SecureTrain: An Approximation-Free and Computationally Efficient Framework for Privacy-Preserved Neural Network Training

Qiao Zhang, Chunsheng Xin, *Senior Member, IEEE*, and Hongyi Wu, *Fellow, IEEE*

***Abstract*—Data privacy is a fundamental challenge for Deep Learning (DL) in many applications. In this work, we propose *SecureTrain*, which aims to *carry out privacy-preserved DL model training efficiently and without accuracy loss*. SecureTrain enables joint linear and non-linear computation based on the Homomorphic Secret Share (HSS) technique,to carry out approximation-free non-polynomial operations, to achieve training stability and prevent accuracy loss. Meanwhile, it eliminates the time consuming Homomorphic *permutation operation* (Perm) and features an efficient piggyback design,by carefully devising the share set and exploiting the dataflow of the whole training process. This design significantly reduces the overall system training time. We analyze the computation and communication complexity of SecureTrain and prove its security. We implement SecureTrain and benchmark its performance with well-known dataset for both inference and training.For inference, SecureTrain not only ensures privacy-preserved inference, but achieves an inference speedup as high as 48× compared with state-of-the-art inference frameworks. For training, SecureTrain maintains the model accuracy and stability comparable to plaintext training, which is a sharp contrast to other schemes. To the best of knowledge, this is the first work that addresses two fundamental challenges, *accuracy loss/training instability, and computation efficiency*, in privacy-preserved deep neural network training.**

***Index Terms*—Homomorphic encryption, neural network training, privacy preserving, secret share.**

I. Introduction

DEEP learning (DL) has demonstrated phenomenal success in recent years, achieving state-of-the-art results for solving complex problems in a multitude of applications such as image classification [1], face recognition [2], and object detection [3]. DL has also become a powerful tool in the field of networking [4], [5] such as mobility analysis [6], network control [7], network security [8] and signal processing [9]. The success of DL relies on three core elements, massive computing power, expertise to construct good DL models, and large datasets for model training [10]–[12]. It is common that the entities possessing data are different than the organizations that own the computing power and DL expertise. For example, end users, enterprises, and regional Internet Service Providers (ISPs) possess large volume of data, while the DL talent and computing power are mostly gathered in technology giants such as Google and Microsoft. The former has a strong motivation to utilize the computing power and DL talent of the latter to solve challenging problems in networks, e.g., to optimize network design. However, a fundamental challenge is data privacy. For example, those data can have precious business value and need to be protected. Moreover, they can be sensitive and protected by laws from disclosure [13]–[19]. To this end, there has been a great interest in the research community to develop privacy-preserved DL systems, such as the CryptoNets [20], SecureML [21], MiniONN [22], EzPC [23], and Xonn [24].

Fig. 1 illustrates a privacy-preserved DL system to provide real-time, networked diagnosis to patients who are covered by Wireless Body Area Networks (WBANs). Each patient is monitored by various sensors, such as ear tempetature sensor, electrocardiograph and pulse oximeter. The health data of patients, e.g., the temperature, heart rate and blood oxygen, is collected by WBANs and reported to a health provider such as a hospital. The latter sends the health data to a server in a cloud (e.g., the Microsoft cloud), which hosts a trained DL model to provide the health diagnosis service. To protect the privacy of sensitive data, the health provider (as a client) encrypts the data before sending to the server. The encrypted data is processed by the server on the crypto domain, and the diagnosis result, which is also on the crypto domain, is returned to the client. The client decrypts the result into plaintext, and uses it to assist patient treatment. The data privacy is fully protected in this process by the underlying encryption scheme, such that the sensitive patient information is not leaked. Furthermore, the privacy-preserved DL system also aims to protect the intellectual property of the DL model on the server to ensure the users (such as health providers) cannot learn the server's proprietary model.

Manuscript received August 7, 2020; revised October 5, 2020; accepted October 27, 2020. Date of publication November 25, 2020; date of current version January 11, 2022. This work was supported in part by the National Science Foundation under Grant CNS-1828593, OAC-1829771, EEC-1840458, and CNS-1950704, Office of Naval Research under Grant N00014-20-1-2065, and the Commonwealth Cyber Initiative, an investment in the advancement of cyber R&D, innovation and workforce development. For more information about CCI, visit cyberinitiative.org. Recommended for acceptance by Dr. Fan Wu. *(Corresponding author: Hongyi Wu.)*

The authors are with the School of Cybersecurity, and the Department of Electrical and Computer Engineering, Old Dominion University, Norfolk, VA 23529 USA (e-mail: qzhan002@odu.edu; cxin@odu.edu; h1wu@odu.edu).

