Neural Network training on homomorphically encrypted data with CKKS bootstrapping

Chanatip Vachirathanusorn
chanatip.va@perm.ai

Phanuruj Suwachirat
phannuruj.su@perm.ai

Pheemmapol Chayanon
pheemmapol.ch@perm.ai

October 2021

Abstract

Concerns about data security have impeded the movement of data between parties for machine learning training purposes. Many works have demonstrated the use of Homomorphic Encryption (HE) with various types of machine learning to overcome this impediment; however, the training of neural networks on encrypted data without any decryption by data-owners during the training process has yet to be explored. In this paper, we demonstrate how a neural network can be trained on data encrypted using the Cheon-Kim-Kim-Song (CKKS) scheme of Homomorphic Encryption (HE). Using the bootstrapping procedure, we're able to perform a sufficient number of multiplication and rescale operations to train a neural network without the need for any decryption. Since CKKS encrypted data are limited to additive and multiplicative operations, we implement bounded-polynomial approximation functions to approximate the traditional activation functions to achieve non-linearity in neural networks. We also propose the design consideration for a neural network's structure to maximize training efficiency on encrypted data; Lastly, we provide the experimental result showing that a neural network trained on homomorphically encrypted MNIST dataset can produce a result that closely resembles a model trained on normal MNIST dataset in term of accuracy with only 1.73% difference testing accuracy;

Keywords— CKKS, Homomorphic Encryption, Neural network, Deep learning, Privacy-preserving machine learning

1 Introduction

Data is one of the most important and valuable assets in the machine learning industry because quality data serves as a foundation for a quality supervised machine learning model. However, not everyone that owns the data has the expertise to extract values out of those data, while those who have the expertise (data processor) may not have access to the data needed. Therefore, the transfer of data between data owners and data processors is crucial. Nevertheless, these transactions of data are often impeded by the secrecy, confidentiality, or privacy of those datasets. These impediments could range from just a concern and distrust that one party has towards the other to regulations, such as the General Data Protection Regulation (GDPR) [1] from the European Union, outlawing these transfers of data entirely. While the former impediment can be overcome with the signing of a Non-Disclosure Agreement, it still carries the risk of accidental data leakage through cyber-attacks, but to overcome the latter impediment those data must be anonymized, which isn't enough for AI as a service business to grow.

One possible solution to these problems is the use of Homomorphic Encryption: a type of encryption, first proposed by Rivest et al. [2], that allows computation to be done on encrypted data without the need for decryption. This will allow the party that owns the data to encrypt their data with a homomorphic encryption scheme and send them to the data processors. The data processors can then perform necessary mathematical operations on that encrypted data to derive the result needed without the need for decryption. That encrypted result can then be sent back to the owner who holds the key for decryption. This whole process can happen securely without the need for the data processor to see the data or the results, while still giving the data owner what they need. This use of homomorphic encryption can be extremely helpful in situation such as the transfer of patients medical records for analyses as demonstrated by Wood et al. [3].

Initially, the use of homomorphic encryption for machine learning training wasn't popular because of its many limitations. These include the limited types of operation available, the limited number of operations allowed, and the computational overhead needed to perform each operation. Nonetheless, as new works are introduced, training a machine learning model on homomorphically encrypted data is becoming

increasingly practical. For example, to tackle the problem of limited number of operations allowed, Gentry [4] introduces Fully Homomorphic Encryption (FHE) with an operation called "bootstrapping" that will reduce the amount of noise – a factor that limits the number of operation performable – in a ciphertext. The computational overhead have also been reduced with homomorphic encryption schemes, such as the BGV [5], TFHE [6, 7] and CKKS [8] scheme, that support operations in single instruction multiple data (SIMD) [9] manner, enabling the potential to parallelize operations. Since most homomorphic encryption scheme only support addition and multiplication operations, Cheon et al. [10] have presented ways to perform operations, such as multiplicative inverse, min/max, or square root, with approximate functions using the available operations. Moreover, programmable bootstrapping is introduced in the TFHE scheme by Chillotti et al. [11], allowing various operations and functions to be performed with homomorphically encrypted data. Lastly, the CKKS scheme also introduces native support of approximate numbers, which further increases the practicality of homomorphic encryption. Due to these contributions, machine learning on homomorphic encryption is becoming increasingly practical. Since most of the significant limitations and impediments have been handled, we believe that homomorphic encryption will be a crucial part of privacy-preserving machine learning.

1.1 Related works

Training of various machine learning models, including Logistic regression [12, 13] and Support Vector Machine [14] using homomorphically encrypted data has been researched and proven. However, when it comes to the use of homomorphic encryption with neural networks, the majority of works have been done only on the evaluation of encrypted data with a trained model. CryptoNets by Dowlin et al. [15] was among the first to demonstrate that a trained Convolutional Neural Network can be used to evaluate homomorphically encrypted images. Subsequent works [16, 17, 18, 19] have suggested ways to improve upon CryptoNets' performance, accuracy, and security using wide ranges of homomorphic encryption schemes. These works have presented us with various techniques to perform forward propagation of a neural network, including ways to evaluate activation functions [16, 18], and data packing method [19].

When it comes to neural network training that involves homomorphic encryption, most works focus on its use with Multi-Party Computation (MPC) to enhance the security of neural network training on joint datasets belonging to different parties. For example, in [20, 21, 22, 23] homomorphic encryption schemes were used as part of their algorithm for multi-party privacy-preserving neural networks training. This type of training is different from ours.

Works that are the most similar to ours focus on the training of neural networks on homomorphically encrypted data. Zhang et al. [24] explore the use of the BFV encryption scheme [25] to encrypt a dataset and perform the forward and backward propagation operations on them to calculate the optimization gradients of a neural network. Their model also uses a polynomial approximation to approximate activation functions in their network similar to ours; however, their training algorithm relies on the data owner to constantly encrypt the parameters and decrypt the gradients once per iteration. Similar to [24], Mihara et al. [26] demonstrated that a neural network can be trained on encrypted data, but with the CKKS scheme [8] which is the same as ours. However, they use a simple neural network with square activation, MSE loss function, and low trainable parameters. Their training algorithm also required a communication with the data owner every iteration to increase their ciphertexts' level through re-encryption; however, they have stated that this re-encryption is not necessary if bootstrapping is implemented.

1.2 Our contributions

Our contributions are as followed:

- We demonstrated the experimental result proving that a neural network with fully connected layers and a high number of trainable parameters can be trained with an MNIST dataset [27] encrypted with CKKS homomorphic encryption scheme.
- We eliminate the need for any decryption during the training process, which is required in previous works [24, 26], by implementing a CKKS bootstrapping method proposed by Bossuat et al. [28]. This bootstrapping operation allowed us to raise the modulus level of a ciphertext efficiently, increasing the number of multiplication and rescaling operations performable.
- We also provide the equation for polynomial approximation of activation functions that we use in our model, including the sigmoid, tanh, and softmax function, and demonstrated that a categotical cross entropy loss can be used to calculate the loss gradient of encrypted model.
- We suggest the factor of consideration when constructing a model to be trained on CKKS encrypted data to maximize speed and efficiency. We also describe the utilization of concurrency to speed up the neural network training process, which is a compute and memory-intensive task using only the CPU.

