Pastoriza, David Froelicher, Jean-Philippe Bossuat, Joao Sa Sousa and Jean-Pierre Hubaux

Abstract
In this paper, we address the problem of privacy-preserving training and evaluation of neural networks in an $N$-party, federated learning setting. We propose a novel system,POSEIDON, the first of its kind in the regime of privacy-preserving neural network training, employing multiparty lattice-based cryptography and preserving the confidentiality of the training data, the model, and the evaluation data, under a passive-adversary model and collusions between up to $N - 1$ parties. To efficiently execute the secure backpropagation algorithm for training neural networks, we provide a generic packing approach that enables Single Instruction, Multiple Data (SIMD) operations on encrypted data. We also introduce arbitrary linear transformations within the cryptographic bootstrapping operation, optimizing the costly cryptographic computations over the parties, and we define a constrained optimization problem for choosing the cryptographic parameters. Our experimental results show that POSEIDON achieves accuracy similar to centralized or decentralized non-private approaches and that its computation and communication overhead scales linearly with the number of parties. POSEIDON trains a 3-layer neural network on the MNIST dataset with 784 features and 60K samples distributed among 10 parties in less than 2 hours.

I. INTRODUCTION
In the era of big data and machine learning, neural networks (NNs) are the state-of-the-art models, as they achieve remarkable predictive performance in various domains such as healthcare, finance, and image recognition [11], [75], [104]. However, training an accurate and robust deep learning model requires a large amount of diverse and heterogeneous data [118]. This phenomenon raises the need for data sharing among multiple data owners who wish to collectively train a deep learning model and to extract valuable and generalizable insights from their joint data. Nonetheless, data sharing among entities, such as medical institutions, companies, and organizations, is often not feasible due to the sensitive nature of the data [115], strict privacy regulations [2], [8] or the business competition between them [102]. Therefore, solutions that enable privacy-preserving training of NNs on the data of multiple parties are highly desirable in many domains.

A simple solution for collective training is to outsource the data of multiple parties to a trusted party that is able to train the neural network (NN) model on their behalf and to retain the data and model’s confidentiality, based on established stringent non-disclosure agreements. These confidentiality agreements, however, require a significant amount of time to be prepared by legal and technical teams [70] and are very costly [60]. Furthermore, the trusted party might become a single point of failure, thus both data and model privacy could be compromised by data breaches, hacking, leaks, etc. Hence, technical solutions originating from the cryptographic community aim to replace and emulate the trusted party with a group of computing servers. In particular, to enable privacy-preserving training of NNs, several studies employ multiparty computation (MPC) techniques and operate on the two [81], [28], three [80], [108], [109], or four [27], [93] server models. Such approaches, however, limit the number of parties among which the trust is split, often assume an honest majority among the computing servers, and require parties to communicate (i.e., secret share) their data outside their premises. This might not be acceptable due to the privacy and confidentiality requirements and the strict data protection regulations. Furthermore, the computing servers do not operate on their own data or benefit from the model training; hence, they do not have an incentive to perform the required computations, which increases the possibility of malicious behaviour.

A recently proposed alternative for privacy-preserving training of NNs – without data outsourcing – is federated learning. Instead of bringing the data to the model, the model is brought (via a coordinating server) to the clients, who perform model updates on their local data. The updated models from multiple parties are averaged to obtain the global NN model [73], [65]. Although federated learning retains locally the sensitive input data and eliminates the need for data outsourcing, the model, that might also be sensitive, e.g., due to proprietary reasons, becomes available to the coordinating server, thus placing the latter in a position of power with respect to the remaining parties. Recent research demonstrates that sharing intermediate model updates among the parties or with the server might lead to various privacy attacks, such as extracting parties’ inputs [53], [111], [117] or membership inference [76], [84]. Consequently, several works employ differential privacy to enable privacy-preserving exchanges of intermediate values and to obtain models that are free from adversarial inferences in federated learning settings [66], [99], [74]. Although differentially private techniques partially limit attacks to federated learning, they decrease the utility of the data and the resulting machine learning model. Furthermore, training robust and accurate models requires high privacy budgets, and as such, the level of privacy achieved in practice remains unclear [55]. Therefore, a distributed privacy-preserving deep learning approach requires strong cryptographic protection of the intermediate model updates during the training process, as well as of the final model weights.

Recent cryptographic approaches for private distributed learning, e.g., [116], [42], not only have limited machine learning functionalities, i.e., regularized or generalized linear models, but also employ traditional encryption schemes that make them vulnerable to post-quantum attacks. This should be cautiously considered, as recent advances in quantum computing [7], [46], [85], [103], [114] increase the need for deploying quantum-resilient cryptographic schemes that eliminate potential risks for applications with long-term sensitive data. Froelicher et al. recently proposed SPINDLE [41], a generic approach for the privacy-preserving training of machine learning models in an $N$-party setting that employs multiparty lattice-based cryptography, thus achieving post-quantum security guarantees. However, the authors [41] demonstrate the applicability of their approach only for generalized linear models, and their solution lacks the necessary protocols and functions that can support the training of complex machine learning models, such as NNs.
In this work, we extend the approach of SPINDLE [41] and build POSEIDON, a novel system that enables the training and evaluation of NNs in a distributed setting and provides end-to-end protection of the parties’ training data, the resulting model, and the evaluation data. Using multiparty lattice-based homomorphic encryption [82], POSEIDON enables NN executions with different types of layers, such as fully connected, convolution, and pooling, on a dataset that is distributed among \( N \) parties that need to trust only themselves for the confidentiality of their data and of the resulting model. POSEIDON relies on mini-batch gradient descent and protects, from any party, the intermediate updates of the NN model by maintaining the weights and gradients encrypted throughout the training phase. POSEIDON also enables the resulting encrypted model to be used for privacy-preserving inference on encrypted evaluation data.

We evaluate POSEIDON on several real-world and publicly available datasets and observe that it achieves training accuracy levels on par with centralized or decentralized non-private approaches. Regarding its execution time, we find that POSEIDON trains a 2-layer NN model on a dataset with 23 features and 30,000 samples distributed among 10 parties, in 8.7 minutes. In a similar setting, POSEIDON trains a 3-layer NN with 64 neurons in each hidden-layer on the MNIST dataset [65] with 784 features and 60K samples shared between 10 parties, in 1.4 hours. Finally, our scalability analysis shows that POSEIDON’s computation and communication overhead scales linearly with the number of parties and logarithmically with the number of features or the number of neurons in each layer.

In this work, we make the following contributions:

- We present POSEIDON, a novel system for privacy-preserving, quantum-resistant, federated learning-based training of and inference on neural networks with \( N \) parties with unbounded \( N \); that relies on multiparty homomorphic encryption and respects the confidentiality of the training data, the model, and the evaluation data.
- We propose the alternating packing approach for the efficient use of single instruction, multiple data (SIMD) operations on encrypted data, and we provide a generic protocol for executing neural networks under encryption, depending on the size of the dataset and the structure of the network.
- We improve the distributed bootstrapping protocol of [82] by introducing arbitrary linear transformations for optimizing computationally heavy operations, such as pooling or a large number of consecutive rotations on ciphertexts.
- We formulate a constrained optimization problem for choosing the cryptographic parameters and for balancing the number of costly cryptographic operations required for training and evaluating neural networks in a distributed setting.
- POSEIDON advances the state-of-the-art privacy-preserving solutions for NNs based on MPC [108], [81], [80], [13], [93], [109], by achieving better flexibility, security, and scalability:
  - **Flexibility.** POSEIDON relies on a federated learning approach, thus eliminates the need for communicating the parties’ confidential data outside their premises which might not be always feasible due to privacy regulations [2], [8]. This is in contrast to MPC-based solutions which require parties to distribute their data among several servers, and thus, fall under the cloud outsourcing model.
  - **Security.** POSEIDON splits the trust among multiple parties, and guarantees its data and model confidentiality properties under a passive-adversarial model and collusions between up to \( N-1 \) parties, for unbounded \( N \). On the contrary, MPC-based solutions limit the number of parties among which the trust is split (typically, 2, 3, or 4-servers) and assume an honest majority among them.
  - **Scalability.** POSEIDON’s communication overhead scales linearly with the number of parties, whereas MPC-based solutions scale quadratically.
- Unlike differential privacy-based approaches for federated learning [66], [99], [74], POSEIDON does not degrade the utility of the data, and the impact on the model’s accuracy is negligible.

To the best of our knowledge, POSEIDON is the first system that enables quantum-resilient, distributed learning on neural networks with \( N \) parties in a federated learning setting, and that preserves the privacy of the parties’ confidential data, the intermediate model updates, and the final model weights.

II. RELATED WORK

**Privacy-Preserving Machine Learning (PPML).** Previous PPML works focus exclusively on the training of (generalized) linear models [19], [57], [25], [34], [48], [61], [62]. They rely on centralized solutions where the learning task is securely outsourced to a server, notably using homomorphic encryption (HE) techniques. As such, these works do not solve the problem of privacy-preserving distributed machine learning, where multiple parties collaboratively train a machine learning model on their data. To address the latter, several works propose multi-party computation (MPC) solutions where learning tasks, such as clustering and regression, are distributed among 2 or 3 servers [54], [26], [80], [43], [44], [14], [98], [23], [31]. Although such solutions enable multiple parties to collaboratively train ML models on their data, the trust distribution is limited to the number of computing servers that train the model, and they rely on assumptions such as non-collusion, or an honest majority among the servers. There exist only a few works that extend the distribution of machine learning computations to \( N \)-parties (\( N \geq 4 \)) and that remove the need for outsourcing [33], [16], [42], [41]. For instance, Zheng et al. propose Helen, a system for privacy-preserving learning of linear models that combines HE with MPC techniques [116]. However, the use of the Paillier additive HE scheme [87] makes their system vulnerable to post-quantum attacks. To this end, Froelicher et al. introduce SPINDLE [41], a system that provides support for generalized linear models and security against post-quantum attacks. These works have paved the way for privacy-preserving machine learning computations in the N-party setting, but none of them addresses the challenges associated with the privacy-preserving training of and the inference on neural networks (NNs).

**Privacy-Preserving Inference on Neural Networks.** In this research direction, the majority of works operate on the following setting: a central server holds a trained neural network model and clients communicate their evaluation data to obtain predictions-as-a-service [37], [71], [59]. Their
Neural networks (NNs) are machine learning algorithms that extract complex non-linear relationships between the input and output data. They are used in a wide range of fields such as pattern recognition, data/image analysis, face recognition, forecasting, and data validation in the medicine, banking, finance, marketing, and health industries [11]. Typical NNs are composed of a pipeline of layers where feed-forward and backpropagation steps for linear and non-linear transformations (activations) are applied to the input data iteratively [47]. Each training iteration is composed of one forward pass and one backward pass, and the term epoch refers to processing once all the samples in a dataset.

A. Neural Networks

The training of neural networks in a distributed setting where the parties' intermediate updates and the final model remain under encryption.

Accuracy NN models requires high privacy budgets [94]. As such, it is hard to quantify the level of privacy protection that can be achieved with privacy guarantees [56], [106]. While DP-based learning aims to mitigate inference attacks, it significantly degrades model utility, as training differentially private gradients [66]. McMahan et al. propose differentially private federated learning [74], by employing the moments accountant parameter update stages, and Li et al. design a privacy-preserving federated learning system for medical image analysis where the parties exchange model updates and averages them. Although this approach does not require clients to send their local data to the central server, several works show that the clients' model updates leak information about their local data [91]. To counter this, some works focus on secure aggregation techniques for distributed NNs, based on HE [90], [91] or MPC [24]. Although encrypting the gradient values prevents the leakage of parties' confidential data to the central server, these solutions do not account for potential leakage from the aggregate values themselves. In particular, parties that decrypt the received model before the next iteration are able to infer information about other parties' data from its parameters [53], [76], [84], [117]. Another line of research relies on differential privacy (DP) to enable privacy-preserving federated learning for NNs. Shokri and Shmatikov [99] apply DP to the model before the next iteration are able to infer information about other parties' data from its parameters [53], [76], [84], [117]. Another line of research relies on differential privacy (DP) to enable privacy-preserving federated learning for NNs. Shokri and Shmatikov [99] apply DP to the zero-leakage privacy guarantees [56], [106]. While DP-based learning aims to mitigate inference attacks, it significantly degrades model utility, as training accurate NN models requires high privacy budgets [94]. As such, it is hard to quantify the level of privacy protection that can be achieved with these approaches [55]. To account for these issues, our work employs multiparty homomorphic encryption techniques to achieve zero-leakage training of neural networks in a distributed setting where the parties' intermediate updates and the final model remain under encryption.

III. Preliminaries

We introduce here the background information about NNs as well as the multiparty homomorphic encryption (MHE) scheme on which POSEIDON relies to achieve privacy-preserving training of and inference on NN models in a federated $N$-party setting.

