Research Article

t-BMPNet: Trainable Bitwise Multilayer Perceptron Neural Network over Fully Homomorphic Encryption Scheme

Joon Soo Yoo and Ji Won Yoon

School of Cybersecurity, Korea University, 145 Anam-ro, Anam-dong, Seongbuk-gu, Seoul, Republic of Korea

Correspondence should be addressed to Ji Won Yoon; jiwon_yoon@korea.ac.kr

Received 6 August 2021; Accepted 15 October 2021; Published 4 December 2021

Academic Editor: Jungab Son

Copyright © 2021 Joon Soo Yoo and Ji Won Yoon. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Homomorphic encryption (HE) is notable for enabling computation on encrypted data as well as guaranteeing high-level security based on the hardness of the lattice problem. In this sense, the advantage of HE has facilitated research that can perform data analysis in an encrypted state as a purpose of achieving security and privacy for both clients and the cloud. However, much of the literature is centered around building a network that only provides an encrypted prediction result rather than constructing a system that can learn from the encrypted data to provide more accurate answers for the clients. Moreover, their research uses simple polynomial approximations to design an activation function causing a possibly significant error in prediction results. Conversely, our approach is more fundamental; we present t-BMPNet which is a neural network over fully homomorphic encryption scheme that is built upon primitive gates and fundamental bitwise homomorphic operations. Thus, our model can tackle the nonlinearity problem of approximating the activation function in a more sophisticated way. Moreover, we show that our t-BMPNet can perform training—backpropagation and feedforward algorithms—in the encrypted domain, unlike other literature. Last, we apply our approach to a small dataset to demonstrate the feasibility of our model.

1. Introduction

1.1. Background. Homomorphic encryption is a cryptographic scheme that has long been considered holy grails for some cryptographers. Its intriguing property is that any function in plaintext can be constructed just as in the encrypted domain while maintaining the same functionality. Moreover, homomorphic encryption based on the learning with error (LWE) scheme [1] ensures high-level security even in the postquantum computing environment. Thus, by using homomorphic encryption, one can construct an environment that is both secure and private, since no information about the data being leaked to an adversary based on these properties.

In recent years, some global companies and institutions have strived to construct a system to provide secure and privacy-preserving services to the clients. In the system, the client sends the encrypted data along with its query to the cloud company; the client's encrypted data are evaluated in a

“magic box” to output an encrypted result. Next, the cloud sends back the result, and the client decrypts the encrypted output using its secret key to obtain the desired result. Homomorphic encryption enables constructing the “magic box,” in which the encrypted data are being processed without revealing any information to the cloud. Moreover, in some homomorphic encryption schemes, the client is unable to retrieve any information about the design of the circuit from the cloud. Therefore, homomorphic encryption can provide privacy for both the cloud and the client.

The primary source of our model (t-BMPNet) is homomorphic encryption and neural network. Particularly, we are interested in designing a multilayer perceptron neural network (MLPNN)—a foundational model of deep learning—under a fully homomorphic encryption (FHE) scheme. Our goal is to construct a model that can learn from the clients' encrypted data to update parameters in the network and provide accurate results for the clients. Specifically, we use the Boolean circuits approach in the FHE

scheme; plaintexts are encrypted in bit-by-bit basis and computations are expressed as Boolean circuits. Our code is freely available at <https://github.com/joonsooyoo/t-bmpnet>.

1.2. Related Works and Some Problems. In general, most of the works [2, 3] related to the “training” of neural networks such as the DLPNN and convolutional neural network (CNN) are performed in a nonencrypted state (Figure 1(a)). As a representative example, Gilad-Bachrach et al. [2] proposed CryptoNet which is based on the modular arithmetic FHE scheme to construct a CNN model. In this model, a client uses a public-key homomorphic cryptosystem to encrypt data $\mathcal{D}_A$ by her public key pk_A and send it to the server that has already pretrained the network. The server returns the evaluated result $\text{Enc}_{pk}(f(\mathcal{D}_A))$ to the original source, and the user can decrypt the prediction result by sk_A . Thus, the limitation is trivial—it can only provide prediction results through the pretrained neural network that performs only the feedforward algorithm. Likewise, CryptoDL [3] can be categorized in this line of research where the training is executed in advance unencrypted.

The construction of a nonlinear sigmoid function is a critical problem in designing a neural network. However, much of the literature approach this problem with a rather simple idea; they approximate the sigmoid function with polynomials. This is because their works are mainly based on homomorphic binary operations—addition and multiplication. Mohassel and Zhang [6] used Taylor series to approximate the sigmoid function in logistic regression. Kim et al. [7] improved the accuracy of the polynomial approximation by using the least square method. Designing the nonlinear function with a polynomial approximation is apparently working in some sense; however, for the sake of designing arbitrary nonlinear functions, it cannot be applied in every circumstance. Also, it does not guarantee high accuracy as well; a more general approach should be discussed.

The work that includes training in the encrypted phase proposed by Phong et al. [4, 5]. They demonstrated training the multilayer perceptron neural network with the partial homomorphic operation, namely, additive operations (Figure 1(b)). The model learns from the client (A_i)’s data by their given gradient vectors $\alpha \nabla c_{A_i}$ homomorphically added to the global weight parameters $\mathbf{p}_{\text{global}}$. The server interacts with clients multiple times to enhance the optimization of the parameters, and later, these parameters $\text{Enc}_{pk_{A_i}}(\mathbf{p}_{\text{global}})$ are distributed to each client. The scheme is suitable for the model that assumes the client to be an honest entity such as a big organization; however, it is vulnerable to a semihonest or malicious client where the client can violate the circuit privacy [8] of the server. Moreover, the calculation of gradient vectors $\alpha \nabla c_{A_i}$ adds a potential burden to the client.

1.3. Contributions. From this point of view, our key contributions of this study are summarized as follows:

- (i) We propose a novel approach of implementing the most accurate homomorphic sigmoid function among the existing literature from the basis of bitwise operations
- (ii) We present t-BMPNet—a general framework for designing a multilayer perceptron neural network over a fully homomorphic encryption scheme that performs training in the encrypted domain
- (iii) We propose a trainable FHE neural network under the minimum interaction between the client and the server compared to other FHE trainable neural networks
- (iv) Our approach broadens the horizon of feasibility in an application to various deep learning studies that require a secure cloud computing model

2. Preliminaries

2.1. Homomorphic Encryption. To discuss a simple notion of homomorphic encryption, we consider two messages m_0 and m_1 from a message space $\mathcal{M}$ and their corresponding ciphertexts c_1 and c_2 . We say that the encryption scheme is additively homomorphic if there exists an operation $*$, such that $\text{Enc}(m_1 + m_2) = \text{Enc}(m_1) * \text{Enc}(m_2)$, where $*$ is not necessarily be the same as the addition in the plaintext. Generally, $*$ operation is more complex and requires more computation than the normal addition. Likewise, we can consider the multiplicative homomorphism in the same way.

We refer to fully homomorphic encryption (FHE) when both of the algebraic operations are supported. Initially, the idea of FHE was proposed by Rivest et al. [9] in 1978 and had not been successful until 30 years later in 2005; the first attempt was realized by Boneh et al. [10]. They managed to perform a somewhat homomorphic encryption scheme (SWHE) that allows numerous additions, but the caveat is that only a single multiplication can be performed. Before then, public cryptosystems such as RSA and Paillier [11] use partial homomorphisms allowing only a single algebraic operation that is multiplication and addition, respectively. Typically, when we refer to a homomorphic encryption scheme, they follow a structure of basic 4 steps: key generation, encryption, decryption, and evaluation. Formally, the definition of HE is as follows.

Definition (public-key homomorphic encryption): homomorphic encryption is a probabilistic polynomial-time algorithm that involves four stages of the steps as follows.

- (i) Key generation: the algorithm takes a security parameter κ as an input and outputs a secret key sk , a public key pk , and an evaluation key ek . We write $\text{KeyGen}(1^\kappa) \rightarrow (sk, pk, ek)$.

FIGURE 1: Related works. (a) Pretrained neural network [2, 3]. (b) Backpropagation using additive HE [4, 5].

- (ii) Encryption: the algorithm takes a single message bit m_i from the message space $\mathcal{M} (= \{0, 1\})$ and a public key pk to output a ciphertext c_i of a single bit. We write $\text{Enc}(m_i, pk) \rightarrow c_i$.
- (iii) Decryption: the algorithm takes a secret sk and its corresponding ciphertext c_i and outputs a message m_i .
- (iv) Evaluation: the algorithm takes l bits of ciphertexts $c_1, c_2, \dots, c_l$ with a function f to output a ciphertext c^* using an evaluation circuit Eval. We write $\text{Eval}(f, c_1, c_2, \dots, c_l) = c^*$. Also, the evaluation circuit Eval needs to satisfy $\text{Dec}(c^*) = f(m_1, m_2, \dots, m_l)$.

In 2009, Gentry and Boneh [12] made a significant breakthrough in FHE in which its contribution can be divided into two pieces: leveled HE (or LHE in short) and bootstrapping. LHE supports both addition and multiplication, while the number of operations is limited. The reason for the restricted number of operations is that noise is accumulated after an evaluation of a Boolean circuit. After some number of evaluations of Boolean circuits (depending on the level of noise parameter designated) because of the noise accumulation, the ciphertext is not guaranteed to provide a correct decryption output. The problem of noise is handled by the bootstrapping procedure such that the noise is reduced after a “well-calculated” number of operations in the ciphertext in order to provide a “fresh” ciphertext.

Various FHE schemes have been actively proposed and improved based on the work of Gentry et al. in 2013 [13]. These FHE schemes provide basic homomorphic addition and multiplication in common; however, they have several different features; it is important to select the right scheme for implementation considering different aspects of the schemes. In general, FHE schemes can be classified into three different categories—Boolean circuits, modular arithmetic, and approximate number arithmetic.

Note that each encryption scheme shows different performance in terms of precision, accuracy, and throughput; we briefly describe each scheme to help the understanding of the experimental section that includes comparisons of our scheme with the existing literature that are constructed based on different FHE schemes. In particular, we provide more details and emphasis on the Boolean circuit method since our work is based on this approach.

2.1.1. Modular Arithmetic Approach (BFV). The modular arithmetic approach is generally used without bootstrapping (leveled HE) and thus perform a fast evaluation of ciphertexts. It is based on integer arithmetic and provides the exact result after decryption. This approach is efficient for SIMD computations over vectors of integers and supports fast scalar multiplication. In this study, we provide an encryption

scheme proposed by Bos et al. [14] which is one of the modular arithmetic approaches and is the basis encryption model for CryptoNets [2].