One of the use-cases of our work is when a party who owns a dataset wants to train a neural network model on that dataset; however, the data owner does not have the computational power and expertise to train that model properly, so the data owner has to send their dataset to a data processor, who will do it for them. For security, that data owner can send the encrypted version of that dataset to the processor,

who can then create a model and train it without the need to decrypt that data. This whole process could be completed without requiring the data owner to perform any computation except encrypting the dataset before training and decrypting the result after the training is completed.

1.3 Paper organization

We present the required preliminaries in section 2. The explanation of the approximate activation functions is in section 3. The model optimization including the description of the implementation of loss functions and stochastic gradient descent is in section 4. Section 5 explains our implementation of the neural network's fully connected layer for both forward and backward propagation, while also providing the facts that need to be taken into account when constructing a neural network for training on CKKS encrypted data, including level and bootstrapping. Our experimentation results are shown in section 6 with the detail on the model we used. Lastly, section 7 is the conclusion.

2 Preliminaries

As previously mentioned, Homomorphic Encryption schemes are becoming increasingly practical. In this section we will introduce the basic architecture of Homomorphic Encryption and other related works on Homomorphic Encryption. Moreover we will introduce the latest contribution to Homomorphic Encryption, an efficient bootstrapping method that significantly reduces computational time in CKKS. Lastly, we will address the security aspect of Homomorphic Encryption and especially of CKKS.

2.1 Basic Architecture

In this section, we will briefly go over the basic functions of modern homomorphic encryption scheme. This usually starts with public-private key pair generation by the data owner. Since the private key can be used to decrypt data, the data owner must securely store it to protect the security of encrypted ciphertext, while the public key can be shared as it can only be used for encryption. Data, in the form of numbers or encoded plaintext, can then be encrypted with the private or public key generating a ciphertext that, for most modern HE schemes, can undergo basic operations of addition, subtraction, or multiplication. These operations will return a ciphertext containing the result of that mathematical operation. Note that the party performing these operations does not necessarily need to know the encrypted values inside the ciphertext. After performing an arbitrary number of computations, the party that has the private key, in this case the data owner, will be able to decrypt the ciphertext revealing the result of calculations performed.

2.2 Homomorphic Encryption

Gentry first introduced Homomorphic Encryption in 2009 [4]. However, Gentry's version of HE had many practical limitations. Despite its many improvements to previous works in the field of Homomorphic Encryption, its real-world usability is still limited. For example, despite being the first scheme that is able to perform both additions and multiplications, there is an aspect of HE that is called "noise" which increases with each computation and when it is more than a certain threshold, the decryption will not yield the correct result. In order to perform further computations, the data processor has to perform a "bootstrapping" process that involves generating new keys and switching them in order to reduce the noise which takes an incredibly long time.

After what is considered a very important period for this field, many schemes and libraries have been created which can be categorized into two branches. The first branch has a different approach to Gentry's first paper in which the schemes aim to reduce bootstrapping time in order to evaluate an arbitrary circuit. The second branch follows what Gentry outlined in his first paper which is to use a noise reduction method in order to turn a Leveled Homomorphic scheme in to a Fully Homomorphic scheme, or put simply, to turn a limited computation circuit into one that can evaluate an arbitrary number of circuits by various techniques of noise reduction procedures. In this section, we will only focus on the second branch which will be further elaborated.

The major contributions in the second branch are the works of BGV [5], BFV [25] and most recently CKKS [8]. Each of these schemes improved upon Gentry's first paper in various different ways. BGV, which was proposed in 2012, utilized another technique called modulus switching in order to reduce noise. The BFV scheme then made many modifications to the BGV scheme by implementing a more simple bootstrapping procedure and a new modulus switching technique. BFV also supports a relinearization technique in order to prevent the ciphertext from outgrowing its bound during multiplications. Finally, the CKKS scheme which was proposed in 2016 made significant improvements to both of these schemes. First of all, both BGV and BFV only support computations on the integers, meaning that their real world usage is very limited, especially in the use of machine learning where computations on real numbers are vital. This is why CKKS is the most viable scheme for machine learning purposes by treating the encryption noise as part of the error that is natural in normal approximate arithmetic. However, when CKKS was first proposed, the scheme did not support bootstrapping which still makes machine learning

on homomorphic encryption impractical. Since its release, there have been many improvements which includes a bootstrapping procedure which turns it from a levelled HE scheme into a FHE scheme which we will further expand on in the next section.

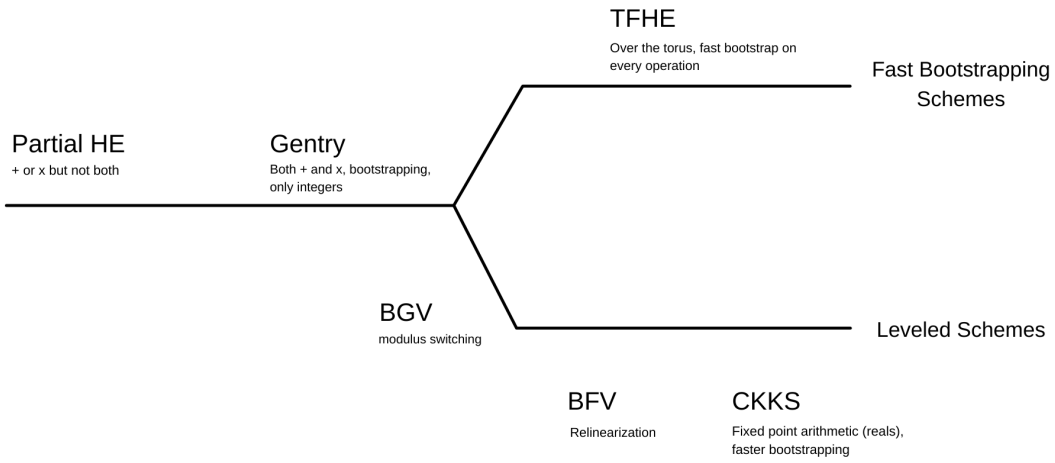

Figure 1: Two main branches of modern Homomorphic Encryption scheme [4, 5, 6, 8, 25].

2.3 CKKS Bootstrapping

One of the most important parts of current state of the art Fully Homomorphic Encryption schemes is, without a doubt, the bootstrapping procedure. Bootstrapping allows Levelled Homomorphic Encryption schemes (LHE) to be able to perform an arbitrary number of computations, thus turning it into a FHE scheme. Recently, there has been significant increases in performance of these bootstrapping procedures, so much so that the once impractical LHE schemes are now one of the best tools out there for privacy preserving machine learning. In this section, we will expand further on bootstrapping in the CKKS scheme.