A. Neural Networks

Neural networks (NNs) are machine learning algorithms that extract complex non-linear relationships between the input and output data. They are used in a wide range of fields such as pattern recognition, data/image analysis, face recognition, forecasting, and data validation in the medicine, banking, finance, marketing, and health industries [11]. Typical NNs are composed of a pipeline of layers where feed-forward and backpropagation steps for linear and non-linear transformations (activations) are applied to the input data iteratively [47]. Each training iteration is composed of one forward pass and one backward pass, and the term epoch refers to processing once all the samples in a dataset.
Multilayer perceptrons (MLPs) are fully-connected deep neural network structures which are widely used in the industry, e.g., they constitute 61% of Tensor Processing Units’ workload in Google’s datacenters [53]. MLPs are composed of an input layer, one or more hidden layer(s), as well as an output layer, and each neuron is connected to all the neurons in the following layer. At iteration \( k \), the weights between layers \( j \) and \( j+1 \), are denoted by a matrix \( W_j^k \), whereas the matrix \( L_j \) represents the activation of the neurons in the \( j^{th} \) layer. The forward pass requires first the linear combination of each layer’s weights with the activation values of the previous layer, i.e., \( U_j = W_j^k \times L_{j-1} \). Then, an activation function is applied to calculate the values of each layer as \( L_j = \varphi(U_j) \).

Backpropagation, a method based on gradient descent, is then used to update the weights during the backward pass. Here, we describe the update rules for mini-batch gradient descent where a random batch of sample inputs of size \( B \) is used in each iteration. The aim is to minimize each iteration’s error based on a cost function \( E \) (e.g., mean squared error) and update the weights accordingly. The update rule is \( W_j^{k+1} = W_j^k - \eta \nabla W_j^k \), where \( \eta \) is the learning rate and \( \nabla W_j^k \) denotes the gradient of the cost function with respect to the weights and calculated as \( \nabla W_j^k = \frac{\partial E}{\partial W_j^k} \).

We note that backpropagation requires several transpose operations applied to matrices/vectors and we denote transpose of a matrix/vector as \( W^T \).

Convolutional neural networks (CNNs) follow a very similar sequence of operations, i.e., forward and backpropagation passes, and typically consist of convolutional (CV), pooling, and fully connected (FC) layers. It is worth mentioning that CV layer operations can be expressed as FC layer operations by representing them as matrix multiplications; in our protocols, we simplify CV layer operations by employing this representation [108], [3]. Finally, pooling layers are downsampling layers where a kernel, i.e., a matrix that moves over the input matrix with a stride of \( a \), is convoluted with the current sub-matrix. For a kernel of size \( k \times k \), the minimum, maximum, or average (depending on the pooling type) of each \( k \times k \) sub-matrix of the layer’s input is computed.

### B. Distributed Deep Learning

We employ the well-known MapReduce abstraction to describe the training of data-parallel NNs in a distributed setting where multiple data providers hold their respective datasets [119], [32]. We rely on a variant of the parallel stochastic gradient descent (SGD) [119] algorithm, where each party performs \( b \) local iterations and calculates each layer’s partial gradients. These gradients are aggregated over all parties and the reducer performs the model update with the average of gradients [32]. This process is repeated for \( m \) global iterations. Note that averaging the gradients from \( N \) parties is equivalent to performing batch gradient descent with a batch size of \( b \times N \). Thus, we differentiate between the local batch size as \( b \) and the global batch size \( B = b \times N \).

### C. Multiparty Homomorphic Encryption (MHE)

In our system, we rely on the Cheon-Kim-Kim-Song (CKKS) [29] variant of the MHE scheme proposed by Mouchet et al. [82]. In this scheme, a public collective key is known by all parties while the corresponding secret key is distributed among them. As such, decryption is only possible with the participation of all parties. Our motivations for choosing this scheme are: (i) It is well suited for floating point arithmetic, (ii) it is based on the ring learning with errors (RLWE) problem [72], making our system secure against post-quantum attacks [12], (iii) it enables secure and flexible collaborative computations between parties without sharing their respective secret key, and (iv) it enables a secure collective key-switch functionality, that is, changing the encryption key of a ciphertext without decryption. Here, we provide a brief description of the cryptographics scheme’s functionalities that we use throughout our protocols. The cyclotomic polynomial ring of dimension \( N' \), where \( N' \) is a power-of-two integer, defines the plaintext and ciphertext space as \( R_{Q_L} = \mathbb{Z}_{Q_L}[X]/(X^{N'}+1) \), with \( Q_L = \prod_0^b q_i \) in our case. Each \( q_i \) is a unique prime, and \( Q_L \) is the ciphertext modulus at an initial level \( L \). Note that a plaintext encodes a vector of up to \( N'/2 \) values. Below, we introduce the main functions that we use in our system. We denote by \( c = \{c_0, c_1\} \in \mathbb{Z}_{Q_L}^2 \) and \( p \in \mathbb{Z}_{Q_L} \), a ciphertext (indicated as boldface) and a plaintext, respectively. \( \bar{p} \) denotes an encoded(packed) plaintext. We denote by \( L_{\text{ce}}, S_{\text{ce}}, L, \text{ and } S, \) the current level of a ciphertext \( c \), the current scale of \( c \), the initial level, and the initial scale (precision) of a fresh ciphertext respectively, and we use the equivalent notations for plaintexts. The functions below that start with ‘D’ are distributed, and executed among all the secret-key-holders, whereas the others can be executed locally by anyone with the public key.

- **\text{SecKeyGen}(1^k)\text{ : Returns the set of secret keys } \{sk_i\}, i.e., sk_i for each party \( P_i \), for a security parameter \( \lambda \).**
- **\text{DKeyGen}(sk_i)\text{ : Returns the collective public key } pk.**
- **\text{Encode}(\text{msg})\text{ : Returns a plaintext } \bar{p} \in \mathbb{Z}_{Q_L} \text{ with scale } S, \text{ encoding } \text{msg}.**
- **\text{Decode}(\bar{p})\text{ : For } \bar{p} \in \mathbb{Z}_{Q_L}, \text{ and scale } S_p, \text{ returns the decoding of } p.**
- **\text{DDecrypt}(c, \{sk_i\})\text{ : For } c \in \mathbb{Z}_{Q_L}, \text{ and scale } S_c, \text{ returns the plaintext } p \in \mathbb{Z}_{Q_L}, \text{ with scale } S_c.**
- **\text{Enc}(pk, \bar{p})\text{ : Returns } \bar{c}_{pk} \in \mathbb{Z}_{Q_L}^2 \text{ with scale } S \text{ such that } \text{DDecrypt}(\bar{c}_{pk}, \{sk_i\}) \approx \bar{p}.**
- **\text{Add}(c_{pk}, c'_{pk})\text{ : Returns } (c+c')_{pk} \text{ at level } \min(L_c, L_{c'}) \text{ and scale } \max(S_c, S_{c'}).**
- **\text{Sub}(c_{pk}, c'_{pk})\text{ : Returns } (c-c')_{pk} \text{ at level } \min(L_c, L_{c'}) \text{ and scale } \max(S_c, S_{c'}).**
- **\text{Mul}_{L, L_p}(c_{pk})\text{ : Returns } (cp)_{pk} \text{ at level } \min(L_c, L_{c'}) \text{ with scale } (S_c \times S_p).**
- **\text{RotL}L(c_{pk}, k)\text{ : Homomorphically rotates } c_{pk} \text{ to the left/right by } k \text{ positions}.**
- **\text{Res}(c_{pk})\text{ : Returns } c_{pk} \text{ with scale } S/c_{g_{lm}} \text{ at level } L_c - 1.**
- **\text{SetScale}(c_{pk}, S)\text{ : Returns } c_{pk} \text{ with scale } S \text{ at level } L_c - 1.**
- **\text{KS}(c_{pk} \in R^2)\text{ : Returns } c_{pk} \in R^2.**
- **\text{DKeySwitch}(c_{pk}, pk')\text{ : Returns } c_{pk'}\text{.**}
- **\text{DBootstrap}(c_{pk}, L_c, S, \{sk_i\})\text{ : Returns } c_{pk} \text{ with initial level } L \text{ and scale } S.**
IV. SYSTEM OVERVIEW

We introduce POSEIDON’s system and threat model, as well as its objectives (Sections IV-A and IV-B). Moreover, we provide a high level description of its functionality (Sections IV-C and IV-D).

A. System and Threat Model

We introduce POSEIDON’s system and threat model below.

System Model. We consider a setting where \( N \) parties, each locally holding its own data \( X_i \), and one-hot vector of labels \( y_i \), collectively train a neural network (NN) model. At the end of the training process, a querier – which can be one of the \( N \) parties or an external entity – queries the model and obtains prediction results \( y_i \) on its evaluation data \( X_i \). The parties involved in the training process are interested in preserving the privacy of their local data, the intermediate model updates, and the resulting model. The querier obtains prediction results on the trained model and keeps its evaluation data confidential. We assume that the parties are interconnected and organized in a tree-network structure for communication efficiency, as shown in Figure 1 (thick lines). However, our system is fully distributed and does not assume any hierarchy, therefore remaining agnostic of the network topology, e.g., we can consider a fully-connected network, or a star topology in which each party communicates with a central server (dotted lines in Figure 1).

Threat Model. We consider a passive-adversary model with collusions of up to \( N-1 \) parties; i.e., the parties follow the protocol but up to \( N-1 \) parties might share among them their inputs and observations during the training phase of the protocol, to extract information about the other parties’ inputs through membership inference or federated learning attacks [76], [84], [53], [117], [111], prevented by our work. Inference attacks on the model’s prediction phase, such as membership [100] or model inversion [40], exploit the final prediction result, and are out-of-the-scope of this work. We discuss complementary security mechanisms that can bound the information a querier infers from the prediction results and an extension to the active-adversary model in Appendix I-A.

B. Objectives

POSEIDON’s main objective is to enable the privacy-preserving training of and the evaluation on neural networks in the above system and threat model. During the training process, POSEIDON protects both the intermediate updates and the final model weights — that can potentially leak information about the parties’ input data [53], [76], [84], [117] — from any party. In the inference step, the parties holding the protected model should not learn the querier’s data, or the prediction results, and the querier should not obtain the model’s weights. Therefore, POSEIDON’s objective is to protect the parties’ and querier’s data confidentiality, as well as the trained model confidentiality, as defined below:

- Data Confidentiality. During training and prediction, no party \( P_i \) (including the querier \( P_q \)) should learn more information about the input data \( X_j \) of any other honest party \( P_j \) \((j \neq i, \text{ including the querier } P_q \)) than what can be deduced from its own input data \( X_i, y_i \) (or the input \( X_q \) and output \( y_q \), for the querier).
- Model Confidentiality. During training and prediction, no party \( P_i \) (including the querier \( P_q \)) should gain more information about the trained model weights, other than what can be deduced from its own input data \( X_i, y_i \) (or \( X_q, y_q \) for the querier).
Weight Initialization.

1) PREPARE:

   Outputs: \( W_{1,1}^{n_1}, W_{2,1}^{n_2}, ..., W_{l,1}^{n_l} \)

   PREPARE:
   1: Parties collectively agree on \( \ell, h_1, ..., h_{\ell}, \eta, \varphi(), m, b \)
   2: Each \( P_i \) generates \( sk_i \leftarrow \text{SecKeyGen}(1^* \) \)
   3: Parties collectively generate \( pk \leftarrow \text{DKeyGen}(\{sk_i\}) \)
   4: Each \( P_i \) encodes its local data as \( X_i, y_i \)
   5: \( P_i \) initializes \( W_{1,1}^{n_1}, W_{2,1}^{n_2}, ..., W_{l,1}^{n_l} \)
   6: for \( k = 0 \rightarrow m – 1 \)
      7: \( P_i \) sends \( W_{1,1}^{k}, W_{2,1}^{k}, ..., W_{l,1}^{k} \) down the tree
   8: Each \( P_i \) does:
      9: Local Gradient Descent Computation:
      10: \( \nabla W_{1,1}^{k}, \nabla W_{2,1}^{k}, ..., \nabla W_{l,1}^{k} \)
   COMBINE:
   11: Parties collectively aggregate:
   \( \nabla W_{1,1}^{k}, ..., \nabla W_{l,1}^{k} \leftarrow \sum_{i=1}^{N} \nabla W_{1,1}^{k}, ..., \nabla W_{l,1}^{k} \) \)
   12: \( P_i \) obtains \( \nabla W_{1,1}^{k}, \nabla W_{2,1}^{k}, ..., \nabla W_{l,1}^{k} \)
   REDUCE (performed by \( P_1 \)) :
   13: for \( j = 1 \rightarrow \ell \) do
      14: \( W_{i,j+1}^{k+1} + = \eta \cdot \nabla W_{j,i}^{k} \) \)
   15: end for
   16: end for

C. Overview of Poseidon

Poseidon achieves its objectives by exploiting the MHE scheme described in Section II-C. In particular, the model weights are kept encrypted, with the parties’ collective public key, throughout the training process. The operations required for the communication-efficient training of neural networks are enabled by the scheme’s computation homomorphic properties, which enables the parties to perform operations between their local data and the encrypted model weights. To enable oblivious inference on the trained model, Poseidon utilizes the scheme’s key-switching functionality that allows the parties to collectively re-encrypt the prediction results with the querier’s public key.