The encryption model takes a plaintext message m from a ring $R_t^n = \mathbb{Z}_t[x]/(x^n + 1)$ to the ciphertext c of a ring R_q^n . Observe that the rings have coefficients of integers over the modular space t and q . For a secret key polynomial $f \in R_q^n$, we choose two random polynomials f' and g in R_q^n , such that f satisfies an equation $f = tf' + 1$ as the first criterion of choosing a secret key. Next, we check if f has an inverse for the public key $h = tgf^{-1}$, or else, we discard f and iterate steps till we obtain the keys that satisfy the criterion.

Now, we encrypt message m by the public key h :

$$c = \left[\left\lfloor \frac{q}{t} \right\rfloor m + e + hs \right]_q, \quad (1)$$

where e, s are the random noise polynomials and $[a]_q$ is the reduction of the coefficients of $a \bmod q$ to the symmetric interval around 0 with the same length q . The decryption process simply takes the ciphertext c and performs multiplication followed by rounding and modular operations: $m = [t/qc]_t$.

For the addition of two messages m_1 and m_2 , it is done by simply adding two corresponding ciphertexts c_1 and c_2 ; one can verify by taking decryption of $c_1 + c_2$ to show that it matches with the result $m_1 + m_2$. However, in the case of the multiplication, an additional step, the relinearization process is required, so that the secret polynomial remains f (not f^2) as it is after the multiplication of two ciphertexts c_1 and c_2 .

One last note is that this modular encryption scheme takes only integers for the input values; it cannot handle floating-point arithmetics. As we will discuss in the later sections, this feature prevents the calculation of nonlinear sigmoid function by approximated polynomials since the coefficients are represented as real numbers. Thus, when designing the neural network (e.g., CryptoNets [2]) with the modular arithmetic approach, the calculation of sigmoid function is a huge obstacle.

2.1.2. Approximate Number Arithmetic Approach (CKKS). The approximate number arithmetic method is the most recently published FHE scheme proposed by Cheon et al. [15] in 2017 and considered as a nearly practical HE scheme to compute over real data. It can perform efficient SIMD computations over vectors of real numbers using batching and evaluates fast polynomial approximation. It demonstrates effectiveness in deep approximate computations such as logistic regression learning and often used without bootstrapping (i.e., leveled HE). One distinctive feature is that the result of the encryption model is approximated; the result includes an error that was generated from the encryption process. Thus, contrary to the modular arithmetic approach, it provides an approximate (not exact) result.

One notable feature that makes HEAAN distinguishable from other HE schemes is its encoding (and decoding) technique from a vector of complex (or real) numbers $\mathbb{C}^{n/2}$ to plaintext message space of a polynomial ring R^n by an isomorphic mapping of ϕ , such that $\phi: \mathbb{Z}[x]/$

$(x^n + 1) \rightarrow \mathbb{C}^{n/2}$. Then, a plaintext vector of real numbers $\mathbf{z} = (z_1, z_2, \dots, z_{n/2})$ can be encoded as a plaintext message of a polynomial $m[x]$ by computing $\Delta \cdot \phi^{-1}(\mathbf{z})$, where $\Delta > 1$ is a scaling factor.

We omit the detailed process of the model [15], but provide the simplified methodology of the whole process to compare with other approaches. First, the parameter generation process outputs n , a modulus $Q = q^2$, and discrete Gaussian distribution χ selected from the choice of a security parameter λ . KeyGen yields $sk \leftarrow (1, s)$ and $pk \leftarrow (b = -a \cdot s + e, a)$, where a secret polynomial $s \in R$ of its coefficients randomly selected from a sparse distribution $\{0, 1, -1\}^n$, a polynomial a sampled uniformly random from R_q^n and $e \leftarrow \chi$. Also, $evk = (b', a')$ is selected by polynomials s', a', e' , such that $s' \leftarrow s^2$, $a' \leftarrow R_q^n$, and $e' \leftarrow \chi$ that are combined to output $b' \leftarrow -a' \cdot s + e' + q \cdot s' \pmod{Q}$.

Enc takes pk and outputs $c \leftarrow v \cdot pk + (m + e_0, e_1)$, where $e_0, e_1 \leftarrow \chi$, and a polynomial v of its coefficients is randomly chosen from $\{0, 1, -1\}$. For Dec of $c = (c_0, c_1)$, $m \leftarrow c_0 + c_1 \cdot s \pmod{q}$. For addition of two ciphertexts c and c' , we simply $c_{\text{add}} \leftarrow c + c'$. Last, a multiplicative result of two ciphertexts $c = (c_0, c_1)$ and $c' = (c'_0, c'_1)$, and we let $(d_0, d_1, d_2) = (c_0 \cdot c'_0, c_0 \cdot c'_1 + c_1 \cdot c'_0, c_1 \cdot c'_1) \pmod{q}$. Then, $c_{\text{mult}} \leftarrow (d_0, d_1) + q^{-1} \cdot d_2 \cdot evk \pmod{q}$.

As discussed in the later section, CKKS is not suitable for evaluating complex circuits, particularly in a neural network with a nonlinear activation function. This is because CKKS primarily utilizes addition and multiplication; it inevitably requires approximation by polynomials for nonlinear functions. In contrast, our approach (bitwise operation) can benefit from atomic operations, resulting in better accuracy and less multiplicative depths.

2.1.3. Boolean Circuit Approach (TFHE). Unlike the previous two approaches, modular and approximate, the Boolean circuit method considers plaintext as bits and evaluates an arbitrary function by a sequence of Boolean gates. As of the proposal from Gentry et al. in 2013 [13], FHEW [16] and TFHE [17] schemes are the most eminent form of this approach. In particular, TFHE has made some significant improvements from FHEW in terms of faster computation of bootstrapping of noise generated from Boolean gates switching to a nearly practical scheme. Specifically, an execution time of a binary gate (AND, OR, and NAND) is about 13 milliseconds single core time by a factor of 53 improvement compared to FHEW. Therefore, throughout the study, our work utilizes the TFHE scheme for implementation; note that our work is not limited only to the TFHE scheme but can also be demonstrated through other Boolean circuit methods.

Our work considers two factors: (1) encrypted bits and (2) evaluation of Boolean circuit that works on encrypted bits. We explain in detail designing a Boolean circuit (e.g., neural network) that operates on encrypted bits in the later section of the study. But we emphasize to the readers that our approach is very different from the perspective of the previous approaches that evaluate integer or floating-point arithmetic circuits using basic operations such as HE

addition and multiplication. Also, these works most often use a leveled version of FHE for the evaluation of circuits but fail to evaluate the deep depth of a circuit. However, they almost surely guarantee faster performance time than the Boolean circuits method in shallow network circuits. But, since our work is based on the TFHE scheme, we can evaluate an arbitrary depth of circuit facilitated by the bootstrapping procedure after an evaluation of each bootstrapping gate.

We briefly explain the basics of TFHE to bridge the understanding of its underlying work and our approach. TFHE operates over the real torus $\mathbb{T}$, that is, $\mathbb{R}/\mathbb{Z}$ of real numbers mod 1. Notice that $\mathbb{T}$ is an additive Abelian group; however, it is not a ring (not closed under multiplication). Instead, $\mathbb{T}$ is a $\mathbb{Z}$ -module enabling a mapping of $\mathbb{Z}$ with $\mathbb{T}$ under operation: $\mathbb{Z} \times \mathbb{T} \rightarrow \mathbb{T}$ with some properties [17]. Based on the newly defined torus $\mathbb{T}$, we can encrypt a message $\mu \in \{0, 1/4\}$ by $c = \mathbf{a} \cdot \mathbf{s} + u + e$, where $\mathbf{s} \leftarrow \mathbb{B}^n$ is a secret key for $\mathbb{B} = \{0, 1\}$, $\mathbf{a} \leftarrow \mathbb{T}^n$ is an uniform-random LWE sample (or TLWE), and $e \leftarrow \chi$, where χ is a sub-Gaussian distribution. One can retrieve the original message μ by (traditionally) performing $c - \mathbf{a} \cdot \mathbf{s}$ and rounding it to the nearest message in the message space.

The novelty of TFHE lies in the bootstrapping procedure of refreshing the ciphertext's noise—taking most of the execution time—by defining an external product $\boxtimes$ between TGSW and T (R) LWE: $\text{TGSW} \times \text{TLWE} \rightarrow \text{TLWE}$, where TGSW and T (R) LWE are the polynomials in $\mathbb{Z}[x]$ and $\mathbb{T}[x]/(x^n + 1)$, respectively. In short, with a newly defined external product, TFHE manages to improve the performance time for bootstrapping that includes a series of procedures: $\text{KeySwitch} \circ \text{SampleExtract} \circ \text{BlindRotate}$ (We refer the interested readers to [17, 18] for more details.).

We can construct bootstrapping FHE Boolean gates using the previous procedures. For example, AND bootstrapping gate between TLWE samples c and c' over the message space $\{0, 1/4\}$ can be constructed by $((0, -1/8) + c + c')$, followed by the bootstrapping procedure. Likewise, other FHE Boolean gates such as OR, XOR, and NAND are designed. In practice, TFHE uses a message μ from the message space $\{-1/8, 1/8\}$ for encryption. Note that $-1/8$ and $1/8$ correspond to 0 and 1, respectively, from the integer space $\mathbb{B}$ for the Boolean arithmetics.

Note that the detailed process of TFHE is out of scope to understand our work; the importance lies in understanding the difference in the TFHE approach to other methods. Based on the bootstrapping FHE gates provided by TFHE—10 binary gates such as NAND, NOT, AND, and OR—we can play with encrypted bits and gates to construct an arbitrary circuit for evaluation (in our work, the neural network).

2.2. Bitwise Operations. We start from the ground truth that any evaluation FHE function, in theory, can be devised from the universal homomorphic gates. Some FHE schemes [13, 16, 18] allow the construction of FHE Boolean gates such as AND and OR that support bootstrapping of noise after each operation. Therefore, we utilize the FHE Boolean

gates from the scheme and construct more complicated evaluation functions based on bitwise operations. However, unlike operations on the plaintext, it is necessary to consider designing evaluation functions from the worst-case scenario in FHE, or else, it means that the server knows about the path in the encrypted domain which is a contradiction. Having considered this fact, we designed various fundamental operations such as shift, compare, arithmetic, and nonlinear functions based on the bitwise operation, that is, considering movements of the encrypted bits. Table 1 provides the average execution time for the basic types of operations in our scheme.