CKKS was, when released, a LHE scheme. Despite its ability to perform approximate arithmetics, which could open up many possibilities for its usage, it was still impractical in real world usage. Being an LHE, the user has to know the evaluation circuits exactly, meaning that the amount of computations are very limited. Due to the nature of the CKKS scheme, which does not support modular arithmetic, the bootstrapping procedure is hard because we have to resort to a modular reduction method in order to decrypt for bootstrapping. Cheon et al. [29] first proposed a way to raise the level of CKKS ciphertexts in 2018. By using an approximated Taylor Polynomial sine function as a replacement of the modular reduction function, they were able to perform bootstrapping with the cost of computation growing linearly. Chen et al. [30] further improved the bootstrapping procedure by two orders of magnitude. They used a new technique for evaluating ciphertexts and replaced the approximated Taylor Polynomial sine function with a more accurate Chebyshev approximation. The most recent work, by Bossuat et al. [28], has greatly improved both the performance and the speed of bootstrapping. They introduced a new algorithm for polynomial evaluation, optimized the key-switching procedure, and also were able to assess the probability of bootstrapping failure. The most recent work was implemented in an open-source library called Lattigo which we are currently using for this paper.

2.4 On the security of HE and CKKS

Security is arguably the most important aspect of any cryptographic scheme. The ability to withstand attacks against malicious parties will determine the viability of the cryptographic system itself. In this section, we will briefly go over the underlying technology and the security of Homomorphic Encryption and especially the CKKS scheme.

The most crucial part of many Homomorphic Encryption schemes is the underlying hard problem called Learning With Error (LWE). LWE which was introduced in 2005 by Regev [31] revolutionized the Homomorphic Encryption literature that was only a theoretical idea just decades ago. Usually with a system of random linear secret-key equations, the secret key can easily be recovered by gaussian elimination which could be solved in polynomial time, meaning that once a quantum computer is developed, it would easily render the encryption useless. However, if we incorporate a small random distribution of errors

into those equations, the current best known algorithms for LWE run in exponential time. Unlike other factoring based cryptographic systems such as RSA, LWE couldn't be rendered useless even in the post quantum world [32] because of the polynomial run time. CKKS on the other hand, uses a variant of LWE called Ring Learning With Error (RLWE). The main reason RLWE is preferred over LWE in this case is that traditional LWE requires key sizes on the order of n^2 which would limit the practicality when dealing with large key lengths. By assuming there exists some structure in LWE, we can interpret the original vectors in smaller quantities which speeds up many operations compared to traditional LWE.

Regarding the security of Homomorphic Encryption cryptosystems, the security itself is evaluated by the notion of Indistinguishability (IND). There are many variants of IND on the likes of IND-CPA, and IND-CCA. CKKS itself is IND-CPA secure, but there have been many studies indicating that the traditional definition of IND-CPA is not adequate when dealing with approximate computations like CKKS [33] which created a new notion of IND-CPA+. However, many schemes have made changes to address new security notions [34].

2.5 Training data packing

Since the CKKS ciphertext is an encrypted vector of numbers with $N/2$ slots where N is the power-of-two ring degree, operation done on a ciphertext can be parallelized in the single instruction multiple data (SIMD) [9] manner. This means that no matter how much data is packed into a ciphertext, the computational cost will stay constant, incentivising us to pack as much data as possible into each ciphertext.

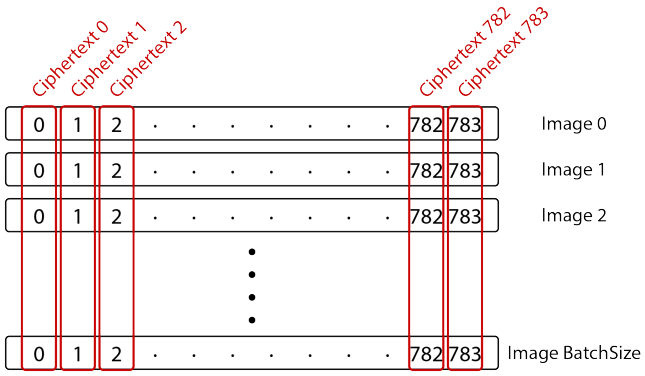

Figure 2: Ciphertext training data packing.

Similar to the packing method proposed in [15, 19], each ciphertext we pack represents a training batch of a single normalized pixel from many images, as shown in Figure 2. This packing method provides a good parallelization, since one training batch can be completed at the same time, while still maintaining a simple training algorithm. However, there is a downside to our approach: since we have to use a high power-of-two ring degree $N = 2^{16}$ to maintain 128 bits of security when bootstrapping, $N/2 - BatchSize$ slots will be wasted; therefore, with this packing method, a high bactch size is incentivized to achieve faster training per epoch. This packing method is also applied to the training labels.

2.6 Training data encryption parameters

Encryption parameters are the important factor to consider when using homomorphic encryption; therefore, it has to be correctly set when encrypting the training dataset to minimize the storage size and ensure accuracy of encrypted dataset. We remark that in this section we will only focus on the initial scale and level when encrypting the training dataset.

In the CKKS scheme, the scale determines how accurate a number is when encoded; the lower the scale is the less accurate the numbers become, and since small numbers are common in machine learning, the scale should be high enough that an error is insignificant enough. However, it shouldn't be too high that it risks overflowing when performing operations.

Level	Size (MB.)
24	26.21
18	19.92
9	10.48
1	2.1

Table 1: Size of ciphertext encrypt with parameter I from table 2 at each sampled level implemented using the Lattigo library [35].

In order to minimize encrypted data size, communication cost, and memory cost, the initial modulus level of the training dataset has to be carefully chosen. Since the encrypted training data will have to be multiplied with the model's weight ciphertexts when training, the extra level that the data ciphertext has over the weight ciphertext will have to be removed; therefore, the data ciphertext shouldn't have a level unnecessarily higher than the maximum post bootstrap level, as unnecessary level will only result in unnecessarily large ciphertext. Table 1 shows the relationship between ciphertext size and modulus level.

3 Activation functions

In order to introduce nonlinearity and complexity to neural networks, activation functions, which are not linear functions, are applied to the output of the artificial neurons. The most frequently used activation functions in neural networks include Sigmoid function, Tanh function, ReLU function, and Softmax function. Since the CKKS scheme only supports the additive and multiplicative operations between ciphertexts, with subtraction achievable through additive inverse, it lacks the ability to calculate the multiplicative inverse ($\frac{1}{x}$) and the exponential function (e^x), which are required to evaluate the Sigmoid, Tanh, and Softmax functions. Moreover, it also lacks a computationally efficient way to evaluate the max function necessary for the Relu activation function. Therefore, in order to use activation functions effectively on homomorphically encrypted data, a HE-friendly version of those functions have to be implemented.

In addition, we also need the derivative of all activation functions except softmax as we'll use them in backward propagation. Those functions must be all converted to a HE-friendly version as well.