Poseidon employs several packing schemes to enable Single Instruction, Multiple Data (SIMD) operations on the weights of various network layers (e.g., fully connected or convolutional ones) and uses approximations that enable the evaluation of multiple activation functions (e.g., Sigmoid, Softmax, ReLU) under encryption. Furthermore, to account for the complex operations required for the forward and backward passes performed during the training of a neural network, Poseidon uses the scheme’s distributed (collective) bootstrapping capability that enables us to refresh ciphertexts. In the following subsection, we provide a high-level description of Poseidon’ phases, the cryptographic operations and optimizations are described in Section V.

We present Poseidon as a synchronous distributed learning protocol throughout the paper. An extension to asynchronous distributed NNs is presented in Appendix I-B.

D. High-Level Protocols

To describe the distributed training of and evaluation on NNs, we employ the extended MapReduce abstraction for privacy-preserving machine learning computations introduced in SPINDLE [41]. The overall learning procedure is composed of four phases: PREPARE, MAP, COMBINE, and REDUCE. Protocol 1 describes the steps required for the federated training of a neural network with \( N \) parties. The bold terms denote encrypted values and \( W_{j,i}^{k} \) represents the weight matrix of the \( j^{th} \) layer, at iteration \( k \), of the party \( P_i \). When there is no ambiguity or we refer to the global model, we replace the sub-index \( i \) with \( \cdot \) and denote weights by \( W_{j}^{k} \). Similarly, we denote the local gradients at party \( P_i \) by \( \nabla W_{j,i}^{k} \) for each network layer \( j \) and iteration \( k \). Throughout the paper, the \( n^{th} \) row of a matrix that belongs to the \( j^{th} \) party is represented by \( X_{j}[n] \) and its encoded (packed) version as \( \tilde{X}_{j}[n] \).

1) PREPARE: In this offline phase, the parties collectively agree on the learning parameters: the number of hidden layers (\( \ell \)), the number of neurons (\( h_j \)) in each layer \( j \), \( j \in [1, 2, ..., \ell] \), the learning rate (\( \eta \)), the number of global iterations (\( m \)), the activation functions to be used in each layer (\( \varphi() \)) and their approximations (see Section V-B), and the local batch size (\( b \)). Then, the parties generate their secret keys \( sk_i \) and collectively generate the public key \( pk \). Subsequently, they collectively normalize or standardize their input data with the secure aggregation protocol described in [22]. Each \( P_i \) encodes (packs) its input data samples \( X_i \) and output labels \( y_i \) (see Section V-A) as \( \tilde{X}_i, \tilde{y}_i \). Finally, the root of the tree (\( P_1 \)) initializes and encrypts the global weights.

Weight Initialization.

To avoid exploding or vanishing gradients, we rely on commonly used techniques: (i) Xavier initialization for the sigmoid or tanh activated layers: \( W_j \leftarrow r \times h_{j-1} \) where \( r \) is a random number sampled from a uniform distribution in the range \([-1, 1]\) [45], and (ii) He initialization [50] for ReLU activated layers, where the Xavier-initialized weights are multiplied twice by their variance.

2) MAP: The root of the tree \( P_1 \) communicates the current encrypted weights, to every other party for their local gradient descent (LGD) computation.
### Protocol 2: Local Gradient Descent (LGD) Computation

**Inputs:** $W^1, W^2, \ldots, W^k$.  
**Outputs:** $\nabla W^1_{k,1}, \nabla W^2_{k,1}, \ldots, \nabla W^k_{k,1}$. Note that $i$ and $k$ indices are omitted in this protocol.

1. for $t = 1 \rightarrow b$ do  
2. $L_0 = \bar{X}[t]$  
3. for $j = 1 \rightarrow k$ do  
4. $U_j = L_{j-1} \times W_j$  
5. $L_j = \varphi(U_j)$  
6. end for  
7. $E_t = \tilde{y}[t] - L_k$  
8. $E_t = \varphi'(U_j) \odot E_t$  
9. $\nabla W_{t+1} = L_{t+1}^T \times E_t$  
10. for $j = t-1 \rightarrow 1$ do  
11. $E_j = E_{j+1} \times W_{j+1}^T$  
12. $E_j = \varphi'(U_j) \odot E_j$  
13. $\nabla W_j = L_j^T \times E_j$  
14. end for  
15. end for

**LGD Computation:** Each $P_i$ performs $b$ forward and backward passes, i.e., to compute and aggregate the local gradients, by processing each sample of its respective batch. Protocol 2 describes the LGD steps performed by each party $P_i$, at iteration $k$; $\odot$ represents an element-wise product and $\varphi'(\cdot)$ the derivative of an activation function. As the protocol refers to one local iteration for a specific party, we omit $k$ and $j$ from the weight and gradient indices. This protocol describes the typical operations for the forward pass and backpropagation using stochastic gradient descent (SGD) with the L2 loss (see Section III). We note that the operations in this protocol are performed over encrypted data.

3) **COMBINE:** In this phase, each party communicates its encrypted local gradients to their parent, and each parent homomorphically sums the received gradients with their own ones. At the end of this phase, the root of the tree ($P_1$) receives the globally aggregated gradients.

4) **REDUCE:** $P_i$ updates the global model weights by using the averaged aggregated gradients. The averaging is done with respect to the global batch size $|B| = b \times N$, as described in Section III-B.

**Training Termination:** In our system, we stop the learning process after a predefined number of epochs. We discuss other well-known techniques for the termination of NN training and how to integrate them in POSEIDON in Appendix I-B.

**Prediction:** At the end of the training phase, the model is kept in an encrypted form such that no individual party or the querier can access the model weights. To enable oblivious inference, the querier encrypts its evaluation data $X_q$ with the parties’ collective key. We note that an oblivious inference is equivalent to one forward pass (see Protocol 2), except that the first plaintext multiplication ($\text{Mul}_a(\cdot)$) of $L_0$ with the first layer weights is substituted with a ciphertext multiplication ($\text{Mul}_a(\cdot)$). At the end of the forward pass, the parties collectively re-encrypt the result with the querier’s public key by using the key-switch functionality of the underlying MHE scheme. Thus, only the querier is able to decrypt the prediction results. Note that any party $P_i$ can perform the oblivious inference step, but the collaboration between all the parties is required to perform the distributed bootstrap and key-switch functionalities.

### V. Cryptographic Operations and Optimizations

We first present the alternating packing (AP) approach that we use for packing the weight matrices of NNs (Section V-A). We then explain how we enable activation functions on encrypted values (Section V-B) and introduce the cryptographic building blocks and functions employed in POSEIDON (Section V-C), together with their execution pipeline and their complexity (Sections V-D and V-E). Finally, we formulate a constrained optimization problem that depends on a cost function for choosing the parameters of the cryptoscheme (Section V-F).

#### A. Alternating Packing (AP) Approach

For the efficient computation of the forward pass and backpropagation described in Protocol 2, we rely on the packing capabilities of the cryptoscheme that enables Single Instruction, Multiple Data (SIMD) operations on ciphertexts. Packing enables coding a vector of values in a ciphertext and to parallelize the computations across its different slots, thus significantly improving the overall performance.

Existing packing strategies that are commonly used for machine learning operations on encrypted data [41], e.g., the row-based [61] or diagonal [49], require a high-number of rotations for the execution of the matrix-matrix multiplications and matrix transpose operations, performed during the forward and backward pass of the local gradient descent computation (see Protocol 2). We here remark that the number of rotations has a significant effect on the overall training time of a neural network on encrypted data, as they require costly key-switch operations (see Section V-E). As an example, the diagonal approach scales linearly with the size of the weight matrices, when it is used for batch-learning of neural networks, due to the matrix transpose operations in the backpropagation. We follow a different packing approach and process each batch sample one by one, making the execution embarrassingly parallelizable. This enables us to optimize the number of rotations, to eliminate the transpose operation applied to matrices in the backpropagation, and to scale logarithmically with the dimension and number of neurons in each layer.

We propose an “alternating packing (AP) approach” that combines row-based and column-based packing, i.e., rows or columns of the matrix are vectorized and packed into one ciphertext. In particular, the weight matrix of every FC layer in the network is packed following the opposite approach from that used to pack the weights of the previous layer. With the AP approach, the number of rotations scales logarithmically with the dimension...
of the matrices, i.e., the number of features ($d$), and the number of hidden neurons in each layer ($h_i$). To enable this, we pad the matrices with zeros to get power-of-two dimensions. In addition, the AP approach reduces the cost of transforming the packing between two consecutive layers.

Protocol 3 describes a generic way for the initialization of encrypted weights for an $\ell$-layer MLP by $P_1$ and for the encoding of the input matrix ($X_i$) and labels ($y_i$) of each party $P_i$. It takes as inputs the NN parameters: the dimension of the data ($d$) that describes the shape of the input layer, the number of hidden neurons in the $j^{th}$ layer ($h_j$), and the number of outputs ($h_\ell$). We denote by $|gap|$ a vector of zeros, and by $\cdot$ the size of a vector or the number of rows of a matrix. $\text{Replicate}(v,k,|gap)$ returns a vector that replicates $v$, $k$ times with a gap in between each replica. $\text{Flatten}(W,|gap,\cdot,dim)$, flattens the rows or columns of a matrix $W$ into a vector and introduces gap in between each row/column. If a vector is given as input to this function, it places gap in between all of its indices. The argument $\cdot$ indicates flattening of rows ('r') or columns ('c') and $\cdot$ indicates 'emb' for the case of vector inputs.

We observe that the rows (or columns) packed into one ciphertext, must be aligned with the rows (or columns) of the following layer for the next layer multiplications in the forward pass and for the alignment of multiplication operations in the backpropagation, as depicted in Table IV (e.g., see steps F1, F6, B3, B5, B6). We enable this alignment by adding gap between rows or columns and using rotations, described in the next section. Note that these steps correspond to the weight initialization and to the input preparation steps of the PREPARE (offline) phase.

**Convolutional Layer Packing.** To optimize the SIMD operations for convolutional (CV) layers, we decompose the $i^{th}$ input sample $X_i[n]$ into $t$ smaller matrices that are going to be convoluted with the weight matrix. We pack these decomposed flattened matrices into one ciphertext, with a gap in between each matrix that is defined with respect to the number of neurons in the next layer, similarly to the AP approach. The weight matrix is then replicated $t$ times with the same gap between each replica. Protocol 5 in Appendix C shows how to pack a CV layer weight matrix and the input data in case of a convolutional layer. If the next layer is another convolution or downsampling layer, the gap is not needed and the values in the slots are rearranged during the training execution (see Section VC).

**Downsampling (Pooling) Layers.** As there is no weight matrix for downsampling layers, they are not included in the offline packing phase. The cryptographic operations for pooling are described in Section V-D.