Yoo et al. [19] explained the specific design of each bitwise function in detail, and therefore, we briefly introduce the concept of its design in this study.

2.3. Multilayer Perceptron Neural Network. The multilayer perceptron neural network (MLPNN) [20–22] is one of the representative algorithms of deep learning. It is a network of multiple layers that have perceptrons in each layer that play as neurons in our brain. It is a basic structure in the neural network system that endeavors to learn a representation of a given set of data by mimicking our brain system in a simple form. The technique is widely used in a variety of applications such as computer vision, speech recognition, and natural language processing.

2.3.1. Single Perceptron. To understand the structure of the network, we decompose its network by a set of layers $\mathcal{L}$ and analyze a single perceptron in a layer. A perceptron computes a prediction value as a linear summation $\sum_j w_j x_j + b$ with respect to the input $x_0, x_1, \dots$, where w_j and b represent a weight and a bias, respectively. More compactly, we can write as $\mathbf{w} \cdot \mathbf{x} + b$, where a bold lowercase letter and $\cdot$ represent a vector and an inner product operation, respectively. As an output for the single perceptron, it is a binary number taking 0 or 1. The output is determined by a threshold value 0, and thus, a rule for the perceptron can be written:

$$\text{Output} = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0. \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0. \end{cases} \quad (2)$$

Training the MLPNN is almost the same as training a single perceptron, but we append one more assumption to the structure; we want a small change in the weights and biases to cause a small change in the corresponding output. The concept can be immediately implemented when we define a sigmoid function σ that outputs a value between 0 and 1 with a “smoothness” property allowing differentiation of the function. In this way, the weighted sum is “squashed” in a small interval in which a small change in parameters affects a small change in the output. Also, the sigmoid function has a property that its derivative can be calculated by $\sigma' = \sigma(1 - \sigma)$ that involves only a single multiplication and a single subtraction. This algebraic property is especially helpful when we update the parameters in the back-propagation algorithm. Therefore, our rule for the output of the single sigmoid neuron is the following:

TABLE 1: Average execution time for the basic types of functions in our scheme with respect to $l = 8, 16, 32$ -bit measured in seconds.

Type	Operation	8-bit (s)	16-bit (s)	32-bit (s)
Shift	Left	$3.00E-06$	$4.00E-06$	$6.00E-06$
	Right	$8.00E-06$	$1.00E-05$	$1.20E-05$
Min/max	Minimum	0.53	1.07	2.13
	Maximum	0.53	1.10	2.15
Compare	Equivalent	0.22	0.45	0.90
	Larger than	0.32	0.67	1.29
Basic	Two's complement	0.18	0.40	0.84
	Addition	0.51	1.06	2.21
	Subtraction	0.51	1.07	2.17
	Multiplication	5.61	22.95	89.41
	Division	8.87	34.59	133.55
Nonlinear	Exponential	18.17	69.57	285.56

$$\text{Output} = \begin{cases} 0 & \text{if } \sigma(z) \leq 0 \\ 1 & \text{if } \sigma(z) > 0 \end{cases}, \quad (3)$$

where $\sigma(z) = 1/(1 + e^{-z})$ and $z = \mathbf{w} \cdot \mathbf{x} + b$.

2.3.2. Feedforward. MLPNN has a series of layers $\{1, 2, \dots, L\} \in \mathcal{L}$ that is divided into three categories: an input layer, hidden layer (s), and an output layer. The input layer takes a set of data $\mathbf{d}_i \in \mathcal{D}$ for $i \in \{1, 2, \dots, N\}$, where each $\mathbf{d}_i$ is a vector of n_1 elements (n_L denotes the number of neurons in a layer L). The purpose of the network is to learn from the training data $\mathcal{M}_{\text{train}} \in \mathcal{M}$ that can estimate its corresponding label $\mathbf{y}$ as accurately as possible. This whole process of the training can be divided into two steps: feedforward and backpropagation algorithms.

The feedforward algorithm takes an encoding of data that are fed into the input layer of $a_0^{(1)}, a_1^{(1)}, \dots, a_{n_1}^{(1)}$. Then, in each layer k , we perform the same procedure as in the derivation of the outcome of the single sigmoid neuron with slightly different notations:

$$\begin{aligned} z_j^{(k+1)} &= \sum_{i=0}^{n_k-1} w_{i,j}^{(k+1)} a_i^{(k)} + b_j^{(k+1)}, \\ a_j^{(k+1)} &= \sigma(z_j^{(k+1)}). \end{aligned} \quad (4)$$

We refer to $w_{i,j}^{(k+1)}$ as the weight between the two neurons, $a_i^{(k)}$ and $a_j^{(k+1)}$. The equation (4) can be simplified as $\mathbf{z}^{(k+1)} = (\mathbf{W}^{(k+1)})^T \mathbf{a}^{(k)}$ and $\mathbf{a}^{(k+1)} = \sigma(\mathbf{z}^{(k+1)})$ in a matrix form. Therefore, the feedforward algorithm aims to calculate the value $\mathbf{a}^{(L)}$ —the likelihood that the data $\mathbf{d}_i$ are classified as the label j —in the output layer.

2.3.3. Backpropagation. The backpropagation algorithm is a popular way of training the neural network using the gradient descent method. Its purpose is to train the network, such that it can correctly at its best estimate the label given its data. In order to achieve this, we define the cost function C_i of a single training example $\mathbf{d}_i \in \mathcal{M}_{\text{train}}$ by

$$C_i = \frac{1}{2} \sum_{j=0}^{n_i} (a_j^{(L)} - y_j)^2, \quad (5)$$

where y_j is the desired value that takes a value of 0 or 1. The total cost function C for the given training dataset $\mathcal{M}_{\text{train}}$ is the mean squared error or MSE, that is, $C = (1/N_{\text{train}}) \sum_{i=1}^{N_{\text{train}}} C_i$. Therefore, our goal of training is to find the values for weights and biases that minimize the cost function.

We use the gradient descent algorithm which is finding partial derivatives of C with respect to $w_{i,j}^{(k)}$ and $b_i^{(k)}$ to iteratively adjust weights and biases:

$$\begin{aligned} w_{i,j}^{(k)} &\leftarrow w_{i,j}^{(k)} - \alpha \frac{\partial C}{\partial w_{i,j}^{(k)}}, \\ b_i^{(k)} &\leftarrow b_i^{(k)} - \alpha \frac{\partial C}{\partial b_i^{(k)}}, \end{aligned} \quad (6)$$

where α is a learning rate. We also denote gradient vector $\nabla \mathbf{c}_p$ which has elements of all partial derivatives $\partial C / \partial w_{i,j}^{(k)}$ and $\partial C / \partial b_i^{(k)}$ derived from a single data d_p . Therefore, for a general gradient descent algorithm, we update our parameters by the mean of gradient vectors $\nabla \mathbf{C} = 1/N_{\text{train}} \sum_{p \in \mathcal{D}_{\text{train}}} \nabla \mathbf{c}_p$ as in the equation (6).

3. Our Model: t-BMPNet over FHE

3.1. Overview. Our model is based on FHE that involves both the training and making predictions under the encrypted states (Figure 2). We set the model to be a server with a client participating in communication protocol where the client is “only” responsible for the encryption of the data and transmitting it to the server through a secure channel.

3.1.1. Threat Model. We assume that both the server and the client to be honest-but-curious entities which are different from [4, 5], where the clients are assumed as honest entities such as financial institutions or hospitals. In our model, the client should not necessarily be limited to these trustable

FIGURE 2: t-BMPNet learns from the encrypted data obtained from each client by performing feedforward and backpropagation algorithms.

systems but can be applied in a broader scope involving individuals.

3.1.2. High-Level Illustration. The communication protocol for learning from the client's encrypted data in our model is as follows (Figure 2):

- (1) Client A encrypts data and outsource the encrypted data $\text{Enc}(\mathcal{D}_A)$ using its public key pk_A to the server
- (2) The server evaluates feedforward and backpropagation algorithms in the encrypted state to learn from the client's data to optimize the global weight parameters
- (3) The server performs classification work of each data using the feedforward algorithm and sends the result $\text{Enc}(f(\mathcal{D}_A))$ to client A
- (4) Client A receives and decrypts the given encrypted result that outputs $f(\mathcal{D}_A)$

Based on this model, we propose (1) an encrypted training algorithm and (2) a more accurate design of the sigmoid function based on the bitwise operations on the ciphertexts as our main points of this study. Specifically, (1) and (2) are evaluation functions (Eval) that operate on ciphertexts.

3.2. Number Formatting, Encoding/Decoding, and Encryption/Decryption. Our number system uses a fixed-point number for the bitwise operation. We use a fixed-point number representation rather than a floating-point number representation due to the efficiency of designing various FHE functions.

3.2.1. Encoding. We encode $x \in \mathbb{R}$ to a plaintext vector $\mathbf{a}$ of size l : $(a_0, a_1, \dots, a_{l-1})$, where $a_i \in \mathbb{B}$ (or $\{0, 1\}$) and concatenated. To prevent confusion, we represent the plaintext $\mathbf{a}$ by $(\underbrace{a_0, a_1, \dots, a_{l/2-2}}_{\text{fractional part}} \mid \underbrace{a_{l/2-1}, \dots, a_{l-2}, a_{l-1}}_{\text{integer part}})$, where $\mid$ denotes the separation of the fractional part and integer part of $\mathbf{a}$. Notice that we enumerate the plaintext elements in the reverse order and allocate the left $l/2 - 1$ numbers of bits to the fractional part, $l/2$ to the integer part, and one for the signed bit (or a_{l-1}). For instance, $x = 2.5$ of a real number is encoded as $\mathbf{a} = (0, 0, 1 \mid 0, 1, 0, 0, 0)$ for the $(l = 8)$ -bit number system.

3.2.2. Decoding. A plaintext $\mathbf{a}$ is decoded to a real number x by a rule of usual conversion from the fixed-point number to a real number. For a positive $\mathbf{a}$, we perform $\sum_{i=l/2-1}^{l-1} 2^i a_i$, whereas for a negative $\mathbf{a}$, we perform the two's complement

of $\mathbf{a}$ followed by the same conversion to a real number x with different signs. For instance, 8-bit plaintext $\mathbf{a} = (0, 0, 1|0, 1, 1, 1, 1)$ is decoded to $x = -1.5$.

