Privacy-preserving machine learning inference and training with Homomorphic Encryption

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Giacomo Mosca

Advisor: Prof. Manuel Roveri

Co-advisor: Alessandro Falcetta

Academic year: 2021-2022

1. Introduction

Machine Learning (ML) techniques have proved to be extremely powerful tools capable of autonomously extracting and learning information from large sets of data, with applications ranging from data mining to medicine and finance. ML tools have become even more accessible thanks to *Cloud computing* and *Machine-Learning-as-a-Service (MLaaS)* solutions, which offer powerful scalable environments for deep learning models at manageable costs.
Outsourcing ML computations to third-party providers however raises important privacy concerns when sensitive data needs to be elaborated. Novel *privacy-preserving machine learning* techniques have recently emerged to address this issue, in particular making use of **Homomorphic Encryption** (HE) schemes to perform calculations on encrypted data without ever accessing its contents [1].
The **PINPOINT** family of models [3] offers a successful application of homomorphic techniques to a privacy-preserving deep learning model for *time series prediction*. PINPOINT can obtain forecasts on encrypted private data with comparable accuracy to other state-of-the-art privacy-violating solutions. However, because of the limitations imposed by HE schemes the model is only able to perform inference on encrypted data.
Our research presents the new privacy-preserving **PINStack** model for time series prediction, which extends the previous PINPOINT architecture by implementing a privacy-preserving *training* algorithm. PINStack uses the *Cheon-Kim-Kim-Song (CKKS)* homomorphic encryption scheme [2] to successfully train the parameters of a single fully connected layer on encrypted data without breaching its privacy. The performance of the model is tested under settings that model real use-case scenarios and compared with other state-of-the-art time series prediction solutions executed on plain data.

2. Background

Homomorphic Encryption (HE) is a type of encryption scheme that supports the computation of specific operations directly over encrypted data without any knowledge of the encrypted information [1]. Under some assumptions, the result of a homomorphic operation between encrypted values or ciphertexts, when decrypted, will be equal to the one between the corresponding plain values or plaintexts. Most practical implementations of HE schemes fall under the category of *Leveled Fully Homomorphic Encryption*, which enables calculations on ciphertexts with the following restrictions:

– Only homomorphic additions and multiplications are supported.
– Only a set number of operations (specifically multiplications) can be performed on the same ciphertext before its information is lost. Indeed, a *noise* term is embedded into each ciphertext during encryption to make the scheme secure to decryption, but this noise grows with each homomorphic operation until it overwrites the original encrypted value.
– Homomorphic operations come at the cost of important computational overheads.

The **Cheon-Kim-Kim-Song (CKKS)** homomorphic encryption scheme [2] is chosen for our model due to its native support of floating point values. CKKS is a leveled FHE scheme based on the Ring Learning With Errors (RLWE) problem designed for approximate arithmetic on vectors of complex numbers. The scheme operates on the plaintext space of the polynomial ring $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ and offers operations to homomorphically evaluate additions, multiplications and rotations over ciphertexts. A Rescale operation is used to manage the noise term between multiplications, which in turn determines a maximum multiplicative depth for each ciphertext. The plaintext polynomials of the scheme can encode up to $N/2$ plain values, allowing the encryption of vectors and the application of Single Instruction, Multiple Data (SIMD) operations through *batching*.

Our research aims to apply the privacy-preserving properties of the CKKS scheme to a **time series prediction** task. Given a time series $X = (X_1, \dots, X_n)$ being an ordered set of real value observations, a prediction task involves finding a predictor $\hat{Y} = (\hat{X}_{n+1}, \dots, \hat{X}_{n+h})$ for the upcoming values of the series up to a forecast horizon h. Multiple machine and deep learning models have had great success in this field. In particular, **Temporal Convolutional Neural Networks (TCNNs)** approach the time series forecasting task through the use of convolutional layers, which apply a convolution operation to an input sequence with a set of filters or kernels. This operation is used to return information about the relationship between neighboring inputs so that each output of a convolutional layer is a function of a sequence of previous inputs. The convolutional and fully connected layers of a TCNN operate using only additions and multiplications, making this architecture a prime candidate for implementation with homomorphic encryption. Other non-linear layers commonly used by TCNNs are not compatible with the limitations of HE schemes but can be approximated with HE-compliant versions, such as non-linear activation functions being replaced by square layers.

The **Privacy-preservINg temPoral cOnvolutIonal Neural neTworks (PINPOINT)** family of models [3] is a collection of privacy-preserving time series forecasting models inspired by TCNNs. These networks operate on time series data encrypted with the BFV homomorphic encryption scheme and are specifically designed for privacy-preserving deep learning on the Cloud. PINPOINT models are built from a sequence of one or more convolutional blocks followed by fully connected layers that output the final prediction. Although effective at making accurate predictions, the main drawback with this solution is that models are limited to privacy-preserving inference and cannot learn from encrypted data. Instead, PINPOINT networks are first trained on plain publicly available data and then encoded to support homomorphic calculations. In addition, the BFV homomorphic scheme natively supports the encryption of integer values only, while time series data often comes in the form of floating point values.

3. Proposed solution

The goal of our work is to develop a privacy-preserving framework for machine learning that suports both encrypted inference and encrypted training. We also prove its effectiveness by developing a working model for univariate time series prediction that pushes the limits of these encrypted training capabilities within the constraints given by homomorphic encryption.

Our starting point is the PINPOINT family of models, which is first reworked to implement the *CKKS* encryption scheme and use more efficient *batched vectorized operations*. The model is adapted to take as input a batched ciphertext encrypting multiple sequences of observations, and returns a batched encryption of all the corresponding predictions. For each layer in the PINPOINT architecture, we present a forward pass algorithm that can operate on batched inputs and consumes only one level of multiplication. In particular, fully connected layers and convolutional layers implement an algorithm for efficient homomorphic matrix multiplication, making use of diagonal order matrix encoding to reduce the number of operations required. Appropriate solutions for square and flatten layers are also developed.

Given this optimized framework, a new homomorphic training procedure can be introduced. Full training of a deep learning model is effectively impossible under HE settings because of the extremely limited number of operations allowed on ciphertexts. Instead, we focus on implementing a backward step solely for a single *fully connected layer* and maximizing its efficiency in terms of multiplication usage. A homomorphic version of the *batch gradient descent* optimization algorithm is developed to optimize the weights and bias of a fully connected layer given its prediction error on a set of training data. In addition, a momentum term in the form of *Nesterov's Accelerated Gradient (NAG)* is introduced. Momentum has proved to speed-up convergence of gradient descent and help it navigate through local optima, with NAG offering a good trade-off in terms of effectiveness and compatibility with homomorphic environments. The resulting gradient descent backward pass algorithm only has a multiplicative depth of one and is shown to be effective even in low-epoch training environments.

The proposed implementations of HE-enabled neural network layers are combined in our trainable privacy-preserving architecture for time series prediction, called **PINStack** (Fig. 1). PINStack makes use of the limited training allowed on its single final fully connected layer to fine-tune a model previously trained on plain data through a *transfer learning* approach. In particular, a *model stacking* architecture is developed to leverage plain transfer data coming from multiple sources, which can even be mostly unrelated to the encrypted data specific to the task. A PINStack model is composed of n *level-0 models* $\varphi_i^{(0)}$ built with the previous PINPOINT architecture, each trained on a separate plain

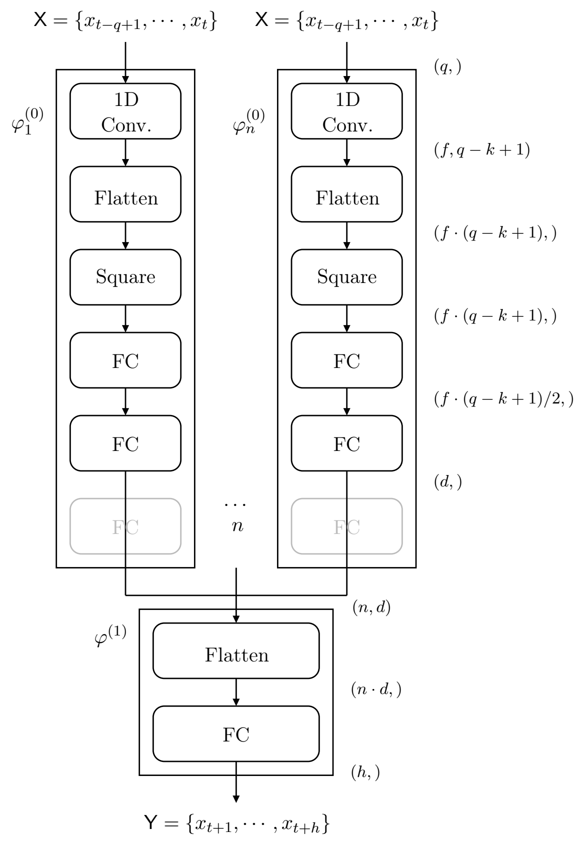


Figure 1: Diagram of the PINStack model. An example of a forward pass is reported with the size of the input and output of each layer.

transfer dataset D_i. Given a specific learning task on an encrypted time series dataset Z composed of input-prediction sequence pairs, the level-0 models are encoded and used to preprocess the encrypted data of Z. Their combined outputs are then fed into a single *level-1 model* or *generalizer* $\varphi^{(1)}$, composed of a flatten layer and a trainable fully connected layer. The level-0 models $\varphi_i^{(0)}$ act as feature extractors for the encrypted dataset Z, each identifying patterns similar to those of their corresponding transfer dataset D_i; the level-1 model $\varphi^{(1)}$ then learns how to best combine these new features to produce the final predictions. Using this approach, users of the model are not required to provide any plain data in addition to their private training dataset, as publicly available datasets can be used for transfer learning instead. The generalizer $\varphi^{(1)}$ is initialized to output an average of the predictions of the original level-0 models, which acts as a good unbiased starting point for the following fine-tuning and greatly speeds up training.

dataset	*range*	*P.P.*	PINStack *P.V.*		*P.V.*			
		PINStack	*transfer*	*training*	PINPOINT	Naive	ARIMA	Prophet
France	1435.904	83.09	126.89	71.50	63.50	95.21	89.18	84.87
Luxembourg	66.577	3.378	3.356	3.376	4.018	4.522	4.019	4.752
Norway	620.595	16.31	16.76	15.27	15.77	18.36	17.07	18.31
Portugal	75.975	6.912	6.878	6.931	6.228	10.332	8.405	7.729

(a) Similar transfer data.

dataset	*range*	*P.P.*	PINStack *P.V.*		*P.V.*			
		PINStack	*transfer*	*training*	PINPOINT	Naive	ARIMA	Prophet
Airline Passengers	518	34.44	57.83	24.29	35.47	66.67	54.66	34.08
Beer Production	153	12.14	16.46	11.42	8.49	38.00	20.31	9.64
India Monthly Rainfall	13352.7	1160.7	1590.7	1059.3	938.5	2440.3	2241.5	830.6
Lake Superior Water Level	1.19	0.0857	0.1067	0.0834	0.0719	0.0988	0.1180	0.2439
Monthly Sunspots	253.8	34.74	40.16	34.29	22.40	31.70	24.23	27.15
New Home Sales	107	5.618	4.786	5.427	5.015	12.354	6.077	26.639
Sales For Retail	439784	18340.4	45702.0	22361.6	10785.2	44849.3	27872.5	12422.0
Total Energy Consumption	4.2286	0.2585	0.3288	0.2495	0.2159	0.6525	0.4510	0.2498
Unemployment Rate	9.0	0.3541	0.4606	0.3582	0.2430	0.3625	0.4842	2.4809

(b) Unrelated transfer data.

Table 1: Prediction MAE of PINStack in the two experiments, compared with other time series prediction models. The PINStack model is run in privacy-preserving (*P.P.*) mode, while the other state-of-the-art references are run in privacy-violating (*P.V.*) mode. PINStack *P.V.* results are reference points obtained on plain data.

4. Experimental results

The proposed PINStack time series prediction model is implemented in the Python library **PyCNN-CKKS**. The library is built on the Pyfhel [4] implementation of the CKKS scheme and offers a framework for a general-purpose encrypted network with privacy-preserving training capabilities. The performance of PINStack using the PyCNN-CKKS library is tested in two scenarios that exemplify two different realistic use cases:

1. *Similar transfer data.* A user wants to train a PINStack model on an encrypted dataset Z and gives it access to other similar plain datasets D_i as well for transfer learning.
2. *Unrelated transfer data.* A user wants to train a PINStack model on an encrypted dataset Z and only has access to sensitive data.

The first experiment is run on a collection of six publicly available datasets coming from similar sources (namely datasets of the daily overall power consumption of several European countries), two of which are used as transfer datasets while four are encrypted for training. The second experiment instead uses nine unrelated public datasets that only share a monthly time resolution. Each dataset is selected once for encrypted training using the remaining eight datasets for transfer learning. In both experiments, plain transfer data is first used to individually train privacy-violating versions of the level-0 models through the PyTorch library. Their weights are encoded to create a PINStack model which is trained on the encrypted dataset for 9 epochs, using all multiplications allowed under maximized CKKS settings. The performance of the PINStack model is evaluated on the accuracy of its predictions made on encrypted data and compared to other privacy-violating state-

	P.P.		*P.V.*
	similar	unrelated	average
level-0 forward pass	231.35	210.0s	0.17s
level-1 forward pass	26.18	74.8s	0.06s
level-1 backward pass	27.85	118.1s	0.76s

Table 2: Amortized computational time of the execution of different parts of the PINStack model on a single input ciphertext. Results from the two experiments on the real privacy-preserving model and a privacy-violating version run on plain data are compared.

of-the-art time series prediction models (Tab. 1). The Mean Absolute Error (MAE) of the predictions is the error metric of choice.
Results show that the performance of the model on encrypted data is at the very least comparable with (and often better than) the other privacy-violating models, both with and without having access to similar transfer learning data. In particular, PINStack convincingly and consistently produces accurate forecasts despite the approximation errors introduced by the CKKS encryption scheme and the overall unoptimized hyperparameters necessary to work in a generalized setting. The transfer learning process results in an effective initialization for the training procedure, which is proved to significantly reduce the prediction error in almost all cases.
The time and memory requirements for the homomorphic computations of the model are quite significant. A full execution of the training and inference algorithms occupies between 130GB and 210GB of memory across both experiments. The computational times of each section of the execution (Tab. 2) lead to a full training runtime of 6.5 to 46 hours for the first experiment and 13 hours for the second. However, these results act as significant improvements over previous privacy-preserving deep learning solutions, and PINStack is intended to be deployed in a Cloud-based setting where the resources are perfectly capable of satisfying its memory and computational demands.

5. Conclusions

In this work a privacy-preserving solution for time series forecasting is proposed, which is capable of performing both inference and training on encrypted time series data using a homomorphic encryption scheme. Implementations of the main layers of a TCNN compatible with such schemes are presented, together with a gradient descent algorithm that operates on fully connected layers under homomorphic restrictions. The resulting PINStack model combines the features of the previous PINPOINT architecture with a transfer learning approach, optimizes its operations for the CKKS encryption scheme and allows training of its final fully connected layer for fine-tuning on encrypted datasets.
The capabilities of the architecture and its applications to a Cloud-based MLaaS scenario are particularly promising. The proposed experiments prove that PINStack is successful at making accurate predictions without breaching the privacy of a user's data, both with and without having access to plain data related to the task. The research acts as a proof-of-concept to show that training in such low-epoch environments can be effective with the right underlying model, and further developments can push this approach further to a proper implementation for real-world usage.
Several improvements can still be made to the underlying PyCNN-CKKS library to increase efficiency and performance, with optimizations for multi-processor and GPU architectures having the potential to significantly reduce computational times. Another area of interest largely left unattended by our research is that of parameter optimization. More studies can be made into optimizing each model to the specific transfer datasets employed, or designing models with a particular combination of transfer datasets and hyperparameters that are optimized for particular scenarios and data distributions. In addition, in the case where similar plain data is accessible, more classical hyperparameter optimization techniques can be used to better fit the task at hand. Finally, the homomorphic layers and stacking architecture proposed in our library can be adapted to work in different deep learning scenarios or extended to encompass more complex models. Applications to generic classification and regressions tasks using a curated collection of simple machine learning models require further research but look entirely feasible.

References

[1] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (Csur)*, 51(4):1–35, 2018.

[2] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 409–437. Springer, 2017.

[3] Alessandro Falcetta and Manuel Roveri. Privacy-preserving time series prediction with temporal convolutional neural networks. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2022.

[4] Alberto Ibarrondo and Alexander Viand. Pyfhel: Python for homomorphic encryption libraries. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 11–16, 2021.

Privacy-preserving machine learning inference and training with Homomorphic Encryption

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Giacomo Mosca, 10574012

Advisor:
Prof. Manuel Roveri

Co-advisors:
Alessandro Falcetta

Academic year:
2021-2022

Abstract: In a world defined by Deep Learning and big data, Cloud-based computing infrastructures have become a necessary tool to meet the increasing computaional demands of machine learning tasks and give any user access to high performance, scalable and affordable solutions. However, this approach involves processing large amounts of data on a third-party platform, which leads to severe privacy concerns when dealing with sensitive data such as medical and financial records. Privacy-preserving machine learning techniques offer a solution to these issues through the use of Homomorphic Encryption (HE) schemes, but present novel challenges in the way these privacy-preserving networks need to be designed. Previous work in the form of the PINPOINT family of deep learning models has already shown promising results on privacy-preserving time series prediction tasks with potential real-world applications. The aim of this thesis is to extend this work to include a novel training procedure which makes it possible to fine-tune a network directly on encrypted sensisitve data without breaching its privacy. The resulting PINStack model stacking architecture is presented as a general-purpose solution for time series forecasting in a privacy-preserving environment both for inference and training, using the Cheon-Kim-Kim-Song (CKKS) homomorphic scheme to guarantee the privacy of the data. Its performance is evaluated in realistic use-case scenarios and shows great potential for future implementation and further developments.

Key-words: Privacy-preserving machine learning, time series forecasting, Homomorphic Encryption, encrypted training

Contents

1. Introduction

1.1. Problem and motivation

Machine Learning (ML) techniques have proved to be extremely powerful tools capable of autonomously extracting and learning information from large sets of data. Their applications range from image [45] and speech recognition [27] to autonomous driving [23], financial risk management [29] and important fields of medicine such as skin cancer identification [30] and genome analysis [44]. With the advent of Deep Learning frameworks [69] machine learning models have only become more and more complex, requiring in turn more and more powerful computational infrastructures. *Cloud computing* [67] has emerged in recent years as a widely available solution to these issues, providing large computational power on demand at a pay-per-use rate, but outsourcing computation to a third-party provider implies giving them access to the necessary data as well. This can be a major privacy concern when sensitive data is treated (e.g. medical diagnoses, financial records, or personal pictures) and potentially prevent essential services from having access to this powerful technology.
A novel solution to this relevant problem comes from privacy-preserving machine learning techniques [12], of which we focus on the ones based on **Homomorphic Encryption (HE)** [10]. HE schemes support the

computation of some operations directly on encrypted data without ever needing access to the original plain information. By employing such a scheme a non-trustworthy third party can receive data encrypted from a user, process these ciphertexts without knowing their contents and produce a final encrypted result to be sent back to the user, fully ensuring the privacy of the data.
Previous work in the form of the **PINPOINT** family of models [31] has shown that HE schemes can be successfully applied to a privacy-preserving learning task, specifically with a focus on *time series prediction.* The PINPOINT architecture is designed to be deployed in a Cloud-based "as-a-service" setting and obtains forecasting results comparable with other state-of-the-art privacy-violating solutions. However, the model comes with some major drawbacks and specifically lacks the ability to perform training on encrypted data, instead relying on additional plain data to construct it.