3.1 Sigmoid

In forward propagation, the sigmoid function is applied between the hidden layers to compressed the range of output from $(-\infty, \infty)$ to $(0, 1)$.Typically, it is used for calculating the probability from the linear function.

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

However, as the function requires the calculation of e^{-x} which can't be natively computed on homomorphically encrypted data, we have to use an approximation function, favorably a polynomial function. The one we are using is adapted from [12]. One downside to using an approximate function is the limited bound for the input, in this case $x \in [-4, 4]$, that will yield an accurate output.

$$\text{ApproxSigmoid}(x) = 0.5 + 0.197x - 0.004x^3$$

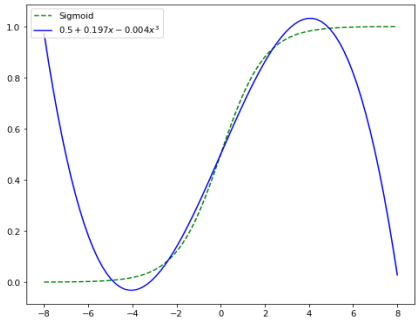

Figure 3: ApproxSigmoid(x) function (blue) compared to actual Sigmoid(x) function.

For the backward propagation, we'll just take the derivative of the function.

$$\text{ApproxBackwardSigmoid}(x) = 0.012x^2 + 0.197$$

3.2 Hyperbolic Tangent (tanh)

Much like Sigmoid, the tanh function is used between the hidden layers of neural network, but the function compressed the range from $(-\infty, \infty)$ to $(-1, 1)$. As tanh has a very similar shape to Sigmoid, we can scale and shift the ApproxSigmoid(x) to have the range we need. The bound for the accurate approximation for this functions is also $x \in [-4, 4]$.

$$\text{ApproxTanh}(x) = -0.00752x^3 + 0.37x$$

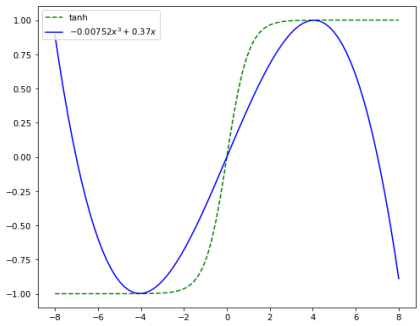

Figure 4: ApproxTanh(x) function (blue) compared to actual tanh(x) function (green)

Similarly, for backward propagation of tanh(x), we take the derivative of the approximation function.

$$\text{ApproxBackwardTanh}(x) = -0.02256x^2 + 0.37$$

3.3 Rectified Linear Unit (ReLU)

Rectified Linear Unit (ReLU) is a piecewise linear function that is the most used activation function in neural network. It is a combination of 2 linear equations.

$$\text{ReLU(x)} = \begin{cases} x & x \geq 0 \\ 0 & otherwise \end{cases}$$

Although we are not using the ReLU activation function in our work, we acknowledge its importance in a neural network; therefore, we are proposing an ReLU approximate polynomial activation function of our own similar to the degree 4 polynomial proposed by Chabanne et al. [16], but with a higher bound at the cost of less accuracy near $x = 0$. We noticed that for the derivative of the ReLU function:

$$\frac{d(\text{ReLU}(x))}{dx} = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$

which is very similar to the function Sigmoid, so we can reuse the pre-existing function ApproxSigmoid(x) to create ApproxReLU(x)

$$\text{ApproxReLU}(x) \approx \int \text{ApproxSigmoid}(x)dx$$

$$\text{ApproxReLU}(x) = -\frac{0.004}{4}x^4 + \frac{0.197}{2}x^2 + 0.5x + 0.7$$

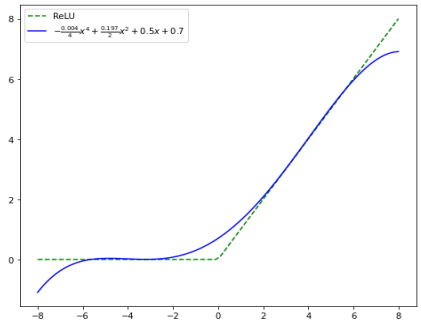

Figure 5: ApproxReLU(x) function (blue) compared to actual ReLU(x) function (green)

For backward propagation of ReLU(x), it can be simply created using the fundamental theorem of calculus.

$$\text{ApproxBackwardReLu}(x) = \frac{d(\text{ApproxReLU}(x))}{dx} = \frac{d(\int \text{ApproxSigmoid}(x)dx)}{dx} = \text{ApproxSigmoid}(x)$$

$$\text{So, ApproxBackwardReLU}(x) = 0.5 + 0.197x - 0.004x^3$$

3.4 Softmax

The softmax function is usually used to evaluate the output of the final layer of a neural network in order to transform the output of a neural network into a probability distribution. This activation function, combined with the cross-entropy loss, can become an important part of the classification neural networks' training process; therefore, it is important to be able to evaluate this activation function on a ciphertext.

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

As shown in equation above, the softmax function is unlike other functions we previously discussed, as it requires the inverse of the sum of all the exponentiated output to evaluate; therefore, a single approximation function couldn't be used to evaluate the softmax function. Instead, using the same approach as suggested by Hong et al. [36], we can approximate it using multiple approximation functions: approximate exponential function and approximate inverse function.

For the approximate exponential function, we can use the 7^{th} degree Taylor polynomial of e^x around x = 0.

$$P_n(x) = \sum_{k=0}^{n} \frac{(x-a)^i f^{(k)}(a)}{k!}$$

where $P_n(x)$ is the taylor polynomial degree n of the function f(x) around x = a.
So, the taylor polynomial degree 7^{th} of e^x around x = 0 would be:

$$ApproxExp(x) = e^x \approx 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040}$$

This polynomial is only an accurate approximation of e^x around the domain $x \in [-1, 2.5]$, so in practice, we have to scale the input down to be in the accurate domain;

For the approximate inverse function, we can use Goldschmidt's division algorithm [37] to approximate the inverse of a positive integer using the following equation:

$$Goldschmidt(x) = \prod_{i=0}^{d-1} (1 + (1-x)^{2^i})$$

However, Goldschmidt's algorithm can only be used to evaluate $x \in (0, 2)$, but as suggested by [36], the Goldschmidt's algorithm $g(x)$ can be expanded to $x \in (0, 2M)$, as demonstrated:

$$\frac{1}{x} = Goldschmidt(\frac{x}{M}) \cdot \frac{1}{M}$$

Using this method we will have to do 2 more multiplications, which is expensive for a ciphertext as it will consume more levels and take more time; however, we propose a method to decrease the numbers of multiplication down to 1 by expanding the Goldschmidt's algorithm into a polynomial, and since the coefficient of that polynomial is known, we can multiply the coefficient of the term n with $(1/M)^n$ as $\mathbb{R}$ before encoding the coefficients into plaintexts to evaluate the function with ciphertexts; with this method, the multiplication of ciphertext with $1/M$ is only needed once after the evaluation of a function. This method is shown below given that $d = 3$.

$$\prod_{i=0}^{2} (1 + (1 - \frac{x}{M})^{2^i}) = -(\frac{x}{M})^7 + 8(\frac{x}{M})^6 - 28(\frac{x}{M})^5 + 56(\frac{x}{M})^4 - 70(\frac{x}{M})^3 + 56(\frac{x}{M})^2 - 28(\frac{x}{M}) + 8$$

$$ApproxInverse(x, M) = -(\frac{1}{M^7})x^7 + (\frac{8}{M^6})x^6 - (\frac{28}{M^5})x^5 + (\frac{56}{M^4})x^4 - (\frac{70}{M^3})x^3 + (\frac{56}{M^2})x^2 - (\frac{28}{M})x + 8$$

Algorithm 1 represent the approximate softmax function.