3.2.3. Encryption. The plaintext $\mathbf{a}$ is encrypted to a ciphertext $\text{ct}.\mathbf{a}$, where each a_i from the message space $\mathbb{B}$ is encrypted over the torus $\mathbb{T}$ under the same secret key $\mathbf{s}$ of the size n , where n is the security parameter. ct is equivalent to Enc , but we use both notations. As mentioned earlier in the preliminaries section, $-1/8$ and $1/8$ are mapped to the message space $\mathbb{B}$; each a_i is encrypted from a randomly selected LWE sample $\mathbf{t} \leftarrow \mathbb{T}^n$ by $c = \mathbf{t} \cdot \mathbf{s} + a_i + e$. We denote an encryption of $\mathbf{a}$ under a secret key $\mathbf{s}$ to be $\text{ct}.\mathbf{a}$. For example, we refer to an encryption of the plaintext $\mathbf{a} = (0, 0, 1|0, 1, 0, 0, 0)$ as $\text{ct}.\mathbf{a} = (\text{ct}.0, \text{ct}.0, \text{ct}.1|\text{ct}.0, \text{ct}.1, \text{ct}.0, \text{ct}.0, \text{ct}.0)$.

3.2.4. Decryption. As mentioned, we bootstrap noise after each gate operation; the noise of the ciphertext does not grow sufficiently large enough for the decryption circuit to output the incorrect message. Therefore, under the same original secret key $\mathbf{s}$, we are guaranteed to successfully retrieve the original plaintext message a_i from the ciphertext $\text{ct}.a_i$ by performing decryption such that $c - \mathbf{t} \cdot \mathbf{s}$ followed by a rounding operation (or taking expectation).

3.3. Some Basic Operations. In this section, we summarize concise details of some of the basic operations given in Table 1 to show insights into how we designed the basis of our system. For the full details of its construction, we recommend the readers to refer to [19, 23]. Note that we use notations $\oplus, \wedge, \vee$, and $\neg$ to denote logical gates XOR, AND, OR, and NOT that are homomorphically designed and provide bootstrapping after each operation. The following is a sketchy knowledge of some of the basic operations in our system.

3.3.1. Shift. Initially, the simplest case of all is the shift operation which basically is moving the bits in an array. Likewise, from a standpoint from the ciphertext bits, it is apparently the same as moving bits in a certain direction. Therefore, we can roughly write the right shift operation of $\mathbf{a}$ by k bits: $\text{ct}.a_{i+k} = \text{ct}.a_i$.

3.3.2. Addition. In a similar manner, the addition operation HE.Add between two ciphertexts $\text{ct}.\mathbf{a}$ and $\text{ct}.\mathbf{b}$ is designed using the binary full-adder circuit: (1) $\text{ct}.s_i = \text{ct}.a_i \oplus \text{ct}.b_i \oplus \text{ct}.c_i$ and (2) $\text{ct}.c_{i+1} = (\text{ct}.a_i \wedge \text{ct}.b_i) \wedge (\text{ct}.c_i \wedge (\text{ct}.a_i \oplus \text{ct}.b_i))$, where $\text{ct}.s_i$ and $\text{ct}.c_i$ represent the encrypted sum and carry at index i , respectively. Additionally, we improved and optimized the full-adder circuit considering the execution time for each homomorphic operation which is elaborated in [23].

3.3.3. Two's Complement. The two's complement method HE.Twos is used for storing a signed number in both the

integers and the real numbers in our system. It is an extension of the addition operation where we take 1's complement followed by the addition of 1: $\text{HE.Add}(\text{ct}.\mathbf{a}, \text{ct}.\mathbf{b})$, where $\text{ct}.\mathbf{a} = (\text{ct}.a_0, \text{ct}.a_1, \dots, \text{ct}.a_{l-1})$ and $\text{ct}.\mathbf{b} = (\text{ct}.1, \text{ct}.0, \dots, \text{ct}.0)$.

3.3.4. Subtraction. We define the subtraction operation HE.Subt using the previous operations: addition and two's complement. Simply, for the subtraction of two ciphertext inputs ($\text{ct}.\mathbf{a}$ and $\text{ct}.\mathbf{b}$), we take two's complement of $\text{ct}.\mathbf{b}$ followed by the addition of $\text{ct}.\mathbf{a}$ that leads to $\text{HE.Subt}(\text{ct}.\mathbf{a}, \text{ct}.\mathbf{b}) = \text{HE.Add}(\text{ct}.\mathbf{a}, \text{HE.Twos}(\text{ct}.\mathbf{b}))$.

3.3.5. Multiplication. We define the multiplication operation by a sum of products of bits where multiplication between two bits is just AND operation. Let $\mathbf{x}$ be the result of a product between two plaintext binary numbers $\mathbf{a} \geq 0$ and $\mathbf{b} \geq 0$. We define c_j by

$$c_j = \left(\underbrace{0, \dots, 0}_{j \text{ number of } 0's}, a_0 \wedge b_j, \dots, a_{l-1} \wedge b_j, \underbrace{0, \dots, 0}_{l-j \text{ number of } 0's} \right). \quad (7)$$

Then, we can express $\mathbf{x}$ by the sum of c_j 's of length $2l$: $\mathbf{x} = \sum_{j=0}^{l-1} c_j$. Typically, we take x_j 's of $\mathbf{x}$ from index $j = l/2 - 1$ to $j = 3l/2 - 1$ for the real number multiplication.

With this idea, we consider the multiplication of $\text{ct}.\mathbf{a}$ and $\text{ct}.\mathbf{b}$ for the case in the ciphertext. Since we have only considered multiplication in positive cases, and also cannot determine whether the $\text{ct}.\mathbf{a}$ and $\text{ct}.\mathbf{b}$ are positive or negative values, the algorithm for "encrypted" multiplication has to follow both of the cases. This is just an outline of the bitwise homomorphic multiplication, and for the interested readers, check [23] for more specific details of the algorithm.

For the sake of the flow of this study, we suggest the readers to refer to [19, 23, 24] for a more insightful understanding of the above operations as well as other homomorphic operations given in Table 1.

4. Learning from the Encrypted Data

4.1. Overall Process. Training a model involves mainly two phases (feedforward and backpropagation) that process back and forth to provide the optimal parameters that minimize the total cost $C^{(i)}$, where superscript (i) of C indicates the number of epoch for i^{th} number of training $\mathcal{D}_{\text{train}}$. The Algorithm 1 illustrates the whole process of the training in plaintext, where lines (7–10) and lines (12, 15, 16) are the feedforward and the backpropagation algorithms, respectively.

Our model that learns from the encrypted data is close to the model in the plaintext with some similarities and differences. We follow the same path as in Algorithm 1 for training; we calculate the cost followed by updating parameters $\mathbf{W}^{(k)}$ and $\mathbf{b}^{(k)}$. However, in the encrypted domain, since every parameters and data are encrypted, we need a different approach. Initially, we replace plaintext operations with homomorphic operations. For instance, in line 8, $\text{ct}.z_j^{(k)} = \text{HE.Add}(\sum_i \text{HE.Mult}(\text{ct}.W_{ij}^{(k)}, \text{ct}.a_i^{(k-1)}), b_j^{(k)})$.


```

Data:  $\mathcal{D}_{\text{train}} \subseteq \mathcal{D}$ 
Output:  $\mathbf{W}^{(k)}$  and  $\mathbf{b}^{(k)}$  for  $k \in \mathcal{L}$ 
(1) Initialize network parameters  $\mathbf{W}^{(k)}$  and  $\mathbf{b}^{(k)}$  for  $k \in \mathcal{L}$ 
(2)  $i \leftarrow 1$ ;
(3) While  $|C^{(i+1)} - C^{(i)}| < \epsilon$  do
(4)   Shuffle  $\mathcal{D}_{\text{train}}$ ;
(5)   For  $\mathbf{d}_j \in \mathcal{D}_{\text{train}}$  of  $j \leftarrow 1$  to  $N_{\text{train}}$  do
(6)      $\mathbf{a}^{(1)} \leftarrow \mathbf{d}_j$ ;
(7)     For  $k \leftarrow 2$  to  $L$  do
(8)        $\mathbf{z}^{(k)} = (\mathbf{W}^{(k)})^T \mathbf{a}^{(k-1)} + \mathbf{b}^{(k)}$ ;
(9)        $\mathbf{a}^{(k)} = \sigma(\mathbf{z}^{(k)})$ ;
(10)    End
(11)     $C_j = 1/2 \|\mathbf{a}^{(L)} - \mathbf{y}_j\|^2$ ;
(12)     $\nabla \mathbf{c}_j \leftarrow \partial C_j / \partial \mathbf{W}^{(k)}, \partial C_j / \partial \mathbf{b}^{(k)}$ ;
(13)  End
(14)   $C^{(i)} = 1/N_{\text{train}} \sum_{j=1}^{N_{\text{train}}} C_j$ ;
(15)   $\nabla \mathbf{C} = 1/N_{\text{train}} \sum_{j=1}^{N_{\text{train}}} \nabla \mathbf{c}_j$ ;
(16)  Update parameters  $\mathbf{W}^{(k)}, \mathbf{b}^{(k)}$  by  $\alpha \nabla \mathbf{C}$ ;
(17)   $i \leftarrow i + 1$ ;
(18) End
(19) Return  $\mathbf{W}^{(k)}$  and  $\mathbf{b}^{(k)}$  for  $k \in \mathcal{L}$ 

```

ALGORITHM 1: Training MLPNN in plaintext.