1.2. Goal and results

The goal of this research is to extend the original work of Falcetta et al. by developing a privacy-preserving machine learning model for inference and training on a time series prediction task.
In particular, the model is designed to process time series sequences encrypted through homomorphic encryption and perform an effective training procedure without breaching the privacy of the training data. This training process needs to be carried out under the harsh limitations introduced by homomorphic schemes, namely:

- HE schemes only support the homomorphic computation of addition and multiplications, requiring that all parts of the training algorithm only use these two operations.
- HE schemes allow only a limited number of operations (and specifically multiplications) to be performed on the same ciphertext before the encrypted information is lost.
- Homorphic encryption introduces important overheads both in terms of computational time and memory usage which need to be managed for an efficient implementation.

This thesis addresses these issues by proposing a new privacy-preserving model called **PINStack**. PINStack is built on the *Cheon-Kim-Kim-Song (CKKS)* homomorphic encryption scheme [24] and uses a model stacking approach based on the original PINPOINT architecture. The model offers a novel implementation of batch gradient descent for the homomorphic training of a single fully connected layer on encrypted inputs. A trainable fully connected layer is used in conjunction with multiple PINPOINT learners trained on plain transfer data, which act as feature extractors for the time series inputs. PINStack also proposes new implementations of all layers used by the original PINPOINT architecture, greatly optimized for operations on vectorized inputs.
The effectiveness of the PINStack architecture is successfully tested on multiple publicly available time series datasets under settings that model real use-case scenarios. Its performance on privacy-preserving encrypted data is shown to be on par with state-of-the-art time series prediction models run on plain data.

1.3. Thesis structure

Section 2 introduces important concepts necessary for the understanding of the work, regarding in particular Machine Learning, Homomorphic Encryption and time series forecasting. *Section 3* discusses the current state-of-the-art for privacy-preserving learning on time series data, specifically focusing on the novel aspects tackled by our research. In addition, this section gives a formal introduction to the PINPOINT family of models that this work is based on. *Section 4* describes in detail the proposed PINStack model, addressing the changes made to the previous PINPOINT implementation and the creation of the new stacking architecture for transfer learning. *Section 5* details the Python library created to implement this solution and its usage. *Section 6* presents a series of experiments that show the efficacy of the PINStack model in realistic time series prediction scenarios. Finally, *Section 7* draws the conclusions and highlights possible directions for future works.

2. Background

2.1. Privacy-preserving machine learning

Machine Learning (ML) [42] is a branch of artificial intelligence that concerns algorithms capable of automatically improving through experience without being explicitly programmed. The goal of ML is to generalize information gathered from data, being a set of inputs with distinct and quantifiable features, and apply it to new scenarios to compute a desired output.
A ML algorithm typically involves two phases. During a *training* phase, the algorithm operates on a dataset of training inputs to learn its representation and update its parameters based on the observations made. The process is repeated over the entire training dataset for a number of iterations called epochs. Then, during an *inference* phase, the algorithm is given new, unseen data and tasked with producing a corresponding output from the information acquired in the training phase.
The way this training phase is performed divides ML algorithms into three main categories (although some additional and hybrid approaches do exist):

- *Supervised learning* associates each input from the training dataset with one or more labels or target values. The objective of the algorithm is to learn the representation of the target values based on the input features and correctly label a set of unlabelled data during inference.
- *Unsupervised learning* offers no labels for the training values, and instead tasks the algorithm with finding a particular structure in the data, such as similarities or specific patterns.
- *Reinforcement learning* has the algorithm learn to perform a set of actions, and rewards or punishes the algorithm based on the outcome of said actions.

Supervised learning tasks are typically divided further into *classification*, where data labels belong to a finite set of classes; *regression*, where data labels are continuous values; and *prediction*, where the model learns to predict a set of upcoming values from a series of observations.

Deep Learning and **Deep Neural Networks (DNNs)** [69] in particular take inspiration from the way information is transmitted and processed by the brain. DNNs are built on interconnected processing units called neurons, further organized into multiple *layers*. Each layer of a neural network takes a set of inputs and extracts a more abstract, higher-level representation of its core features, to then feed these new inputs into the next layer of the network. The word "deep" in deep learning indeed refers to the number of layers that compose a single network, with deeper DNNs being able to extract better features from the inputs at the cost of higher computational complexity. Layers can take multiple forms according to the operations carried out by their neurons and can be combined together to perform complex computations. The basic building blocks are *fully connected layers*, where each neuron computes a weighted sum of all outputs of the previous layer.
DNNs can themselves be categorized into *feed-forward* networks, where inputs are processed sequentially by each layer until an output is returned, and *recurrent* networks, where signals coming from a layer can be fed back into previous layers for further processing.

Privacy-preserving machine learning [12] refers to a sub-category of ML algorithms aimed at preserving the privacy of the processed data. These algorithms are born out of necessity from the rise of Cloud computing and the advent of **Machine-Learning-as-a-Service (MLaaS)** paradigms [59]. In MLaaS, an expensive machine learning procedure is outsourced for computation to a Cloud service, which offers resources far greater than everyday-use machines at a fraction of the cost that would be necessary to purchase such computational power. The Cloud provider can offer platforms already optimized for specific ML tasks and highly scalable environments according to a user's needs.
However, outsourcing computation to a Cloud service typically requires users to send their data to a potentially untrusted third-party, as it needs to have full access to it in order to perform the machine learning task. Privacy-preserving machine learning addresses these privacy concerns by implementing algorithms that do not require explicit access to the necessary input data in order to compute a corresponding output. This is mainly achieved through the use of *cryptography*, which obfuscates data and makes it unreadable to any party who does not have access to the specific key used to encrypt it.

2.2. Homomorphic encryption

Homomorphic Encryption (HE) is a type of encryption scheme that supports the computation of specific operations directly over encrypted data without any knowledge of the encrypted information [10]. In particular, an encryption scheme with an encryption function $\mathsf{Encrypt}$ and decryption function $\mathsf{Decrypt}$ is *homomorphic* with respect to a class of functions $\mathcal{F}$ if, for any $f \in \mathcal{F}$, a function g can be constructed such that:

$$f(x) = \mathsf{Decrypt}(g(\mathsf{Encrypt}(x))$$

where x is any set of possible inputs [16]. As an example, an *additively* homomorphic encryption scheme supports an operation $\oplus$ such that, given two input messages m_1 and m_2, $\mathsf{Encrypt}(m_1 + m2)$ can be computed as $\mathsf{Encrypt}(m_1) \oplus \mathsf{Encrypt}(m2)$ without ever needing to access the unencrypted data. The homomorphic properties of the scheme guarantee that (under some assumptions) the result of the operation between encrypted values, when decrypted, will be equal to the one between corresponding plain values.
HE schemes build upon classical asymmetric encryption schemes (although it is also possible to transform them into symmetric schemes and viceversa [61]) by introducing the HE-specific Eval operation on top of the usual KeyGen, $\mathsf{Encrypt}$, and $\mathsf{Decrypt}$ of conventional cryptography [13]:

- $\mathsf{KeyGen}(\Theta)$ (*key generation*) takes as input a set of encryption parameters Θ and returns a key pair (pk, sk), where pk is the public key used for encryption and sk is the secret key used for decryption.
- $\mathsf{Encrypt}(pk, m)$ (*encryption*) takes as input the public key pk and a plaintext message m and returns the corresponing ciphertext c.
- $\mathsf{Decrypt}(sk, c)$ (*decryption*) takes as input the secret key sk and a ciphertext c and returns the original plaintext message m.
- $\mathsf{Eval}(f, C)$ (*evaluation*) takes as input a supported function f and a set of ciphertexts C that encrypt the plaintexts M, and returns an evaluation ciphertext c' such thath $\mathsf{Decrypt}(sk, c') = f(M)$.

In practice the Eval operation only ever needs to support *addition* and *multiplication* in order to allow for the homomorphic evaluation of arbitray functions, as addition and multiplication are functionally complete sets over finite sets: any boolean circuit can be implemented using only AND (addition) and XOR (multiplication) gates.

First theorized by Rivest et al. [60], it is only the breakthrough work of Gentry [34] and the advent of **Fully Homomorphic Encryption** (FHE) that opened the way for applications of homomorphic encryption to machine learning. FHE schemes as theorized allow for unlimited types of homomorphic operations (namely any combination of additions and multiplications) to be performed on ciphertexts an unlimited number of times. However, their practical implementation raises some complications.
Gentry's and all following FHE schemes inject *noise* into ciphertexts during encryption in order to make the scheme robust to decryption attempts. The introduction of noise enables the property of probabilistic encryption [36] and guarantees that multiple encryptions of the same plaintext will produce different ciphertexts. Typically this noise term is small enough that the original message can be fully recovered after a proper decryption, however the application of homomorphic additions and multiplications to ciphertexts results in the addition or multiplication of their noise as well. With enough operations the noise will grow to the point where the original message will be overwritten and decryption will fail. The number of operations allowed on a ciphertext before a failure is a function of the parameters chosen for the HE scheme and the type of operations used. In particular, homomorphic addition makes the noise grow linearly while homomorphic multiplication makes it grow exponentially.
FHE schemes overcome this issue by implementing a *bootstrapping* operation, which can refresh a noisy ciphertext without decrypting it. Bootstrapping creates a new ciphertext from an existing one that encrypts the same message but with a lower noise, by executing the decryption circuit in the homomorphic domain. The technique as proposed by Gentry however is extremely costly, and while faster bootstrapping procedures have been designed [25] this makes true FHE schemes still too expensive for real-world applications.

A more practical approach is taken by **Leveled Fully Homomorphic Encryption** which instead skips bootstrapping entirely. Leveled FHE schemes still support any combination of homomorphic additions and multiplications but limit the number of operations allowed. Several efficient HE schemes that fall under this category have been developed in the last decade, such as the BFV [19, 32], BGV [20] and CKKS schemes [24]. These schemes are the main focus for current applications of HE functionalities to complex scenarios such as deep learning. Indeed, a machine learning model can operate entirely on encrypted data by replacing its operations with the homomorphic counterparts of a leveled FHE scheme, assuming that all operations necessary for the learner are supported by the encryption scheme.

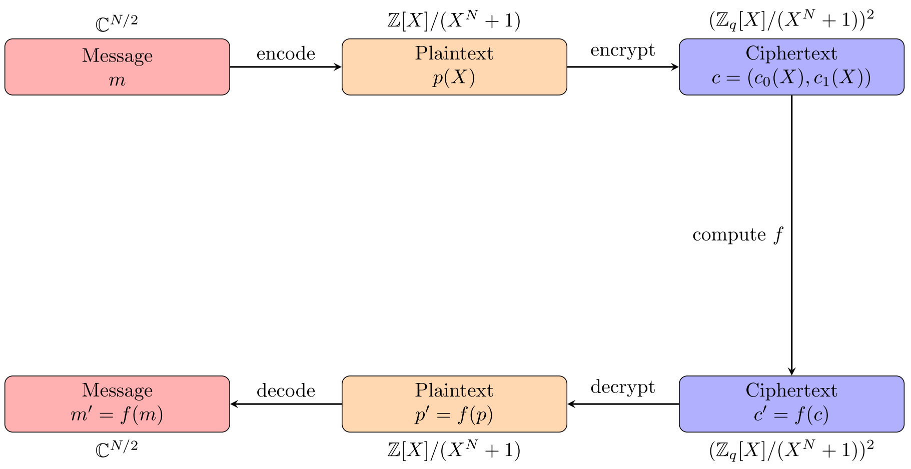


Figure 1: High level view of the CKKS scheme [40].

2.3. CKKS

The **Cheon-Kim-Kim-Song (CKKS)** scheme [24] is a leveled FHE scheme proposed by Cheon et al. based on the *Ring Learning With Errors* (RLWE) problem. The scheme is designed for approximate homomorphic arithmetic on vectors of complex numbers, differentiating it from other schemes such as BFV [19, 32] and BGV [20] which perform exact computations on integer values. The scheme is approximate because CKKS embeds noise into ciphertexts as part of the message itself, treating it as an error that cannot be removed during decryption but that stays small enough not to affect the significant figures of the plaintext. Given an encryption c of message m its decryption will then have the form $\mathsf{Decrypt}(sk, c) = m + e \pmod q$, where e is a small error, sk is the secret key and q a ciphertext modulus. The new value $m' = m + e$ is close enough to the original message m that it can replace it for approximate arithmetic.
A high level view of the full cycle for the homomorphic evaluation of function $f(\cdot)$ on a plain message m is presented in *Figure 1*.

2.3.1 Ring Learning With Errors

The security of the CKKS scheme is given by the hardness of the **Ring Learning With Errors (RLWE)** problem, introduced in [50]. In particular, the RLWE problem is built on *polynomial rings* with coefficients from a finite field. In order to formally introduce this concept we first define a parameter N or polynomial degree, being a power of 2, and a parameter q or coefficient modulus, being a prime integer. We denote with $\phi_M(X) = X^N + 1$ the M-th cyclotonic polynomial [46], with $M = 2N$. We also denote with $\mathbb{Z}_q$ the set of integers modulo q defined as $\mathbb{Z} \cap (-q/2, q/2]$.
We can now define the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[X]/\phi_M(X)$ as the ring of all polynomials of degree $N-1$ with coefficients in $\mathbb{Z}_q$, meaning that $\mathcal{R}_q = \{a_0 + a_1 x + \ldots + a_{n-1}x^{n-1} : a_i \in \mathbb{Z}_q \forall i\}$.

Under these assumptions we give a formal definition to the RLWE problem, specifically its *decision* version. Let:
- $a_i(X)$ be a set of random but known polynomials from $\mathcal{R}_q$.
- $e_i(X)$ be a set of small random and unknown polynomials relative to a bound b in the ring $\mathcal{R}_q$.
- $s(X)$ be a small unknown polynomial relative to a bound b in the ring $\mathcal{R}_q$.
- $b_i(X) = (a_i(X) \cdot s(X)) + e_i(X)$.

Given a list of polynomial pairs $(a_i(X), b_i(X))$, the RLWE decision problem consists in determining whether the $b_i(X)$ polynomials were indeed constructed as $b_i(X) = (a_i(X) \cdot s(X)) + e_i(X)$ or were generated randomly from $\mathcal{R}_q$.
The RLWE problem is presumed to be difficult to solve even on quantum computers, meaning that it can be used as a base to build secure public-secret key pairs for asymmetric encryption schemes.

2.3.2 Encoding and batching

The CKKS scheme uses the security guarantees of the RLWE problem to build an encryption scheme that can encrypt polynomials from $\mathcal{R}_q$. However, messages to encrypt typically come in the form of real numbers, so it is first necessary to encode these values into proper polynomial plaintexts.
CKKS proposes a way to encode a vector $\mathbf{z} \in \mathbb{C}^{N/2}$ of size $N/2$ into a polynomial $m(X) \in \mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$

using a variant of the complex canonical embedding map $\phi : \mathbb{C}^{N/2} \to \mathcal{R}$ [51]. The *encoding* and *decoding* algorithms are defined as follows:

- $\mathsf{Encode}(\mathbf{z}, \Delta)$. For $\mathbf{z} \in \mathbb{C}^{N/2}$, output the polynomial $m(X) = \phi(\Delta \cdot \mathbf{z}) \in \mathcal{R}$. We multiply $\mathbf{z}$ by a power-of-two scaling factor $\Delta > 1$ to preserve its precision.
- $\mathsf{Decode}(m, \Delta)$. For a plaintext polynomial $m \in \mathcal{R}$, output the original complex vector $\mathbf{z} = \phi^{-1}(m/\Delta) \in \mathbb{C}^{N/2}$.

Using this encoding technique a single plaintext polynomials can actually store up to $N/2$ complex values. We refer to the practice of storing multiple input values into a single plaintext or ciphertext as *batching* [65]. Batching enables the application of parallel processing through *Single Instruction, Multiple Data (SIMD)* operations [33]. Indeed, by operating on one such polynomial, we are effectively applying the same operation to each of the $N/2$ slots of the ciphertext at the same time.

2.3.3 Encrypted operations

We can now define the set of operations implemented by CKKS for plaintext and ciphertext data. Ciphertexts in CKKS come in the form of polynomial pairs $c = (c_0, c_1) \in \mathcal{R}^2_{q_l}$, where l is the level associated with the ciphertext. This level is an integer $0 < l < L$ that defines the coefficient modulus of the ciphertext as $q_l = \Delta^l \cdot q_0$.

In order to create such ciphertexts we first require a way to generate the asymmetric key pairs required for encryption and decryption. These keys are a function of a security parameter λ, ensuring that all attacks against the cryptographic system should take $\Omega(2\lambda)$ bit operations. We can then define:

- $\mathsf{KeyGen}(1^\lambda)$.
  - For the security parameter λ and a maximum coefficient modulus q_L, output the power-of-two ring dimension N.
  - Set the parameters for the small distributions χ_{key}, χ_{err} and χ_{enc} over $\mathcal{R}$.
  - Sample a secret $s \leftarrow \chi_{key}$, a random polynomial $a \leftarrow \mathcal{R}_{q_l}$ and an error $e \leftarrow \chi_{err}$. Set the secret key as $sk \leftarrow (1, s)$ and the public key as $pk \leftarrow (b, a) \in \mathcal{R}^2_{q_l}$, where $b \leftarrow -as + e \pmod{q_L}$.
  - Sample another $a' \leftarrow \mathcal{R}_{q_l^2}$ and $e' \leftarrow \chi_{err}$. Set the evaluation key evk as $evk \leftarrow (b', a') \in \mathcal{R}^2_{q_l^2}$, where $b' \leftarrow -a's + e' + q_L s^2 \pmod{q_L^2}$.

In addition to the public-secret key pair (pk, sk) KeyGen returns an evaluation key evk, which is used for multiplication between ciphertexts.
We can finally define all operations of the scheme:

- $\mathsf{Encrypt}(pk, m)$. For $m \in \mathcal{R}$, sample $v \leftarrow \chi_{enc}$ and $e_0, e_1 \leftarrow \chi_{err}$. Output $c \leftarrow v \cdot pk + (m + e_0, e_1) \pmod{q_L} \in \mathcal{R}^2_{q_l}$.
- $\mathsf{Decrypt}(sk, c)$. For $c = (b, a) \in \mathcal{R}^2_{q_l}$, output $m \leftarrow b + a \cdot s \pmod{q_l} \in \mathcal{R}$.
- $\mathsf{Add}(c_1, c_2)$. For $c_1, c_2 \in \mathcal{R}^2_{q_l}$, output $c_{add} \leftarrow c_1 + c_2 \pmod{q_l} \in \mathcal{R}^2_{q_l}$.
- $\mathsf{CMult}(c, m)$. For $c \in \mathcal{R}^2_{q_l}$ and $m \in \mathcal{R}$, output $c_{mult} \leftarrow m \cdot c \pmod{q_l} \in \mathcal{R}^2_{q_l}$.
- $\mathsf{Mult}(evk, c_1, c_2)$. For $c_1 = (b_1, a_1)$, $c_2 = (b_2, a_2) \in \mathcal{R}^2_{q_l}$, let $(d_0, d_1, d_1) = (b_1 b_2, a_1 b_2 + a_2 b_1, a_1 a_2) \pmod{q_l} \in \mathcal{R}^3_{q_l}$. Output $c_{mult} \leftarrow (d_0, d_1) + \lfloor q_L^{-1} \cdot d_2 \cdot evk \rceil \pmod{q_l} \in \mathcal{R}^2_{q_l}$.