### B. Approximated Activation Functions

For the encrypted evaluation of non-linear activation functions, such as Sigmoid or Softmax, we use least-squares approximations and rely on the optimized polynomial evaluation that, as described in [41], consumes $\lceil \log(d_a+1) \rceil$ levels for an approximation degree $d_a$. For the piece-wise function ReLU, we approximate the smooth approximation of ReLU, softplus (SmoothReLU), $\phi(x) = \ln(1+e^x)$ with least-squares. Lastly, we use derivatives of the approximated functions. We discuss possible alternatives to these approximations in Appendix C.
To achieve better approximation with the lowest possible degree, we apply two approaches to keep the input range of the activation function as small as possible, by using (i) different weight initialization techniques for different layers (i.e., Xavier or He initialization), and (ii) collective normalization of the data by sharing and collectively aggregating statistics on each party’s local data in a privacy-preserving way [42]. Finally, the interval and the degree of the approximations are chosen based on the heuristics on the data distribution in a privacy-preserving way, as described in [51].

| PREPARE: | Representation |
| --- | --- |
| 1. Each $P_i$ prepares $X_i, y_i$ | Encode $X_i, y_i$ to $\tilde{X}_i, \tilde{y}_i$ |
| 2. $P_1$ initializes $W_1$ | Vectorize columns, pack with $\text{gap} = 0$ $W_1^0 = \text{Flatten}(W_1, \text{gap}, \gamma^*)$ |
| 3. $P_1$ initializes $W_2$ | Vectorize rows, pack with $\text{gap} = d - h_t$ $W_2^0 = \text{Flatten}(W_2, \text{gap}, \gamma^*)$ |
| 4. Each $P_i$ generates masks $\tilde{m}_1, \tilde{m}_2$ | |
| 5. Forward Pass (Each $P_i$): | |
| 6. $U_1 = \tilde{L}_0 \times W_1$, $L_1 = \varphi(U_1)$ | $F_1$. $U_1 = \text{Mul}_p(\tilde{L}_0, W_1)$, $L_1 = \text{Res}(U_1)$ |
| 7. $U_2 = \tilde{L}_1 \times W_2$, $L_2 = \varphi(U_2)$ | $F_2$. $U_2 = \text{RIS}(U_1, 1, d)$ |
| 8. $U_3 = \text{Mul}_d(U_2, m_2)$, $L_3 = \text{Res}(U_3)$ | $F_3$. $U_3 = \text{Mul}_d(U_2, m_2)$, $L_3 = \text{Res}(U_3)$ |
| 9. $U_4 = \text{RR}(U_3, 1, h_t)$ | $F_4$. $U_4 = \text{RR}(U_3, 1, h_t)$ |
| 10. $U_5 = \varphi(U_4)$ | $F_5$. $U_5 = \varphi(U_4)$ |
| 11. Backpropagation (Each $P_i$): | |
| 12. $E_2 = \tilde{y}_i[n] - L_2$ | $B_1$. $E_2 = \text{Sub}(\tilde{y}_i[n], L_2)$ |
| 13. $E_2 = (\varphi'(U_2)) \odot E_2$ | $B_2$. $d = \varphi'(U_2)$ |
| 14. $E_3 = \text{Mul}_d(E_2, d)$, $E_3 = \text{Res}(E_2)$ | $B_3$. $E_3 = \text{Mul}_d(E_2, d)$, $E_3 = \text{Res}(E_2)$ |
| 15. $E_4 = \text{RR}(E_2, d, h_t)$ | $B_4$. $E_4 = \text{RR}(E_2, d, h_t)$ |
| 16. $\nabla W_{2,i} = L_1^T \times E_1$ | $B_5$. $\nabla W_{2,i} = \text{Mul}_d(L_1, E_2)$, $\text{Res}(\nabla W_{2,i})$ |
| 17. $E_6 = E_2 \times W_{2,i}^T$ | $B_6$. $E_6 = \text{Mul}_d(E_2, W_{2,i})$, $\text{Res}(E_1)$ |
| 18. $E_7 = \text{RIS}(E_1, 1, d)$ | $B_7$. $E_7 = \text{RIS}(E_1, 1, d)$ |
| 19. $E_8 = \varphi'(E_7)$ | $B_8$. $d = \varphi'(E_7)$ |
| 20. $E_9 = \text{Mul}_d(d, m_1)$ | $B_9$. $E_9 = \text{Mul}_d(d, m_1)$ |
| 21. $E_{10} = \text{Mul}_d(E_9, d)$, $E_{10} = \text{Res}(E_9)$ | $B_{10}$. $E_{10} = \text{Mul}_d(E_9, d)$, $E_{10} = \text{Res}(E_9)$ |
| 22. $E_{11} = \text{DB}^{\varphi'}(E_{10})$ | $B_{11}$. $E_{11} = \text{DB}^{\varphi'}(E_{10})$ |
| 23. $\nabla W_{1,i} = L_0^T \times E_1$ | $B_{12}$. $E_1 = \text{Mul}_d(E_{11}, m_1)$, $\text{Res}(E_1)$ |
| 24. $E_{13} = \text{RR}(E_1, 1, d)$ | $B_{13}$. $E_{13} = \text{RR}(E_1, 1, d)$ |
| 25. $\nabla W_{1,i} = \text{Mul}_p(L_0, E_2)$, $\text{Res}(\nabla W_{1,i})$ | $B_{14}$. $\nabla W_{1,i} = \text{Mul}_p(L_0, E_2)$, $\text{Res}(\nabla W_{1,i})$ |
| Update (at $P_i$): | |
| 26. $W_{j,i} = \gamma \cdot \text{SetScale}(\nabla W_{j,i}, \gamma)$ | $U_1$. SetScale($\nabla W_{j,i}, \gamma$) |
| 27. $W_{j,i} = \text{Add}(W_{j,i}, \nabla W_{j,i})$ | $U_2$. $W_{j,i} = \text{Add}(W_{j,i}, \nabla W_{j,i})$ |
| 28. $W_{j,i} = \text{DB}^{\varphi'}(W_{j,i})$ | $U_3$. $W_{j,i} = \text{DB}^{\varphi'}(W_{j,i})$ |

**TABLE 1:** Execution pipeline for a 2-layer MLP network with Alternating Packing (AP). Orange steps indicate the operations introduced to DBootstrapALT(·).

**C. Cryptographic Building Blocks**

We present each cryptographic function that we employ to enable the privacy-preserving training of NNs with $N$ parties. We also discuss the optimizations employed to avoid costly transpose operations in the encrypted domain.

**Rotations.** As we rely on packing capabilities, computation of the inner-sum of vector-matrix multiplications and transpose operation implies a restructuring of the vectors, that can only be achieved by applying slot rotations. Throughout the paper, we use two types of rotation functions: (i) Rotate For Inner Sum (RIS($c, p, s$)) is used to compute the inner-sum of a packed vector $c$ by homomorphically rotating it to the left with RotL($c, p$) and by adding it to itself iteratively $\log_2(s)$ times, and (ii) Rotate For Replication (RR($c, p, s$)) replicates the values in the slots of a ciphertext by rotating the ciphertext to the right with RotR($c, p$) and by adding to itself, iteratively $\log_2(s)$ times. For both functions, $p$ is multiplied by two at each iteration, thus both yield $\log_2(s)$ rotations. As rotations are costly cryptographic functions (see Table II), and the matrix
when computing the multiplication of a packed matrix (to be transposed) and a vector: without interaction on a secret-shared plaintext. Despite its properties, the protocol that Mouchet et al. propose for the BFV scheme cannot version of bootstrapping \[82\], as it is several orders of magnitude more efficient than the traditional centralized bootstrapping. Then we modify it, to leverage on the interaction to automatically perform some of the rotations, or pooling operations, embedded as transforms in the bootstrapping. 

Mouchet et al. replace the expensive bootstrap circuit by a one-round protocol where the parties collectively switch a Brakerski/Fan-Vercauteren (BFV) \[39\] ciphertext to secret-shares in \(\mathbb{Z}_q^N\). Since the BFV encoding and decoding algorithms are linear transformations, they can be performed without interaction on a secret-shared plaintext. Despite its properties, the protocol that Mouchet et al. propose for the BFV scheme cannot be directly applied to CKKS, as CKKS is a \textit{leveled} scheme: The re-encryption process extends the residue number system (RNS) basis from \(Q_t\) to \(Q_L\). Modular reduction of the masks in \(Q_t\) will result in an incorrect encryption. Our solution is therefore to to collectively switch the ciphertext to a secret-shared plaintext with statistical indistinguishability.

We define this protocol as \textbf{DBootstrapALT}(\cdot) (Protocol \[4\]) that takes as inputs a ciphertext \(c_{ph}\) at level \(\ell\) encrypting a message \(msg\) and returns a ciphertext \(c'_{ph}\) at level \(L\) encrypting \(\phi(msg)\), where \(\phi(\cdot)\) is a linear transformation over the field of complex numbers. We denote by \(|a|\) the infinity norm of the vector or polynomial \(a\). As the security of the RLWE is based on computational indistinguishability, switching to the secret-shared domain does not hinder security. We refer to Appendix \[A\] and \[B\] for additional technical details and the security proof of our protocol, respectively.

### Optimization of the Vector-Transpose Matrix Product

The backpropagation step of the low gradient computation at each party requires several multiplications of a vector (or matrix) with the transposed vector (or matrix) (see Lines 11-13 of Protocol \[2\]). The naïve multiplication of a vector \(v\) with a transposed weight matrix \(W^T\) that is fully packed in one ciphertext, requires converting \(W\) of size \(g \times k\), from column-packed to row-packed. This is equivalent to applying a permutation of the plaintext slots, that can be expressed with a plaintext matrix \(W_{ph \times ak}\) and homomorphically computed by doing a matrix-vector multiplication. As a result, a naïve multiplication requires \(\sqrt{g \times k}\) rotations followed by \(\log_2(k)\) rotations to obtain the inner sum from the matrix-vector multiplication. We propose several approaches to reduce the number of rotations when computing the multiplication of a packed matrix (to be transposed) and a vector:

- For the mini-batch gradient descent, we do not perform operations on the batch matrix. Instead, we process each batch sample in parallel, because having separate vectors (instead of a matrix that is packed into one ciphertext) enables us to reorder them at a lower cost. This approach translates \(\ell\) matrix transpose operations to be operations in vectors (the transpose of the vectors representing each layer activations in the backpropagation, see Line-13, Protocol \[2\]).
- Instead of taking the transpose of the weight matrix, we replicate the values in the vector that will be multiplied with the transposed matrix (for the operation in Line-11, Protocol \[2\]), leveraging the gaps between slots with the AP approach. That is, for a vector \(v\) of size \(k\) and the column-packed matrix \(W\) of size \(g \times k\), \(v\) has the form \([a,0,0,0,\ldots,b,0,0,0,\ldots,c,0,0,0,\ldots]\) with at least \(k\) zeros in between values (due to Protocol \[3\]). Hence, any resulting ciphertext requiring the transpose of the matrix that will be subsequently multiplied, will also include gaps in between values. We apply RR(\(v,1,k\)) that consumes \(\log_2(k)\) rotations to generate \([a,a,a,\ldots,b,b,b,\ldots,c,c,c,\ldots]\). Finally, we compute the product \(P = Mv_{\ell}(v,W)\) and apply Ris(\(P,1\), \(g\)) to get the inner sum with \(\log_2(g)\) rotations.
- We further optimize the performance by using \textbf{DBootstrapALT}(\cdot) (Protocol \[4\]): If the ciphertext before the multiplication must be bootstrapped, we embed the \(\log_2(k)\) rotations as a linear transformation performed during the bootstrapping.

### D. Execution Pipeline

Table \[D\] depicts the pipeline of the operations for processing one sample in LGD computation for a 2-layer MLP. These steps can be extended to an \(\ell\)-layer MLP by following the same operations for multiple layers. The weights are encoded and encrypted using the AP approach, and the shape of the packed ciphertext for each step is shown in the representation column. Each forward and backward pass on a layer in the pipeline consumes one Rotate For Inner Sum (\textsc{Ris}(\cdot)) and one Rotate For Replication (\textsc{Rr}(\cdot)) operation, except for the last layer, as the labels are prepared according to the shape of the \(\ell^{th}\) layer output. In Table \[D\] we assume that the initial level \(L = 7\). When a bootstrapping function is
We apply when a CV layer is followed by a FC layer, the output of the labels in the offline phase, depending on the NN structure (see Protocol 3). Hence, we save during the key-switching, maximum level, current level, and the approximation degree, respectively.

### Corollary

The distributed bootstrapping takes 1 round of communication and the size of the communication scales with the number of parties ($N$) and the size of the ciphertext (see [82] for details).

| Computational Complexity | #Levels Used | Communication | Rounds |
|--------------------------|--------------|---------------|--------|
| FORWARD P. (FP)          | $(\log_2(h_{i-1}) + \log_2(h_{i+1})) \cdot \text{KS} + \text{Mul}_\beta + \text{Mul}_\varphi$ | $2 + \log_2(d_a + 1)$ | $-$ | $-$ |
| BACKWARD P. (BP)         | $(\log_2(h_{i-1}) + \log_2(h_{i+1})) \cdot \text{KS} + 2\text{Mul}_\beta + \text{Mul}_\varphi$ | $3 + \log_2(d_a)$ | $-$ | $-$ |
| MAP                      | $\ell (\text{FP} + \text{BP}) - 2\log_2(h_i)$ | $\ell (5 + \log_2(d_a + 1) + \log_2(d_a))$ | $z(N - 1)\cdot |1/2|
| COMBINE                  | $-$ | $z(N - 1)\cdot |1/2|
| REDUCE                   | $\ell (\text{Mul}_\beta + \text{DB})$ | $-$ | $-$ |
| DBootstrap (DB)          | $N\log_2(N)(L+1) + N\log_2(N)(L_c+1)$ | $-$ | $(N - 1)\cdot |1|
| Mul Plaintext (Mul$_a$)  | $2N(L_c+1)$ | $1$ | $-$ | $-$ |
| Mul Ciphertext (Mul$_a$) | $4N(L_c+1) + \text{KS}$ | $1$ | $-$ | $-$ |
| Approx. Activation Function ($\varphi$) | $(2^a + m - \kappa - 3 + [(d_a + 1)/2^a]) \cdot \text{Mul}_d$ | $\log_2(d_a + 1)$ | $-$ | $-$ |
| RIS($c,p,s$), RR($c,p,s$) | $\log_2(s) \cdot \text{KS}$ | $-$ | $-$ | $-$ |
| Key-switch (KS)          | $O(N\log_2(N)L_c\beta)$ | $-$ | $-$ | $-$ |

**TABLE II:** Complexity analysis of POSEIDON’s building blocks. $N, \alpha, L, L_c, d_a$ stand for the cyclotomic ring size, the number of secondary moduli used during the key-switching, maximum level, current level, and the approximation degree, respectively. $\beta = \lceil L_c + 1/\alpha \rceil$, $m = \log_2(d_a + 1)$, $\kappa = \lfloor m/2 \rfloor$.

followed by a masking (that is used to eliminate unnecessary values during multiplications) and/or several rotations, we perform these operations embedded as part of the distributed bootstrapping (DBootstrap($\cdot$)) to minimize their computational cost. The steps highlighted in orange are the operations embedded in the DBootstrap($\cdot$). The complexity of each cryptographic function is analyzed in Section V.