In particular, the sigmoid function $\sigma(\cdot)$ is crucially important in both feedforward and backpropagation. In the feedforward algorithm, the activation function is used in every node to compute $a_j^{(k)} = \sigma(z_j^{(k)})$ in line 9 for the input to the next layer. Moreover, in the backpropagation algorithm, the partial derivatives are derived from the sigmoid functions. Since line 12 is the step of obtaining partial derivatives $\nabla \mathbf{c}_j$ without details of the actual steps, we state its process in more details as follows:

$$\begin{aligned}
 \frac{\delta C_p}{\delta W_{i,j}^{(k)}} &= \frac{\delta C_p}{\delta a_j^{(k)}} \frac{\delta a_j^{(k)}}{\delta z_j^{(k)}} \frac{\delta z_j^{(k)}}{\delta W_{i,j}^{(k)}} = \xi_j^{(k)} \sigma'(z_j^{(k)}) a_i^{(k-1)} \\
 \frac{\delta C_p}{\delta b_j^{(k)}} &= \frac{\delta C_p}{\delta a_j^{(k)}} \frac{\delta a_j^{(k)}}{\delta z_j^{(k)}} \frac{\delta z_j^{(k)}}{\delta b_j^{(k)}} = \xi_j^{(k)} \sigma'(z_j^{(k)}) \\
 \frac{\delta C_p}{\delta a_i^{(k-1)}} &= \sum_{j=0}^{n_k-1} \frac{\delta C_p}{\delta a_j^{(k)}} \frac{\delta a_j^{(k)}}{\delta z_j^{(k)}} \frac{\delta z_j^{(k)}}{\delta a_i^{(k-1)}} \\
 &= \sum_{j=0}^{n_k-1} \xi_j^{(k)} \sigma'(z_j^{(k)}) W_{i,j}^{(k)},
 \end{aligned} \tag{8}$$

where $\xi_i^{(k)}$ is a partial derivative of C_p with respect to the i^{th} neuron at layer k . Equation (8) demonstrates that all the partial derivatives $\delta C_p / \delta W_{i,j}^{(k)}$, $\delta C_p / \delta b_j^{(k)}$, and $\delta C_p / \delta a_i^{(k)}$ have $\sigma'(z_j^{(k)})$ which can be calculated by $\sigma(z_j^{(k)})(1 - \sigma(z_j^{(k)}))$. Thus, we conclude that approximating the sigmoid function $\sigma(\cdot)$ accurately is significant in both feedforward and

backpropagation algorithms, thereby, increasing the accuracy of the learning model as a whole.

4.2. Key Operations. The sigmoid function $1/(1 + \exp(-x))$ mainly has four operations: addition, two's complement, exponential function, and division. The former two have been discussed in the previous section. Now, we explain the other two key functions for the construction of the sigmoid function in the encrypted domain. The general approach for deriving exponential function and division is discussed in this section; we provide concrete examples in the Appendix section followed by figures illustrating a detailed process of bitwise operations performed behind the scene.

4.2.1. Exponential Function. Our binary exponential function requires several steps to perform in the ciphertext (Figure 3). This is because the algorithms are different depending on the sign of the input value. First, suppose we only consider deriving 2^x of $x \geq 0$ in the plaintext. Then, we can divide x by the fractional part x_{frac} and integer part x_{int} . We express x in a binary vector $\mathbf{a}$ as follows:

$$\begin{aligned}
 \mathbf{a} &= \mathbf{a}_{\text{frac}} + \mathbf{a}_{\text{int}} \\
 &= (a_0, a_1, \dots, a_{(l/2)-1} | 0, \dots, 0) + \dots \\
 &\quad (0, \dots, 0 | a_{l/2}, a_{(l/2)+1}, \dots, a_{l-1}).
 \end{aligned} \tag{9}$$

The algorithm for positive x is as follows: (1) right shift $\mathbf{a}_{\text{frac}}$ by x_{int} , (2) right shift 1 in binary vector by x_{int} , and (3) add the results of (1) and (2).

FIGURE 3: Flow diagram of binary exponential function in ciphertext.

$$\begin{aligned}
 (a_0, a_1, \dots | 0, \dots, 0) &\longrightarrow \left(\underbrace{0, \dots, 0}_{\text{shift by } x_{int}}, a_0, a_1, \dots, 0 \right), \\
 (0, \dots | 1, 0, \dots, 0) &\longrightarrow \left(0, \dots | \underbrace{0, \dots, 0}_{\text{shift by } x_{int}}, 1, 0, \dots, 0 \right).
 \end{aligned} \quad (10)$$

Equation (10) illustrates details of the plaintext positive exponentiation algorithm in steps (1) and (2).

The algorithm for the plaintext negative exponentiation is different from that of the positive exponentiation in terms of direction and magnitude of the shift and subtraction instead of addition. We also need to initiate x to be a positive number by performing an absolute value operation in the beginning. We perform the algorithm with $|x|$: (1) left shift $|x|_{frac}$ by $|x|_{int} + 1$, (2) left shift 1 in binary vector by $|x|_{int}$, and (3) subtract result (2) by (1).

The result of the positive and negative exponentiation yields a set of line segments that approximates the exponential function, that is, a line segment connects a point $(k, 2^k)$ and a point $(k, 2^{k+1})$. A simple intuitive proof of the idea is that, for instance, step (2) in positive exponentiation corresponds to the point $(k, 2^k)$ in 2^x , and step (1) is a proportional increase of $2^{x_{frac}}$ with respect to 2^k . Therefore, $2^{x_{frac}}$ stays in the line from $(k, 2^k)$ to $(k, 2^{k+1})$.

We transform the above plaintext exponentiation to an encrypted version. To achieve this, the crucial task is to find the value of $ct.x_{int}$. In fact, it is impossible to find this value since, otherwise, it means that we know the value of x_{int} from $ct.x_{int}$ which is contradictory. Therefore, we must consider all possible cases that the exponentiation outcome can be. This task involves comparing values of $\text{Enc}(0)$ to $\text{Enc}((l/2) - 2)$ of all possible values that the value $ct.x_{int}$ can have by using HE.Equi function. In short, the function HE.Equi outputs $\text{Enc}(1)$ if the two given ciphertexts are equivalent, and otherwise $\text{Enc}(0)$ [19]. We denote $ct.o_1, ct.o_2, \dots, ct.o_{(l/2)-2}$ to be the results of all the comparisons. We also perform all possible cases of the exponentiation and call it $ct.p_i$ where $ct.p_i$ refers to exponentiation of $ct.x$ by $i - 1$ times. Next, we bitwise HE.Equi $ct.o_i$ and $ct.p_i$ pairwise, which outputs the result of exponentiation if $ct.o_i = ct.x_{int}$, otherwise $\text{Enc}(0)$. Finally, adding all the values of the results of HE.Equi ($ct.o_i, ct.p_i$) is the result of the exponentiation which considered all possible scenarios.

Up to this point, we have proceeded steps in the absolute value operation, all the possible cases of $ct.x_{int}$ and positive and negative exponentiation (Figure 3). Now, we consider the sign of the input a whether to perform the positive or negative exponentiation algorithm. We perform bitwise AND operation of the result of the positive exponentiation $ct.r_1$ with $ct.a_{l-1}$. The result is $ct.r_1$ if a is positive (i.e., sign bit $a_{l-1} = 0$ which means $a_{l-1} = 1$) and returns $ct.0$ if a is negative. Likewise, we perform the similar procedure: bitwise AND operates the result of negative exponentiation $ct.r_2$ with $ct.a_{l-1}$. The procedure only gives $ct.r_2$ whenever a is negative, and otherwise $ct.0$. Last, we add the two results which output $ct.r_1$ or $ct.r_2$ depending on the sign of the input a to obtain $ct.2^x$.

For a general exponential function b^x , we preprocess an input x at the very first of the whole algorithm, that is, we multiply x by $\log_2 b$; logarithm property outputs the following: $x \log_2 b = \log_2 b^x$. If the input x is replaced by $\log_2 b^x$, we obtain $2^{\log_2 b^x} = b^x$. Therefore, in ciphertext, we multiply $ct.x$ by $ct.\log_2 b$ in the first place to obtain the general formula $ct.b^x$.

Remark: In this study, we restrict the exponential function to be e^x . Therefore, we multiply the input value by $\log_2 e$ as a constant to make the algorithm work for e^x .

4.2.2. Division. The binary division algorithm in ciphertext takes two real number ciphers $ct.a$ and $ct.b$ and performs $ct.a$ divided by $ct.b$. The result of the function is $ct.q$, where $ct.q$ is a quotient (Figure 4). The algorithm is somewhat similar to the exponential function and intuitive to understand.

Suppose we consider the algorithm in plaintext, where a, b, q are all unencrypted positive real number vectors and have a vector of size l . First, we concatenate a and q , where q is set to be a zero vector. We denote the vector as $w = (a \| q)$. Then, we repeat the steps as follows:

- (1) Right shift w by 1 and compare q with b
- (2) If q is less than b , set w_0 to be 0, and otherwise, set w_0 to be 1 and subtract q by b .

We iterate the steps for $3l/2$ times and obtain $q = (w_0, \dots, w_{l-1})$ for the result of the plaintext real number binary division algorithm. The repeated steps are analogous to the normal division process; at each step of the iteration,

FIGURE 4: Flow diagram of binary division in ciphertext.

we compare and subtract the values of the divisor **b** and the dividend **a** until we acquire quotient of size l .

As for the binary division in ciphertext, it is necessary to consider the sign of the inputs in the same manner as in the binary exponential function. The difference is that we now consider two variables as our inputs compared to one variable in the exponential function algorithm. Thus, we consider two cases: (1) **a**, **b** are both positive or negative and (2) both have different signs. The result of division from the former condition is positive, whereas it is negative in (2). One good way is using XOR operation between the sign bits of **a** and **b**, where the operation outputs 0 if both have the same sign and else 1. We use $ct.s$ to denote the result of the operation: $ct.s = ct.a_{l-1} \oplus ct.b_{l-1}$.

Let $ct.r$ be the result of the binary division step in Figure 4 which outputs a positive division result. Suppose $ct.a$ and $ct.b$ are the case (1) variables. Then, since $ct.s$ is $ct.1$, bitwise AND operation between $ct.s$ and $ct.r$ is the $ct.r$ itself, whereas if $ct.a$ and $ct.b$ are the case (2) variables, then the bitwise AND operation will output $ct.0$ (since $ct.s = ct.0$). Thus, the result of the bitwise AND operation between $ct.s$ and $ct.r$ is case (1) output. Likewise, for the negative result of (2), we instead perform two's complement operation in the beginning to make $ct.r$ a negative division result. The bitwise AND operation of $ct.s$ follows and outputs the negative result for case (2) and $ct.0$ for case (1). Last, we add the former results: HE.ADD (bitwise $ct.s \wedge ct.r$ and bitwise $ct.s \wedge HE.Twos(ct.r)$).

The main part of the binary division where the goal is to obtain $ct.r$ is designed in the following way. We first refer HE.Compare function [23] to an operation that outputs $ct.0$ if the former input is less than the latter and otherwise, $ct.1$. This function is extremely useful since we can use it in each step of the iteration to set the value for w_0 , that is, $ct.w_0 = HE.Compare(ct.q, ct.b)$. We also use $ct.w_0$ to perform bitwise AND operation with the subtraction of $ct.q$ by $ct.b$ since the subtraction operation only performs when $ct.w_0$ is $ct.1$. Hence, we can update $ct.q$ in both cases of $ct.w_0$ being $ct.0$ and $ct.1$.