Note that CKKS enables both ciphertext-ciphertext operations and ciphertext-plaintext operations, as the two share similar spaces. The latter is particularly useful in the case of multiplications, as ciphertext-plaintext multiplications CMult are less complex than the ciphertext-ciphertext Mult version.
The homomorphic Mult and CMult operations pose however some issues. Because plaintexts are scaled before encryption, multiplying two ciphertexts together (or a ciphertext and a plaintext) multiplies their scale as well. Not only that, but the noises introduced during encryption, which in CKKS are part of the messages themselves, get multiplied too and grow with each operation. To prevent this a $\mathsf{Rescale}$ operation is introduced:

- $\mathsf{Rescale}_{l \to l'}(c)$. For $c \in \mathcal{R}^2_{q_l}$, output $c' \leftarrow \lfloor \frac{q_{l'}}{q_l} c \rceil \pmod{q_{l'}} \in \mathcal{R}^2_{q_{l'}}$.

$\mathsf{Rescale}$ should be used after each homomorphic multiplication to manage the scale of the result by reducing its level from l to $l-1$. This entails that a scheme with maximum level L (and maximum coefficient modulus q_L) has a *multiplicative depth* of L, meaning that it can only support exactly L multiplications on the same

N	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
q_L **bit length**	≤ 27	≤ 54	≤ 109	≤ 218	≤ 438	≤ 881

Table 1: Maximum coefficient modulus q_L bit length for a given polynomial modulus degree N, using a security level λ of 128 bits [41].

ciphertext before decryption fails. When the parameters of the scheme are first defined it is then necessary to know the maximum number of multiplications that will be performed on each ciphertext.

CKKS also supports a rotation operation that shifts the values of the original plain vector across the slots of the encrypted ciphertext. A rotation of r slots of a vector $(v_0, \dots, v_{k-1})$ would return the shifted vector $(v_r, \dots, v_{k-1}, v_0, \dots, v_{r-1})$. Rotation requires the generation of an additional key rk, called rotation key. We can define it over ciphertexts as:

- $\mathsf{Rotate}(rk, c, r)$. For $c \in \mathcal{R}^2_{q_l}$, output $c' \in \mathcal{R}^2_{q_l}$ encrypting the plaintext vector of c rotated by r positions.

2.3.4 Parameters analysis

We summarize here the relevant parameters of the CKKS scheme and their role in practical applications.

- L is the maximum possible level for ciphertexts: freshly encrypted ciphertext start from this level. More importantly, L corresponds to the maximum number of multiplications allowed on the same ciphertext. L is usually the first parameter to be set and needs to be known in advance when designing an application using CKKS.
- N and specifically $N/2$ determines the number of values that can be encoded in a single ciphertext. It is also the main indicator for memory requirement and computation times of homomorphic algorithms, as a bigger N increases the complexity of encrypted operations.
- Δ and q_0 determine the precision of the encrypted values. In particular, a decrypted value will retain $(\log_2 \Delta)$ bits of precision for the decimal part and $(\log_2 q_0 - \log_2 \Delta)$ bits of precision for the integer part. In actual implementations of the scheme these values are typically approximated by a list of primes in order to greatly speed up computations. Instead of having $q_L = \Delta^L \cdot q_0$, a list of prime integers is chosen such that $p_0 \approx q_0$ and $p_1, \dots, p_L \approx \Delta$. The maximum coefficient modulus is then set as $q_L = \prod_{i=0}^{L} p_i$.

For a given security level λ and a polynomial modulus degree N there is an upper bound on the maximum coefficient modulus q_L. *Table 1* illustrates typical values for such upper bounds. Parameters L, N and Δ are indeed strictly interconnected and pose practical limitations to the capabilities of the scheme, requiring a trade-off between accuracy, performance and number of operations allowed.

2.4. Time series forecasting

Time series forecasting [22] is a task which involves the prediction of upcoming events given a set of timestamped observations. In particular, given a time series $X = (X_1, \dots, X_n)$ being an ordered set of real values, a prediction task involves finding a predictor $\hat{X}_{n+1}$ for the next value in the series X_{n+1}. This process can be expanded to produce multiple forecasts, resulting in an output vector $\hat{Y} = (\hat{X}_{n+1}, \dots, \hat{X}_{n+h})$ of future predictions, where h is denoted as the *forecast horizon.*
In particular, a *univariate* time series is composed of single-feature observations for each time instant, while a *multivariate* time series has multiple features for each data point. We also refer to *seasonal* time series data when the observations experience regular and predictable changes according to a specific cycle, where the seasonality of the data is the lenght of said cycle.

Multiple machine learning and specifically deep learning models have had great success historically on time series forecasting tasks. Some examples of machine learning models commonly used in the literature are Recurrent Neural Networks (RNNs) [62], AutoRegressive Integrated Moving Average (ARIMA) models [18], Long Short-Term Memory (LSTM) models [63] and transformer models [73].

2.5. TCNNs and privacy-preserving forecasting

Temporal Convolutional Neural Networks (TCNNs) [14] are a variation of *Convolutional Neural Networks (CNNs)* [49] designed for sequence modelling tasks. Both architectures make use of convolutional layers, which apply a convolution operation to a multi-dimensional input with a set of filters or kernels. This operation is used to return information about the relationship between neighboring inputs: the size of the filter defines a receptive field in which input values are weighted by the kernel and summed together. The filter "slides" across the dimensions of the input to compute an output or activation map for each of the filters used.
In traditional CNNs this convolution operation is used to analyze 2-D images, where 2-D filters extract features from patches of neighboring pixels. Instead, in TCNNs the same process is applied to 1-D time series inputs using 1-D kernels to perform a *causal convolution*. Each output of the convolutional layer is indeed a function of a sequence of previous inputs of length corresponding to the receptive field of the filter. Multiple convolutional layer can be chained together to increase the size of the receptive field as shown in *Figure 2*.

Among all deep learning time series solutions TCNNs are a prime candidate for implementation with homomorphic encryption due to their simple structure. Indeed, all other state-of-the-art models listed in *Section 2.4* are *stateful* recursive models, meaning that they require a persistent internal representation which is updated at every new input. The model uses this internal state to retain information between multiple inputs and compute a prediction based on the history of the inputs, rather than just the last value fed to the network. TCNNs instead use a *stateless*, feed-forward approach where all inputs are processed at the same time and the size of the receptive field determines how many inputs are considered for the computation of each prediction. The model does not hold an internal representation of the inputs, instead relying solely on the weights of the convolutional filters to evaluate the relationship between sequences of observations.
This property is particularly useful in a homomorphic environment where the number of operations on the same encrypted value is a main limiting factor. Indeed, the recurring updates of the internal state of stateful models are not compatible with a ciphertext of fixed multiplicative depth.

In order to implement a TCNN (or CNN) model using homomorphic encryption all of its layers need to be adapted to work using homomorphic operations, as leveled HE schemes natively only support addition and multiplication. While convolutional layers and fully connected layers only use these operations for computation, other non-linear layers commonly used by TCNNs are not compatible with the limitations of homomorphic encryption and as such need to be approximated with HE-compliant versions:

- *Activation functions*: Activation functions introduce non-linearities in a neural network in order to allow the computation of more complex functions. Typical activation functions such as the Rectified Linear Unit (ReLU) and tanh functions are too complex for computation in homomorphic environments. Low degree polynomials are instead much easier to calculate, and in particular the simple square function $f(x) = x^2$ is commonly used without a significant loss of complexity [35].
- *Pooling layers*: Maximum Pooling is typically used to reduce the dimensionality of convolutional activation maps, where each batch of values of a specific pooling size is reduced to its maximum. The computation of the maximum requires a comparison operator which is not available on ciphertexts. Instead, an Average Pooling layer can be used, as the average of multiple values can be easily computed through a sum and a division.
- *Normalization layers*: Normalization layers use the mean and standard deviation of the input data distribution to reduce the values of the inputs to a known range. As the inputs cannot be observed when encrypted, their mean and standard deviation are not accessible. These layers are replaced by batch normalization layers that calculate an average of the currently processed data, which uses a division by a fixed value.

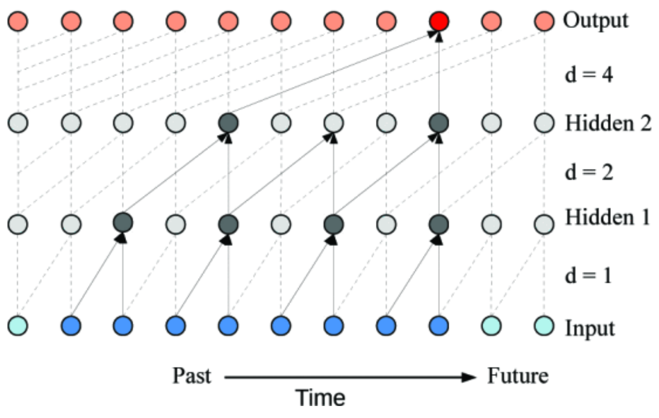


Figure 2: Receptive field of a multi-layer convolutional network of filter size 2. Convolutions in each layer are dilated in order to take into consideration a wider range of inputs. [54]

3. Related literature

After the groundbreaking introduction of FHE and the development of homomorphic encryption schemes for the practical use, the research field in HE-based privacy-preserving machine learning has been quickly and steadily growing. Multiple solutions with promising applications have been proposed in recent years, especially on the topic of privacy-preserving inference using leveled FHE schemes [35] [21] [47].
Our specific solution however aims to address three specific issues that have seen little to no research due to their complexity:

- The solution should be capable of *privacy-preserving training* on encrypted data as well as inference.
- The solution should be tailored to the specific *time series prediction* task, although capable of generalization to multiple different contexts.
- The solution should be implemented in a *Cloud learning* scenario and fully transparent to users in order to minimize their involvement.

Section 3.1 and *Section 3.2* analyze the few results found in the literature that at least partially address some of these points. *Section 3.3* then describes an initial answer to this problem in the form of the PINPOINT family of models, proposed by Falcetta et al. Our work acts as a continuation of their research, which offers a basic framework that will be extended with the addition of privacy-preserving training.

3.1. Privacy-preserving training

Privacy-preserving training of machine learning models is still a widely unexplored field in the literature. Most accepted soltuions for privacy-preserving learning focus soleley on encrypted inference, instead relying on the encryption of models pre-trained on plain data. Indeed, the iterative nature of the training process greatly increases the number of operations required on ciphertexts, which can easily go beyond the capabilities of the practical homomorphic schemes currently available.

One solution to this issue comes with *interactive* training, where the entirety or part of the computational demand is outsourced to the owners of the private data. Secure Multi-Party Computation (SMPC) approaches [79] offer a distributed computing environment where multiple parties can jointly calculate an objective function, each using their own private data. A Cloud environment can contribute as one of the parties themselves, and homorphic encryption and garbled circuits [76] allow users to securely share the results of private computations without any privacy loss. Bakshi et al. [15] use SMPC to introduce a Recurrent Neural Network model (RNN) with no network approximation, where the expensive non-linear activation functions are calculated by the client on ciphertexts received by the server.
Federated learning [48] instead takes a model sharing approach. Each party privately trains a machine learning model on their own data, then securely sends the model parameters to some intermediate server to aggregate the training results without revealing any information about the training data. Tran et al. [72] develop a general-purpose model sharing framework for decentralized networks with low communication bandwith, implementing an efficient protocol to jointly calculate a sum of private inputs. Zhang et al. [78] have participating clients share local gradient updates on their private data instead of the entire models. Gradient information is

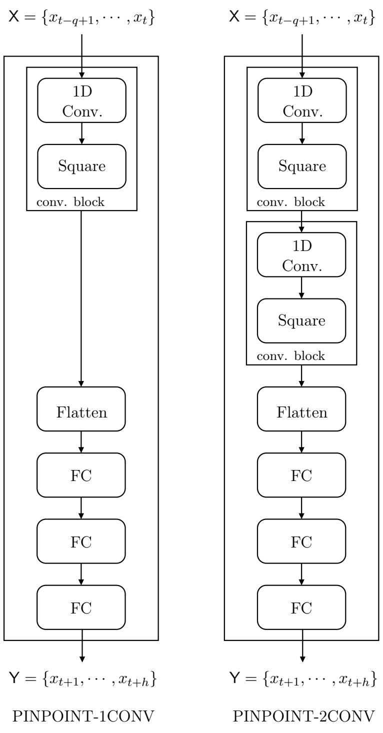
quantized and batched in order to improve the efficiency of the protocol and then aggregated by a server using additively homomorphic encryption.
Another interactive training solution that minimizes client-side computations is to have users refresh noisy ciphertexts by decrypting and recrypting their content. Mihara et al. [52] develop privacy-preserving implementations of neural network layers where ciphertexts are sent back to the user for recryption after each homomorphic multiplication. Takabi et al. [70] study the same approach in a SMPC scenario with data contributions from multiple users, and monitor the noise level of ciphertexts in order to use recryption only when strictly necessary.

All of these approaches are not a good fit for the Cloud-based learning scenario we propose. Our MLaaS model aims to be completely transparent to users, requiring no active participation from their side and making no computational demands. The goal is a *non-interactive* training solution, where all computations are performed server-side so that clients can simply send encrypted data and receive the desired encrypted outputs.
Non-interactive training for deep learning networks must come at the cost of severe constraints on the training process to be somewhat practical. Most studies in this area focus on simple networks for tasks such as linear regression and logistic regression. Kim et al. [43] perform privacy-preserving logistic regression analysis with the CKKS encryption scheme, proposing an efficient polynomial approximation of the sigmoid function and an algorithm for the homomorphic evaluation of gradient descent with ciphertext batching.
The first results on privacy-preserving training of a full deep learning neural network come from Nandakumar et al. [53]. They propose an effective implementation of the bootstrapping technique to refresh noisy ciphertexts, allowing an unlimited number of homomorphic operations, and employ a mini-batch version of gradient descent with ciphertext batching. However, the bootstrapping implementation has an extremely high computational cost, requiring the network to be massively simplified to make learning feasible. The solution is tested on the MNIST image classification task [26], which even after downscaling the inputs to 8×8 images requires almost 28 days of computation for a single epoch of training on the full dataset.
Similarly, Al Badawi et al. [11] develop a non-interactive deep learning training framework for text classification using the CKKS homomorphic scheme. In particular, a novel GPU implementation of CKKS is used for a 1 to 2 orders of magnitude speedup against existing implementations. The resulting network is composed of fully connected layers only and trained using batch gradient descent with backpropagation. The trade-off in this case is that the high number of operations required by the training phase allows only for 2 epochs of training on encrypted data, and the procedure still takes around 5 days to compute.

3.2. Privacy-preserving time series prediction

Time series analysis is a particularly challenging task under the limitations of homomorphic encryption. Similarly to the issues with training, typical stateful time series prediction models require iterative updates on the internal state of the layers even during inference, which is expensive in terms of noise propagation and discourages privacy-preserving implementations with HE. The result is that the existing literature on this topic is extremely scarce, with only a few highly specific privacy-preserving solutions being present.

Yue et al. [77] introduce a privacy-preserving hybrid deep learning framework for the classification of time-series medical images. In particular, convolutional blocks are used to extract the spatial features of the images, while Long-Short Term Memory (LSTM) cells encode the temporal information of the encrypted image sequences. Both types of networks are adapted to work with the BFV homomorphic encryption scheme [19, 32] and their results are combined through a sequence voting procedure. However, the model is only capable of performing inference on encrypted outputs, as training an LSTM network in the homomorphic domain would be unfeasible in terms of multiplicative depth. This solution also performs classification instead of prediction.
Bos et al. [17] propose a privacy-preserving time series prediction tool for smart grids using the Fan-Vercauteren encryption scheme [32]. Instead of neural networks, the simpler Group Method of Data Handling (GMDH) is implemented with homomorphic operations and used to make predictions on encrypted data regarding the electric load of smart grids. Although this approach is quite effective in this limited scenario, GMDH is a far less powerful tool than neural networks and cannot be easily extended to a general-purpose application.
Finally, Paul et al. [56] use a collective learning protocol to train a privacy-preserving LSTM network on time series classification. The work focuses on fine-tuning the last fully connected layer of the network using encrypted logistic regression. Collective learning can then be used to securely share the model parameters obtained by multiple workers who have access to different encrypted data in order to create a better aggregated learner. This approach is once again only designed for time series classification and cannot be easily adapted to a MLaaS scenario, as it requires users to train their models locally.



$\mathsf{X} = \{x_{t-q+1}, \cdots, x_t\}$
1D Conv.
Square
conv. block
Flatten
FC
FC
FC
$\mathsf{Y} = \{x_{t+1}, \cdots, x_{t+h}\}$
PINPOINT-1CONV
PINPOINT-2CONV

Figure 3: Diagram of the PINPOINT Temporal Convolutional Neural Network family.

3.3. PINPOINT

The **Privacy-preservINg temPoral cOnvolutIonal Neural neTworks (PINPOINT)** family of models [31] is a collection of privacy-preserving time series forecasting models inspired by TCNNs. PINPOINT models are built on the BFV homomorphic encryption scheme [19, 32] and implemented in Python through an extension of the *PyCrCNN* library [28], which itself uses the Pyfhel library [41] to perform homomorphic computations on encrypted data.
Unlike the other solutions described so far, the PINPOINT architecture is specifically designed with the limitations of homomorphic encryption in mind for the purpose of privacy-preserving deep learning on the Cloud. In particular, the choice of using a TCNN minimizes the number of operations necessary on ciphertexts, as described in *Section 2.5*.
The structure of PINPOINT is shown in *Figure 3*. It receives as input a vector of q encrypted time series observations and returns an encrypted vector of the predictions for the next h time instants. The model is built from a sequence of convolutional blocks, followed by a flatten layer and three fully connected layers that return the actual output. Each convolutional block is composed of a 1-D convolutional layer and a square layer, chosen as a non-linear activation function compatible with homomorphic constraints. The original paper presents two different configurations of the model, one which uses a single convolutional block (PINPOINT-1CONV) and one which uses two (PINPOINT-2CONV). The higher complexity of the PINPOINT-2CONV version leads to a slight increase in performance at the cost of a higher number of operations required. The model does not use any pooling or normalization layers, instead keeping the network structure as simple as possible to facilitate the implementation of homomorphic operations.