### Convolutional Layers

As we flatten, replicate, and pack the kernel in one ciphertext, a CV layer follows the exact same execution pipeline as a FC layer. However, the number of RIS($\cdot$) operations for a CV layer is smaller than for a FC layer. That is because the kernel size is usually smaller than the number of neurons in a FC layer. For a kernel of size $h = f \times f$, the inner sum is calculated by $\log_2(f)$ rotations. Note that when a CV layer is followed by a FC layer, the output of the $i^{th}$ CV layer ($L_i$) already gives the flattened version of the matrix in one ciphertext. We apply RR($L_i, 1, h_{i+1}$) for the preparation of the next layer multiplication. When a CV layer is followed by a pooling layer, however, the RR($\cdot$) operation is not needed, as the pooling layer requires a new arrangement of the slots of $L_i$. We avoid this costly operation by passing $L_i$ to DBootstrap($\cdot$), and by embedding both the pooling and its derivative in DBootstrap($\cdot$).

### Pooling Layers

In POSEIDON, we evaluate our system based on average pooling as it is the most efficient type of pooling that can be evaluated under encryption [87]. To do so, we exploit our modified collective bootstrapping to perform arbitrary linear transformations. Indeed, the average pooling is a linear function and so is its derivative (note that this is not the case for the max pooling). Therefore, in the case of a CV layer followed by a pooling layer, we apply DBootstrap($\cdot$) and use it both to rearrange the slots and to compute the convolution of the average pooling in the forward pass and its derivative, which is used later in the backward pass. For a $h = f \times f$ kernel size, this saves $\log_2(h)$ rotations and additions (RIS($\cdot$)) and one level if masking is needed. For max/min pooling, which are non-linear functions, we refer the reader to Appendix D and highlight that evaluating these functions by using encrypted arithmetic remains impractical due to the need of high-precision approximations.

**E. Complexity Analysis**

Table II displays the communication and worst-case computational complexity of POSEIDON’s building blocks. This includes the MHE primitives, thus facilitating the discussion on the parameter selection in the following section. We define the complexity in terms of key-switch KS($\cdot$) operations and recall that this is a different operation than DKeySwitch($\cdot$), as explained in Section III-C. We note that KS($\cdot$) and DBootstrap($\cdot$) are 2 orders of magnitude slower than an addition operation, rendering the complexity of an addition negligible.

We observe that POSEIDON’s communication complexity depends solely on the number of parties ($N$), the number of total ciphertexts sent in each global iteration ($z$), and the size of one ciphertext ($|c|$). The building blocks that do not require communication are indicated as $-$.

In Table II, forward and backward passes represent the per-layer complexity for FC layers, so they are an overestimate for CV layers. Note that the number of multiplications differs in a forward pass and a backward pass, depending on the packing scheme, e.g., if the current layer is row-packed, it requires 1 less Mul$_\beta$($\cdot$) in the backward pass, and we have 1 less Mul$_\varphi$($\cdot$) in several layers, depending on the masking requirements. Furthermore, the last layer of forward pass and the first layer of backpropagation take 1 less RR($\cdot$) operation that we gain from packing the labels in the offline phase, depending on the NN structure (see Protocol 3). Hence, we save $2\log_2(h_x)$ rotations per one LGD computation.

In the MAP phase, we provide the complexity of the local computations per $P_i$, depending on the total number of layers $\ell$. In the COMBINE phase, each $P_i$ performs an addition for the collective aggregation of the gradients in which the complexity is negligible. To update the weights, REDUCE is done by one party ($P_i$) and divisions do not consume levels when performed with SetScale($\cdot$). The complexity of an activation function ($\varphi(\cdot)$) depends on the approximation degree $d_a$. We note that the derivative of the activation function ($\varphi'(\cdot)$) has the same complexity as $\varphi(\cdot)$ with degree $d_a - 1$.

For the cryptographic primitives represented in Table II, we rely on the CKKS variant of the MHE cryptosystem in [82], and we report the dominating terms. The distributed bootstrapping takes 1 round of communication and the size of the communication scales with the number of parties ($N$) and the size of the ciphertext (see [82] for details).
F. Parameter Selection

We first discuss several details to optimize the number of $\text{Res} (\cdot)$ operations and give a cost function which is computed by the complexities of each functionality presented in Table II. Finally, relying on this cost function we formulate an optimization problem for choosing POSEIDON’s parameters.

As discussed in Section III.C, we assume that each multiplication is followed by a $\text{Res} (\cdot)$ operation. The number of total rescaling operations, however, can be further reduced by checking the scale of the ciphertext. When the initial scale $S$ is chosen such that $Q / S = r$ for a ciphertext modulus $Q$, the ciphertext is rescaled after $r$ consecutive multiplications. This reduces the level consumption and is integrated into our cost function hereinafter.

Cryptographic Parameters Optimization. We define the overall complexity of an $\ell$-layer MLP aiming to formulate a constrained optimization problem for choosing the cryptographic parameters. We first introduce the total number of bootstrapping operations ($B$) required in one forward and backward pass, depending on the multiplicative depth as

$$B = \ell \left[ 5 + \lceil \log_2 (d_a + 1) \rceil + \lceil \log_2 (d_an) \rceil \right],$$

where $r = Q / S$, for a ciphertext modulus $Q$ and an initial scale $S$. The number of total bootstrapping operations is calculated by the total number of consumed levels (numerator), the level requiring a bootstrap $(L - \tau)$ and $r$ which denotes how many consecutive multiplications are allowed before rescaling (denominator). The initial level of a fresh ciphertext $L$ has an effect on the design of the protocols, as the ciphertext should be bootstrapped before the level $L_c$ reaches a number $(L - \tau)$ that is close to zero, where $\tau$ depends on the security parameters. For a cyclotomic ring size $N$, the initial level of a ciphertext $L$, and for the fixed neural network parameters such as the number of layers $\ell$, the number of neurons in each layer $h_1, h_2, \ldots, h_{\ell}$, and for the number of global iterations $m$, the overall complexity is defined as

$$C(N, L) = m \sum_{i=1}^{\ell} \left\{ (2 \log_2 (q_{i-1} + 1) + \log_2 (q_{i+1})) \cdot \text{KS} + 3 \cdot \text{Mul}_{\text{ct}} + 2 \cdot \text{Mul}_{\text{ct}} \cdot \varphi + \varphi' - 2 \log_2 (h_i) + B \cdot \text{DB} \right\}.$$  \hspace{1cm} (1)

Note that the complexity of each $\text{KS}(\cdot)$ operation depends on the level of the ciphertext that it is performed on (see Table II), but we use the initial level $L$ in the cost function for the sake of clarity. The complexity of $\text{Mul}_{\text{ct}}, \text{Mul}_{\text{ct}}, \text{DB}$, and $\text{KS}$ is defined in Table II. Then, the optimization problem for a fixed scale (precision) $S$ and a security level $\lambda$, which defines the security parameters, can be formulated as

$$\min_{N, L} C(N, L) \hspace{1cm} \text{(2)}$$

subject to $mc = \{q_1, \ldots, q_{\ell}\}: L = |mc|Q = \prod_{i=1}^{\ell} q_i: Q = kS, k \in \mathbb{R}^+$;

where $\text{postQsec}(Q, S, \lambda)$ gives the necessary cyclotomic ring size $N$, depending on the ciphertext modulus ($Q$) and on the desired security level ($\lambda$), according to the homomorphic encryption standard whitepaper [16]. Eq. (2) gives the optimal $N$ and $L$ for a given NN structure. We then pack each weight matrix into one ciphertext. It is worth mentioning that the solution might give an $N$ that has fewer slots than the required number to pack the big weight matrices in the neural network. In this case, we use a multi-cipher approach where we pack the weight matrix using more than one ciphertext and do the operations in parallel.

Multi-cipher Approach. In the case of a big weight matrix, we divide the flattened weight vector into multiple ciphertexts and carry out the neural network operations on several ciphertexts in parallel. E.g., for a weight matrix of size $1,024 \times 64$ and $N / 2 = 4,096$ slots, we divide the weight matrix into $1,024 \times 64 / 4,096 = 16$ ciphers.

VI. SECURITY ANALYSIS

We demonstrate that POSEIDON achieves the Data and Model Confidentiality properties defined in Section IV.B under a passive-adversary model with up to $N - 1$ colluding parties. We follow the real/ideal world simulation paradigm [68] for the confidentiality proofs.

The semantic security of the CKKS scheme is based on the hardness of the decisional RLWE problem [29, 72, 69]. The achieved practical bit-security against state-of-the-art attacks can be computed using Albrecht’s LWE-Estimator [16, 17]. The security of the used distributed cryptographic protocols, i.e., $\text{DKeyGen}(\cdot)$ and $\text{DKeySwitch}(\cdot)$, relies on the proofs by Mouchet et al. [82]. They show that these protocols are secure in a passive-adversary model with up to $N - 1$ colluding parties, under the assumption that the underlying RLWE problem is hard [82]. The security of $\text{DBBootstrap}(\cdot)$, and its variant $\text{DBBootstrapALT}(\cdot)$ is based on Lemma I which we state and prove in Appendix I.

Remark 1. Any encryption broadcast to the network in Protocol I is re-randomized to avoid leakage about parties’ confidential data by two consecutive broadcasts. We omit this operation in Protocol I for clarity.

Proposition 1. Assume that POSEIDON’s encryptions are generated using the CKKS cryptosystem with parameters $(N, Q_L, S)$ ensuring a post-quantum security level of $\lambda$. Given a passive adversary corrupting at most $N - 1$ parties, POSEIDON achieves Data and Model Confidentiality during training.

Proof (Sketch). Let us assume a real-world simulator $\mathcal{S}_I$ that simulates the view of a computationally-bounded adversary corrupting $N - 1$ parties, as such having access to the inputs and outputs of $N - 1$ parties. As stated above, any encryption under CKKS with parameters that ensure a post-quantum security level of $\lambda$ is semantically secure. During POSEIDON’s training phase, the model parameters that are exchanged
We use Mininet [78] to evaluate POSEIDON in a virtual network with an average network delay of 0.17ms and 1Gbps bandwidth. All the experiments are performed on 10 Linux servers with Intel Xeon E5-2680 v3 CPUs running at 2.5GHz with 24 threads on 12 cores and 256 GB RAM. Unless otherwise stated, in our default experimental setting, we instantiate POSEIDON with $N=10$ parties. When we run experiments with more than $N=10$ parties, we employ multiple cores on the same 10 Linux machines. As for the parameters of the cryptographic scheme, we fix precision to 32 bits, the number of levels with more than 6 parties, and due to Remark 1. As such, there is no dependency between the random values that an adversary can leverage on. Moreover, the adversary is not able to decrypt the communicated values of an honest party because decryption is only possible with the collaboration of all the parties. Following this, POSEIDON protects the data confidentiality of the honest parties.

Analogously, the same argument follows to prove that POSEIDON protects the confidentiality of the trained model, as it is a function of the parties’ inputs, and its intermediate and final weights are always under encryption. Hence, POSEIDON eliminates federated learning attacks [53], [76], [84], [117], that aim at extracting private information about the parties from the intermediate parameters or the final model.

**Proposition 2.** Assume that POSEIDON’s encryptions are generated using the CKKS cryptosystem with parameters $(N, Q, L, S)$ ensuring a post-quantum security level of $\lambda$. Given a passive adversary corrupting at most $N-1$ parties, POSEIDON achieves Data and Model Confidentiality during prediction.