5. Result

Generally, a goal for an encrypted model is to mimic the plaintext model with the purpose of providing the same level of functionality. In this sense, a key factor for constructing a “well” homomorphic MLPNN is how “well” the underlying functions are designed so as to evaluate the accurate result. That is, the output after decryption should be as close to the result of the plaintext model under the same configuration such as the number of layers and neurons. In this sense, our

model has every homomorphic operation that can evaluate output with the same level of accuracy as the plaintext model except the sigmoid function; in fact, this is the same for the previous literature that we have discussed. Therefore, the “real” key factor to construct such a well-encrypted MLPNN model narrows down to the construction of the activation function—approximating the sigmoid function is crucial.

5.1. Sigmoid Function: Low-Degree Polynomial vs. Binary Approximation. Approximating the sigmoid function is important in two ways: it is used in (1) feedforward activation of neurons and (2) computation of partial derivatives in backward propagation. First, during the feedforward phase, the sigmoid function is computed in every neuron except the neurons in the input layer. Therefore, a model becomes error-prone when the sigmoid function is not well approximated. Also, in case (2), partial derivatives of the cost with respect to the parameters are calculated by derivatives of the sigmoid (equation (8)). Moreover, the number of sigmoid derivatives required is about triple times compared to that of (1) because the derivative of the sigmoid function is used for every partial derivative of C with respect to parameters w , b , and a neuron a_i^k .

Figure 5 shows graphs of sigmoid functions and their derivatives in terms of different approaches. We refer the true sigmoid function to the nonapproximated sigmoid function evaluated in plaintext and use our binary sigmoid function that takes 32-bit input. Moreover, we present the two most famous approximation techniques—Taylor series [6] and the least square method [7]—for comparison. The Taylor series method is the most common method for approximating a nonlinear function as sums of polynomials (denoted by $t(x)$). Along with the line of approximation by polynomials, the least square method aims to approximate the nonlinear function $f(x)$ by minimizing mean squared error (MSE) of $(1/|I|) \int_I (f(x) - g(x))^2 dx$, where $g(x)$ is our target function, and I is defined by $(1/|I|) \int_I f(x)^2 dx$. In our experiment, we use the Taylor series of degree 7 and the least square method of degree 9 polynomials. The equations are the following:

$$t(x) = \frac{1}{2} + \frac{1}{4}x - \frac{1}{48}x^3 + \frac{1}{480}x^5 - \frac{17}{80640}x^7, \quad (11)$$

$$g(x) = b_0 + b_1\left(\frac{x}{8}\right) + b_3\left(\frac{x}{8}\right)^3 + b_5\left(\frac{x}{8}\right)^5 + b_7\left(\frac{x}{8}\right)^7,$$

where the coefficients of $g(x)$ are $b_0 = 0.5$, $b_1 = 1.73496$, $b_3 = -4.19407$, $b_5 = 5.43402$, $b_7 = -2.50739$.

FIGURE 5: Graphs of sigmoid functions and their derivatives with respect to different approaches: true (or plaintext), binary, Taylor, and LS (least square). (a) Graphs of sigmoid functions. (b) Graphs of sigmoid function derivatives.

5.1.1. Comparison Result. In Figure 5(a), a valid range for each approach is different—the binary sigmoid function shows a wider valid range than the two other approaches. The binary approximation provides the valid range of -11 to 11 , while the Taylor series and the least square approximation provides the ranges of -2 to 2 and -8 to 8 , respectively. The reason for a failure of precision from the point $x = -11$ of the binary sigmoid function is because of the design of our exponential function and the number of bits that are allocated to the integer part in our number formatting system; increasing the size of bits for an array or allocation of more bits to integers can provide a wider range.

Figure 5(b) shows graphs of sigmoid function derivatives regarding each approach. The binary sigmoid derivative demonstrates the highest similarity to the derivative of the true sigmoid function within all ranges in the figure, whereas the sigmoid function of the Taylor series approximation only works in the interval of around -2 to 2 in accordance with the range shown in Figure 5(a). An interesting result is that the least square approximation is unstable—less accurate from -7.5 to 3 and 3 to 7.5 compared to its graph shown in Figure 5(a). This implies that the least square method is not a recommended approach for the sigmoid function in the backpropagation. Therefore, we conclude that our binary sigmoid approximation is significantly more accurate compared to the other two famous approaches that utilize low-degree polynomials to approximate the sigmoid function—unlike our binary approximation method not conforming to this tradition but builds upon the bitwise operations—considering the results shown in Figure 5.

5.2. Time Performance of t-BMPNet

5.2.1. Environment Setting. Our research is conducted in AMD Ryzen 5 3500X 6-Core 3.60 GHz, 8.0 GB RAM, Ubuntu 20.04.1 LTS, and we use TFHE version 1.0 to implement the bitwise scheme of our approach.

We experiment execution time of our binary approximation with respect to three key algorithms—sigmoid function, feedforward, and backpropagation—with different input bits (Table 2). Since it takes a considerable amount of time for the execution of some key functions in our system, a time performance measure of feedforward and backpropagation is conducted on MLPNN with the simplest setting—three layers, one neuron per layer, and a single data training.

As a result of the time performance of the training algorithm under t-BMPNet, we obtain 1.87, 7.14, and 28.02 minutes for 8, 16, and 32 input bits, respectively. Our main focus—feedforward and backpropagation algorithms—is measured at 17.00 and 11.02 minutes, respectively. As analysis for the number of operations evaluated in each algorithm, the feedforward algorithm takes 2 additions, multiplications, and sigmoid functions of total 6 operations, whereas the backpropagation algorithm requires 3 subtractions and 7 multiplications resulting in 10 operations for the training of a single data.

Since addition and subtraction operations are negligible compared to other heavy operations in terms of computational costs—for instance, the execution time for multiplication is 58(99/1.7) times of addition and subtraction—concerning only the remaining operations, the net amount of operations for feedforward and

TABLE 2: Execution time (minute) of key operations—multiplication, division, exponential, and sigmoid function—and training algorithms measured in minute unit.

Time (Min)	Trainable bitwise multilayer perceptron neural network (t-BMPNet)				Training algorithm		Total
	Multiplication	Division	Exponential	Sigmoid	Feedforward	Backpropagation	
8-bit	0.08	0.11	0.23	0.36	0.89	0.57	1.46
16-bit	0.37	0.45	0.93	1.41	3.6	2.63	6.23
32-bit	1.65	1.8	3.78	5.6	14.59	11.64	26.23

backpropagation is 2 sigmoids and 5 multiplications, respectively. However, since the sigmoid function in 32-bit is approximately equivalent to 3(336/99) multiplications, the feedforward algorithm takes a longer time than the backpropagation algorithm. In fact, the sigmoid derivative property $\sigma' = \sigma(1 - \sigma)$ reduces a significant amount of computational cost in the backpropagation algorithm. It facilitates bypassing computationally expensive derivatives of the exponential function in the sigmoid function evaluating only one multiplication and subtraction operation per function.

In general, for a small network in t-BMPNet, we expect that the feedforward is a more computationally expensive algorithm than the backpropagation algorithm since most of the time-consuming procedure takes in deriving values for sigmoid functions, whereas the backpropagation does not participate in this process. However, as the network becomes more complex, there is a turnover between the algorithms in terms of time performance. This is because, suppose for any two layers containing n, m neurons, the time complexity of deriving sigmoid functions in the feedforward algorithm is $O(n + m)$, whereas, in the backpropagation algorithm, updating parameter requires evaluation of multiplication operations, and thus, the time complexity increases by $O(nm)$ (equation (8) and Algorithm 1).

5.3. Comparison with Other Approaches. We implement using SEAL version 3.10 to construct MLPNN in two different schemes—BFV and CKKS—for comparison with t-BMPNet. Both schemes are provided by the SEAL library and are the most actively used cryptographic schemes for implementing the neural network algorithms. Specifically, the BFV scheme is the cryptographic basis for the Low-Latency CryptoNets (LoLa) [25], the most recent version of CryptoNets designed for private inference. Likewise, the CKKS scheme is applied in neural networks since it is considered one of the most practical FHE schemes; much of the literature [27, 28] focus on the enhancement of throughput and memory efficiency of the neural network.

We implement MLPNN with the network architecture of [2, 3, 6, 7] (each entry represents the number of nodes in the layer) based on the three schemes—TFHE, BFV, and CKKS—to compare and analyze their characteristics (Table 3). We assume that the network parameters are encrypted, i.e., weights and biases are encrypted, and thus, operations are between ciphertexts. Also, we measure the time performance of the three approaches in two different ways by selecting different types of the activation

function—square and sigmoid. Particularly, we measure the square function since CryptoNets and its variants use the square operation for the activation function.

5.3.1. Our Approach. t-BMPNet is designed as in the previous section using TFHE library with the difference in the architecture of the network. We measure the throughput of inferencing a single encrypted data considering the lengths of the input ($l = 8, 16, 32$) and obtain 4.94, 19.33, and 75.91 minutes, respectively, in the setting where the square function is an activation function. As for the total execution time using the sigmoid function, we obtain increased execution time 9.48, 39.32, and 147.5 minutes, respectively.

5.3.2. Other Approaches. MLPNN using BFV and CKKS schemes are designed in the leveled HE scheme, that is, unlike TFHE that uses FHE which performs bootstrapping for every gate operation, the bootstrapping is not supported; parameters are carefully chosen before the execution, so that the whole procedure is performed without having to undergo an expensive bootstrapping procedure. We pack a single data followed by encryption to yield a ciphertext polynomial in $\mathbb{Z}_q[x]/(x^n + 1)$ in both schemes. The encrypted weights and biases are encoded row-wise to perform matrix multiplication with some rotations involved.

In the BFV setting, we obtain 0.15 minutes for evaluation with square function with a polynomial degree of $n = 16384$. Since the BFV scheme can only support integer arithmetic, the normal sigmoid function is not applicable to the scheme. In the CKKS environment, we use $n = 32768$ and acquire 0.57 and 0.97 minutes for the evaluation using the square and sigmoid functions, respectively.

5.3.3. Analysis. One of the major benefits of using the TFHE scheme is that it is not constrained by the complexity of the neural network, that is, we can implement as many layers without having to consider parameters in the first place (this is because TFHE is a fully HE scheme with bootstrapping). However, in both leveled CKKS and BFV schemes, a complex neural network is not applicable. In our experiment setting with CKKS, a single inference on the neural network requires a total of 16 multiplicative depths of the circuit; in each layer, 6 multiplication operations are performed—one between weights and the input and the rest are allocated for the sigmoid function. In particular, the sigmoid function requires at least 5 multiplications since it is approximated by the least square polynomials with the highest exponent equal

TABLE 3: Comparison of t-BMPNet with other approaches—BFV and CKKS—in terms of execution time (minute), applicability to complex neural networks, and whether it is trainable under the encrypted domain.