Both configurations of PINPOINT are shown to have a prediction accuracy directly comparable with other state-of-the-art time series forecasting models run in a privacy-violating way (meaning on plain data).
The main drawback with this approach is that the model is limited to privacy-preserving inference and cannot learn from encrypted data. Instead, the model is first trained on plain publicly available data, requiring processing in the clear that violates its privacy. After the training phase, the model weights are fixed and encoded to support operations on ciphertexts, and a new encrypted dataset can be used for encrypted inference.
In addition, when compared to plain algorithms the homomorphic operations introduce an important overhead both in terms of time and memory for the computation of each forecast. The BFV homomorphic scheme also natively supports the encryption of integer values only, while the model takes floating point values as inputs. The PyCrCNN library of PINPOINT gets around this issue by implementing a fractional encoder within the encryption procedure, which comes at the cost of some precision in the encrypted values.

4. Proposed solution

The goal of our work is to develop a privacy-preserving framework for machine learning that suports both encrypted inference and encrypted training. We also prove its effectiveness by developing a working model for univariate time series prediction that pushes the limits of these encrypted training capabilities within the constraints given by homomorphic encryption. The architecture should be fully compatible with Cloud computing, providing a privacy preserving time series forecasting application "as-a-service" with general-purpose models that can be fine-tuned by users using encrypted data. Its usage is exemplified in the following scenario.
A user wants to use a Cloud-based application to provide a time series prediction on sensitive data without breaching its privacy. They have access to a dataset Z of univariate time series observations up to time t and want to compute a prediction $\hat{Z}$ for its values in the next h time instants. However, they do not want to disclose the actual values of Z to a third-party provider. By using our solution, the user can first encrypt their data Z locally using their public key and send over to our Cloud provider $\underline{\mathsf{Z}} = \mathsf{Encrypt}(pk, Z)$ instead. Our machine learning model on the Cloud platform can use part of the encrypted data $\underline{\mathsf{Z}}_{train}$ to fine-tune its own parameters through a series of homomorphic computations, without ever accessing the original information. The rest of the encrypted dataset $\underline{\mathsf{Z}}_{inf}$ is then used to compute the h predicted values $\hat{\underline{\mathsf{Z}}}$, still in encrypted form. The output is finally sent back to the user who can use their secret key to recover the plain prediction $\hat{Z} = \mathsf{Decrypt}(sk, \hat{\underline{\mathsf{Z}}})$.

Our starting point is the **PINPOINT** family of models described in *Section 3.3* and the TCNN architecture in general. PINPOINT already provides a solution for inference on encrypted data and the underlying PyCrCNN library offers HE-compatible implementations of the basic layers of a TCNN. The architecture shows promising results both on plain and encrypted data and proves that homomorphic encryption can be applied to feed-forward models for effective privacy-preserving learning. However, PINPOINT does not implement a training procedure, instead relying on encrypting a model pre-trained on plain data. Although this solution can be effective even in Cloud-based scenarios, it does not allow users to tune the model parameters to their specific prediction task unless they have access to equivalent plain data not protected by privacy.

Several steps have to be taken in order to add encrypted training functionalities to the PINPOINT architecture in its current state. *Section 4.1* first describes the implementation of the CKKS encrption scheme and batched vectorized operations for PINPOINT, which should minimize the number of operations required on ciphertexts. *Section 4.2* then expands this initial framework by introducing a homomorphic training algorithm that can work with the implemented neural network layers. Finally, *Section 4.3* describes a novel model for time series prediction based on PINPOINT that uses these new functionalities to effectively perform training on encrypted data.

4.1. CKKS implementation

The current implementation of the PINPOINT family of models poses several challenges to the addition of a training algorithm. First, as explained in *Section 3.3*, PINPOINT is built on the PyCrCNN library which uses the BFV scheme for homomorphic computations. The BFV scheme natively supports only integer operations, requiring additional steps in order to implement a fractional encoder that lead to lower overall accuracy and performance. More importantly however, PyCrCNN does not make use of ciphertext batching, meaning that every private value is encrypted by itself in a new ciphertext. Implementing a training phase requires a much larger ciphertext size and number of operations than simple inference, which in turn quickly makes this approach completely unfeasible both in terms of time and resources.
Before we can add any new functionalities to the model, we first need to restructure the PyCrCNN library to be compatible with **ciphertext batching** and in particular with the **CKKS** encryption scheme, chosen for its native support of floating point values. Ciphertext batching enables SIMD processing by operating on multiple input values at once, at the expense of the ability to access individual values of the encrypted vector. Our first goal then becomes translating all operations performed in the forward pass of all PINPOINT layers into matrix and vector operations compatible with batched ciphertexts.

For each proposed algorithm, we highlight the number of multiplications and rotations used (being the relevant operations in terms of computation time) and, most importantly, the multiplicative depth, meaning the number of multiplications that need to be applied in sequence to the same ciphertext. The multiplicative depth is what determines the number of multiplications allowed under the chosen HE settings, and consequently what we are mainly looking to minimize. An efficient algorithm needs to use up at most one level of multiplication on the input in its forward pass. Any number of additions and rotations are allowed, although they should also be minimized under the previous constraints to improve efficiency.

4.1.1 Encoding and basic operations

Time series forecasting tasks use input sequences of multiple observed values in order to compute an output sequence of multiple upcoming predictions. Batching can be applied to such sequences on multiple levels when encrypting them to CKKS ciphertexts.
The first obvious step is to encrypt an entire input sequence into a single ciphertext vector. The current single value operations in PINPOINT can then be replaced with much more efficient vector and matrix formulations. We can use the operations described in *Section 2.3.2* and *Section 2.3.3* to encrypt a vector of inputs $\mathbf{x} = \begin{pmatrix} x_0 & x_1 & \cdots & x_n \end{pmatrix}^{\mathbf{T}}$ of size $n+1$ into the ciphertext:

$$\mathsf{X} = \mathsf{Encrypt}(pk, \mathsf{Encode}(\mathbf{x}, \Delta)).$$

We will abuse this notation and simply refer to this operation as $\mathsf{X} = \mathsf{Encrypt}(\mathbf{x})$, and similarly avoid mentioning scales or keys for all further operations. With this encoding, the resulting ciphertext X has the form:

$$\mathsf{X} \;=\; \{\; x_0 \;\; x_1 \;\; \cdots \;\; x_n \;\; 0 \;\; \cdots \;\; 0 \;\}_{(l)}$$

where we denote with $\{\}$ a set of encrypted values and where (l) is the current level of the ciphertext.
The ciphertext as described can encode up to $N/2$ complex values, which in practical applications is a number order of magnitudes larger than the size of inputs $n+1$. We can then make efficient use of the ciphertext slots by applying another batching step and stacking together multiple input sequences into the same ciphertext. We define for each ciphertext a number of rows r (being a power of 2 to ease future operations) that can contain up to $\frac{N}{2r}$ elements each. Multiple input vectors $\mathbf{x}_i$ can now be encrypted into the same ciphertext as:

$$\mathsf{X}_r \;=\; \{\; x_{0,0} \;\cdots\; x_{0,n} \;\; 0\cdots0 \;\; x_{1,0} \;\cdots\; x_{1,n} \;\; 0\cdots0 \;\; x_{r-1,0} \;\cdots\; x_{r-1,n} \;\; 0\cdots0 \;\}_{(l)}.$$

The input vectors are all padded with 0s to fill each row. A 0 padding is extremely important as it prevents unwanted overlaps between rows after a Rotate operation on the ciphertext. Indeed, rotating a ciphertext makes it so that the encrypted values at the beginning or end of the sequence wrap around to the other end of the vector, such as:

$$\mathsf{Rotate}(\mathsf{X}, 1) \;=\; \{\; x_1 \;\; x_2 \;\; \cdots \;\; x_n \;\; 0 \;\; \cdots \;\; 0 \;\; x_0 \;\}_{(l)}$$

Without the zero padding, rotating a ciphertext would cause some values of each row to spill into a neighboring one. In order to avoid this issue the size of each row should always be at least twice as big as the maximum size of the vectors to encrypt, or $r < \frac{N}{4(n+1)}$.

We can now study the implementation of the basic building blocks of PINPOINT using these encrypted input vectors. In order to simplify their discussion, the following algorithms are explained using only ciphertexts with $r = 1$ where no batching of multiple input vectors is performed. However, unless stated otherwise, the SIMD operations employed by the algorithms work in the exact same way even with ciphertexts of $r > 1$. The only difference is that any vector encrypted or encoded within the algorithms will have to be padded with 0s and replicated along all rows, making it so that the input values in each row are all subject to the same exact operations. The encryption operation of additional parameter vectors in the case of multiple rows can then be viewed as:

$$\mathsf{Encrypt}(\mathbf{x}) \to \mathsf{Encrypt}([\mathbf{x} \;\&\; [0] * (r - n - 1)] * r).$$

where & denotes a concatenation and $*$ a replication of an array.

4.1.2 Fully Connected layer

Fully Connected (FC) layers take an input vector $\mathbf{x}$ of size $n+1$ and perform a matrix multiplication using a weight matrix $\mathbf{W}$ and a bias term $\mathbf{b}$. Given:

$$\mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{pmatrix}, \; \mathbf{W} = \begin{pmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \cdots & & & \\ w_{m,0} & w_{m,1} & \cdots & w_{m,n} \end{pmatrix}, \; \mathbf{b} = \begin{pmatrix} b_0 \\ b_1 \\ \cdots \\ b_m \end{pmatrix},$$

a fully connected layer will output the feature vector $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$ of size $m+1$. Each element of the output vector $\mathbf{y}$ has the form, for $0 \le i \le m$,

$$y_i = b_i + \sum_{j=1}^{n} w_{i,j} x_j. \tag{1}$$

In order to operate on an encrypted input vector, we present an algorithm for homomorphic matrix-vector multiplication with a multiplicative depth of one. We also analyze in detail each relevant line of the algorithm, showing its output.

Algorithm 1 Fully Connected layer forward pass

```
1: for $0 \le i \le n$ do
2:     $\mathsf{W}_i = \mathsf{Encrypt}(d_i(\mathbf{W}))$
3: end for
4: $\mathsf{B} = \mathsf{Encrypt}(\mathbf{b})$
5: $\mathsf{Xd} = \mathsf{Add}(\mathsf{X}, \mathsf{Rotate}(\mathsf{X}, -(n+1)))$
6: for $0 \le i \le n$ do
7:     $\mathsf{C1}_i = \mathsf{Rotate}(\mathsf{Xd}, i)$
8:     $\mathsf{C2}_i = \mathsf{Rescale}(\mathsf{Mult}(\mathsf{C1}_i, \mathsf{W}_i))$
9: end for
10: $\mathsf{C3} = \mathsf{Add}(\mathsf{C2}_0, \dots, \mathsf{C2}_n)$
11: $\mathsf{Y} = \mathsf{Add}(\mathsf{B}, \mathsf{C3})$
```

Line 2. We encrypt the weight matrix $\mathbf{W}$ of shape $(m+1) \times (n+1)$. As suggested in [38], we can greatly simplify the following matrix multiplication by representing $\mathbf{W}$ in *diagonal order* and encrypting it into $n+1$ separate ciphertexts. Namely, $\mathbf{W}$ has $n+1$ generalized diagonals $d_0, \dots, d_n$ of $m+1$ elements $d_i[0], \dots, d_i[m]$ such that $d_i[j] = w_{j\%(n+1),(j+i)\%(n+1)}$, where % is the modulo operation. Assuming for our example $m < n$ (although it is not required by the algorithm), this results in the following ciphertexts:

$$\begin{array}{lcl} \mathsf{W}_0 & = & \{ \; w_{0,0} \;\; w_{1,1} \;\; \cdots \;\; w_{m,m} \;\; 0 \;\; \cdots \;\; 0 \; \}_{(l)} \\ \mathsf{W}_1 & = & \{ \; w_{0,1} \;\; w_{1,2} \;\; \cdots \;\; w_{m,m+1} \;\; 0 \;\; \cdots \;\; 0 \; \}_{(l)} \\ \cdots & & \\ \mathsf{W}_n & = & \{ \; w_{0,n} \;\; w_{1,0} \;\; \cdots \;\; w_{m,m-1} \;\; 0 \;\; \cdots \;\; 0 \; \}_{(l)} \end{array}$$

Line 4. We simply encrypt the bias vector $\mathbf{b}$ to a single ciphertext.

$$\mathsf{B} \; = \; \{ \; b_0 \;\; b_1 \;\; \cdots \;\; b_m \;\; 0 \;\; \cdots \;\; 0 \; \}_{(l)}$$

Line 5. We append to the end of the input ciphertext $\mathsf{X} = \mathsf{Encrypt}(pk, \mathbf{x})$ a copy of itself, in order to make its values properly wrap around in the following rotations.

$$\mathsf{Xd} \; = \; \{ \; x_0 \;\; x_1 \;\; \cdots \;\; x_n \;\; x_0 \;\; \cdots \;\; x_n \;\; 0 \;\; \cdots \;\; 0 \; \}_{(l)}$$

Line 7. We rotate Xd $n+1$ times, producing $n+1$ staggered versions of the input with some unnecessary values appended at the end.

$$\begin{array}{lcl} \mathsf{C1}_0 & = & \{ \; x_0 \;\; x_1 \;\; \cdots \;\; x_n \;\; x_0 \;\; \cdots \;\; 0 \;\; 0 \; \}_{(l)} \\ \mathsf{C1}_1 & = & \{ \; x_1 \;\; x_2 \;\; \cdots \;\; x_0 \;\; x_1 \;\; \cdots \;\; 0 \;\; x_0 \; \}_{(l)} \\ \cdots & & \\ \mathsf{C1}_n & = & \{ \; x_n \;\; x_0 \;\; \cdots \;\; x_{n-1} \;\; x_n \;\; 0 \;\; \cdots \;\; x_{n-1} \; \}_{(l)} \end{array}$$

Line 8. We multiply the rotated inputs by the weight diagonals. Note that the resulting ciphertexts only have $m+1$ values, as the weight diagonals act as a mask that sets all other unnecessary values to 0 in the multiplication.

$$\begin{array}{lcl} \mathsf{C2}_0 & = & \{ \; w_{0,0}x_0 \;\; w_{0,1}x_1 \;\; \cdots \;\; w_{m,m}x_m \;\; 0 \;\; \cdots \;\; 0 \; \}_{(l+1)} \\ \mathsf{C2}_1 & = & \{ \; w_{0,1}x_1 \;\; w_{1,2}x_2 \;\; \cdots \;\; w_{m,m+1}x_{m+1} \;\; 0 \;\; \cdots \;\; 0 \; \}_{(l+1)} \\ \cdots & & \\ \mathsf{C2}_n & = & \{ \; w_{0,n}x_n \;\; w_{1,0}x_0 \;\; \cdots \;\; w_{m,m-1}x_{m-1} \;\; 0 \;\; \cdots \;\; 0 \; \}_{(l+1)} \end{array}$$

Line 10. We sum together the previous $n+1$ results. Each element of the resulting vector is the sum of all values of $\mathbf{x}$ multiplied by a different row of the original matrix $\mathbf{W}$.

$$\mathsf{C3} \; = \; \{ \; \textstyle\sum_{j=0}^{n} w_{0,j}x_j \;\; \sum_{j=0}^{n} w_{1,j}x_j \;\; \cdots \;\; \sum_{j=0}^{n} w_{m,j}x_j \;\; 0 \;\; \cdots \;\; 0 \; \}_{(l+1)}$$

Line 11. We add the bias B to obtain the final $m+1$ values of (1).

$$\mathsf{Y} \; = \; \{ \; b_0 + \textstyle\sum_{j=0}^{n} w_{0,j}x_j \;\; b_1 + \sum_{j=0}^{n} w_{1,j}x_j \;\; \cdots \;\; b_m + \sum_{j=0}^{n} w_{m,j}x_j \;\; 0 \;\; \cdots \;\; 0 \; \}_{(l+1)}$$

Algorithm 1 uses $\mathcal{O}(n)$ rotations and multiplications per input and has a depth of one multiplication.

4.1.3 Convolutional layer

Convolutional layers take an input $\mathbf{x}$ and perform a convolution with a filter $\mathbf{w}$, called kernel. In particular, TCNNs use one-dimensional convolutional layers, which operate over 1-D inputs (in our case a time series) using 1-D kernel vectors. Here the kernel sets a temporal window and convolutes neighboring inputs together in order to evaluate their relationship, returned in a so called activation map.
Ignoring the concepts of stride and padding which are not relevant to our application (namely, we always use a

stride of 1 and valid padding), for a single kernel $\mathbf{w}$ and a bias value b, given:

$\mathbf{x} = \begin{pmatrix} x_0 & x_1 & \cdots & x_n \end{pmatrix}^{\mathbf{T}}$, $\mathbf{w} = \begin{pmatrix} w_0 & w_1 & \cdots & w_k \end{pmatrix}^{\mathbf{T}}$,

with $n > k$, a 1-D convolutional layer will output a vector $\mathbf{y}$ of length $n-k+1$ such that, for $0 \leq i \leq (n-k+1)$,

$$y_i = b + \sum_{j=0}^{k} w_j x_{j+i}. \tag{2}$$

We can observe that this result is equivalent to the following matrix multiplication:

$$\mathbf{y} = \begin{pmatrix} w_0 & \cdots & w_k & 0 & 0 & \cdots & 0 \\ 0 & w_0 & \cdots & w_k & 0 & \cdots & 0 \\ \cdots & & & & & & \\ 0 & 0 & \cdots & 0 & w_0 & \cdots & w_k \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{pmatrix} + \begin{pmatrix} b \\ b \\ \cdots \\ b \end{pmatrix},$$

or $\mathbf{y} = \mathbf{W}'\mathbf{x} + \mathbf{b}'$, where $\mathbf{W}'$ is a matrix of size $(n-k+1) \times (n+1)$ and $\mathbf{b}'$ is a vector of size $n-k+1$. As such, we can adapt the algorithm used in *Section 4.1.2* to compute homomorphic convolution as well.