**Proof (Sketch).** (a) Let us assume a real-world simulator $S_P$ that simulates the view of a computationally-bounded adversary corrupting $N-1$ computing nodes (parties). The Data Confidentiality of the honest parties and Model Confidentiality is ensured following the arguments of Proposition 1 as the prediction protocol is equivalent to a forward-pass performed during a training iteration by a computing party. Following similar arguments to Proposition 1, the encryption of the querier’s input data (with the parties common public key $pk$) can be simulated by $S_P$. The only additional function used in the prediction step is $DKeySwitch(\cdot)$ that is proven to be simulatable by $S_P$ [82]. Thus, POSEIDON ensures Data Confidentiality of the querier. (b) Let us assume a real-world simulator $S_P'$ that simulates a computationally-bounded adversary corrupting $N-2$ parties and the querier. Data Confidentiality of the querier is trivial, as it is controlled by the adversary. The simulator has access to the prediction result as the output of the process for $P_q$, so it can produce all the intermediate (indistinguishable) encryptions that the adversary sees (based on the simulatability of the key-switch/collective decrypt protocol [82]). Following this and the arguments of Proposition 1, Data and Model Confidentiality are ensured during prediction. We remind here that the membership inference [100] and model inversion [40] are out-of-the-scope attacks (see Appendix I-A for complementary security mechanisms against these attacks).

## VII. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate POSEIDON’s performance and present our empirical results. We also compare POSEIDON to other state-of-the-art privacy-preserving federated learning solutions.

### A. Implementation Details

We implement POSEIDON in Go [65] building on top of the Lattigo lattice-based library [77] for the multiparty cryptographic operations. We make use of OneT [4] and build a decentralized system where the parties communicate over TCP with secure channels (TLS).

### B. Experimental Setup

We use Mininet [78] to evaluate POSEIDON in a virtual network with an average network delay of 0.17ms and 1Gbps bandwidth. All the experiments are performed on 10 Linux servers with Intel Xeon E5-2680 v3 CPUs running at 2.5GHz with 24 threads on 12 cores and 256 GB RAM. Unless otherwise stated, in our default experimental setting, we instantiate POSEIDON with $N=10$ parties. When we run experiments with more than $N=10$ parties, we employ multiple cores on the same 10 Linux machines. As for the parameters of the cryptographic scheme, we fix precision to 32 bits, the number of levels $L=6$, and $N=2^{13}$ for the datasets with $d<32$, and $N=2^{14}$ for those with $d>32$, following the multi-cipher approach (see Section V-F).

### C. Datasets

For the evaluation of POSEIDON’s performance, we use the following real-world and publicly available datasets: (a) the Breast Cancer Wisconsin dataset (BCW) [20] with $n=699, d=9, h_f=2$, where the aim is to model the presence of breast cancer as a function of the patients’ input data, (b) the hand-written digits (MNIST) dataset [65] with $n=70,000, d=784, h_f=10$ for modelling hand-written digits, (c) the Epileptic seizure recognition (ESR) dataset [38] with $n=11,500, d=179, h_f=2$ that is used to model seizure, and (d) the default of credit card clients (CREDIT) dataset [12] with $n=30,000, d=23, h_f=2$ where the goal is to model the status of the clients’ default payment. Recall that $h_f$ represents the number of neurons in the last layer of a neural network (NN), i.e., the number of output labels. Moreover, since we pad with zeros each dimension of a weight matrix to the nearest power-of-two (see Section V-A), for the experiments using the CREDIT, ESR, and MNIST datasets, we actually perform the NN training with $d=32$, 256 and 1,024 features, respectively. To evaluate the scalability of our system, we generate synthetic datasets and vary the number of features or samples. Finally, for our experiments we evenly and randomly distribute all the above datasets among the participating parties. We note that the data and label distribution between the parties, and its effects on the model accuracy is orthogonal to this paper (see Appendix I-B for extensions related to the data and label distribution).
TABLE III: POSEIDON’s accuracy and execution times for different settings. The trained model accuracy is compared to several non-private approaches.

| Dataset   | Accuracy C1 | Accuracy C2 | Accuracy L | Accuracy D | POSEIDON Training | POSEIDON Inference |
|-----------|-------------|-------------|------------|-------------|-------------------|-------------------|
| BCW       | 97.8%       | 97.4%       | 93.9%      | 97.4%       | 96.9%             | 91.06             |
| ESR       | 93.6%       | 91.2%       | 89.9%      | 91.1%       | 90.4%             | 851.84            |
| CREDIT    | 81.4%       | 80.9%       | 79.6%      | 80.6%       | 80.2%             | 516.61            |
| MNIST     | 92.1%       | 91.3%       | 87.8%      | 90.6%       | 89.9%             | 5,283.1           |

(a) Increasing number of features \(d\), (b) Increasing number of parties \(N\), (c) Increasing number of parties \(N\) and features \(d\) each having 200 samples \(n\).

Figure 2: POSEIDON’s training execution time and communication overhead with increasing number of parties, features, and samples, for 1 training epoch.

D. Neural Network Configuration

For the BCW, ESR, and CREDIT datasets, we deploy a 2-layer fully connected NN with 64 neurons per layer, and we use the same NN structure for the synthetic datasets used to test POSEIDON’s scalability. For the MNIST dataset, we train a 3-layer fully connected NN with 64 neurons per-layer. We use the approximated sigmoid and/or the approximated SmoothReLU activation functions (see Section V-B), depending on the dataset. We train the above models for 100, 600, 500, and 1,000 global iterations for the BCW, ESR, CREDIT, and MNIST datasets, respectively. Finally, we set the local batch size \(b\) to 10 and, as such, the global batch size is \(B = 100\) in our default setting with 10 parties.

E. Empirical Results

We experimentally evaluate POSEIDON in terms of accuracy of the trained model, execution time for both training and prediction phases, and communication overhead. We also evaluate POSEIDON’s scalability with respect to the number of parties \(N\), as well as the number of data samples \(n\) and features \(d\) in a dataset. For the interested readers, we provide microbenchmark timings and communication overhead for the various functionalities and operations for FC, CV, and pooling layers in Appendix H-A. These can be used to extrapolate POSEIDON’s execution time for different NN structures. We further give per-global-iteration execution times of various NN architectures in Appendix H-C and various CNN architectures in Appendix H-D.

Model Accuracy. Table III displays POSEIDON’s accuracy results on the used real-world datasets. The accuracy column shows four baselines with the following approaches: two approaches where the data is collected to a central party in its clear form: centralized with original activation functions (C1), and centralized with approximated activation functions (C2); one approach where each party trains the model only with its local data (L), and a decentralized approach with approximated activation functions (D), where the data is distributed among the 10 parties, but the learning is performed on cleartext data, i.e., without any protection of the gradients communicated between the parties. For all experiments, we use the same NN structure with the same learning parameters. These baselines enable us to evaluate POSEIDON’s accuracy loss due to the approximation of the activation functions, distribution, encryption, and the impact of privacy-preserving federated learning. We observe that the accuracy loss between C1, C2, D, and POSEIDON is 0.9—3% when 32-bits precision is used. For instance, POSEIDON achieves 90.4% training accuracy on the ESR dataset, a performance that is equivalent to a decentralized (D) non-private approach and only slightly lower compared to centralized approaches. Finally, we compare POSEIDON’s accuracy with that achieved by one party using its local dataset, that is 1/10 of the overall data, with exact activation functions. We observe that even with the accuracy loss due to approximation and encryption, POSEIDON still achieves 1—3% increase in the model accuracy due to privacy-preserving collaboration.

Execution Time. As shown on the right-hand side of Table III, POSEIDON trains the BCW, ESR, and CREDIT datasets in less than 15 minutes and the MNIST in 1.4 hours, when each dataset is evenly distributed among 10 parties. Note that POSEIDON’s overall training time for MNIST is less than an hour when the dataset is split among 20 parties that use the same local batch size. The per-sample inference times presented in Table III include the forward pass, the DKeySwitch(·) operations that re-encrypt the result with the querier’s public key, and the communication among the parties. We note that as all the parties keep the model in encrypted form, any of them can process the prediction query. Hence, taking the advantage of parallel query executions and multi-threading, POSEIDON achieves a throughput of 864,000 predictions per hour on the MNIST dataset with the chosen NN structure.
Scalability. Figure 2a shows the scaling of POSEIDON with the number of features \( d \) when the one-cipher and multi-cipher with parallelization approaches are used for a 2-layer NN with 64 hidden neurons. The runtime refers to one epoch, i.e., a processing of all the data from \( N = 10 \) parties, each having 2,000 samples, and employing a batch size of \( b = 10 \). For small datasets with a number of features between 1 and 64, we observe no difference in execution time between the one-cipher and multi-cipher approaches. This is because the weight matrices between layers fit in one ciphertext with \( N = 2^{13} \). However, we observe a larger runtime of the one-cipher approach when the number of features increases further. This is because each power-of-two increase in the number of features requires an increase in the cryptographic parameters, thus introducing overhead in the arithmetic operations.

We further analyse POSEIDON’s scalability with respect to the number of parties \( (N) \) and the number of total samples in the distributed dataset \((n)\). Figures 2b and 2c display POSEIDON’s execution time, when the number of parties ranges from 3 to 24, and one training epoch is performed, i.e., all the data of the parties is processed once. For Figure 2b, we fix the number of data samples per party to 200 to study the effect of an increasing number of members in the federation. We observe that POSEIDON’s execution time is almost independent of \( N \) and is affected only by increasing communication between the parties. When we fix the global number of samples \((n)\), increasing \( N \) results in a runtime decrease, as the samples are processed by the parties in parallel (see Figure 2c). Then, we evaluate POSEIDON’s runtime with an increasing number of data samples and a fixed number of parties \( N = 10 \), in Figure 2d. We observe that POSEIDON scales linearly with the number of data samples. Finally, we remark that POSEIDON also scales proportionally with the number of layers in the NN structure, if these are all of the same type, i.e., FC, CV, or pooling, and if the number of neurons per layer or the kernel size is fixed (see Appendix I-B).

F. Comparison with Prior Work

A quantitative comparison of our work with the state-of-the-art solutions for privacy-preserving NN executions is a non-trivial task. Indeed, the most recent cryptographic solutions for privacy-preserving machine learning in this setting, i.e., Helen [116] and SPINDLE [41], support the functionalities of only regularized [116] and generalized [41] linear models respectively.

POSEIDON operates in a federated learning setting where the parties keep their data locally. This is a substantially different setting compared to that envisioned by MPC-based solutions [51], [80], [108], [109], [27], [93] for privacy-preserving NN training. In these solutions, the parties’ data has to be communicated (i.e., secret shared) outside their premises, and the data and model confidentiality is preserved as long as there exists an honest majority among a limited number of computing servers (typically, 2 to 4, depending on the setting). Since a quantitative comparison with such solutions is not relevant, we provide a detailed qualitative comparison with these works in Table V, Appendix B.

Federated learning approaches based on differential privacy (DP), e.g., [66], [29], [74], train the NN while introducing some noise to the intermediate values to mitigate adversarial inferences. However, in contrast with POSEIDON, DP-based approaches significantly degrade the utility of the data. Furthermore, training an accurate NN model requires a high privacy budget [24], hence it remains unclear what privacy protection is obtained in practice [55].

Finally, existing HE-based solutions [51], [83], [107] focus only on a centralized setting where the NN learning task is outsourced to a central server. These solutions, however, employ non-realistic cryptographic parameters [107], [83], and their performance is not practical [51] due to their costly homomorphic computations. Our system, focused on a federated learning-based and a multiparty homomorphic encryption scheme, can improve the response time in 3 to 4 orders of magnitude: (a) The execution times produced by Nandakumar et al. [83] for processing one batch of 60 samples in a single thread and 30 threads for a NN structure with \( d = 64 \), \( h_1 = 32 \), \( h_2 = 16 \), \( h_3 = 2 \), are respectively 33,840s and 2,400s. When we evaluate the same setting, but with \( N = 10 \) parties, we observe that POSEIDON processes the same batch in 6.3s and 1s, respectively. We also achieve stronger security guarantees (128 bits) than [83] (80 bits). (b) For a NN structure with 2-hidden layers of 128 neurons each, for the MNIST dataset, CryptoDL [51] processes a batch with \( B = 192 \) in 10,476.29s, whereas our system in the distributed setting processes the same batch in 34.72s.

VIII. CONCLUSION

In this work, we presented POSEIDON, a novel system for zero-leakage privacy-preserving federated neural network learning among \( N \) parties. Based on lattice-based multiparty homomorphic encryption, our system protects the confidentiality of the training data, of the model, and of the evaluation data, under a passive adversary model with collusions of up to \( N - 1 \) parties. By leveraging on packing strategies and an extended distributed bootstrapping functionality, POSEIDON is the first system demonstrating that secure federated learning on neural networks is practical under multiparty homomorphic encryption. Our experimental evaluation shows that POSEIDON significantly improves on the accuracy of individual local training, bringing it on par with centralized and decentralized non-private approaches. Its computation and communication overhead scales linearly with the number of parties that participate in the training, and is between 3 to 4 orders of magnitude faster than equivalent centralized outsourced approaches based on traditional homomorphic encryption. This work opens up the door of practical and secure federated training in passive-adversarial settings. Future work (see Appendix I) involves extensions to other scenarios with active adversaries and further optimizations to the learning process.