4 Optimizing loss function

Neural network training is an optimization problem with the objective of achieving the minimum error. This error is calculated using a loss function, and to optimize it, we calculate the gradient of this loss function then pass it on to an optimization algorithm, usually the stochastic gradient descent algorithm, performing the backpropagation of gradient and update the weight of each layer. Therefore, choosing the correct loss function and the optimization algorithm is extremely important to achieve the best training result.

Algorithm 1 Approximate Softmax Algorithm

Input: Inverse stretch scale $M \in \mathbb{R}$, a vector of ciphertexts $\vec{c} = (c_1, ..., c_t)$
Output: Approximate output of $softmax(\vec{c})$

for $i \leftarrow 1$ `to` t **do**
    $x_i \leftarrow ApproxExp(c_i)$
end for
$InverseSum \leftarrow ApproxInverse(\sum_{i=1}^{t} x_i, M) \cdot \frac{1}{M}$
for $i \leftarrow 1$ `to` t **do**
    $x_i \leftarrow InverseSum \cdot x_i$
end for
return $(x_1, ..., x_t)$

4.1 Loss functions

The two most popular loss functions are the mean squared error (MSE) loss function and cross-entropy loss function. While the former is popular and effective to optimize a regression problem, the latter is better at optimizing a classification problem.

$$MSE(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{1}$$

$$\frac{\partial L}{\partial a_i} = 2(y_i - a_i) \tag{2}$$

The MSE loss function can be calculated by taking the average squared difference between the predicted values $\hat{y}$ and the observed values y as shown in equation 1, while its derivative – the gradient – can be calculated using equation 2 where L is the MSE loss value and a is the activation of the last layer. Both of which can be performed easily with ciphertexts.

$$CE(y, \hat{y}) = -\sum_{i=0}^{n} y_i \cdot \mathbf{log}(\hat{y}_i) \tag{3}$$

The cross-entropy loss function is used to calculate the error between two probability distributions, which can be calculated using the equation 3; However, to use cross-entropy to calculate the error of multi-class classification network, the predicted values $\hat{y}$ have to be the probability distribution calculated using the softmax function mentioned in the previous section. This variant of cross-entropy is called categorical cross-entropy.

$$\frac{\partial L}{\partial z_i} = a_i - y_i \tag{4}$$

Calculating the value of the cross-entropy loss function with ciphertext will be a challenge since it requires the calculation of logarithmic function; however, when training a neural network, it is not necessary to calculate the output of a loss function, since we only need its gradient, and luckily, the gradient of the categorical cross-entropy loss function L with regards to the linear output of the last layer z_i can be calculated by subtracting the softmax activated output of the last layer a_i by the one-hot encoded label y_i, as shown in equation 4. This can easily be calculated between ciphertexts, and this loss function is what we use in our implementation explain in the next section.

4.2 Optimization algorithm

After we have calculated the loss gradient, backpropagation is done to propagate this loss gradient backward, and the optimization algorithm will perform the update of the weights and biases in our network. As explained in section 2, we will be doing our computation in batches; therefore, the best optimization algorithm we will use is the batch gradient descent algorithm. This optimization algorithm is straightforward; after we calculated the batch loss gradient with regards to weight ∇_w of a batch, we will calculate the average of all the gradients in this batch:

$$\overline{\nabla}_w = \frac{1}{n} \sum_{i=0}^{n} \nabla_{w_i}$$

We then update the weight by subtracting the weight of iteration t with the average gradient, now ∇_w, by the learning rate:

$$w_{t+1} = w_t - \textit{learning rate} \cdot \overline{\nabla}_w$$

However, to reduce the number of multiplication needed to be performed with the ciphertext, we can combine the learning rate with the average scale, resulting in the following equation:

$$w_{t+1} = w_t - \frac{\textit{learning rate}}{n} \sum_{i=0}^{n} \nabla_{w_i}$$

When implementing this optimization algorithm on ciphertext with each ciphertext representing a batch of gradients, the average of all gradients can be calculated by calculating the inner-sum of a ciphertext using the rotation operation of the CKKS scheme then multiply with $\frac{learning\ rate}{n}$ encoded in plaintext. We remark that the same algorithm is utilized for the bias as well.

5 Neural network implementation

This section will be outlining our implementation of a neural network that will be used to train on homomorphically encrypted data. Overall, this implementation will be similar to a typical neural network, but the main differences include the replacement of activation functions into HE-friendly activation functions as described in section 3, level management (bootstrapping) which will be described in this section, and higher computational and memory overhead.

5.1 Forward propagation

Forward propagation is performed to obtain the output of a neural network. This process starts at the first layer called the input layer, as it consists of the input of the neural network. It then proceeds into the next layer called the hidden layer, which are all the layers that are between the input and the output layer of a neural network and every neural network consists of at least one hidden layer. There are many types of hidden layers, such as fully connected layers or convolutional layers, and here, we will only be utilizing the fully connected layer.

A fully connected (FC) layer will consist of a predetermined number of artificial neurons which represent the length of the output vector of that layer. Each neuron will consist of n weights (w) where n is the number of inputs, a single bias value (b), and a selected activation function f [38]. The following represents a single neuron as function g with input a:

$$g(a) = f(\sum_{i=1}^{n}(w_i \cdot a) + b)$$

The outputs of each artificial neuron in a layer combined form the output vector of this layer, which serves as the input vector for the next layer. This continues until it reaches the last layer, known as the output layer. All of these calculations can be performed with ciphertext using the standard CKKS operations, given that the selected activation function is a HE-friendly function as discussed in section 3. One thing that will differ when the forward propagation algorithm is implemented for a ciphertext is the need to perform bootstrapping after the multiplication or activation function if told or necessary. The forward propagation algorithm of a single fully connected layer of a neural network is shown in algorithm 2. This process is also highly parallelizable; therefore, to speed this process up, some form of parallelization is recommended; A fully connected homomorphic layer with bias will have a total of $(input \times output) + output$ ciphertexts serving as its trainable parameters, which will have to be updated for every batch. Each trainable parameter ciphertext will be packed with vector of numbers with length equal to the batch size, similar to how a training data ciphertext is packed as explained in section 3; however, every non-zero number in a ciphertext will always be the same, both when initiated and after update.