Algorithm 2 1-D Convolutional layer forward pass

```
1: for $0 \leq i \leq k$ do
2:     $\mathsf{W}_i = \mathsf{Encrypt}([w_i] * (n-k+1))$
3: end for
4: $\mathsf{B} = \mathsf{Encrypt}([b] * k)$
5: for $0 \leq i \leq k$ do
6:     $\mathsf{C1}_i = \mathsf{Rotate}(\mathsf{X}, i)$
7:     $\mathsf{C2}_i = \mathsf{Rescale}(\mathsf{Mult}(\mathsf{C1}_i, \mathsf{W}_i))$
8: end for
9: $\mathsf{C3} = \mathsf{Add}(\mathsf{C2}_0, \ldots, \mathsf{C2}_n)$
10: $\mathsf{Y} = \mathsf{Add}(\mathsf{B}, \mathsf{C3})$
```

Line 2. We encrypt the kernel $\mathbf{w}$ into $k+1$ ciphertexts corresponding to the diagonals of matrix $\mathbf{W}'$. We only need to encrypt the first $k+1$ diagonals as the remaining ones are all empty. Each diagonal $d_i(\mathbf{W}')$ simply contains $n-k+1$ copies of the kernel parameter w_i.

$$\begin{array}{lcl} \mathsf{W}_0 & = & \{ \; w_0 \; w_0 \; \cdots \; w_0 \; 0 \; \cdots \; 0 \; \}_{(l)} \\ \cdots & & \\ \mathsf{W}_k & = & \{ \; w_k \; w_k \; \cdots \; w_k \; 0 \; \cdots \; 0 \; \}_{(l)} \end{array}$$

Line 4. For the bias we once again encrypt a vector containing $n-k+1$ copies of b.

$$\mathsf{B} \; = \; \{ \; b \; b \; \cdots \; b \; 0 \; \cdots \; 0 \; \}_{(l)}$$

Line 6-7. We proceed with multiplying the rotated inputs with the encrypted diagonals as before. Note that duplicating the input ciphertext X is no longer necessary, as the kernel only convolutes with the input up to its final value without any looping. Again we only obtain $k+1$ ciphertexts as the rest would all be set to 0.

$$\begin{array}{lcl} \mathsf{C2}_0 & = & \{ \; w_0x_0 \;\; w_0x_1 \;\; \cdots \;\; w_0x_{n-k} \;\; 0 \; \cdots \; 0 \; \}_{(l+1)} \\ \mathsf{C2}_1 & = & \{ \; w_1x_1 \;\; w_1x_2 \;\; \cdots \;\; w_1x_{n-k+1} \;\; 0 \; \cdots \; 0 \; \}_{(l+1)} \\ \cdots & & \\ \mathsf{C2}_k & = & \{ \; w_kx_k \;\; w_kx_{k+1} \;\; \cdots \;\; w_kx_n \;\; 0 \; \cdots \; 0 \; \}_{(l+1)} \end{array}$$

Line 9-10. We sum the $k+1$ results with the bias to obtain (2).

$$\mathsf{Y} \; = \; \{ \; b + \sum_{j=0}^{k} w_jx_j \;\; b + \sum_{j=0}^{k} w_jx_{j+1} \;\; \cdots \;\; b + \sum_{j=0}^{k} w_jx_{j+(n-k)} \;\; 0 \; \cdots \; 0 \; \}_{(l+1)}$$

Algorithm 2 uses $\mathcal{O}(k)$ rotations and multiplications per input and has a depth of one multiplication.
Note that typically convolutional layers deal with multichannel data. *Algorithm 2* can be extended to deal with both multiple output and input channels while keeping a multiplicative depth of one:

- A convolutional layer with multiple output channels uses m different kernels $\mathbf{w}_i$, each of which is convoluted with the same input to produce a multichannel activation map. In this case *Algorithm 2* is simply repeated m times for m different kernels, resulting in m separate output vectors. The resulting complexity is of $\mathcal{O}(m \cdot k)$ rotations and multiplications.
- A convolutional layer with multiple input channels takes c different vectors $\mathbf{x}_i$ as input. In this case a different kernel $\mathbf{w}_i$ is applied to each input channel using *Algorithm 2* c times, and the resulting c

activation maps are summed together to produce a single output vector. The resulting complexity is of $\mathcal{O}(c \cdot k)$ rotations and multiplications.

4.1.4 Square layer

Square layers are the non-linear activation layer of choice for PINPOINT, being easy to implement under homomorphic conditions. Indeed, square activation layers simply return the square of an input $\mathbf{x}$, which is an operation natively supported on ciphertexts.
The layer uses one multiplication per input, meaning a depth of one multiplication too.

4.1.5 Flatten layer

Flatten layers take $m+1$ separate inputs $\mathbf{x}_i$ of size $n+1$ and concatenate them in a single vector. Flatten layers are used between convolutional blocks and fully connected blocks in order to transform the multichannel outputs of convolutional layers into a single input vector compatible with fully connected layers. Given:

$\mathbf{x}_0 = \begin{pmatrix} x_{0,0} & \cdots & x_{0,n} \end{pmatrix}^{\mathbf{T}}, \ \ldots, \ \mathbf{x}_m = \begin{pmatrix} x_{m,0} & \cdots & x_{m,n} \end{pmatrix}^{\mathbf{T}},$

a flatten layer will output a vector of length $(m+1) \cdot (n+1)$

$$\mathbf{y} = \begin{pmatrix} x_{0,0} & \cdots & x_{0,n} & x_{1,0} & \cdots & x_{m-1,n} & x_{m,0} & \cdots & x_{m,n} \end{pmatrix}^{\mathbf{T}}. \tag{3}$$

While this can easily be achieved under homomorphic conditions with a set of rotations and additions, adding up multiple rotated ciphertexts in CKKS can quickly build up their approximation errors and give suboptimal results. Instead, we take an extra masking step that improves accuracy at the cost of one multiplication.

Algorithm 3 Flatten layer forward pass

```
1: for $0 \le i \le m$ do
2:     $\mathsf{C1}_i = \mathsf{Rotate}(\mathsf{X}_i, -i(n+1))$
3: end for
4: $M = [1] * (n+1)$
5: for $0 \le i \le m$ do
6:     $\mathsf{M}_i = \mathsf{Encode}(\mathsf{Rotate}(M, -i(n+1)))$
7:     $\mathsf{C2}_i = \mathsf{Rescale}(\mathsf{Mult}(\mathsf{M}_i, \mathsf{C1}_i))$
8: end for
9: $\mathsf{Y} = \mathsf{Add}(\mathsf{C2}_0, \ldots, \mathsf{C2}_m)$
```

Line 2. We rotate each encrypted input $\mathsf{X}_i = \mathsf{Encrypt}(\mathbf{x}_i)$ to the position it will occupy in the final concatenated output, which means shifting it by $-i(n+1)$ positions. The rotations leave some unwanted errors where 0s should be, especially in the first few ciphertext slots.

$$\begin{array}{lcl} \mathsf{C1}_0 & = & \{ \ x_{0,0} \ \cdots \ x_{0,n} \ \ 0 \ \cdots \ 0 \ \ 0 \ \cdots \ 0 \ \ 0 \ \cdots \ 0 \ \}_{(l)} \\ \mathsf{C1}_1 & = & \{ \ \varepsilon \ \cdots \ 0 \ \ x_{1,0} \ \cdots \ x_{1,n} \ \ 0 \ \cdots \ 0 \ \ 0 \ \cdots \ 0 \ \}_{(l)} \\ \cdots & & \\ \mathsf{C1}_m & = & \{ \ \varepsilon \ \cdots \ 0 \ \ 0 \ \cdots \ 0 \ \ x_{m,0} \ \cdots \ x_{m,n} \ \ 0 \ \cdots \ 0 \ \}_{(l)} \end{array}$$

Line 4-6. We define an encoded mask for each rotated input $\mathsf{C1}_i$, where the positions filled with useful values are marked with 1s and the rest with 0s. Note that the values of these masks are known and only depend on the size of the ciphertexts, so they can simply be encoded instead of encrypted.

$$\begin{array}{lcl} \mathsf{M}_0 & = & \{ \ 1 \ \cdots \ 1 \ \ 0 \ \cdots \ 0 \ \ 0 \ \cdots \ 0 \ \ 0 \ \cdots \ 0 \ \} \\ \mathsf{M}_1 & = & \{ \ 0 \ \cdots \ 0 \ \ 1 \ \cdots \ 1 \ \ 0 \ \cdots \ 0 \ \ 0 \ \cdots \ 0 \ \} \\ \cdots & & \\ \mathsf{M}_m & = & \{ \ 0 \ \cdots \ 0 \ \ 0 \ \cdots \ 0 \ \ 1 \ \cdots \ 1 \ \ 0 \ \cdots \ 0 \ \} \end{array}$$

Line 7-9. We multiply each rotated input with its mask in order to reset any errors in the unused ciphertext slots to 0, then add them all up to obtain (3).

$$\mathsf{Y} \ = \ \{ \ x_{0,0} \ \cdots \ x_{0,n} \ \ x_{1,0} \ \cdots \ x_{m-1,n} \ \ x_{m,0} \ \cdots \ x_{m,n} \ \ 0 \ \cdots \ 0 \ \}_{(l+1)}$$

Algorithm 3 uses $\mathcal{O}(m)$ rotations and multiplications and has a depth of one multiplication.

4.2. Encrypted training

Section 4.1 analyzes how to optimize the current PINPOINT model to work with the CKKS scheme and with batched inputs. We can now build upon it and discuss the implementation of a proper training step on encrypted data.
While implementing a training algorithm for each layer of the model is theoretically possible, the practicality of this operation in terms of multiplicative depth is a major issue. Most implementations of CKKS use $2^{15} = 32768$ as the maximum possible polynomial degree N for cyphertexts, as going above that would be too demanding in terms of time and memory occupation. This in turn limits the maximum bit-length of the coefficient modulus q_L to 881 (see *Table 1*). If we want to guarantee good enough precision of the encrypted data with a scale $\Delta > 2^{30}$, this leaves us with a maximum multiplicative depth L of 25-30 levels depending on the chosen parameters (we recall that $q_L = \Delta^L q_0$).
Under this harsh limitation, training the entire model for a sufficient number of epochs becomes impossible. Indeed, in order to train a layer on encrypted data its parameters need to be encrypted at the start of the training algorithm, and will be iteratively updated at each epoch. If the layer's backward step uses up even just one level of multiplication this will stack up with subsequent iterations, posing a limit to the number of epochs allowed. Instead, we focus on implementing a backward step solely for *fully connected layers*, as they make up the final layers of the PINPOINT model, and maximizing its efficiency in terms of multiplication usage. We will analyze how this single backward step can produce sufficient training results in the following section.

4.2.1 Gradient descent

The weights of a fully connected layer can be trained using **gradient descent**. Gradient descent is an optimization algorithm that minimizes an objective function by moving its parameters in the direction opposite to its gradient. The objective function we want to minimize is a cost or loss function C, which given the inputs $\mathbf{x}_i$ measures the error between the predicted outputs of our layer $\mathbf{y}_i$ and the actual target values $\mathbf{t}_i$ of the same inputs. We specifically use the *batch* version of gradient descent, which evaluates the next set of parameters over the entirety of the predicted outputs. Stochastic gradient descent, which instead updates the weights and bias for each sample output individually, is typically preferred in privacy-violating algorithms for large enough datasets as it converges faster and more consistently to the optimal solution. However, under homomorphic conditions this approach would quickly run through all multiplication levels of the weight ciphertexts due to the frequency of their updates.
Given a learning rate α and a fully connected layer with weights $\mathbf{W}_t$ and bias $\mathbf{b}_t$ at iteration t of the batch gradient descent algorithm, we can find the next set of weights and bias at iteration $t+1$ as:

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \alpha \frac{\partial C}{\partial \mathbf{W}} \tag{4}$$

$$\mathbf{b}_{t+1} = \mathbf{b}_t - \alpha \frac{\partial C}{\partial \mathbf{b}}. \tag{5}$$

This update can be repeated any number of times until it eventually reaches convergence to a local minimum. Our loss function of choice over $k+1$ predictions is the *sum-of-squares* error $C = \frac{1}{2k}\sum_{i=0}^{k}(\mathbf{y}_i - \mathbf{t}_i)^2$, where we recall that $\mathbf{y}_i = \mathbf{W}\mathbf{x}_i + \mathbf{b}$. The gradient of C with respect to $\mathbf{W}$ and $\mathbf{b}$ can be computed using the derivative chain rule, resulting in the update equations:

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \frac{\alpha}{k}\sum_{i=0}^{k}(\mathbf{y}_i - \mathbf{t}_i)\mathbf{x}_i^{\mathbf{T}} \tag{6}$$

$$\mathbf{b}_{t+1} = \mathbf{b}_t - \frac{\alpha}{k}\sum_{i=0}^{k}(\mathbf{y}_i - \mathbf{t}_i). \tag{7}$$

The point of interest here for homomorphic computation is the matrix product $(\mathbf{y}_i - \mathbf{t}_i)\mathbf{x}_i^{\mathbf{T}}$, also referred to as the *outer product* $(\mathbf{y}_i - \mathbf{t}_i) \otimes \mathbf{x}_i$. The outer product is defined as a $(m+1) \times (n+1)$ matrix obtained by multiplying each element of the first vector with each element of the second. In particular we are interested in obtaining the *diagonal order* formulation of this matrix, as the results of the outer products have to be subtracted from the weight matrix $\mathbf{W}$ which we represent as a set of encrypted diagonals (see *Section 4.1.2*).
We propose the following algorithm for the homomorphic computation of the generalized diagonals of matrix

$$\mathbf{A} = \mathbf{u} \otimes \mathbf{v} = \mathbf{u}\mathbf{v}^{\mathbf{T}} = \begin{pmatrix} u_0v_0 & u_0v_1 & \cdots & u_0v_n \\ u_1v_0 & u_1v_1 & \cdots & u_1v_n \\ \cdots & & & \\ u_mv_0 & u_mv_0 & \cdots & u_mv_n \end{pmatrix},$$

where **u** and **v** are vectors of $m+1$ and $n+1$ elements respectively, encrypted as U and V.

Algorithm 4 Outer product of two vectors

```
1: $\mathsf{Vd} = \mathsf{Add}(\mathsf{V}, \mathsf{Rotate}(\mathsf{V}, -(n+1)))$
2: for $0 \leq i \leq n$ do
3:     $\mathsf{V1}_i = \mathsf{Rotate}(\mathsf{Vd}, i)$
4:     $\mathsf{A}_i = \mathsf{Rescale}(\mathsf{Mult}(\mathsf{U}, \mathsf{V1}_i))$
5: end for
```

Line 1-3. As with the matrix multiplication of *Algorithm 1*, we duplicate the encrypted input V and rotate it $n+1$ times to obtain staggered copies of it.

$$\begin{array}{lcllllllll}
\mathsf{V1}_0 & = & \{ & v_0 & v_1 & \cdots & v_n & x_0 & \cdots & 0 & 0 & \}_{(l)} \\
\mathsf{V1}_1 & = & \{ & v_1 & v_2 & \cdots & v_0 & v_1 & \cdots & 0 & v_0 & \}_{(l)} \\
\cdots \\
\mathsf{V1}_n & = & \{ & v_n & v_0 & \cdots & v_{n-1} & v_n & 0 & \cdots & v_{n-1} & \}_{(l)}
\end{array}$$

Line 4. We simply multiply each rotated ciphertext $\mathsf{V1}_i$ for U in order to obtain the i-th diagonal of the outer product A_i. The 0 values of U mask out only $m+1$ elements in the resulting ciphertext, assuming for this example $m < n$.

$$\begin{array}{lcllllllll}
\mathsf{A}_0 & = & \{ & u_0v_0 & u_1v_1 & \cdots & u_mv_m & 0 & \cdots & 0 & \}_{(l+1)} \\
\mathsf{A}_1 & = & \{ & u_0v_1 & u_1v_2 & \cdots & u_mv_{m+1} & 0 & \cdots & 0 & \}_{(l+1)} \\
\cdots \\
\mathsf{A}_n & = & \{ & u_0v_n & u_1v_0 & \cdots & u_mv_{m-1} & 0 & \cdots & 0 & \}_{(l+1)}
\end{array}$$

Algorithm 4 uses $\mathcal{O}(n)$ rotations and multiplications per input pair and has a depth of one multiplication.

4.2.2 Nesterov's Accelerated Gradient

The gradient descent step for the weights of a fully connected layer as formulated in (6) can be easily computed using *Algorithm 4*. However, basic gradient descent, especially in batch form, gives suboptimal results when used only for the < 30 epochs that we have available as it often gets stuck in a local mimimum. *Momentum* [58] has proved to be an effective tool in almost all scenarios to accelerate gradient descent and help it navigate through local optima. While many implementations of momentum are present in the literature, we focus on **Nesterov's Accelerated Gradient (NAG)** [68], which offers a good trade-off in terms of effectiveness and compatibility with homomorphic environments.
NAG introduces a velocity parameter $\mathbf{V}$ to "look ahead" at the next expected values of the parameters to optimize and calculate the gradient in that position. Using NAG the gradient descent update equation (14) for the weights of a fully connected layer becomes:

$$\mathbf{V}_{t+1} = \mu\mathbf{V}_t - \alpha\frac{\partial C}{\partial(\mathbf{W}_t + \mu\mathbf{V}_t)} \tag{8}$$

$$\mathbf{W}_{t+1} = \mathbf{W}_t + \mathbf{V}_{t+1} \tag{9}$$

where μ is a momentum term. In order to simplify these equations and keep the gradient calculations relative to the current weights instead of the predicted future position, the Keras deep learning library [37] proposes the following equivalent implementation of NAG:

$$\mathbf{V}_{t+1} = \mu\mathbf{V}_t - \alpha\frac{\partial C}{\partial\mathbf{W}_t} \tag{10}$$

$$\mathbf{W}_{t+1} = \mathbf{W}_t + \mu\mathbf{V}_{t+1} - \alpha\frac{\partial C}{\partial\mathbf{W}_t}. \tag{11}$$

For the purpose of homomorphic implementation, we can save up on one multiplication on ciphertexts by further rewriting (10) and (11) as:

$$(\mu\mathbf{V}_{t+1}) = \mu(\mu\mathbf{V}_t) - \frac{\alpha\mu}{k}\sum_{i=0}^{k}(\mathbf{y}_i - \mathbf{t}_i)\mathbf{x}_i^{\mathbf{T}} \tag{12}$$

$$\mathbf{W}_{t+1} = \mathbf{W}_t + (\mu\mathbf{V}_{t+1}) - \frac{\alpha}{k}\sum_{i=0}^{k}(\mathbf{y}_i - \mathbf{t}_i)\mathbf{x}_i^{\mathbf{T}} \tag{13}$$

effectively replacing parameter $\mathbf{V}$ with a new parameter $\mu\mathbf{V}$.
The following algorithm implements these equations together with (7) in order to compute a full iteration of the

backward step of a fully connected layer, given the results of the forward step $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$. We pay particular attention to the levels of the ciphertexts in question. Considering that the encryptions of $\mathbf{W}_t$, $\mathbf{b}_t$ and $\mu\mathbf{V}_t$ as well as the inputs $\mathbf{x}_i$ start at level l, the encryption of the output $\mathbf{y}$ already starts at level $l+1$ as the forward pass of *Algorithm 1* has a multiplicative depth of one.