REFERENCES

[1] Microsoft SEAL (release 3.3). https://github.com/Microsoft/SEAL
[2] Centers for Medicare & Medicaid Services. The Health Insurance Portability and Accountability Act of 1996 (HIPAA). [http://www.cms.hhs.gov/hipaa/](http://www.cms.hhs.gov/hipaa/)
[3] Convolutional Neural Networks. [https://cs231n.github.io/convolutional-networks/](https://cs231n.github.io/convolutional-networks/) (Accessed: 2020-04-04).
[4] Cethohy network library. [https://github.com/dedis/onet](https://github.com/dedis/onet) (Accessed: 2019-11-10).
[5] Data61. MP-SPDZ - Versatile framework for multi-party computation. [https://github.com/data61/MP-SPDZ](https://github.com/data61/MP-SPDZ)
[6] Go Programming Language. [https://golang.org/](https://golang.org/) (Accessed: 2019-11-10).
T. Graepel, K. Lauter, and M. Naehrig. ML confidential: Machine learning on encrypted data. In I. Goodfellow, Y. Bengio, and A. Courville.

Y. Aono, T. Hayashi, L. Trieu Phong, and L. Wang. Scalable and secure logistic regression via homomorphic encryption. In A. Al Badawi, L. Hoang, C. Mun, K. Laine, and K. Aung. Privft: Private and fast text classification with homomorphic encryption, 08 2019.

J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In S. V. Algesheimer J., Camenisch J.

A. Akavia, H. Shaul, M. Weiss, and Z. Yakhini. Linear regression on packed encrypted data in the two-server model. In A. Acar, H. Aksu, A. S. Uluagac, and M. Conti. A survey on homomorphic encryption schemes: Theory and implementation.

B. Hie, H. Cho, and B. Berger. Realizing private and practical pharmacological collaboration.

E. Hesamifard, H. Takabi, M. Ghasemi, and R. Wright. Privacy-preserving machine learning as a service.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In ACM Conference on Computer and Communications Security (CCS), 2016.

I. O. Abiodun, A. Jantan, A. E. Omolar, K. V. Dada, N. A. Mohamed, and H. Arshad. State-of-the-art in artificial neural network applications: A survey. Helcyon, 4(1):e00938, 2018.

A. Aono, T. Hayashi, L. Trieu Phong, and L. Wang. Scalable and secure logistic regression via homomorphic encryption. In ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAC), 2019.

R. Ostrovsky. Secure two-party k-means clustering. In P. Bunn and R. Ostrovsky. Secure two-party k-means clustering. In Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS ’07, pages 486–497, New York, NY, USA, 2007. ACM.

M. ByáI, H. Chaudhari, A. Patra, and A. Suresh. Flash: Fast and robust framework for privacy-preserving machine learning. Cryptology ePrint Archive, Report 2019/1365, 2019. https://eprint.iacr.org/2019/1365

J. L. Crawford, C. Gentry, S. Halevi, D. Pfatt, and V. Shoup. Doing real work with fhe: The case of logistic regression. In ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAC), 2018.

A. Sleivert, D. Escudero, and M. Keller. Secure evaluation of quantized neural networks. Cryptology ePrint Archive, Report 2019/131, 2019.

J. Dean, G. Corrado, R. Monga, M. Devin, K. Chen, G. Tucker, P. J. Wang, M. Irving, M. Loyd, L. Chen, and B. Barham. Large scale distributed deep networks. In M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In ACM Conference on Computer and Communications Security (CCS), 2016.
Our analysis shows that the latter achieves a better approximation for a degree $d$ than the former. However, for keeping the error bounded throughout the whole interval, but requires a larger degree for a high accuracy approximation.

The squared error over an interval, whereas Chebyshev asymptotically minimizes the maximum error. Hence, Chebyshev is more appropriate and an efficient algorithm to evaluate polynomials in standard or Chebyshev basis. The least-squares is the optimal solution for minimizing the squared error over an interval, whereas Chebyshev asymptotically minimizes the maximum error. Hence, Chebyshev is more appropriate for keeping the error bounded throughout the whole interval, but requires a larger degree for a high accuracy approximation.
TABLE IV: Frequently Used Symbols and Notations.

APPENDIX D

APPROXIMATION OF THE MAX/MIN POOLING AND ITS DERIVATIVE

The challenge of max/min pooling resides in finding the index of the maximum/minimum value in a given vector. For the sake of clarity, we describe the max pooling. Given a vector \( x = (x[0],...,x[n-1]) \) we compute a vector \( y \) such that \( y[i] = 1 \) if \( y[i] = \text{max}(x) \) and \( y[i] = 0 \) otherwise. Once \( y \) is computed, we can also compute the vector \( x_{\text{max}} = x \odot y \) which stores \( \text{max}(x) \) at the index of the maximum and zeros at all other indices. For approximating the max index, we follow a protocol similar to the one given in [30], described below.
which guarantees the statistical indistinguishability of the shares in. This algorithm can be easily generalized to vectors: Given a vector

$$\lambda$$

Proof:

$$|$$

TABLE V: Qualitative comparison of private deep learning frameworks. Conf. stands for confidentiality. A and P stand for active and passive adversarial capabilities, respectively. GC, SS, HE denote garbled-circuits, secret sharing, and homomorphic encryption. Adversarial model* and collision* take into account the servers responsible for the training/inference. 1’ denotes our interpretation as [37], [83], and [51] do not present an adversarial model. NA stands for not applicable.

Given two real values $a, b$, with $0 < a, b < 1$, we observe the following: If $a > b$, then $a - b < a^d - b^d$ for $d > 1$, i.e., with increasing $d$, smaller values converge to zero faster than greater values and the ratio between the maximum value and all other values increases. The process can be repeated to further increase the ratio between $a$ and $b$ but, unless $a = 1$, both values will eventually converge to zero. To avoid this, we add a second step that consists in re-normalizing $a$ and $b$ by computing $a = a/(a+b)$ and $b = b/(a+b)$. Doing so, we ensure that after each iteration, $a + b = 1$ and since $b$ will eventually converge to zero, $a$ will tend towards 1. In the special case where $a = b$, both values will converge to 0.5. This algorithm can be easily generalized to vectors: Given a vector $x = (x[0],...,x[n-1])$, for each iteration compute $x[i] = x[i]^d / \sum_{j=0}^{n-1} x[j]^d$.

Although theoretically possible, this iterative algorithm for max pooling is a costly and time-consuming procedure. Indeed, at each iteration, it requires computing an inverse, which is an expensive operation, especially if a high accuracy is desired. As such, several collective bootstrapping operations are required for each max-pooling layer. For this reason, we suggest using the average pooling, which is much more efficient, e.g., Dowlin et al. [37] show that low-degree approximations of max pooling will converge to a scalar multiple of the mean of $k$ values. Hence, using average pooling is much more efficient in the encrypted domain when the degree of the approximation for max pooling is kept small.

APPENDIX E

TECHNICAL DETAILS OF DISTRIBUTED BOOTSTRAPPING WITH ARBITRARY LINEAR TRANSFORMATIONS (DBOOTSTRAPALT(·))

A linear transformation $\phi(·)$ over a vector of $n$ elements can be described by a $n \times n$ matrix. As evaluating a matrix-vector multiplication requires a number of rotations proportional to the square-root of its non-zero diagonals, this operation becomes prohibitive when the number of non-zero diagonals is large.

Such a linear transformation can be, however, efficiently carried out locally and without interactions on a secret-shared plaintext, as $\phi(msg + M) = \phi(msg) + \phi(M)$ due to the linear characteristic of $\phi(·)$. Moreover, because of the magnitude of $msg + M$ (100 to 200 bits), arbitrary precision complex arithmetic with sufficient precision should be used for $\text{Encode}(·)$, $\text{Decode}(·)$, and $\phi(·)$ to preserve the lower bits. The collective bootstrapping protocol in [82] performs the bootstrapping through a conversion of an encryption to secret shared values and a re-encryption in a refreshed ciphertext. We can leverage this conversion to perform the aforementioned linear transformation in the secret-shared domain, before the refreshed ciphertext is reconstructed. This is what we call our $\text{DBootstrapALT}(·)$ protocol (Protocol 4).

When the linear transformation is simple, i.e., it does not involve a complex permutation or has only a small number of rotations, the $\text{Encode}(·)$ and $\text{Decode}(·)$ operations in Line 8, Protocol 4 can be skipped. Indeed, those two operations need to be carried out using arbitrary precision complex arithmetic. In such cases, it is more efficient to perform the linear transformation directly on the encoded plaintext.

APPENDIX F

SECURITY ANALYSIS OF DISTRIBUTED BOOTSTRAPPING WITH ARBITRARY LINEAR TRANSFORMATIONS (DBOOTSTRAPALT(·))

The protocol $\text{DBootstrapALT}(·)$ is a modification of the protocol $\text{DBootstrap}(·)$ of Mouchet et al. [82], with the difference that it includes a product of a public matrix. Both $\text{DBootstrap}(·)$ and $\text{DBootstrapALT}(·)$ for CKKS differ from the BFV version proposed in [82] in which the shares are not unconditionally hiding, but statistically or computationally hiding due to the incomplete support of the used masks. Therefore, the proof follows analogously the passive adversary security proof of the BFV $\text{DBootstrap}(·)$ protocol in [82], with the addition of Lemma 1 which guarantees the statistical indistinguishability of the shares in C. While the RLWE problem and Lemma 1 do not rely on the same security assumptions, the first one being computational and the second one being statistical, given the same security parameter, they share the same security bounds. Hence $\text{DBootstrap}(·)$ and $\text{DBootstrapALT}(·)$ provide the same security as the original protocol of Mouchet et al. [82].

Lemma 1. Given the distribution $P_0 = (a+b)$ and $P_1 = c$ with $0 \leq a < 2^b$ and $0 \leq b, c < 2^{\lambda + \delta}$ and $b, c$ uniform, then the distributions $P_0$ and $P_1$ are $\lambda$-indistinguishable; i.e., a probabilistic polynomial adversary $A$ cannot distinguish between both with probability greater than $2^{-\lambda}$: $|\Pr[A \rightarrow 1|P = P_1] - \Pr[A \rightarrow 1|P = P_0]| \leq 2^{-\lambda}$.

Proof: We refer to Algesheimer et. al. [18], Section 3.2 and Schoenmakers and Tuyls [97], Appendix A, for the proof of the statistical $\lambda$-indistinguishability.
We recall that an encoded message $msg$ of $N/2$ complex numbers with the CKKS scheme is an integer polynomial of $\mathbb{Z}[X]/(X^N + 1)$. Given that $||msg|| < 2^d$, and a second polynomial $M$ of $N$ integer coefficients with each coefficient uniformly sampled and bounded by $2^{\lambda+\delta} - 1$ for a security parameter $\lambda$, Lemma 1 suggests that $\Pr||msg^{(i)} + M^{(i)}|| \geq 2^{\lambda+\delta} \approx 2^{-\lambda}$, for $0 \leq i < N$ and where $i$ denotes the $i^{th}$ coefficient of the polynomial. That is, the probability of a coefficient of $msg + M$ to be distinguished from a uniformly sampled integer in $[0, 2^{\lambda+\delta})$ is bounded by $2^{-\lambda}$. Hence, during Protocol 4 each party samples its polynomial mask $M$ with uniform coefficients in $[0, 2^{\lambda+\delta})$. The parties, however, should have an estimate of the magnitude of $msg$ to derive $\delta$, and a probabilistic upper-bound for the magnitude can be computed by the circuit and the expected range of its inputs.

In Protocol 4, the masks $M_i$ are added to the ciphertext of $R_Q$, during the decryption to the secret-shared domain. To avoid a modular reduction of the masks in $R_Q$, and ensure a correct re-encryption in $R_Q$, the modulus $Q_i$ should be large enough for the additions of $N$ masks. Therefore, the ciphertext modulus size should be greater than $(N+1) \cdot ||M||$ when the bootstrapping is called. For example, for $N = 10$, a $Q_i$ composed of a 60 bits modulus, a message $msg$ with $||msg|| < 2^{55}$ (taking the scaling factor $\Delta$ into account) and $\lambda = 128$, we should have $||M_i|| \geq 2^{183}$ and $Q_i > 11 \cdot 2^{183}$. Hence, the bootstrap should be called at $Q_2$ because $Q_2 \approx 2^{180}$ and $Q_3 \approx 2^{240}$. Although the aforementioned details suggest that $DBootstrap(\cdot)$ is equivalent to a depth 3 to 4 circuit, depending on the parameters, it is still compelling, as it enables us to refresh a ciphertext and apply an arbitrary complex linear transformation at the same time. Thus, its cost remains negligible compared to a centralized bootstrapping where any transformation is applied via rotations.