Execution time comparison of MLPNN [2, 3, 6, 7]								
Method	Type	Scheme	Parameter	Feedforward		Complex network?	Trainable?	Related works
				Square	Sigmoid			
t-BMPNet	Fully	TFHE	$l = 8$	4.94	9.48	Yes	Yes	—
			$l = 16$	19.33	39.32			
			$l = 32$	75.91	147.5			
Other approaches	Leveled	BFV	$n = 16384$	0.15	—	No (requires FHE scheme)	No	[2, 25]
		CKKS	$n = 32768$	0.57	0.97

FIGURE 6: Comparison of the real training MLPNN model, simulated t-BMPNet, simulated Taylor series, and simulated least square method concerning classification result, mean squared error (MSE), and number of epochs. (a) Result of the real MLPNN training model at interval level $c = 8$. (b) Simulated result of t-BMPNet with 32-bit input at interval level $c = 8$. (c) Simulated result of the Taylor series method of degree 7 at interval level $c = 5$. (d) Simulated result of the least square method of degree 9 at interval level $c = 7$.

to 7; it requires at least 4 multiplications for evaluation, and finally, we need a multiplication between the coefficients and the exponents. Therefore, expanding the network follows an additional 6 multiplicative depths which is not practical for a complex network since the growth of the multiplicative

depths requires an exponential increase of the modular space $\mathbb{Z}_q$. In our experiment, we set the parameter $n = 32768$ to be the largest degree that the SEAL library can support in order to perform inference on encrypted data. A larger neural network cannot be implemented using our approach unless

FIGURE 7: Example of $x = -1.5$ for deriving binary exponential function 2^x for the input length 8-bit. (a) Encode, encrypt, and absolute value operation of $x = -1.5$ of length $l = 8$. (b) Process of deriving ct: x_{int} ($\odot$: HE Equi). (c) Positive exponentiation process. (d) Negative exponentiation process. (e) Process of considering the sign of the input ct.a. (f) Decryption and decoding process.

FIGURE 8: Example of division—dividend $x = 4.5$ and $y = -2$ —for the input length 8-bit. (a) Encode, encrypt, and absolute value operation of $x = 4.5$ and $y = -2$. (b) Iterative steps in the binary division process (Δ : HE.Compare). (c) Rescaling of $\text{ct. } w$ to obtain $\text{ct. } r$ at iteration step $(3l/2)$. (d) Process of considering signs of the inputs $\text{ct. } a$ and $\text{ct. } b$.

the library supports a higher degree of polynomials and use a lesser degree in the least square polynomials (however, as a tradeoff, the accuracy of the network will decrease).

t-BMPNet can train the model without the constraint of considering parameters from the beginning since it provides bootstrapping—similar discussion with the previous complexity problem. However, both CKKS and BFV schemes in the leveled environment cannot support training under the encrypted domain since training requires a feedforward and a backpropagation step for every iteration of the training procedure. One can choose the fully homomorphic encryption for BFV and CKKS scheme to facilitate complex neural networks; however, in terms of bootstrapping efficiency, TFHE is much more time-saving, e.g., less than a second per gate, whereas the CKKS bootstrapping procedure requires several minutes to refresh the ciphertext.

5.4. Application to a Dataset. We have previously demonstrated that the training of our model works in the encrypted domain within a small network consisting of three neurons. Now, we expand our model to experiment in a larger network—we compare the real training model with our simulated model implemented in Matlab 2019a in terms of mean squared error (MSE), a number of epochs, and classification result for the measure.

We use a small dataset from the Sharky neural network consisting of 32 coordinates of x and y with their labels. The goal of the dataset is to classify the points with a decision boundary that is determined by minimizing the cost. We randomly chose parameters of weights and biases and fixed their values as an initial setting and set the learning rate equal to 0.15. The network consists of 3 layers with 2, 4, and 2 number of neurons in each layer. We perform the backpropagation algorithm with the normal gradient descent method.

Sharky data consists of data in the range of -1 to 1 . We compare each MLPNN model with respect to different types of approximation techniques—plaintext, our approach, Taylor series, and least square method—to test accuracy for each model. In our experiment with the original Sharky dataset, we obtain 100% accuracy for all the models. We suspect that the results of 100% accuracy are because all of the techniques approximate sigmoid functions without a significant loss of precision within a certain degree of range. Thus, we perturb the original dataset by multiplying the data with a constant value c —without changing labels—to test each model with different input ranges. For $c = 5$, the Taylor series approximation fails to locate the decision boundary, whereas the least square method fails at $c = 7$. The result in Figure 6 shows that both—real training MLPNN and t-BMPNet—at $c = 8$ classify 32 points with the mean squared error equal to 0 at epoch number 35.

6. Discussion and Conclusion

In this study, we presented t-BMPNet that can evaluate multilayer perceptron neural net over fully homomorphic encryption scheme from the basis of bitwise operations. We

introduced our basic FHE operations with the emphasis on the key operations—exponential function and division—for the construction of the sigmoid function. Our work is different from the other literature in several aspects: privacy for both the user and the server, backward and forward training in the encrypted domain, and minimal requirement of operations performed by users. Moreover, our work has achieved a highly accurate design of nonlinear sigmoid function compared to other literature that almost uses the polynomial approximation to construct the function; our research guarantees more accurate prediction result. Despite notable results, our research has shown low time performance. This is because our work is at the foundational stage of the bitwise neural network—our work leaves much room for improvements in time performance.

First, the overall time performance can be enhanced from improvements in bootstrapping of binary gates. In fact, since Gentry's [13] proposal of bootstrapping of noisy ciphertexts in 2013, there has been a significant increase in the time performance for the bootstrapping procedures. FHEW [16] library reduced the time for bootstrapping in less than a half-second, and TFHE [17] library improved the speed by a factor of 53 resulting in 13 milliseconds single core time for evaluation. Next, some hardware acceleration techniques can be used to boost the time performance. For example, GPU and FPGA can increase the computational speed to improve latency and throughput. Moreover, not dramatically but to some extent, algorithms for the basic operations can be optimized to enhance the performance. For instance, the full-adder binary addition can be improved by using 9 NAND gates instead, since, in our scheme, NAND gates have better performance than other binary gates. Last, as a further step, using the Chimera [29] scheme can enhance the time performance in a significant amount. This is because the scheme performs as a bridge to enabling the exploitation of only the advantages of each FHE scheme such as TFHE, BFV [30], and HEAAN [26].

Appendix

A. Binary Exponential Function Example

We provide an illustration of the exponential function being performed behind the scene with an example of $x = -1.5$. In this example, our goal is to perform a binary exponential operation of -1.5 , that is, $2^{-1.5}$. Note that we only perform 2^x in this example. For the general result of b^x , we multiply ct.a by $\text{ct.log}_2 b$ in the first place; we omit this procedure as in Figure 7 and show only the remaining (important) parts of the procedure.

Initially, we encode the number $x = -1.5$ into the plaintext vector $\mathbf{a}$ of size $l = 8$ (Figure 7(a)). We refer to dotted rectangular boxes as a plaintext vector, whereas its encryption is referred to as solid-line boxes. We designate the number written in red letters as the fractional part of the fixed-point number in order to distinguish it from the integer part. Then, the plaintext vector $\mathbf{a}$ is encrypted bitwise under the same secret key $\mathbf{s} \leftarrow \mathbb{B}^n$ to output ct.a followed by absolute value operation to yield $|\text{ct.a}|$. We present numbers

in parentheses next to the boxes to indicate its decimal form of encryption. For instance, in Figure 7(a), we write (1.5) for $(\text{Enc}(0), \text{Enc}(0), \text{Enc}(1) | \text{Enc}(1), \text{Enc}(0), \dots, 0)$.

The core technique of the binary exponentiation algorithm is to search for the integer value of $|x|$ (denote it by $|x|_{\text{int}}$) which is the indicator of deciding the direction and number of shifts. However, since it is encrypted, in other words, we are unable to use $\text{Enc}(|x|_{\text{int}})$ to perform such actions, we exhaustively perform all possible cases of positive exponentiations (likewise negative). The key part is at Figure 7(b) where HE.Equi operation (or $\odot$ in Figure 7(b)) yields $\text{Enc}(1)$ only when $|\text{ct.a}|_{\text{int}}$ is equal to one of the values of $\text{Enc}(0), \text{Enc}(1), \dots, \text{Enc}(l/2 - 1)$; each result is denoted by $\text{ct.o}_1, \text{ct.o}_2, \dots$, respectively. In this example, since $|x|_{\text{int}} = 1$ ($|\text{ct.a}|_{\text{int}} = \text{Enc}(1)$), only ct.o_2 is $\text{Enc}(1)$.

In Figures 7(c) and 7(d), ct.p_i and ct.n_i are the outcomes of positive and negative binary exponentiations of $|\text{ct.a}|$, respectively. For instance, ct.p_2 is $\text{HE.Add}(\text{ct.u}, \text{ct.v})$, where ct.u is the outcome of right shift $\text{Enc}(1)$ by 1 and ct.v is the outcome of right shift $|\text{ct.a}|_{\text{frac}}$ by 1. In contrast, ct.n_2 is $\text{HE.Subt}(\text{ct.u}', \text{ct.v}')$, where $\text{ct.u}'$ is the outcome of left shift $\text{Enc}(1)$ by 1 and $\text{ct.v}'$ is the outcome of left shift $|\text{ct.a}|_{\text{frac}}$ by 2.

Given ct.o_i 's and their corresponding ct.p_i 's (likewise, ct.n_i 's), we perform bitwise AND operation (denote $\wedge_{\text{bitw}}$) between the two ciphertexts. The result of the operation is non- $\text{Enc}(0)$ only if ct.o_i is $\text{Enc}(1)$ for some i . In our example, ct.p_2 and ct.p_2 are the only non- $\text{Enc}(0)$ values. Thus, after adding (i.e., HE.Add or denote Σ in Figures 7(c) and 7(d)) ct.p_i 's and ct.n_i 's, we obtain ct.p_2 and ct.p_2 , respectively. (We refer to the results as ct.r_1 and ct.r_2 , respectively.)