Algorithm 2 HE Fully Connected Layer Forward Propagation Algorithm

Input: an input ciphertext vector $\vec{a}^{(L-1)} = (a_1^{(L-1)}, ..., a_n^{(L-1)})$,
HE-friendly activation function f,
$o \times n$ weight ciphertext matrix W,
a bias ciphertext vector $\vec{b} = (b_1, ..., b_o)$,
bootstrap output boolean *btspOut*,
bootstrap activation output boolean *btspActivOut*

Output: Output and activated output of fully connected later

for $i \leftarrow 1$ `to` o **do**
  $z_i^{(L)} \leftarrow \sum_{j=1}^{n}(W_{i,j} \cdot a_j^{(L-1)}) + b_i$
  if *btspOut* **then**
    $z_i^{(L)} \leftarrow bootstrap(z_i^{(L)})$
  end if
  $a_i^{(L)} \leftarrow f(z_i^{(L)})$
  if *btspActivOut* **then**
    $a_i^{(L)} \leftarrow bootstrap(a_i^{(L)})$
  end if
end for
return $\{\vec{z}^{(L)}, \vec{a}^{(L)}\}$

5.2 Backward propagation

Backward propagation is performed to propagate the loss gradient backward throughout all the previous layers of the network to obtain the loss gradient with regards to each trainable parameter for an optimization algorithm, like stochastic gradient descent or batch gradient descent mentioned in section 4, to update them.

This can be simply done using the chain rule [38]. For example, to find the loss gradient with regards to a single weight w in the second-to-last layer $L-1$ of a neural network that only composes of fully connected layers, while using the catagorical cross-entropy loss C, we can simply calculate the following, with $a^{(n)}$ denoting the activated output of layer n and $z^{(n)}$ denoting the output of layer n:

$$\frac{\partial C}{\partial w^{(L-1)}} = \frac{\partial C}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial z^{(L-1)}}{\partial w^{(L-1)}}$$

This can fully be calculated with the CKKS homomorphic encryption, given that we uses HE-friendly activation functions. The backward propagation algorithm on homomorphic encryption will be similar to the backward propagation of a normal neural network except the existance of bootstrapping function after calculating loss gradient with regard to output or after calculating loss gradient with regard to input if told. The implementation of backward propagation of a fully connected layer is showed in algorithm 3.

Algorithm 3 HE Fully Connected Layer Backward Propagation Algorithm

Input: $\vec{a}^{(L-1)} = (a_1^{(L-1)}, ..., a_n^{(L-1)})$, $\vec{z}^{(L)} = (z_1^{(L)}, ..., z_o^{(L)})$, $\frac{\partial \vec{C}}{\partial a^{(L)}} = (\frac{\partial C}{\partial a_1^{(L)}}, ..., \frac{\partial C}{\partial a_o^{(L)}})$
HE-friendly activation function f,
$o \times n$ weight ciphertext matrix W
bootstrap activation gradient boolean *btspOut*,
bootstrap loss gradient boolean *btspInput*

Output: Bias gradient vector $\vec{\nabla}_b$, $o \times n$ weight gradient matrix ∇_W,
Loss gradient with regards to input $\frac{\partial \vec{C}}{\partial a^{(L-1)}} = (\frac{\partial C}{\partial a_1^{(L-1)}}, ..., \frac{\partial C}{\partial a_n^{(L-1)}})$

for $i \leftarrow 1$ `to` o **do**
  $\frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} \leftarrow f'(z_i^{(L)})$
  $\frac{\partial C}{\partial z_i^{(L)}} \leftarrow \frac{\partial C}{\partial a_i^{(L)}} \cdot \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}}$
  if *btspOut* **then**
    $bootstrap(\frac{\partial C}{\partial z_i^{(L)}})$
  end if
  $\nabla_{b_i} \leftarrow \frac{\partial C}{\partial z_i^{(L)}}$
  for $j \leftarrow 1$ `to` n **do**
    $\nabla_{W_{i,j}} \leftarrow \frac{\partial C}{\partial z_i^{(L)}} \cdot a_j^{(L-1)}$
  end for
end for
$\frac{\partial C}{\partial a^{(L-1)}} \leftarrow W^T \frac{\partial \vec{C}}{\partial z^{(L)}}$
if *btspInput* **then**
  $bootstrapEach(\frac{\partial \vec{C}}{\partial a^{(L-1)}})$
end if
return $\{\vec{\nabla}_b, \nabla_W, \frac{\partial \vec{C}}{\partial a^{(L-1)}}\}$

5.3 Efficient level management

Since our implementation of neural networks is based on a leveled homomorphic encryption scheme, the CKKS scheme, modulus level management becomes integral to the success of this implementation. Three main influences that the modulus level of ciphertext has on our neural network implementation are memory, speed, and amount of bootstrapping required. As highlighted in table 1, a higher ciphertext level correlates with a larger ciphertext size; moreover, it takes more computational power to perform operations on ciphertext with a higher modulus level, and the bootstrapping procedure is necessary but computationally expensive to perform. With this variation in memory and computational requirements, it is best to design a neural network in a way that the change in ciphertext's level will remain consistent throughout the training process. This means that the weights and biases ciphertexts' modulus level in a layer will always be constant, even after each update. This will allow us to design a neural network that takes advantage of the ciphertext level to optimize our training speed.

To optimize the efficiency of neural network training on encrypted data we need to maximize the number of operations done on ciphertext with a lower level while minimizing the number of the costly bootstrapping

procedure. This simply means that a layer with high trainable parameters should be evaluated at the lowest level possible, accelerating operation speed and decreasing memory requirements, when there's a good trade-off with the number of bootstrapping procedures needed to be performed.

5.4 Scale management

Scale of a ciphertext is another important factor when performing operations on encrypted data as it determine the accuracy of encrypted data and also the correctness of the bootstrapping operations: a scale that is too high or too low could result in incorrect bootstrapping; therefore, we suggest keeping the scale of the weight in a layer that contain bootstrapping operations in mind when designing a neural network algorithm to train on encrypted datasets.

6 Experimental results

With overall higher computational costs, neural network training on encrypted data will be suited for certain tasks with the good trade-off between the benefits that it provides, such as security and privacy, and its drawbacks, such as the higher computational and memory cost. In this section, we will demonstrate the experimental results of neural network training on encrypted data showing exactly how much higher the computational cost is, and break down the time taken to perform certain operations. This experimental result may be helpful for those considering training a neural network on encrypted data.

6.1 Experimental test environment

The experimental results that will be shown in this section were all implemented using the CKKS scheme on the open-source Lattigo library [35] in Go, and our computations were completed on an AWS EC2 server: specifically, the r5.24xlarge EC2 instance that uses Intel Xeon Platinum 8000 series processor with 96 vCPU and 768 GB of memory.