This article has supplementary downloadable material available at https://doi.org/10.1109/TNSE.2020.3040704, provided by the authors.

Digital Object Identifier 10.1109/TNSE.2020.3040704

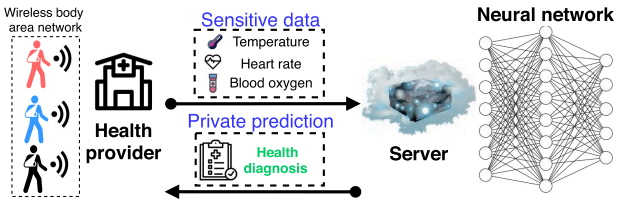

Fig. 1. A Framework of privacy-preserved inference.

While there has been good progress on privacy-preserved DL, the current systems are primarily designed for inference only, and face great challenges for model training. Due to the intractability of privacy-preserved non-polynomial computations (e.g., activation and softmax), the current approaches have chosen to approximate the non-polynomial functions to enable computation over the crypto domain. As to be discussed in Sec. II, such approximation comes with a price that leads to system accuracy drop [25], [26]. Moreover, applying it to model training (backpropagation) results in unwarranted stability [21], [27]–[29]. Although one can infinitely approximate a function to alleviate those problems, e.g., using piecewise linear functions, the resulted large-size approximation function hinders the system efficiency and usability. The detailed analysis is given in Sec. II. Furthermore, the current systems use a large number of Homomorphic *permutation operation* (Perm) to achieve inference over the crpto domain, since it is needed to compute the weighted sum and convolution, two critical operations in DL. As discussed in Sec. III-B, the Perm operation is very time-consuming, and results in poor system efficiency of current systems.

In this paper, we propose a novel framework, called *secure model training* (SecureTrain), to address the two fundamental challenges faced by privacy-preserved DL model training: (1) model accuracy loss and training instability due to use of function approximation, and (2) computation efficiency. The overarching goal is to eliminate the use of function approximation to *carry out training without accuracy loss and instability*, and reduce the use of Perm operation to improve *computation efficiency*. First of all, in order to achieve approximation-free computation, SecureTrain features an innovative design that enables joint linear and non-linear computation based on the Homomorphic Secret Share (HSS) [30]–[32]. Second, it eliminates the time consuming Perm operations by carefully designing the share set. Moreover, SecureTrain exploits the data flow in both forward propagation and backpropagation to enable an efficient piggybacking, thus further accelerating the overall computation and reducing the communication cost.

We analyze the computation and communication complexity of SecureTrain and prove its security using the standard simulation approach [33], [34]. The proposed SecureTrain is benchmarked with well-known datasets for both inference and training. For inference, our results show that SecureTrain not only ensures privacy-preserved inference, but achieves an inference speedup of 48×, 10×, 7× and 1.3×, respectively, compared with state-of-the-art privacy preserved inference systems: SecureML [21], MiniONN [22], EzPC [23], and Xonn [24]. For training, SecureTrain achieves the training accuracy and stability comparable to plaintext learning, which is a sharp contrast to current systems such as [21], [25]–[29] that suffer unwarranted stability as to be shown in Sec. VII-B. To the best of knowledge, this is the first work that addresses the two challenges, accuracy-loss/training instability, and computation efficiency in privacy-preserved deep neural network learning.

The rest of the paper is organized as follows. Sec. II discusses the related work about privacy-preserved DL. The background for neural network training and security primitives are introduced in Sec. III. The threat model is defined in Sec. IV. Sec. V details the design of SecureTrain. Sec. VI gives the security analysis. Experimental results are shown in Sec. VII. Finally, Sec. VIII concludes the work.

II. Related Work and Challenges

The existing efforts to enable privacy-preserved DL largely focus on inference and can be broadly classified into six categories based on their underlying cryptographic techniques.