Algorithm 5 Fully Connected layer backward pass using NAG

```
 1: for $0 \le i \le k$ do
 2:     $\mathsf{E}_i = \mathsf{Add}(\mathsf{Y}_i, -\mathsf{T}_i)$
 3:     $\mathsf{XA}_i = \mathsf{Rescale}(\mathsf{Mult}(\mathsf{X}_i, \frac{\alpha}{k}))$
 4:     $\mathsf{XAMu}_i = \mathsf{Rescale}(\mathsf{Mult}(\mathsf{X}_i, \frac{\alpha\mu}{k}))$
 5:     $\mathbf{GA}_i = \mathsf{E}_i \otimes \mathsf{AX}_i$
 6:     $\mathbf{GAMu}_i = \mathsf{E}_i \otimes \mathsf{AMuX}_i$
 7:     $\mathsf{EA}_i = \mathsf{Rescale}(\mathsf{Mult}(\mathsf{E}_i, \frac{\alpha}{k}))$
 8: end for
 9: $\mathbf{GA} = \mathsf{Add}(\mathbf{GA}_0, \ldots, \mathbf{GA}_k)$
10: $\mathbf{GAMu} = \mathsf{Add}(\mathbf{GAMu}_0, \ldots, \mathbf{GAMu}_k)$
11: $\mathbf{MuMuV} = \mathsf{Rescale}(\mathsf{Mult}(\mathbf{MuV}, \mu))$
12: $\mathbf{MuV} \leftarrow \mathsf{Add}(\mathbf{MuMuV}, -\mathbf{GAMu})$
13: $\mathbf{W} \leftarrow \mathsf{Add}(\mathbf{W}, \mathbf{MuV}, -\mathbf{GA})$
14: $\mathsf{B} \leftarrow \mathsf{Add}(\mathsf{B}, -\mathsf{E}_0, \ldots, -\mathsf{E}_k)$
```

Line 2. We compute the error between the predictions $\mathsf{Y}_{i\ (l+1)}$ and the target values $\mathsf{T}_{i\ (l)}$ as $\mathsf{E}_{i\ (l+1)}$.
Line 3-4. We compute the products between the inputs $\mathsf{X}_{i\ (l)}$ and the two coefficients $\frac{\alpha}{k}$ and $\frac{\alpha\mu}{k}$. We perform these operations separately and before the upcoming sums in order to obtain ciphertexts of level $l+1$ instead of $l+2$. While this is less efficient in terms of overall number of operations, it allows the algorithm to consume only one level of multiplication.
Line 5-6. Using *Algorithm 4* we compute two outer products that give us respectively the encryptions of $\frac{\alpha}{k}(\mathbf{y}_i - \mathbf{t}_i)\mathbf{x}_i^{\mathbf{T}}$ and $\frac{\alpha\mu}{k}(\mathbf{y}_i - \mathbf{t}_i)\mathbf{x}_i^{\mathbf{T}}$. The results are two matrices represented by two sets of $n+1$ encrypted diagonals, each being a ciphertext of level $l+2$.
Line 7. For the bias calculations we multiply the errors $\mathsf{E}_{i\ (l+1)}$ by $\frac{\alpha}{k}$, obtaining the encryptions of $\frac{\alpha}{k}(\mathbf{y}_i - \mathbf{t}_i)$ at level $l+2$.
Line 9-10. We sum together the results of Line 5-6 to obtain $\frac{\alpha}{k}\sum_{i=0}^{k}(\mathbf{y}_i - \mathbf{t}_i)\mathbf{x}_i^{\mathbf{T}}$ and $\frac{\alpha\mu}{k}\sum_{i=0}^{k}(\mathbf{y}_i - \mathbf{t}_i)\mathbf{x}_i^{\mathbf{T}}$, still as sets of $n+1$ ciphertexts at level $l+2$.
Line 11. We compute $\mu(\mu\mathbf{V}_t)$ by multiplying $\mu\mathbf{V}_t$ by μ, giving us the matrix of ciphertexts $\mathbf{MuMuV}_{(l+1)}$. Note that $\mu\mathbf{V}_t$ is also a matrix stored through its encrypted diagonals.
Line 12. We update $\mu\mathbf{V}_t$ with the sum described in (12). Because of $\mathbf{GAMu}_{(l+2)}$ the result is a set of level $l+2$ ciphertexts too.
Line 13. We update $\mathbf{W}$ with the sum described in (13), which also results in a set of level $l+2$ encrypted diagonals because of $\mathbf{MuV}_{(l+2)}$ and $\mathbf{GA}_{(l+2)}$.
Line 14. We subtract all the scaled errors from the bias to update it according to (7). This results in a single level $l+2$ ciphertext.

Algorithm 5 uses $\mathcal{O}(k \cdot n)$ multiplications and has a depth of one multiplication starting from the forward step.

In the case of *Algorithm 5* we need to make a distinction if we are using batched ciphertexts that encrypt multiple input rows at once. The additions at *Line 9-10* need to sum together all input rows of the operands and not just the ciphertexts themselves. We can achieve this by applying an $\mathsf{Accumulate}$ procedure to all batched ciphertexts before said additions. $\mathsf{Accumulate}$ takes as input a ciphertext X of r rows of size $s = \frac{N}{2r}$ and adds it recursively to its rotations:

$$\mathsf{X} \leftarrow \mathsf{Add}(\mathsf{X}, \mathsf{Rotate}(\mathsf{X}, s \cdot 2^i)), \text{ for } i = 0, \ldots, (\log_2 r) - 1.$$

The result is a ciphertext where each row contains a copy of the sum of all rows of the original input. The chosen approach is faster than simply rotating the ciphertext by each row and adding it to the original, which would take $\mathcal{O}(r)$ rotations instead of $\mathcal{O}(\log r)$.

4.2.3 Backpropagation

In a typical feed-forward network the gradient descent algorithm is extended to train all layers at once through the use of *backpropagation*. With backpropagation, the loss computed at the final output of the network is

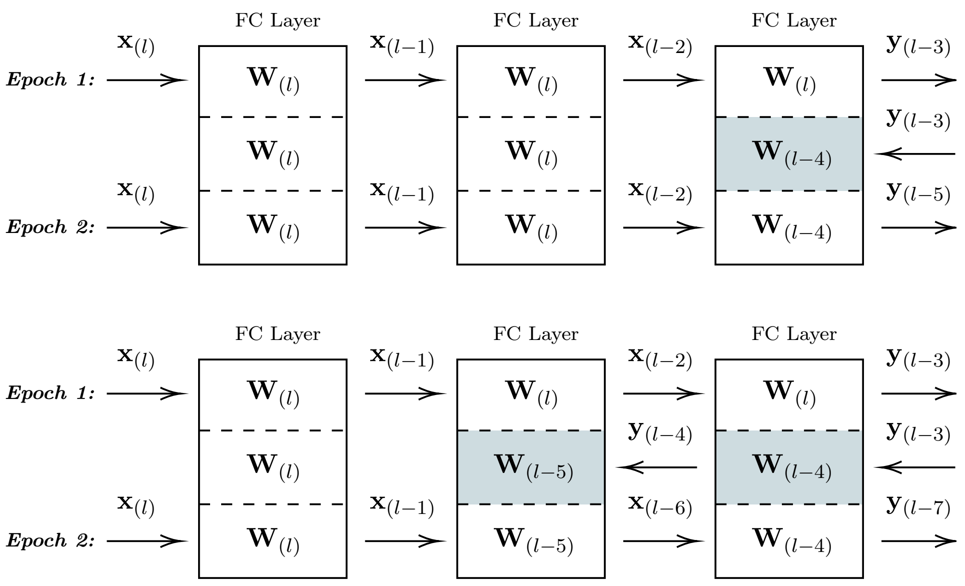


Figure 4: Comparison of backpropagation of gradient descent between one and two fully connected layers. Backpropagating the error through multiple layers uses up more multiplications both on the backward step and on the forward step of the following epoch. Even with a backward step of multiplicative depth of 1, each epoch of training consumes 2 levels of multiplication in the first case and 4 in the second case, effectively halving the amount of epochs available if opting to backpropagate through two layers.

propagated backwards through the previous layers, iterating the gradient descent algorithm using the derivative chain rule. For instance, after updating the weights of the final layer L of our network using (14), layer $L-1$ can be updated as:

$$\mathbf{W}_{t+1}^{(L-1)} = \mathbf{W}_{t}^{(L-1)} - \alpha \frac{\partial C}{\partial \mathbf{W}^{(L-1)}} = \mathbf{W}_{t}^{(L-1)} - \alpha \frac{\partial C}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{W}^{(L-1)}} \tag{14}$$

considering that in a feed-forward network the input of a layer $\mathbf{x}^{(L)}$ is the output of the previous layer $\mathbf{y}^{(L-1)}$.
In order to implement this update rule under homomorphic restrictions we need to be able to compute $\frac{\partial C}{\partial \mathbf{x}^{(L)}}$, which for fully connected layers specifically requires the transposition of the weight matrix $\mathbf{W}^{(L)}$. Matrix transposition is not easy to compute using the diagonal matrix encoding adopted so far and would require one extra multiplication per layer. In addition, backpropagating the error through multiple layers would increasingly lower the level of their weight ciphertexts, which in turn would bring down the level of the inputs at the following epoch. *Figure 4* illustrates how even with a backward step that consumes only one level of multiplication per layer, training two fully connected layers doubles the amount of multiplications required per epoch.
As such, we opt not to implement backpropagation for our encrypted training phase, and instead focus on training only the final layer of the network for as many epochs as possible.

4.3. Trainable model

Thanks to the implementation of homomorphic gradient descent for fully connected layers described in *Section 4.2*, we can now optimize the PINPOINT architecture to maximize the efficacy of this training phase.
As stated in *Section 4.2.3* we choose to apply training only to the final fully connected layer of the model, meaning that the rest of the network needs to be trained on plain data first and then encrypted. This goes against the original privacy-preserving mission of our encrypted network and its compatibility with fully transparent Cloud-based training. Users should not be required to send over plain reference data as there might be situations where only sensitive data is available.
Ideally, we want to pre-train a base network on plain, publicly available data that might not even be strictly related to a user's specific task, and then use the user's private encrypted data to fine-tune the results. The approch of taking an existing learner and applying the knowledge it gained to a different task is referred to as **transfer learning**. The resulting model should be robust enough to give accurate results with very few epochs of fine-tuning of a single fully connected layer, and work as a general-purpose solution that does not require any assumptions on the specific private training data.

4.3.1 Stacking model

Because no knowledge on the type of training data can be used, our approach is to train multiple transfer models on different sets of plain time series data and then combine their predictions on encrypted data using a modified **model stacking** approach. Stacking [75] is an ensemble machine learning technique that combines multiple types of models to create a more powerful architecture:

- Multiple *level-0 models* make a prediction on the training data.
- A single *level-1 model* or *generalizer* uses the outputs of the level-0 models as input data and learns how to best combine them.

The level-1 model is trained on the predictions of the level-0 models, which in practice act as feature extractors. By using level-0 models trained on different types of data distributions, the generalizer can then select which ones best match the actual target outputs.

We can adapt this idea for our purposes by using multiple PINPOINT models as the level-0 models, and a trainable fully connected layer as the level-1 generalizer. A diagram of our trainable stacking model implemented using the PINPOINT architecture, called **PINStack**, is shown in *Figure 5*.
Given a time series forecasting task of forecast horizon h, n plain time series datasets D_i used for transfer learning, and an encrypted time series dataset Z for fine-tuning, a PINStack model can be constructed and trained on encrypted data with the following steps:

1. We train in parallel n PINPOINT-1CONV transfer models $\varphi_i^{(0)}$ on the plain datasets D_i. The PINPOINT-1CONV version of the model is chosen in order to minimize the number of layers and, in turn, the number of multiplications required. Because we only use one convolutional layer, we also swap the flatten and square layers from the original architecture seen in *Figure 3* in order to speed up computation. Note that this training is done on plain versions of the models and does not require any encryptions or consume ciphertext levels.
2. We remove the final fully connected layer of each level-0 model $\varphi_i^{(0)}$ in order to increase the number of inputs for the generalizer. We denote with d the new output size of the models, corresponding to the output size of the second-to-last fully connected layer.
3. We encode the parameters of the models $\varphi_i^{(0)}$. These parameters only depend on the plain transfer data and will no longer be modified by the encrypted data, as such they can simply be encoded without any privacy loss.
4. We create a new feed-forward model $\varphi^{(1)}$ composed of a flatten layer and a fully connected layer. The fully connected layer has input size $n \cdot d$ and output size h. It is initialized using the parameters of the removed fully connected layers, which are then encrypted. This is used as our level-1 generalizer.
5. We run the training dataset Z through a forward pass of each level-0 model $\varphi_i^{(0)}$. The outputs are n sets of predictions of size d for each input. Even though the level-0 models were not trained on this dataset, we can still use their inner representation of this data to learn its real distribution.
6. We stack together the outputs of all models by running them through the flatten layer of $\varphi^{(1)}$, obtaining a new set of $|Z|$ inputs of size $n \cdot d$.
7. We use the new encrypted inputs to homomorphically train the fully connected layer of $\varphi^{(1)}$.

To summarize, the level-0 models are each trained on a different plain dataset D_i (named the *transfer datasets*) and then used to pre-process the encrypted dataset Z (named the *training dataset*), so that the level-1 generalizer can be trained to produce the final predictions.
To perform inference on a PINStack model, new encrypted time series inputs are first passed through all level-0 models $\varphi_i^{(0)}$ in parallel and through the new flatten layer. The input obtained this way is compatible with the level-1 generalizer, which can then output the final prediction.

4.3.2 Model details

The *multiplicative depth* necessary to run encrypted training and then encrypted inference on the PINStack model illustrated in *Figure 5* is:

- 5 levels for initial the forward pass of the $\varphi_i^{(0)}$ models, as we run for each layer the forward pass algorithms implemented in *Section 4.1* that only require one level of multiplication each;
- 1 level for the forward pass of the flatten layer of $\varphi^{(1)}$;
- 2 levels for each epoch of training of the fully connected layer of $\varphi^{(1)}$, which requires one multiplication on the input for the forward pass and one on the weights for the backward pass;

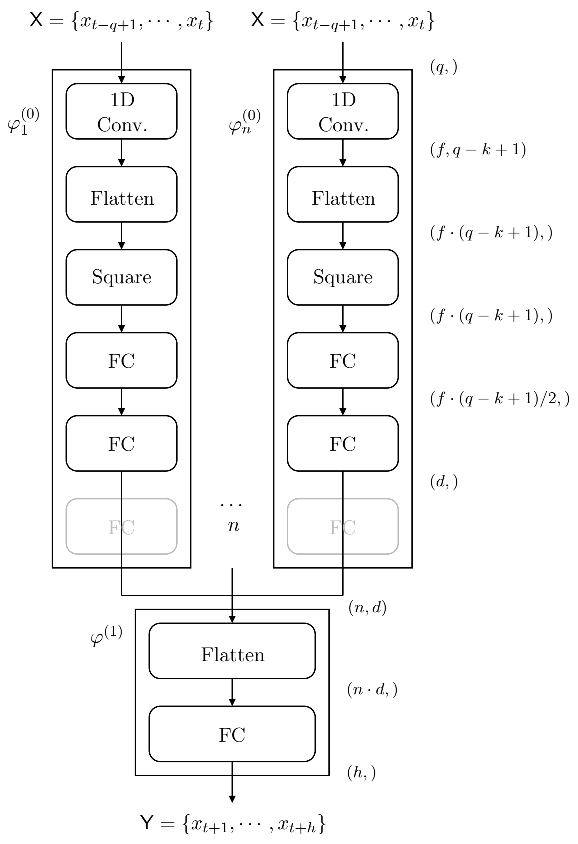


Figure 5: Diagram of the PINStack model. An example of a forward pass is reported with the size of the input and output of each layer.

– 1 level for the final forward pass of the fully connected layer of $\varphi^{(1)}$ performed during inference.

For e epochs of training, this results in an overall multiplicative depth of $L = 7 + 2 \cdot e$.

Because of the very limited number of epochs of training allowed, the *initialization* of the trainable parameters (namely the weights $\mathbf{W}^{(1)}$ and bias $\mathbf{b}^{(1)}$ of the generalizer $\varphi^{(1)}$) plays a key role in the final performance of the model. As anticipated this initialization is done using the parameters of the n final fully connected layers that are cut from the level-0 models.

Given a forecast horizon h, the weights $\mathbf{W}_i^{(0)}$ of these layers are matrices of shape $h \times d$, while their biases $\mathbf{b}_i^{(0)}$ are vectors of size h. The weigth matrix $\mathbf{W}^{(1)}$ and bias $\mathbf{b}^{(1)}$ can be initialized to:

$$\mathbf{W}^{(1)} = \begin{pmatrix} \frac{1}{n}\mathbf{W}_1^{(0)} & \cdots & \frac{1}{n}\mathbf{W}_n^{(0)} \end{pmatrix}, \ \mathbf{b}^{(1)} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{b}_i^{(0)},$$

resulting in a $h \times (n \cdot d)$ matrix and a vector of size h respectively. The result is that the fully connected layer is initialized to output an average of the predictions of the original level-0 models, which acts as a good unbiased starting point for the following fine-tuning.

Using this approach, no assumptions are required on the plain transfer datasets D_i other than that they should have the same input size and forecast horizon as the encrypted training dataset Z. This limitation is easy to overcome in an actual Cloud-based implementation, as TCNNs for time series prediction tasks typically set the input sequence length according to the time resolution of the target. A consistent time resolution and input size allows the level-0 models to capture seasonal trends in the data and then identify similar patterns in the final encrypted dataset. A provider can exploit this in the Cloud scenario by preparing a collection of PINStack models that each use data with a different time resolution for transfer learning, and letting users select which ones they want to train with their encrypted data.

Of course, results can be improved if a user has access to plain data similar to their private encrypted data (for instance historical data that is no longer subject to confidentiality). We can use this plain data to prepare the trasfer datasets D_i and initialize a PINStack model using them, then fine-tune the models with the encrypted data. The plain level-0 models will likely already have good prediction performance on the new encrypted dataset, resulting in even better accuracy after training.

5. Implementation

The proposed PINStack time series prediction model is implemented in the Python library **PyCNN-CKKS**. The library includes all algorithms previously described and offers a framework for a general-purpose encrypted network with privacy-preserving training capabilities. This section describes the overall structure of the new library, as well as providing some notes on its usage.

5.1. External libraries

PyCNN-CKKS uses the following external libraries:

- Pyfhel [41]: Homomorphic encryption library that supports the CKKS encryption scheme. It is a Python wrapper for the open-source C++ Microsoft SEAL [64] library.
- numpy [39]: Scientific computation.
- PyTorch [55]: Open source machine learning framework.

In addition, the following supporting libraries are useful to run the final experiments:

- pandas [74]: Data analysis library. Used for database handling.
- scikit-learn [57]: Open source library for machine learning tools. Used for data processing.

5.2. Structure

PyCNN-CKKS is a Python library built on top of the implementation of the CKKS scheme from the Pyfhel library [41]. It provides tools for the creation and training of privacy-preserving feed-forward machine learning models compatible with the network structure of the PyTorch library [55]. The library only implements the layers necessary for our proposed architecture, namely:

– Fully Connected layers (corresponding to `torch.nn.Linear`);
– 1-D Convolutional layers (corresponding to `torch.nn.Conv1d`);
– Square layers (our custom activation layer);
– Flatten layers (corresponding to `torch.nn.Flatten`).