6.2 CKKS encryption parameters

Set	h	N	$\log(QP)$	L	$\log(q_i)$				$\log(p_j)$
					$q_{0\leq i\leq(L-k)}$	StC	Sine	CtS	
I	192	2^{16}	1546	24	$60+9\cdot 40$	$3\cdot 39$	$8\cdot 60$	$4\cdot 56$	$5\cdot 61$

Table 2: Detail of bootstrappable full-RNS CKKS parameter set I from [28]. h represents the secret-key density and q represents a modulus size. $+$ denote concatenation into the chain and $a\cdot b$ denotes consecutive concatnation of a moduli of size b. This table is adapted from [28]

With bootstrapping, it is important to use the correct encryption parameters to ensure the accuracy, efficiency and security of ciphertexts. Here, we utilize parameter I proposed in [28] that was also implemented in the Lattigo library. With this parameter a ciphertext will have up to 24 modulus level when initially encrypted. When bootstrapping 15 level is consumed, leaving up to 9 maximum post bootstrapping level. This parameter, with $N = 2^{16}$, has the total of $N/2$ or 2^{15} slots. The detail of this parameter is shown in table 2.

6.3 Concurrent operations

Operation	Number of operations performed concurrently			
	50	100	150	200
Addition	3.44s.	5.78s.	5.33s.	6.85s.
Multiplication	1.92s.	4.12s.	8.17s.	8.1s.
Bootstrap	83.14s.	153.82s.	197.78s.	253.56s.

Table 3: Time recorded when performing each operation on CKKS ciphertext concurrently with different numbers of operation performed simultaneously. The addition and multiplication operations were performed between two ciphertexts with modulus level 2.

Since our implementation is done on Go, a language with native support for concurrency, we can utilize it to accelerate each homomorphic operation by completing multiple computations simultaneously on different threads of the CPU. This can speed up our computations tremendously; for example, as shown in table 3, 200 bootstrapping operations can be performed in just 253.56 seconds, or the average of 1.27 seconds per ciphertext. This is what we utilized in our implementation of the neural network training on encrypted data algorithms.

6.4 Model Structure

layer's name	forward propagation					backward propagation			
	input		output		btp	output level			btp
	size	level	size	level		bias	weights	loss wrt. input	
FC layer	784	2	20	$1 \rightarrow 9$	true	7	6	-	false
Tanh	20	9	20	7	false	-	-	8	false
FC layer	20	2	10	$1 \rightarrow 9$	true	9	6	$1 \rightarrow 9$	true
Softmax (M=1/20)	10	9	10	$3 \rightarrow 9$	true	-	-	9	false

Table 4: Implemented model structure and level management for both forward and backward propagation where $x \rightarrow y$ means that level x was bootstrapped to level y.

The model that we implemented is a model with two hidden layers and a total of 15,910 trainable parameters. This model is designed to train on the MNIST dataset [27] with a total of 784 input and 10 output corresponding to the one-hot encoded label. We utilize the HE-friendly approximate Tanh activation function for the first layer and the approximate Softmax activation function for the last, turning the output of this neural network into a probability distribution ready for use with cross-entropy loss function. The detail on the structure and level management of this model is shown in table 4.

We have decided to bring the level of the input of the first layer down to a minimum of 2 which will leave output with 1 level to rescale and perform bootstrapping. This is because, as mentioned in section 5, a lower modulus level of ciphertexts means lower computational cost and memory cost, and since this layer has over 15,700 trainable ciphertexts and 15,680 ciphertext multiplication to be performed, having them at a lower modulus level will speed up this process significantly. We then bootstrap the outputs of this layer bringing their level up to 9 before calculating the activated output of this layer with the Tanh activation function. Since the approximate Tanh function we use is a degree 3 polynomial, it will consume 2 levels.

For the next fully connected layer, we also bring the modulus level of the input down to 2, since we will need to perform bootstrapping on the output as the approximate softmax activation function consume 7 levels. We then perform another bootstrapping: this time after the approximate softmax activation function, raising the level of the loss gradient up to the 9 to minimize the number of bootstrapping needed for backpropagation.

We then propagate backward until the level goes to 1 at the loss gradient with regards to the input of the second fully connected layer. Here we perform the only bootstrapping in the backpropagation process. With this arrangement, we will perform a total of 60 bootstrapping when forward and backward propagating. Most of the operations are also done on low modulus level ciphertexts which maximizes the training speed.

6.5 Training duration and accuracy

layer's name	forward	forward btp	backward	backward btp	batch gradient descent
FC layer	90.72s.	35.65s.	148.46s.	-	373.57s.
Tanh	1.34s.	-	1.15s.	-	-
FC layer	1.66s.	32.52s.	3.4s.	35.98s.	5.40s.
Softmax	12.98s.	32.66s.	0s.	-	-

Table 5: Time recorded when training a neural network on encrypted data.

From our experiment, it took approximately 775.49 seconds, or 12.92 minutes, to perform the forward propagation, backward propagation, and batch gradient descent with a single batch. The process that took the longest is the batch gradient descent since this process requires the calculation of inner-sum of a ciphertext, which requires $\log(N/2-1)$ rotation operations to be perform. The bootstrapping processes is also computationally expensive which result in longer computing time.

		Training example			
		15,000	30,000	45,000	60,000
Testing accuracy	Plain	62.34%	71.39%	75.69%	78.30%
	Encrypted	62.23%	71.67%	74.54%	76.57%
Encrypted training correctness		99.82%	100.39%	98.48%	97.79%

Table 6: Comparison between the accuracy of a neural network trained on plain MNIST dataset and encrypted MNIST dataset taken after training on 15,000 - 60,000 unique training data. The accuracy shown is tested on 10,000 seperate testing data. Both plain and encrypted model are trained with the same algorithm and parameters with batch size of 2,500 and the learning rate of 0.5 over 1 epoch. Testing for accuracy of the encrypted model is done with the decrypted weights and biases. The correctness is calculated by dividing the encrypted model accuracy with the plain model accuracy

Our experiment proved that a neural network trained on the encrypted MNIST dataset can achieve a result that accurately resembles the one trained on normal MNIST dataset, as shown in table 6, with over 97.79% correctness when 1 epoch of training was completed. However, one main difference between the training of these two model is the training run time: the plain model took less than half a minute to train, while the encrypted model took over 5 hours to train with 1 epoch.

7 Conclusions

In this work, we have demonstrated that the training of a neural network consisting of 2 fully connected layers with approximate HE-friendly activation functions on the MNIST datasets encrypted with the CKKS homomorphic encryption scheme with bootstrapping is possible without the need of any decryption. Moreover, it produced a great result comparable to those trained on normal plain datasets. We strongly believe that our work can be applied to build a deeper model that consists of more than 2 fully connected layers to train on any encrypted dataset. We also believe there are various real-world use-cases for our method of privacy-preserving machine learning (PPML) that can benefit from the security that the training of neural networks on encrypted data without any decryption can provide.

A more efficient implementation of neural networks with HE in future works can improve what we have achieved by speeding the training process up while requiring less computational power; this will make this method of PPML a more viable option for more cases by enabling various benefits that this method provides, such as a high degree of security, to outweigh its drawbacks, which are mainly its computational overheads.