(1) Homomorphic Encryption (HE)-Based Approaches. In CryptoNets [20], Faster CryptoNets [35] and CryptoDL [27], the client encrypts data using HE and sends the encrypted data to the server. The server performs polynomial computation over encrypted data to calculate an encrypted result. The client obtains the result and decrypts it. E2DM [36] adopts a more efficient HE (i.e., packed HE [37]) which packs multiple messages into one ciphertext and thus improves the computation efficiency.

(2) Garbled Circuit (GC)-Based Approaches. DeepSecure [38] and Xonn [24] binarize the computations in the neural network and then employ GC [39] to let the client obliviously obtain the private prediction without leaking its sensitive data.

(3) Secret Share (SS)-Based Approaches. SS was employed in [40] and [29] to split sensitive client data into shares. The server owns only one share of the data and cannot recover the original data. The computations are completed by interactive share exchange.

(4) Secure Enclave (SE)-Based Approaches. The trusted processor module such as Intel SGX was introduced in [41]–[44] to construct a secure environment. The client and server load their data and model to such a secure environment and perform secure computation.

(5) Differential Privacy (DP)-Based Approaches. DP was adopted in [45]–[47] to inject noise during the model training such that the trained model can be released from the server to the client with controlled leakage of the training data. The client is thus able to run the released model in plaintext.

(6) Mixed Protocol (MP)-Based Approaches. In order to deal with different properties of linearity (i.e., the weighted sum and convolution functions) and nonlinearity (i.e., activation and pooling functions) in neural network computations, the MP approaches aim to orchestrate various cryptographic techniques to achieve better performance [21]–[23], [25], [26], [28], [48]–[53]. Among them, the schemes with the HE-based linear computation and GC-based nonlinear computation demonstrate superior

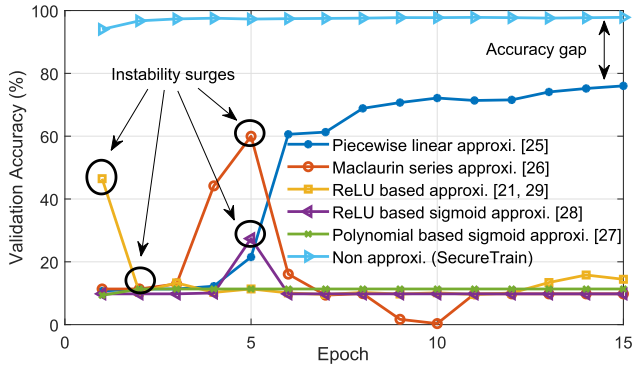

Fig. 2. Validation accuracy of different approximations.

TABLE I
KEY NOTATIONS

Symbol	Definition
$\mathcal{C}/\mathcal{S}$	Client/Server
$\boldsymbol{x}$	Input data from client
$\boldsymbol{w}/\boldsymbol{b}$	Initial model weight and bias
$\boldsymbol{w}^{\mathcal{C}}/\boldsymbol{w}^{\mathcal{S}}$	Weight share at client/server
$\boldsymbol{w}_1^{\mathcal{C}}/\boldsymbol{w}_2^{\mathcal{C}}$	Weight shares of $\boldsymbol{w}^{\mathcal{C}}$
$\boldsymbol{b}^{\mathcal{C}}/\boldsymbol{b}^{\mathcal{S}}$	Bias share at client/server
$\widehat{\boldsymbol{w}}/\widehat{\boldsymbol{b}}$	Updated model weight/bias
$[\,]_{\mathcal{C}}/[\,]_{\mathcal{S}}$	Ciphertext encrypted by client/server
$\boldsymbol{z}$	Linear output of neural network
r, r_1, r_2, h_1, h_2	Random numbers
$\boldsymbol{\delta}$	Backpropagation error

performance [22], [49], [50]. Specifically, GAZELLE [50] achieves a speedup of three orders of magnitude than the classic CryptoNets [20].

Although it is highly desirable to extend the above privacy-preserved inference frameworks to enable privacy-preserved training, there exist intrinsic challenges to achieve this ambitious goal. The challenges stem from the intractability and inefficiency of performing privacy-preserved non-polynomial computations, in particular the softmax function that is critical for model training.