PyCNN-CKKS is divided into the following sub-packages:

- *he*: Contains the `Pyfhel2d` class, which implements all HE functionalities of the CKKS scheme. It inherits the base `Pyfhel.Pyfhel` class (from the Pyfhel library) by adding support for matrix encryption and encrypted matrix operations as described in *Section 4.1.2*, as well as useful operations related to ciphertext batching.
- *linear*: Contains the `EncLinear` class, which implements an encrypted fully connected layer together with its forward pass and backward pass algorithms as described in *Section 4.1.2* and *Section 4.2.2* respectively.
- *convolutional*: Contains the `EncConv1d` class, which implements an encrypted 1-D convolutional layer together with its forward pass as described in *Section 4.1.3*.
- *functional*: Contains the `EncSquare` and `EncFlatten` classes, which implement encrypted square and flatten layers together with their forward pass as described in *Section 4.1.4* and *Section 4.1.5* respectively.
- *model*: Contains the `EncSequential` class, which takes a `torch.nn.Sequential` network (from the PyTorch library) that only contains the previously implemented layers and builds an HE-compatible version of the network using said layers. The network can include training functionalities on its last fully connected layer.
  Also contains the `EncStacking` class specific to our proposed architecture, which groups together multiple `EncSequential` models to create the stacking model described in *Section 4.3*

The main methods of each of these classes are highlighted here:

◦ A homomorphic encryption context can be generated by the `Pyfhel2d` class using the `contextGen` method inherited by `Pyfhel.Pyfhel`, which we call through:

```
Pyfhel2d.contextGen(scheme='ckks',
                    n=N,
                    qi_sizes=[int_scale] + [bits_scale]*L + [int_scale],
                    scale=2**bits_scale)
```

Here `N` is the ciphertext polynomial degree N, `bits_scale` is the bit-length of the scale Δ (which will be approximated by prime integers), `int_scale` is the bit-length of final coefficient modulus q_0, and `L` is the maximum number of multiplications L.

- An instance of each implemented encrypted layer is created in a similar way to the corresponding PyTorch modules, as:

  ```
  EncLinear(HE, row_size, weight, bias, enc_mode)
  EncConv1d(HE, row_size, weight, bias, in_size, enc_mode)
  EncSquare(HE)
  EncFlatten(HE, row_size, in_size)
  ```

  All layers are given an instance `HE` of the `Pyfhel2d` class with its context already initialized, which will handle their homomorphic operations, as well as a `row_size` used for batched ciphertexts. `EncLinear` and `EncConv1d` are given an initial plain `weight` and `bias`, which will either be encrypted or encoded according to parameter `enc_mode`. `EncConv1d` and `EncFlatten` also need to specify an additional `int_size` parameter, the size of their input vectors. This parameter can typically be inferred from the inputs in a plain execution but cannot be known from encrypted ciphertext vectors of fixed size $N/2$.
  Each of these classes implements their respective forward pass algorithm on an encrypted input sequence `x` in the function:

  ```
  layer.forward(x)
  ```

  `EncLinear` has the additional function:

  ```
  EncLinear.forward_backward(X, Y, num_inputs, learning_rate, momentum)
  ```

  The function performs encrypted training on the layer by implementing both the forward and backward pass algorithms and executing them on the set of input-target sequence pairs `(X, Y)`.

- The model classes are initialized as:

  ```
  EncSequential(HE, model, row_size, seq_length, trainable)
  EncStacking(HE, lv0_models, lv1_model)
  ```

  `EncSequential` takes as input a `torch.nn.Sequential model` and encrypts it, specifying if the last layer should be `trainable`. `EncStacking` instead takes a collection of `EncSequential lv0_models` and a single `EncSequential lv1_model` to construct a stacking model. Both classes can perform a single forward pass on an input `x` through all their layers as:

  ```
  model.forward(x)
  ```

  To train them on labeled inputs `(X, Y)` for `num_epochs` iterations the following function is used:

  ```
  model.train(X, Y, num_inputs, num_epochs, learning_rate, momentum)
  ```

  The `train` function of `EncStacking` implements the training framework described in *Section 4.3.1* by first running the inputs `X` through each of the `lv0_models`, and then using these new inputs to train the `lv1_model`.

5.3. Usage

The `EncLinear` and `EncConv1d` classes are constructed from a set of plain weights and biases. The library assumes that all models to encrypt have already been trained on plain data, which can be easily carried out through PyTorch. In addition, both `EncLinear` and `EncConv1d` layers support an *encrypted* and *encoded* mode, where the latter can be used for layers that do not require encrypted training (and as such do not need to be modified according to any encrypted data) in order to speed up computation.

The encryption parameters for the CKKS scheme are chosen at initialization of the `Pyfhel2d` class, which is used to build all encrypted layers and models and carry out their homomorphic operations. These encryption parameters must be shared with any user who wants to access a model encrypted with said instance of `Pyfhel2d`. Indeed, the user must have access to an equivalent `Pyfhel2d` instance in order to encrypt their data into ciphertexts compatible with the model weights and biases. Once these encryption parameters are known by both parties, the user can generate a pair of public and secret keys and share their public key with the machine

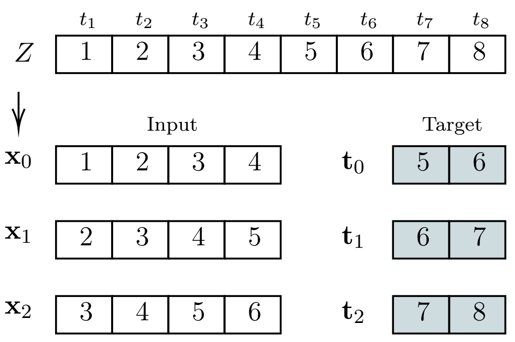


Figure 6: Example of the creation of a supervised training dataset starting from a time series dataset Z. Here, the sequence length is $q = 4$ and the forecast horizon is $h = 2$. A dataset of $|Z|$ time series observations can be used to create a training dataset of $|Z| - (q + h - 1)$ input-target sequence pairs.

learning provider. The same public key can then be used to encrypt the weights of the model that need to be fine-tuned, with the provider no longer able to access their plain values without the secret key.

We also need to define how the inputs themselves are presented to the network. `EncSequential` and `EncStacking` models have a set input size equal to a sequence length q and output size equal to a forecast horizon h, and will learn to predict h upcoming time series values given the previous q observations. The models are trained in a supervised manner and as such require that training data comes in the form of vector pairs: an input sequence of q time series observations and a target sequence corresponding to the next h observations. The data should also be normalized first in order to avoid issues with the scale parameters of CKKS, and it is important that the same scaling is used across all transfer and training datasets for the learning process to be effective.
In the case of private data, these operations must necessarily be carried out by users on plain data before encryption, meaning locally before sending it over to the chosen model when considering a Cloud-based setting. Starting from any time series dataset, a user can easily construct a training dataset (either for the plain transfer learning of level-0 models or for the encrypted fine-tuning of the generalizer) by splitting it into pairs of inputs of length q and labels of length h, as illustrated in *Figure 6*. All data from the same dataset can also be normalized together first, keeping the scaling consistent between samples and making it easy to rescale back the outputs received by the encrypted network.

Note that because we are training on encrypted data we cannot use a validation set, as the model has no knowledge of the actual values of its outputs. This should not be a concern in our case however, as with the extremely limited number of training epochs we have available we do not risk overfitting the training dataset in any significant way.

6. Experimental results

In this section the performance of the PyCNN-CKKS implementation of the PINStack time series prediction model is tested in two scenarios that exemplify two different realistic use cases:

1. *Similar transfer data.* A user wants to train a PINStack model on an encrypted dataset Z and gives it access to other similar plain datasets D_i as well for transfer learning.
2. *Unrelated transfer data.* A user wants to train a PINStack model on an encrypted dataset Z and only has access to sensitive data.

The second case is of particular interest to our mission for a privacy-preserving Cloud-based solution, as it does not demand from the user any information about their data and completely preserves the privacy of their task. The goal of these experiments is to show that the privacy-preserving training capabilities of the model, although limited to a single fully connected layer, are effective at producing similar results to other privacy-violating architectures commonly used in the field.

6.1. Datasets

In order to test the two scenarios previously described, a collection of publicly available univariate time series datasets is used to create the transfer and training datasets. In the first case, both the plain datasets for training of the level-0 models and the encrypted dataset for training of the level-1 generalizer come from similar sources. In the second case, a collection of unrelated datasets with the same time resolution is used as transfer data for training on a different encrypted dataset.
Each dataset is split into a train dataset used for training (either of the level-0 models or the generalizer) and a test dataset used for inference to evaluate their performance. When training level-0 models on plain data, a further 5% of the train datasets is kept for validation in order to help prevent overfitting.
All datasets are scaled before encryption to the range $[-1, 1]$ through the `sklearn.preprocessing.MinMaxScaler` from the scikit-learn library. The same reversed scaler is applied to the decrypted outputs of the network to retrieve the final predictions.

6.1.1 Similar transfer data

The first experiment is run on a collection of *Western Europe Power Consumption* datasets [4]. These datasets describe the daily overall power consumption of several European countries, in GigaWatts, between 2015 and 2020. They are each split into a train dataset with values from 01-01-2015 to 20-01-2020 (1846 observations) and a test dataset with values from 21-01-2020 to 31-08-2020 (224 observations).
In particular, the following power consumption datasets are chosen for encrypted training:

- *France*: Data range 1435.904.
- *Luxembourg*: Data range 66.577.
- *Norway*: Data range 620.595.
- *Portugal*: Data range 75.975.

In each of these cases the same two datasets are used exclusively for transfer learning:

- *Germany*: Data range 2806.167.
- *Italy*: Data range 657.905.

6.1.2 Unrelated transfer data

The following nine unrelated public datasets have been selected for the second experiment. All datasets have the same time resolution, with observations being made on a monthly basis. This is necessary in order to better capture any information about seasonal trends in the level-0 models and choose a consistent input sequence size across all datasets. Outside of this restriction, the datasets are specifically chosen to be different in nature from each other and offer a wide range of distribution types and trends in the data.

- *Airline Passengers* [2]: Monthly total passengers of an airline company, from Jan-1949 to Dec-1960. Train dataset from Jan-1949 to Dec-1959 (126 observations), test dataset from Jan-1960 to Dec-1960 (12 observations). Data range 518.
- *Beer Production* [1]: Monthly beer production in Australia, in units of volume, from Jan-1956 to Aug-1996. Train dataset from Jan-1956 to Dec-1993 (444 observations), test dataset from Jan-1994 to Aug-1996 (32 observations). Data range 153.

- *India Monthly Rainfall* [6]: Monthly rainfall normal data in India, in millimeters, from Jan-1901 to Dec-2002. Train dataset from Jan-1901 to Dec-1992 (1104 observations), test dataset from Jan-1993 to Dec-2002 (120 observations). Data range 13352.7.
- *Lake Superior Water Level* [3]: Monthly mean lakewide average water level of Lake Superior (USA), in meters, from Jan-1918 to Dec-2021. Train dataset from Jan-1901 to Dec-2015 (1176 observations), test dataset from Jan-2016 to Dec-2021 (72 observations). Data range 1.19.
- *Monthly Sunspots* [7]: Monthly mean number of observed sunspots, from Jan-1749 to Aug-1983. Train dataset from Jan-1749 to Dec-1981 (2796 observations), test dataset from Jan-1982 to Aug-1983 (20 observations). Data range 253.8.
- *New Home Sales* [8]: Monthly new single-family houses sold in the United States, in thousands of units, from Jan-1963 to Dec-2019. Train dataset from Jan-1963 to Dec-2015 (636 observations), test dataset from Jan-2016 to Dec-2019 (48 observations). Data range 107.
- *Sales For Retail* [8]: Monthly sales for retail and food services in the United States, in millions of dollars, from Jan-1992 to Dec-2019. Train dataset from Jan-1992 to Dec-2017 (312 observations), test dataset from Jan-2018 to Dec-2019 (24 observations). Data range 439784.
- *Total Energy Consumption* [9]: Monthly total energy consumption in the United States, in quadrillion BTU, from Jan-1973 to Dec-2019. Train dataset from Jan-1973 to Dec-2015 (516 observations), test dataset from Jan-2016 to Dec-2019 (48 observations). Data range 4.2286.
- *Unemployment Rate* [5]: Monthly unemployment rate in the United States, percentage, from Jan-1948 to Dec-2019. Train dataset from Jan-1948 to Dec-2015 (816 observations), test dataset from Jan-2016 to Dec-2019 (48 observations). Data range 9.0.

Each dataset is selected once for encrypted training using the remaining eight datasets for transfer learning.

6.2. Parameter selection

Both experiments use the same underlying encryption scheme, which we initialize using the following parameters:

- Security level $\lambda = 128$ (default value for the this implementation of CKKS).
- Polynomial degree $N = 2^{15}$. This is the maximum possible value for N allowed by the SEAL library implementation of the CKKS scheme used by Pyfhel. Maximizing this parameter is necessaryan to allow a sufficient number of operations on ciphertexts.
- Bit-size of scale $\Delta = 32$. This results in a precision of 32 bits for the decimal part of the encrypted values.
- Bit-size of final coefficient modulus $q_0 = 40$. This results in a precision of 8 bits for the decimal part of the encrypted values.
- Maximum ciphertext level and maximum multiplicative depth $L = 25$.
- Row size $r = 1024$. This allows each ciphertext (encrypting up to $N/2 = 2^{14}$ values) to store 16 batched vectors of size $< 1024/2 = 512$ in order to leave room for padding.

Because $L = 7 + 2 \cdot e$ (see *Section 4.3.2*), a multiplicative depth of 25 lets us train our proposed model for only 9 epochs. While this number could be slightly increased at the cost of some accuracy, our results show that it is sufficient to obtain good transfer learning performance.
For this training phase, a learning rate of 0.01 and a momentum parameter of 0.99 are chosen across all experiments. A high NAG momentum parameter in particular is the key factor in making training possible under these harsh conditions, as test runs without a momentum term showed significantly lower performances.

The PINStack model follows the structure illustrated in *Figure 5* for a sequence length q and a forecast horizon h. It is implemented through an instance of the `EncStacking` class, built from the following sub-models:

- *Level-0 models.* For each of the n transfer learning datasets, one `EncSequential` model implementing a PINPOINT-1CONV architecture with the following layers:
  - `EncConv1d` layer: `in_size` $= q$, `weight` $=$ matrix of size $32 \times 1 \times 3$ (input channels $= 1$, number of filters $= 32$, kernel size $= 3$), `bias` $=$ vector of length 32.
  - `EncFlatten` layer: `in_size` $= (q - 3 + 1) = (q - 2)$. Flattens the $32 \times (q - 2)$ outputs of the `EncConv1d` layer.
  - `EncSquare` layer.
  - `EncLinear` layer: `weight` $=$ matrix of size $16(q-2) \times 32(q-2)$ (input size $= 32(q-2)$, output size $= 16(q-2)$), `bias` $=$ vector of length $16(q-2)$.
  - `EncLinear` layer: `weight` $=$ matrix of size $d \times 16(q-2)$ (input size $= 16(q-2)$, output size $= d$), `bias` $=$ vector of length d.
  - `EncLinear` layer: `weight` $=$ matrix of size $h \times d$ (input size $= d$, output size $= h$), `bias` $=$ vector

of length h. Only used to train on plain transfer data, then cut from the model.

- *Level-1 model.* Single `EncSequential` generalizer model with the following layers:
  - `EncFlatten` layer: `in_size` $= d$. Flattens the $n \times d$ outputs of the level-0 `EncSequential` models.
  - `EncLinear` layer: `weight` $=$ matrix of size $h \times n \cdot d$ (input size $= n \cdot d$, output size $= h$), `bias` $=$ vector of length h, `trainable` $=$ `True`.

The specific model parameters for each experiment are:

1. **Similar transfer data**:
   - Sequence length $q = 2 \cdot$ seasonality $= 14$ (for daily data).
   - Forecast horizon $h = 7$.
   - Number of level-0 models $n = 2$.
   - Level-0 output size $d = 64$.
2. **Unrelated transfer data**:
   - Sequence length $q = 2 \cdot$ seasonality $= 12$ (for monthly data).
   - Forecast horizon $h = 6$.
   - Number of level-0 models $n = 8$.
   - Level-0 output size $d = 32$.

All of these model parameters have been mostly selected through trial and error, with hyperparameter optimization not being a main focus for our goals.

6.3. Model performance

In both experiments, plain transfer data is first used to individually train privacy-violating versions of the level-0 models through the PyTorch library. These models are then passed to an instance of the `EncStacking` class, which encrypts their weights and initializes the generalizer. The `train` function of `EncStacking` is used to train the generalizer on the specific encrypted train dataset, and the `forward` function if finally used to produce the final predictions from the encrypted test dataset.
Testing was performed on a 48-core CPU run at 3GHz with 384GB of memory, but no multiprocessing was used.

The performance of the PINStack model is evaluated on the accuracy of its predictions by comparing the forecast and the true target values. The error metric of choice, which we are looking to minimize, is the *Mean Absolute Error (MAE)* $\frac{\sum_{i=1}^{n} |y_i - t_i|}{n}$, where y_i is a single value prediction returned by the model (corresponding to one of the values of each output sequence), t_i is the true target value and n is the total number of predictions.

Table 2 shows the performance of the PINStack model with similar and unrelated transfer data and compares it to other state-of-the-art time series prediction models trained on the same tasks. The prediction MAE for the following models is reported:

- *PINPOINT*: Base PINPOINT-1CONV model with analogous parameters to the PINStack model, run in this case on plain data.
- *Naive*: Naive forecast model, which simply uses the last know time series value for the forecast. As such, given an input sequence $\mathbf{x} = (x_{t-q+1}, \dots, x_t)$, it will output the prediction $\mathbf{y} = (x_{t-h+1}, \dots, x_t)$. This is used as a baseline reference for performance.
- *ARIMA*: AutoRegressive Integrated Moving Average (ARIMA) model, implemented and trained using the pmdarima Python library [66] with default settings.
- *Prophet*: Facebook Prophet model [71], implemented and trained using its official Python library with default settings.

Note that these are all privacy-violating models, meaning that plain versions of the datasets were used for training and inference, while our PINStack experiments were executed on encrypted data.
In addition, a privacy-violating version of PINStack (that runs the same operations on plain data instead of encrypted data) was compared to the privacy-preserving model in order to better highlight its strengths and weaknesses:

- *PINStack P.V. transfer*: Obtained by training the level-0 models of PINStack on the transfer data, constructing the model and then running it on plain data with no further fine-tuning. This is a reference point for the performance of the model before training, using only the initialization provided by the level-0 models.
- *PINStack P.V. training*: Obtained by training a privacy-violating version of PINStack on plain data. This removes any approximation errors given by CKKS encryption from the training results.

dataset	*range*	*P.P.*	PINStack *P.V.*		*P.V.*			
		PINStack	*transfer*	*training*	PINPOINT	Naive	ARIMA	Prophet
France	1435.904	83.09	126.89	71.50	63.50	95.21	89.18	84.87
Luxembourg	66.577	3.378	3.356	3.376	4.018	4.522	4.019	4.752
Norway	620.595	16.31	16.76	15.27	15.77	18.36	17.07	18.31
Portugal	75.975	6.912	6.878	6.931	6.228	10.332	8.405	7.729

(a) Similar transfer data.

dataset	*range*	*P.P.*	PINStack *P.V.*		*P.V.*			
		PINStack	*transfer*	*training*	PINPOINT	Naive	ARIMA	Prophet
Airline Passengers	518	34.44	57.83	24.29	35.47	66.67	54.66	34.08
Beer Production	153	12.14	16.46	11.42	8.49	38.00	20.31	9.64
India Monthly Rainfall	13352.7	1160.7	1590.7	1059.3	938.5	2440.3	2241.5	830.6
Lake Superior Water Level	1.19	0.0857	0.1067	0.0834	0.0719	0.0988	0.1180	0.2439
Monthly Sunspots	253.8	34.74	40.16	34.29	22.40	31.70	24.23	27.15
New Home Sales	107	5.618	4.786	5.427	5.015	12.354	6.077	26.639
Sales For Retail	439784	18340.4	45702.0	22361.6	10785.2	44849.3	27872.5	12422.0
Total Energy Consumption	4.2286	0.2585	0.3288	0.2495	0.2159	0.6525	0.4510	0.2498
Unemployment Rate	9.0	0.3541	0.4606	0.3582	0.2430	0.3625	0.4842	2.4809

(b) Unrelated transfer data.