The training of other types of neural networks on encrypted data, such as the Convolution Neural Network, is also possible which could allow models such as the object detection models to be trained securely on encrypted data.

8 Acknowledgement

This paper was written under the guidance of Dr.Ronnakorn Vaiyavuth (ronnakorn.v@chula.ac.th) and Dr.Poomjai Nacaskul (poomjai.n@chula.ac.th) for a Startup project: Perm.

References

[1] Council of European Union, "General data protection regulation," 2016. https://eur-lex.europa.eu/eli/reg/2016/679/oj.

[2] R. Rivest, L. Adleman, and M. Dertouzos, "On data banks and privacy homomorphisms," in *Foundations on Secure Computation, Academia Press*, pp. 169–179, 1978.

[3] A. Wood, K. Najarian, and D. Kahrobaei, "Homomorphic encryption for machine learning in medicine and bioinformatics," *ACM Comput. Surv.*, vol. 53, Aug. 2020.

[4] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, (New York, NY, USA), p. 169–178, Association for Computing Machinery, 2009.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, (New York, NY, USA), p. 309–325, Association for Computing Machinery, 2012.

[6] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: Fast fully homomorphic encryption over the torus." Cryptology ePrint Archive, Report 2018/421, 2018. https://ia.cr/2018/421.

[7] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *Advances in Cryptology – ASIACRYPT 2016* (J. H. Cheon and T. Takagi, eds.), (Berlin, Heidelberg), pp. 3–33, Springer Berlin Heidelberg, 2016.

[8] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers." Cryptology ePrint Archive, Report 2016/421, 2016. https://ia.cr/2016/421.

[9] N. Smart and F. Vercauteren, "Fully homomorphic simd operations." Cryptology ePrint Archive, Report 2011/133, 2011. https://ia.cr/2011/133.

[10] J. H. Cheon, D. Kim, D. Kim, H. H. Lee, and K. Lee, "Numerical method for comparison on homomorphically encrypted numbers." Cryptology ePrint Archive, Report 2019/417, 2019. https://ia.cr/2019/417.

[11] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks." Cryptology ePrint Archive, Report 2021/091, 2021. https://ia.cr/2021/091.

[12] M. Kim, Y. Song, S. Wang, Y. Xia, and X. Jiang, "Secure logistic regression based on homomorphic encryption: Design and evaluation." Cryptology ePrint Archive, Report 2018/074, 2018. https://ia.cr/2018/074.

[13] A. Kim, Y. Song, M. Kim, K. Lee, and J. H. Cheon, "Logistic regression model training based on the approximate homomorphic encryption." Cryptology ePrint Archive, Report 2018/254, 2018. https://ia.cr/2018/254.

[14] S. Park, J. Byun, J. Lee, J. H. Cheon, and J. Lee, "He-friendly algorithm for privacy-preserving svm training," *IEEE Access*, vol. 8, pp. 57414–57425, 2020.

[15] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, p. 201–210, JMLR.org, 2016.

[16] H. Chabanne, A. de Wargny, J. Milgram, C. Morel, and E. Prouff, "Privacy-preserving classification on deep neural network." Cryptology ePrint Archive, Report 2017/035, 2017. https://ia.cr/2017/035.

[17] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks." Cryptology ePrint Archive, Report 2017/1114, 2017. https://ia.cr/2017/1114.

[18] C. Boura, N. Gama, M. Georgieva, and D. Jetchev, "Simulating homomorphic evaluation of deep learning predictions." Cryptology ePrint Archive, Report 2019/591, 2019. https://ia.cr/2019/591.

[19] A. A. Badawi, J. Chao, J. Lin, C. F. Mun, J. J. Sim, B. H. M. Tan, X. Nan, K. M. M. Aung, and V. R. Chandrasekhar, "Towards the alexnet moment for homomorphic encryption: Hcnn, thefirst homomorphic cnn on encrypted data with gpus," 2020.

[20] T. Chen and S. Zhong, "Privacy-preserving backpropagation neural network learning," *IEEE Transactions on Neural Networks*, vol. 20, no. 10, pp. 1554–1564, 2009.

[21] A. Bansal, T. Chen, and S. Zhong, "Privacy preserving back-propagation neural network learning over arbitrarily partitioned data," *Neural Computing and Applications*, vol. 20, pp. 143–150, 2010.

[22] J. Yuan and S. Yu, "Privacy preserving back-propagation neural network learning made practical with cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 212–221, 2014.

[23] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai, "Privacy-preserving deep learning via additively homomorphic encryption," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 5, pp. 1333–1345, 2018.

[24] Q. Zhang, L. T. Yang, and Z. Chen, "Privacy preserving deep computation model on cloud for big data feature learning," *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1351–1362, 2016.

[25] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." Cryptology ePrint Archive, Report 2012/144, 2012. https://ia.cr/2012/144.

[26] K. Mihara, R. Yamaguchi, M. Mitsuishi, and Y. Maruyama, "Neural network training with homomorphic encryption," 2020.

[27] Y. LeCun, C. Cortez, and C. C. J. Burges, "The mnist database of handwritten digits." Online: http://yann.lecun.com/exdb/mnist/.

[28] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys." Cryptology ePrint Archive, Report 2020/1203, 2020. https://ia.cr/2020/1203.

[29] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption." Cryptology ePrint Archive, Report 2018/153, 2018. https://ia.cr/2018/153.

[30] H. Chen, I. Chillotti, and Y. Song, "Improved bootstrapping for approximate homomorphic encryption." Cryptology ePrint Archive, Report 2018/1043, 2018. https://ia.cr/2018/1043.

[31] O. Regev, "The learning with errors problem (invited survey)," in *2010 IEEE 25th Annual Conference on Computational Complexity*, pp. 191–204, 2010.

[32] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors." Cryptology ePrint Archive, Report 2015/046, 2015. https://ia.cr/2015/046.

[33] B. Li and D. Micciancio, "On the security of homomorphic encryption on approximate numbers." Cryptology ePrint Archive, Report 2020/1533, 2020. https://ia.cr/2020/1533.

[34] J. H. Cheon, S. Hong, and D. Kim, "Remark on the security of ckks scheme in practice." Cryptology ePrint Archive, Report 2020/1581, 2020. https://ia.cr/2020/1581.

[35] "Lattigo v2.3.0." Online: https://github.com/ldsec/lattigo, Oct. 2021. EPFL-LDS.

[36] S. Hong, J. H. Park, W. Cho, H. Choe, and J. H. Cheon, "Secure multi-label tumor classification using homomorphic encryption," July 2021.

[37] R. E. Goldschmidt, "Applications of division by convergence." Thesis (M.S.)–Massachusetts Institute of Technology, Dept. of Electrical Engineering, 1964. http://hdl.handle.net/1721.1/11113.

[38] T. M. Mitchell, *Machine Learning*. USA: McGraw-Hill, Inc., 1997.