### APPENDIX G

**CONVOLUTIONAL LAYER OFFLINE PACKING**

This work introduces the protocols and algorithms for the training and prediction for feed-forward NNs for the sake of clarity. The system however supports convolutional layers by using the packing described in this section.

Protocol 5 describes the offline packing of one CV layer and the input data $X$ when the first layer is a CV layer. It takes $X$, the initial weight matrix of the first CV layer, the kernel size $h_1 = f \times f$, the stride $s$, and the number of neurons in the next layer ($h_2$). We denote by Type($i$) a function that returns the type of the $i^{th}$ layer as FC, CV, or pooling, whereas Decompose($X, h = f \times f, s$) decomposes the matrix $X$ into $t$ small matrices according to a kernel size $h$, and the stride $s$. The functions Flatten() and Replicate() are defined in Section V-A. The packing for all CV layers of a network is done the same way as described in steps 11-13 of Protocol 5, and for all CV layers of a network is done the same way as described in steps 11-13 of Protocol 5, and

### PROTOCOL 5: CONVOLUTIONAL LAYER PACKING

**Inputs:** $X, W^{(0)}_i, n, h_1, h_2, s$

**Outputs:** $W^{(0)}_i$

1. if Type([2]) == FC & $h_2 > h_1$ then
2. Initialize $|gap| = h_2 - h_1$
3. end if
4. for $t = 1 \rightarrow n$ do
5. $D_0, ..., D_t \leftarrow$ Decompose($X[t], h_1 = f \times f, s$)
6. for $j = 1 \rightarrow t$ do
7. $v \leftarrow \text{Flatten}(D_j, gap, 'r')$
8. end for
9. $W^{(0)}_i[t] = \text{Encode}(pk, v, X)$
10. end for
11. $W_1 = \text{Flatten}(W_1, gap, 'r')$
12. $vW_1 = \text{Replicate}(W_1, i, gap)$
13. $W^{(0)}_i = \text{Enc}(pk, vW_1)$

### APPENDIX H

**SUPPLEMENTARY EXPERIMENTAL RESULTS**

We provide further experimental results of POSEIDON, that were left out of the main text due to space constraints. We provide the microbenchmarks and execution times of various NN architectures.

### A. Microbenchmarks

We present microbenchmark timings for the various functionalities and sub-protocols of POSEIDON in Table VII. These are measured in an experimental setting with $N = 10$ parties, a dimension of $d = 32$ features, $h = 64$ neurons in a layer or kernel size $k = 3 \times 3$, and degree $d_a = 3$ for the approximated activation functions for FC, CV, and average pooling benchmarks. The communication column shows the overall communication between the parties in MB. As several HE-based solutions [37, 59, 51] use square activation functions, we also benchmark them and compare them with the approximated activation functions with $d_a = 3$. 

21
We note that PREPARE stands for the offline phase and it incorporates the collective generation of the encryption, decryption, evaluation, and rotation keys based on the protocols presented in [82]. Most of the time and bandwidth are consumed by the generation of the rotation keys needed for the training protocol.

### Table VI: Execution times per-global-iteration of various NN architectures with batch size \( B = 120 \), \( N = 10 \) parties. MAP:FF, MAP:BP, Comm. stand for MAP:feed-forward, MAP: backpropagation, and communication respectively.

| Topology | MAP:FF (s) | MAP:BP (s) | REDUCE (s) | Comm. (s) | Total (s) |
|----------|------------|------------|------------|-----------|-----------|
| (6, 1, 1, 2) | 0.40 | 0.36 | 0.05 | 0.47 | 1.28 |
| (6, 2, 2, 2) | 0.44 | 0.43 | 0.04 | 0.52 | 1.43 |
| (16, 2, 2, 8) | 0.48 | 0.42 | 0.03 | 0.54 | 1.47 |
| (16, 4, 4, 8) | 0.47 | 0.45 | 0.04 | 0.51 | 1.47 |
| (32, 8, 8, 8) | 0.57 | 0.50 | 0.04 | 0.45 | 1.56 |
| (32, 16, 16, 8) | 0.55 | 0.52 | 0.03 | 0.47 | 1.57 |
| (64, 8, 8, 8) | 0.55 | 0.50 | 0.04 | 0.45 | 1.54 |
| (64, 32, 32, 8) | 0.55 | 0.62 | 0.04 | 0.43 | 1.64 |
| (128, 32, 32, 8) | 0.60 | 0.63 | 0.04 | 0.38 | 1.65 |
| (128, 64, 64, 8) | 0.78 | 0.80 | 0.05 | 0.56 | 2.19 |
| (256, 64, 64, 8) | 1.04 | 1.36 | 0.06 | 0.38 | 2.84 |
| (256, 128, 128, 8) | 2.01 | 2.62 | 0.11 | 0.61 | 5.35 |

**TABLE VII:** Microbenchmarks of different functionalities for \( N = 10 \) parties, \( d = 32, h = 64, N = 2^{13}, d_a = 3 \).

| Functionality | Execution time (s) | Comm. (MB) |
|---------------|-------------------|------------|
| ASigmoid/ASmoothRelu | 0.050 | - |
| ASigmoidD/ASmoothReluD | 0.022 | - |
| Square | 0.01 | - |
| ASoftmax | 0.07 | - |
| SquareD | 0.006 | - |
| DBootstrap | 0.09 | 6.5 |
| DBootstrapALT (\( \log_2(h) \) rots) | 0.18 | 6.5 |
| DBootstrapALT with Average Pool | 0.33 | 6.5 |
| FC layer | 0.09 | - |
| CV layer | 0.03 | - |
| DKeySwitch | 0.07 | 23.13 |
| PREPARE (offline) | 18.19 | 3.8k |
| MAP (only communication) | 0.03 | 18.35 |
| COMBINE | 0.09 | 7.8 |
| REDUCE | 0.08 | 13 |

### B. Benchmarks on Various Neural Network Topologies

We provide execution times of different network topologies in Table VII. “Topology” represents number of features \( d \), hidden neurons in each layer \( h_1, h_2 \), and number of output labels \( h_3 \) as \((d,h_1,h_2,h_3)\). We use local batch size \( b = 12 \) and global batch size \( B = 120 \) for \( N = 10 \) parties. We use ASigmoid with \( d_a = 3 \) as an activation function. The execution times indicate the time required for one global iteration, i.e., a processing of the global batch, and we report forward pass, backpropagation and the number of communications in separate columns. The “Communication” column includes the communication required for the COMBINE phase and for DBootstrap(\() \) operations. We provide the feed-forward and backpropagation times in MAP separately.
Active Adversaries: when there is between Availability, Data Distribution, and Asynchronous Distributed Neural Networks. As such, PQ As. Security Extensions C. Benchmarks on Various Convolutional Neural Network Topologies We provide extrapolated execution times of different CNN topologies in Table VIII. As we introduce several operations (derivative of pooling) in the forward pass to bootstrapping function, we do not separate between forward pass and backpropagation times, and we introduce the overall execution times. “Topology” represents the padded (power-of-two) number of features (d), kernel size for CV layer (CV[n x n]), kernel size for average pooling layer (P[n x n]), and h number of neurons in the last FC layer connected to hℓ output layers (FC[h: hℓ]) as (d, CV[n x n], P[n x n], FC[h: hℓ]).

APPENDIX I Extensions We introduce here several security, learning, and optimization extensions that can be integrated to POSEIDON.

A. Security Extensions We provide several security extensions that can be integrated to POSEIDON as a future work. Active Adversaries: POSEIDON preserves the privacy of the parties under a passive-adversary model with up to N − 1 colluding parties, motivated by the cooperative federated learning scenario presented in Sections I and IV-A. If applied to other different scenarios, our work could be extended to an active-adversarial setting by using standard verifiable computation techniques, e.g., resorting to zero-knowledge proofs and redundant computation. This would, though, come at the cost of an increase in the computational complexity, that will be analyzed as future work. Out-of-the-Scope Attacks: We briefly discuss here out-of-the-scope attacks and countermeasures. By maintaining the intermediate values of the learning process and the final model weights under encryption, during the training process, we protect data and model confidentiality. As such, POSEIDON protects against federated learning attacks [84], [76], [53], [117], [111]. Nonetheless, there exist inference attacks that target the outputs of the model’s predictions, e.g., membership inference [100], model inversion [40], or model stealing [105]. Such attacks can be mitigated via complementary countermeasures that can be easily integrated to POSEIDON: (i) limiting the number of prediction queries for the queriers, and (ii) adding noise to the prediction’s output to achieve differential privacy guarantees. The choice of the differential privacy parameters in this setting remains an interesting open problem.

B. Learning Extensions Early Stop. There are several techniques proposed for the early stopping of the training of a neural network. They also prevent over-fitting as described and evaluated by Prechelt [92]. These approaches are: (i) GLα: stop when the generalization loss exceeds a certain threshold α, (ii) PQα: stop when the quotient of generalization loss and progress exceeds a certain threshold α, and (iii) UPα: stop when the generalization error increased in s successive strips. The generalization error is estimated by the error on a validation set. We note that these methods can be seamlessly integrated into POSEIDON by dividing each party’s data into training and validation sets. Depending on the threshold and the method, the privacy-preserving implementation would require the homomorphic aggregation of the generalization error evaluated on each Pℓ’s validation set and a collective decryption of the error, after a number of global iterations t. As the error is the averaged scalar value, the leakage from the loss remains negligible when there are sufficient validation samples. Availability, Data Distribution, and Asynchronous Distributed Neural Networks. In this work, we rely on a multiparty cryptographic scheme that assumes that the parties are always available. We here note that POSEIDON can support asynchronous distributed neural network training [36] without waiting for all parties to send the local gradients. As such, a time threshold could be used for updating the global model. However, we note that the collective cryptographic protocols require that all the parties be available (e.g., DBootstrap(·) and DBootstrapALT(·)). Changing POSEIDON’s distributed bootstrapping with a centralized one that achieves a practical security level would require increasing the size of the ciphertexts and result in a huge computation and communication overhead. For the evaluation of POSEIDON, we evenly distribute the dataset across the parties; we consider the effects of uneven distributions or the asynchronous SGD to the model accuracy — which are studied in the literature [67], [58], [110] — orthogonal to this work. However, a preliminary analysis with the MNIST dataset and the NN structure defined in our evaluation (see Section VII) shows that asynchronous learning decreases the model accuracy between 1 and 4% when we assume that a server is down with a failure probability between 0.4 and 0.8, i.e., when there is between 40 and 80% chance of not receiving the local gradients from a server in a global iteration. Finally, we find that the uneven distribution of the MNIST dataset for N = 10 parties with one party holding 90% of the data results to a 6% decrease in the model accuracy.

| Topology                     | Execution Time (s) |
|------------------------------|--------------------|
| (256, CV[2 x 2], P[2 x 2], FC[16 : 2]) | 1.42               |
| (512, CV[2 x 2], P[2 x 2], FC[16 : 2]) | 1.52               |
| (512, CV[2 x 2], P[2 x 2], FC[32 : 2]) | 1.88               |
| (784, CV[2 x 2], P[2 x 2], FC[32 : 2]) | 2.12               |
| (784, CV[2 x 2], P[2 x 2], FC[32 : 10]) | 2.56               |
| (784, CV[2 x 2], P[2 x 2], CV[2 x 2], P[2 x 2], FC[32 : 10]) | 3.88               |

TABLE VIII: Execution times per-global-iteration of various CNN architectures, with batch size B = 120, N = 10 parties.
C. Optimization Extensions

Optimizations for Convolutional Neural Networks. We present a scheme for applying the convolutions on the slots, similar to FC layers, by representing them with a matrix multiplication. Convolution on a matrix, however, can be performed with a simple polynomial multiplication by using the coefficients of the polynomial. This operation requires a Fast-Fourier Transform (FFT) from slots (Number Theoretic Transform (NTT)) to coefficients domain, and vice versa (inverseFFT) for switching between CV to pooling or FC layers. Although it achieves better performance for CV layers, domain-switching is expensive. In the case of multiple CV layers before an FC layer, this operation could be embedded into the distributed bootstrapping (DBootstrapALT(·)) for efficiency. The evaluation of the trade-off between the two solutions for larger matrix dimensions is an interesting direction for future work.

Graphics Processing Units (GPUs). In this work, we evaluate our system on CPUs. Using GPUs to improve POSEIDON’s performance requires GPU-compatible cryptographic functions, i.e., extending the underlying cryptographic library Lattigo [77]. In a recent work, Badawi et al. [15] proposed the first GPU implementation of the full RNS-variant of the CKKS scheme, for which they report speedups of one to two orders of magnitude over a CPU implementation. Hence, GPU-accelerated FHE is an option that could greatly improve the practicality of POSEIDON.