So far, we have proceeded the positive and negative exponentiation steps in Figure 3. Now, we combine the positive exponentiation result ct.r_1 and the negative exponentiation result ct.r_2 considering the sign of ct.a . Since x is negative (the sign bit ct.a_{l-1} is $\text{Enc}(1)$), we expect our output to be ct.r_2 . Thus, we perform bitwise AND operation between ct.a_{l-1} and ct.r_1 which results in $\text{Enc}(0)$. On the contrary, $\text{ct.a}_{l-1} \wedge_{\text{bitw}} \text{ct.r}_2$ yields the negative exponentiation result ct.r_2 . By adding both of the results, we obtain ct.r_2 for ct.2^a (Figure 7(e)).

As a final step, we retrieve the encrypted result ct.2^a and perform decryption followed by a decoding procedure. We decrypt ct.2^a using the same secret key $\mathbf{s}$ and obtain plaintext 2^a . We decode the plaintext by performing $\sum_{i=0}^{i=6} 2^{-3} a_i$ and obtain 0.375 for our desired output of the binary exponential procedure.

B. Binary Division Function Example

As an example of binary division function in FHE scheme, we consider two real numbers $x = 4.5$ and $y = -2$ (Figure 8(a)). We first encode both numbers of length $l = 8$ and obtain $\mathbf{a} = (0, 0, 1 | 0, 0, 1, 0)$ and $\mathbf{b} = (0, 0, 0 | 0, 1, 1, 1)$, followed by their encryption ct.a and ct.b . We consider that the rest of the operations are being performed on the positive real ciphertexts; the absolute value operation for both ct.a and ct.b gives $\text{Enc}(4.5)$ and $\text{Enc}(2)$, respectively.

Now, we consider ct.w , concatenation of two ciphertexts $|\text{ct.a}|$ and ct.q (ct.q is initially set to $\text{Enc}(0)$) generating $2l (= 16)$ -bits vector of encrypted bits. We iterate the following steps for $3l/2 (= 12)$ times:

- (1) We right shift ct.w once and $\text{HE.Compare}(\text{ct.q}, |\text{ct.b}|)$, where ct.q is the right $l (= 8)$ element of ct.w and assign the result to ct.w_0 (e.g., in Figure 8(b)), and ct.q is still $\text{Enc}(0)$ after the shift operation from the first iteration. The comparison operation between ct.q and $|\text{ct.b}| (= \text{Enc}(2))$ yields $\text{Enc}(0)$ for ct.w_0 .
- (2) If (1) ct.q is less than or equal to $|\text{ct.b}|$ (i.e., $\text{ct.w}_0 = \text{Enc}(0)$), we assign ct.q as it is. On the other hand, if (2) ct.q is greater than $|\text{ct.b}|$, we assign $\text{HE.Subt}(\text{ct.q}, |\text{ct.b}|)$ to ct.q . As in 7e, we use bitwise AND operation (or $\wedge_{\text{bitw}}$) and ct.w_0 to handle both conditions. We perform the following operations: $\text{ct.w}_0 \wedge_{\text{bitw}} \text{HE.Subt}(\text{ct.q}, |\text{ct.b}|)$ and $\text{ct.w}_0 \wedge_{\text{bitw}} \text{ct.q}$. We add (i.e., HE.Add) both results and assign it to ct.q (e.g., in Figure 8(b)), and ct.q is less than $|\text{ct.b}|$. We need ct.q not $\text{HE.Subt}(\text{ct.q}, |\text{ct.b}|)$. Thus, bitwise AND operation of ct.w_0 , that is, $\text{Enc}(1)$, to elements of ct.q yields ct.q , whereas bitwise AND operation of ct.w_0 (or $\text{Enc}(0)$) to $\text{HE.Subt}(\text{ct.q}, |\text{ct.b}|)$ yields $\text{Enc}(0)$. Thus, we gain ct.q by adding both results.

After $3l/2 (= 12)$ iterations, we obtain the result ct.w of $2l$ -bits vector. Since we are only interested in the precision of l -bit, we rescale ct.w by taking only the left l bits from ct.w and call it ct.r . (Note that one can get better precision at most $2l$ -bits of the result by performing more iterations). In our example, we obtain $\text{Enc}(2.25)$ from ct.w .

Last, we consider the signs of ct.a and ct.b (Figure 8(d)). Using XOR operation between ct.a_{l-1} and ct.b_{l-1} , we obtain ct.s that is $\text{Enc}(0)$ if ct.a and ct.b have the same sign, whereas $\text{Enc}(1)$ if ct.a and ct.b have the opposite sign values. Taking advantage of this fact, we apply $\wedge_{\text{bitw}}$ operation of ct.s to ct.r which yields ct.r only when ct.a and ct.b have the same sign value (i.e., $\text{ct.s} = \text{Enc}(1)$). In Figure 8(d), since ct.a and ct.b have the opposite sign value, ct.s is $\text{Enc}(0)$; bitwise AND operation yields $\text{Enc}(0)$. Likewise, we can design a circuit that outputs only the negative result of ct.r , that is, $\text{HE.Twos}(\text{ct.r})$ by the following: $\text{ct.s} \wedge_{\text{bitw}} \text{HE.Twos}(\text{ct.r})$. In our example, ct.s is $\text{Enc}(1)$; $\wedge_{\text{bitw}}$ yields $\text{HE.Twos}(\text{ct.r})$ which is $\text{Enc}(-2.25)$, the encrypted result of ct.a divided by ct.b . We decrypt the result followed by a decoding process to obtain -2.25 , which is the desired outcome of $x = 4.5$ divided by $y = -2$.

Data Availability

The toy data used for comparison between FHE schemes were generated by the author and included within the article.

Conflicts of Interest

The authors declare that there are no conflicts of interest.

Acknowledgments

This work was supported by Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by the Korea Government (MSIT) (20210005580012003, Development of national statistical analysis system using homomorphic encryption technology).

References

- [1] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM*, vol. 56, no. 6, pp. 1–40, 2009.
- [2] R. Gilad-Bachrach, N. Dowlin, 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 Machine Learning*, vol. 48, pp. 201–210, New York, NY, USA, June 2016.
- [3] E. Hesamifard, H. Takabi, and M. Ghasemi, "CryptoDL: deep neural networks over encrypted data," 2017, <https://arxiv.org/abs/1711.05189>.
- [4] 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.
- [5] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai, "Privacy-preserving deep learning: revisited and enhanced," in *Proceedings of the Applications and Techniques in Information Security*, pp. 100–110, Auckland, New Zealand, July 2017.
- [6] P. Mohassel and Y. Zhang, "SecureML: a system for scalable privacy-preserving machine learning," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP)*, pp. 19–38, San Jose, CA, USA, May 2017.
- [7] M. Kim, Y. Song, S. Wang, Y. Xia, and X. Jiang, "Secure logistic regression based on homomorphic encryption: design and evaluation," *JMIR Medical Informatics*, vol. 6, no. 2, p. e19, 2018.
- [8] V. Vaikuntanathan, "Computing blindfolded: new developments in fully homomorphic encryption," in *Proceedings of the 2011 IEEE 52nd Annual Symposium Foundations Computer Science*, pp. 5–16, Palm Springs, CA, USA, October 2011.
- [9] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundation Security Computer*, vol. 4, no. 11, pp. 169–180, 1978.
- [10] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-DNF formulas on ciphertexts," in *Proceedings of the Theory of Cryptography*, pp. 325–341, Cambridge, MA, USA, February 2005.
- [11] P. Paillier, "Low-cost double-size modular exponentiation or how to stretch your cryptoprocessor," *Public Key Cryptography*, vol. 1560, pp. 223–234, 1999.
- [12] C. Gentry and D. Boneh, *A fully homomorphic encryption scheme*, Stanford Univ. press, Palo Alto, CA, USA, 2009.
- [13] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: conceptually-simpler, asymptotically-faster, attribute-based," in *Proceedings of the Advances in Cryptology-CRYPTO 2013*, pp. 75–92, Santa Barbara, CA, USA, August 2013.
- [14] J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig, "Improved security for a ring-based fully homomorphic encryption scheme," in *Proceedings of the Cryptography and Coding*, pp. 45–64, Oxford, UK, December 2013.
- [15] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," *Advances in Cryptology-ASIACRYPT 2017*, vol. 10624, pp. 409–437, 2017.
- [16] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *Proceedings of the Advances in Cryptology--EUROCRYPT 2015*, pp. 617–640, Sofia, Bulgaria, April 2015.
- [17] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: bootstrapping in less than 0.1 seconds," in *Proceedings of the Advances in Cryptology-ASIACRYPT 2016*, pp. 3–33, Hanoi, Vietnam, December 2016.
- [18] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.
- [19] J. S. Yoo, J. H. Hwang, B. K. Song, and J. W. Yoon, "A bitwise logistic regression using binary approximation and real number division in homomorphic encryption scheme," *Information Security Practice and Experience*, vol. 11879, pp. 20–40, 2019.
- [20] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
- [21] M. A. Nielsen, *How the Backpropagation Algorithm Works* in *Neural Networks and Deep Learning*, Determination Press, San Francisco, CA, USA, 2015.
- [22] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep Feedforward Networks* in *Deep Learning*, MIT Press, Cambridge, MA, USA, 2016.
- [23] B. K. Song, J. S. Yoo, M. Hong, and J. W. Yoon, "A bitwise design and implementation for privacy-preserving data mining: from atomic operations to advanced algorithms," vol. 2019, Article ID 3648671, 2019.
- [24] J. S. Yoo, B. K. Song, and J. W. Yoon, "Logarithm design on encrypted data with bitwise operation," in *Proceedings of the Information Security Applications*, pp. 105–116, Jeju Island, South Korea, August 2019.
- [25] A. Brutzkus, R. Gilad-Bachrach, and O. Elisha, "Low latency privacy preserving inference," *ICML*, vol. 97, pp. 812–821, 2019.
- [26] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *Proceedings of the Advances in Cryptology-EUROCRYPT 2018*, pp. 360–384, Tel Aviv, Israel, April 2018.
- [27] R. Dathathri, O. Saarikivi, H. Chen et al., *CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inference*, pp. 142–156, PLDI, Phoenix, AZ, USA, 2019.
- [28] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, *nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data*, WAHC, London, UK, 2019.
- [29] C. Boura, N. Gama, and M. Georgieva, "Chimera: a unified framework for B/FV, TFHE and HEAAN fully homomorphic encryption and predictions for deep learning," *IACR Cryptology ePrint Archive*, vol. 2018, p. 758, 2018.
- [30] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *Proceedings of the CRYPTO*, pp. 868–886, Santa Barbara, CA, USA, August 2012.