Table 2: Prediction MAE of PINStack in the two experiments, compared with other time series prediction models. The PINStack model is run in privacy-preserving (*P.P.*) mode, while the other state-of-the-art references are run in privacy-violating (*P.V.*) mode. PINStack *P.V.* results are reference points obtained on plain data.

The results show that the performance of the model on encrypted data is at the very least comparable with (and often better than) the other privacy-violating models, both with and without having access to similar transfer learning data. Specifically, the following observations can be made:

– PINStack convincingly and consistently produces forecasts more accurate than the simple Naive model and shows results comparable with the privacy-violating version of PINPOINT, which has the lowest MAE across almost all experiments.

– As expected, the experiments on similar transfer data show more consistent results than the ones using unrelated datasets. Indeed the PINStack *P.V. transfer* MAEs indicate that the transfer learning models by themselves were enough to obtain good performance on the encrypted dataset even without any fine-tuning. However, even in the case of unrelated transfer data the training phase is proved to be effective at correcting an initial low transfer performance.

– The choice of the learning rate proves to be particularly challenging in the privacy-preserving environment of PINStack. As no information about the training dataset is known and the training algorithm is run for only very few epochs, a fixed and quite high base learning rate of 0.01 is employed in all experiments in order to speed up the training process. However, in cases where the starting transfer performance PINStack *P.V. transfer* is already good (such as with the *New Home Sales* dataset), the MAE decreases slightly after training as the network weights are moved away from the initial local minimum. This is a necessary trade-off of the privacy-preserving setting, and the final MAE obtained in these cases is still very good regardless.

– In almost all situations, the privacy-preserving PINStack MAE is closer to its starting PINStack *P.V. transfer* value than its privacy-violating counterpart PINStack *P.V. training*. This is likely a consequence of the approximation errors introduced by CKKS encryption, which somewhat slow down training. The difference however is mostly negligible when compared to the other prediction models, and in turn signifies that the approximation error introduced by the encryption scheme is not a major point of concern even with high settings.

- The chosen PINStack setting is not a perfect solution for all datasets, as the model was unable to beat the Naive performance for the *Monthly Sunspots* dataset. It is possible that this dataset is paricularly incompatible with the other transfer datasets, making it difficult to learn an accurate representation of the data from their pre-processed inputs. This issue can be mitigated in a real Cloud-based implementation, where more transfer datasets can be used or multiple separate models can be trained in parallel on different types of transfer data, giving users the option to choose which one is more suited to their task.

6.4. Time and memory occupation

The main drawback introduced by the application of homomorphic encryption to the training process comes in the form of its execution time and memory usage. The increase in computational requirements is a necessary consequence of the homomorphic encryption scheme and needs to be managed if the privacy of the data is to be preserved.
The number of multiplications used by PINStack requires maximized settings for the CKKS encryption scheme, but as a consequence each encrypted ciphertext occupies a significant amount of memory. The result is that a full execution of the training and inference algorithms for the first experiment on similar transfer data (which uses two level-0 models that both need to be stored in memory for the entire execution) occupies approximately 130GB of memory. An execution of the second experiment on unrelated transfer data (which uses eight level-0 models) occupies approximately 210GB of memory.

As for computational time, *Table 3* shows the amortized time per single input ciphertext of each part of the execution of the PINStack model. The introduction of homomorphic operations increases the execution time of the algorithms by two to three orders of magnitude over their privacy-violating counterparts. Fully connected layers in particular make up for most of the time usage, both in their forward and backward passes, as they use the highest number of multiplications and rotations. Their effect can be clearly seen when comparing the times between the two experiments and considering the parameters described in *Section 6.2*. The experiment with similar transfer data has a number of inputs for the level-1 fully connected layer of $2 \cdot 64 = 128$, while the experiment with different transfer data has $8 \cdot 32 = 256$ inputs. As the number of operations for both the forward and backward pass of fully connected layers depends on the number of inputs, the first experiment runs significantly faster than the second. Instead, the level-0 models have an initial input size for the fully-connected layers of $32 \cdot (14 - 2) = 384$ in the first experiment, and $32 \cdot (12 - 2) = 320$ in the second experiment. This explains the slower level-0 computations in the first experiment.
The overall computational time per input of each phase can be computed as:

- Training: $n \cdot (\textit{level-0 forward time}) + e \cdot ((\textit{level-1 forward time}) + (\textit{level-1 backward time}))$
- Inference: $n \cdot (\textit{level-0 forward time}) + (\textit{level-1 forward time})$

where n is the number of level-0 models and e is the number of epochs.
Thanks to input batching a forward pass or backward pass of the model on a single ciphertext accounts for 16 separate input sequences. As a result, according to the size of the dataset, for the experiments on unrelated trasfer data a full training procedure of PINStack takes between 6.5 hours (for *Airline Passengers* with 126 observations) and 46 hours (for *Monthly Sunspots* with 2796 observations). For the experiments on similar transfer data (all with 1846 observations) the full training time is approximately 13 hours. This runtime is a noticeable improvement over the time requirements of other non-interactive state-of-the-art training solutions, which operate in the order of several days [53] [11].

Although these requirements might appear like significant issues, PINStack is intended to be deployed in a Cloud-based setting, where the resources are perfectly capable of satisfying its memory and computational demands. These experiments are also designed for single-processor execution, although most of the computations required by the algorithms can potentially be performed in parallel with no modifications necessary. Further developments of the PyCNN-CKKS library could greatly improve its performance in real world applications, but they fall outside of the scope of this research.

	P.P.		*P.V.*
	similar	unrelated	average
level-0 forward pass	231.35	210.0s	0.17s
level-1 forward pass	26.18	74.8s	0.06s
level-1 backward pass	27.85	118.1s	0.76s

Table 3: Amortized computational time of the execution of different parts of the PINStack model on a single input ciphertext. Results from the two experiments on the real privacy-preserving model and a privacy-violating version run on plain data are compared.

7. Conclusions

In this work a privacy-preserving solution for time series forecasting is proposed, which is capable of performing both inference and training on encrypted time series data. Our solution makes use of state-of-the-art homomorphic encryption schemes to perform computations on encrypted data without ever needing to know its plain contents. Implementations of the main layers of a TCNN compatible with such schemes are presented, together with a gradient descent algorithm that operates on fully connected layers under homomorphic restrictions. The resulting PINStack model combines the features of the previous PINPOINT architecture with a transfer learning approach, optimizes its operations for the CKKS encryption scheme and allows training of its final fully connected layer for fine-tuning on encrypted datasets.
The capabilities of the architecture and its applications to a Cloud-based MLaaS scenario are particularly promising. The proposed experiments prove that PINStack is successful at making accurate predictions without breaching the privacy of a user's data, both with and without having access to plain data related to the task. The research acts as a proof-of-concept to show that training in such low-epoch environments can be effective with the right underlying model, and further developments can push this approach further to a proper implementation for real-world usage.

Several improvements can still be made to the underlying PyCNN-CKKS library to increase efficiency and performance, with optimizations for multi-processor architectures having the potential to significantly reduce computational times. More efficient implementations of the CKKS scheme, such as the one proposed in [11], could also lead to important speed-ups by making use of GPU computing and introducing further parallelization in the calculations.
Another area of interest largely left unattended by our research is that of parameter optimization. Although having to operate on encrypted data makes it difficult if not outright impossible to optimize a model to a specific encrypted learning task, the transfer learning framework used by PINStack still leaves room for major improvements. More studies can be made into optimizing each model to the specific transfer datasets employed, or designing models with a particular combination of transfer datasets and hyperparameters that are optimized for particular scenarios and data distributions. In addition, in the case where similar plain data is accessible, more classical hyperparameter optimization techniques can be used to better fit the task at hand.
Finally, the homomorphic layers proposed in our library can be adapted to work in different deep learning scenarios or extended to encompass more complex models. The stacking framework and fine-tuning technique described in our research can be applied to any machine learning model, provided that the network is not too deep and that its layers can be implemented with addition and multiplication only. Applications to generic classification and regressions tasks using a curated collection of simple machine learning models require further research but look entirely feasible.

References

[1] Australian Monthly Beer Production. `https://www.kaggle.com/code/mpwolke/australian-monthly-beer-production/`.

[2] Datasets. `https://github.com/jbrownlee/Datasets/`.

[3] Detroit District, U.S. Army Corps of Engineers. `https://www.lre.usace.army.mil/`.

[4] ENTSO-E Transparency Platform. `https://transparency.entsoe.eu/`.

[5] Federal Reserve Economic Data. `https://fred.stlouisfed.org/`.

[6] Open Government Data Platform India. `https://data.gov.in/`.

[7] Solar Influences Data Analysis Center. `https://www.sidc.be/`.

[8] United States Census Bureau. `https://www.census.gov/`.

[9] U.S. Energy Information Administration. `https://www.eia.gov/`.

[10] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (Csur)*, 51(4):1–35, 2018.

[11] Ahmad Al Badawi, Louie Hoang, Chan Fook Mun, Kim Laine, and Khin Mi Mi Aung. Privft: Private and fast text classification with homomorphic encryption. *IEEE Access*, 8:226544–226556, 2020.

[12] Mohammad Al-Rubaie and J Morris Chang. Privacy-preserving machine learning: Threats and solutions. *IEEE Security & Privacy*, 17(2):49–58, 2019.

[13] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A Reuter, and Martin Strand. A guide to fully homomorphic encryption. *Cryptology ePrint Archive*, 2015.

[14] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018.

[15] Maya Bakshi and Mark Last. Cryptornn-privacy-preserving recurrent neural networks using homomorphic encryption. In *Cyber Security Cryptography and Machine Learning: Fourth International Symposium, CSCML 2020, Be'er Sheva, Israel, July 2–3, 2020, Proceedings 4*, pages 245–253. Springer, 2020.

[16] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. ngraph-he: a graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pages 3–13, 2019.

[17] Joppe W Bos, Wouter Castryck, Ilia Iliashenko, and Frederik Vercauteren. Privacy-friendly forecasting for the smart grid using homomorphic encryption and the group method of data handling. In *Progress in Cryptology-AFRICACRYPT 2017: 9th International Conference on Cryptology in Africa, Dakar, Senegal, May 24-26, 2017, Proceedings*, pages 184–201. Springer, 2017.

[18] GEORGE EP Box, Gwilym M Jenkins, and G Reinsel. Time series analysis: forecasting and control holden-day san francisco. *BoxTime Series Analysis: Forecasting and Control Holden Day1970*, 1970.

[19] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Advances in Cryptology–CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 868–886. Springer, 2012.

[20] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[21] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. Low latency privacy preserving inference. In *International Conference on Machine Learning*, pages 812–821. PMLR, 2019.

[22] Chris Chatfield. *Time-series forecasting.* CRC press, 2000.

[23] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE international conference on computer vision*, pages 2722–2730, 2015.

[24] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 409–437. Springer, 2017.

[25] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[26] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[27] Li Deng, Jinyu Li, Jui-Ting Huang, Kaisheng Yao, Dong Yu, Frank Seide, Michael Seltzer, Geoff Zweig, Xiaodong He, Jason Williams, et al. Recent advances in deep learning for speech research at microsoft. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 8604–8608. IEEE, 2013.

[28] Simone Disabato, Alessandro Falcetta, Alessio Mongelluzzo, and Manuel Roveri. A privacy-preserving distributed architecture for deep-learning-as-a-service. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020.

[29] Matthew F Dixon, Igor Halperin, and Paul Bilokon. *Machine learning in Finance*, volume 1170. Springer, 2020.

[30] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *nature*, 542(7639):115–118, 2017.

[31] Alessandro Falcetta and Manuel Roveri. Privacy-preserving time series prediction with temporal convolutional neural networks. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2022.

[32] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.

[33] Michael J Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[34] Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.

[35] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*, pages 201–210. PMLR, 2016.

[36] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Providing sound foundations for cryptography: on the work of Shafi Goldwasser and Silvio Micali*, pages 173–201. 2019.

[37] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[38] Shai Halevi and Victor Shoup. Algorithms in helib. In *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34*, pages 554–571. Springer, 2014.

[39] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[40] Daniel Huynh. OpenMinded - CKKS explained series. `https://blog.openmined.org/ckks-explained-part-1-simple-encoding-and-decoding/`, 2020.

[41] Alberto Ibarrondo and Alexander Viand. Pyfhel: Python for homomorphic encryption libraries. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 11–16, 2021.

[42] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[43] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. Logistic regression model training based on the approximate homomorphic encryption. *BMC medical genomics*, 11(4):23–31, 2018.

[44] Miran Kim and Kristin Lauter. Private genome analysis through homomorphic encryption. In *BMC medical informatics and decision making*, volume 15, pages 1–12. BioMed Central, 2015.

[45] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[46] Tsit-Yeun Lam and Ka Hin Leung. On the cyclotomic polynomial ϕ pq (x). *The American Mathematical Monthly*, 103(7):562–564, 1996.

[47] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access*, 10:30039–30054, 2022.

[48] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *IEEE signal processing magazine*, 37(3):50–60, 2020.

[49] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems*, 2021.

[50] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM)*, 60(6):1–35, 2013.

[51] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-lwe cryptography. In *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32*, pages 35–54. Springer, 2013.

[52] Kentaro Mihara, Ryohei Yamaguchi, Miguel Mitsuishi, and Yusuke Maruyama. Neural network training with homomorphic encryption. *arXiv preprint arXiv:2012.13552*, 2020.

[53] Karthik Nandakumar, Nalini Ratha, Sharath Pankanti, and Shai Halevi. Towards deep neural network training on encrypted data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 0–0, 2019.

[54] Ashutosh Pandey and DeLiang Wang. Tcnn: Temporal convolutional neural network for real-time speech enhancement in the time domain. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6875–6879. IEEE, 2019.

[55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[56] Jestine Paul, Meenatchi Sundaram Muthu Selva Annamalai, William Ming, Ahmad Al Badawi, Bharadwaj Veeravalli, and Khin Mi Mi Aung. Privacy-preserving collective learning with homomorphic encryption. *IEEE Access*, 9:132084–132096, 2021.

[57] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[58] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[59] Mauro Ribeiro, Katarina Grolinger, and Miriam AM Capretz. Mlaas: Machine learning as a service. In *2015 IEEE 14th international conference on machine learning and applications (ICMLA)*, pages 896–902. IEEE, 2015.

[60] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[61] Ron Rothblum. Homomorphic encryption: From private-key to public-key. In *Theory of Cryptography: 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings 8*, pages 219–234. Springer, 2011.

[62] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[63] Jürgen Schmidhuber, Sepp Hochreiter, et al. Long short-term memory. *Neural Comput*, 9(8):1735–1780, 1997.

[64] Microsoft SEAL (release 4.1). `https://github.com/Microsoft/SEAL`, January 2023. Microsoft Research, Redmond, WA.

[65] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71:57–81, 2014.

[66] Taylor G. Smith et al. pmdarima: Arima estimators for Python, 2017–.

[67] Priyanshu Srivastava and Rizwan Khan. A review paper on cloud computing. *International Journal of Advanced Research in Computer Science and Software Engineering*, 8(6):17–20, 2018.

[68] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[69] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[70] Hassan Takabi, Ehsan Hesamifard, and Mehdi Ghasemi. Privacy preserving multi-party machine learning with homomorphic encryption. In *29th Annual Conference on Neural Information Processing Systems (NIPS)*, 2016.

[71] Sean J Taylor and Benjamin Letham. Forecasting at scale. *The American Statistician*, 72(1):37–45, 2018.

[72] Anh-Tu Tran, The-Dung Luong, Jessada Karnjana, and Van-Nam Huynh. An efficient approach for privacy preserving decentralized deep learning models based on secure multi-party computation. *Neurocomputing*, 422:245–262, 2021.

[73] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[74] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[75] David H Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992.

[76] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th annual symposium on foundations of computer science (Sfcs 1986)*, pages 162–167. IEEE, 1986.

[77] Zijie Yue, Shuai Ding, Lei Zhao, Youtao Zhang, Zehong Cao, Mohammad Tanveer, Alireza Jolfaei, and Xi Zheng. Privacy-preserving time-series medical images analysis using a hybrid deep learning framework. *ACM Transactions on Internet Technology (TOIT)*, 21(3):1–21, 2021.

[78] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. Batchcrypt: Efficient homomorphic encryption for cross-silo federated learning. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 2020)*, 2020.

[79] Chuan Zhao, Shengnan Zhao, Minghao Zhao, Zhenxiang Chen, Chong-Zhi Gao, Hongwei Li, and Yu-an Tan. Secure multi-party computation: theory, practice and applications. *Information Sciences*, 476:357–372, 2019.

Abstract in lingua italiana

In un mondo definito da Deep Learning e dai Big Data, le infrastrutture informatiche basate sul Cloud sono diventate uno strumento necessario per soddisfare le cresenti esigenze di calcolo richieste da attività di Machine Learning, in modo da dare a qualsiasi utente accesso a soluzioni ad alte prestazioni, scalabili e a costi contenuti. Tuttavia, questo approccio comporta l'elaborazione di grandi quantità di dati su una piattaforma di terze parti, il che comporta gravi problemi di privacy se bisogna trattare dati sensibili come quelli medici e finanziari. Le tecniche di privacy-preserving machine learning offrono una soluzione a questi problemi grazie all'uso di schemi di crittografia omomorfica (HE), ma presentano nuove sfide nel modo in cui questi privacy-preserving network devono essere progettati. Lavori precedenti, sotto forma della famiglia di modelli di deep learning PINPOINT, hanno già mostrato risultati promettenti nel campo della previsione di serie temporali a rispetto della privacy dei dati, con potenziali applicazioni in situazioni reali. L'obiettivo di questa tesi è estendere tale lavoro per includere una nuova procedura di training, che rende possibile il fine-tuning di un network direttamente su dati sensibili criptati senza violare la loro privacy. Il risultato è la nuova architettura PINStack realizzata con model stacking, che viene presentata come una soluzione generica per la previsione di serie temporali in un ambiente privacy-preserving sia in una fase di inference che di training, utilizzando lo schema omomorfico Cheon-Kim-Kim-Song (CKKS) per garantire la privacy dei dati. Le sue prestazioni sono state valutate in scenari d'uso realistici e mostrano un grande potenziale per future implementazioni e ulteriori sviluppi.

Parole chiave: Privacy-preserving machine learning, previsione di serie temporali, crittografia omomorfica, training su dati criptati