First, the HE-based approaches only support polynomial computations (i.e., addition and multiplication), rendering it impossible to directly compute softmax. The SS-based approaches require multiple interactions between involved parties [29], which are not communication-efficient especially for higher-degree polynomial or nonlinear computations. The GC-based approaches require the boolean circuits for target functions [51], which render large circuit size for state-of-art networks and make the computation inefficient. A common approach is to approximate softmax using a tractable function, e.g., (1) a sigmoid replacement [25]–[28] or (2) a representation of ReLU over the sum of ReLU [21], [29], where sigmoid and ReLU are classic activation functions in DL [10]. Such approximation, though enabling privacy-preserved computation, comes with a price: 1) the approximation granularity has to be predetermined, which is not feasible in a collaborative learning scenario where each client is reluctant to release its data distribution property, 2) the model accuracy drops [25], [26] in inference and 3) unwarranted stability or even failures in training. Fig. 2 shows the change of validation accuracy during training for a 4-layer neural network[1] on MNIST dataset [54], with different softmax approximations. As can be seen, the softmax approximations result in the accuracy drop (e.g., nearly 20% accuracy degradation with the approximation in [25]), or failure to converge, which can be seen from the instability surges during training with the approximations in [21], [26]–[29].

Second, although the softmax function can be infinitely approximated, using piece-wise linear approximation or polynomial expansion, or digitized with GC, the resulting large-size approximation or digitization function hinders the system efficiency. For instance, the GC-based digitalization for softmax with n classes requires $80(n-1)$ gates per input for boolean circuits [38]. Our benchmark experiments show that it takes 11 seconds on a 3.2 GHz Intel Core processor to compute a softmax function with precision up to 12 fractional bits based on the IMAGENET dataset with 1000 classes [55]. Training a standard AlexNet network [10] based on IMAGENET would require roughly 1.2 M softmax computations for each epoch, which alone would take nearly 5 months should the above approximation is adopted.

At last, in addition to softmax, similar challenges are observed for other non-polynomial functions such as activation and pooling. In fact, the critical functions in DL have been carefully devised and proven effective by the machine learning community. The classic neural networks such as AlexNet [10], VGG [11] and ResNet [1] all use softmax to reach remarkable classification performance. Approximating these functions for the sake of privacy-preserved computation may lead to unacceptable performance loss or failure.

In summary, the progress in privacy-preserved DL inference has relied on approximation of non-polynomial functions. While the degraded accuracy and increased computation time resulted from the approximation can be controlled within a tolerance level for some inference applications, this approach is not directly applicable to achieve desired performance in training.

III. PRELIMINARIES

We now introduce the neural network training and the cryptographic background. Table I summaries the key notations that we use in the rest of the paper.

A. Neural Network Training

A neural network consists of multiple computation layers that represent a complex relation between the high-dimensional input and the output. Training a neural network is to fit model parameters to a training dataset. A typical training process consists of both *forward propagation* and *backpropagation*. Consider a multiclass classification problem to classify m-dimension input $\boldsymbol{x} = (x_1, x_2, \ldots, x_m)$ into a number of l classes, i.e., $\boldsymbol{y}_n = (y_1, y_2, \ldots, y_l)$. Assume that there are totally n layers except for the input layer. $\boldsymbol{w}_i$ and $\boldsymbol{b}_i$ are the

[1] The network structure is 784-128-128-10 where the input has 784 pixels for each digit image and the output has 10 classes from 0 to 9.

weight and bias matrices corresponding to the i-th $(1 \leq i \leq n)$ layer. Generally, the input $\boldsymbol{x}$ is denoted as the 0-th layer.

Forward Propagation: The forward propagation calculates weighted-sums.[2] layer by layer. The output of the i-th layer is activation $\boldsymbol{a}_i = f(\boldsymbol{z}_i)$, where $\boldsymbol{z}_i$ is the weighted-sum $\boldsymbol{a}_{i-1}\boldsymbol{w}_i + \boldsymbol{b}_i$; $f(\cdot)$ is the activation function, e.g., the *ReLU function* $f(x) = \max\{0, x\}$, *sigmoid function* $f(x) = \frac{1}{1+e^{-x}}$, and *tanh function* $f(x) = \frac{e^{2x}-1}{e^{2x}+1}$. The last layer adopts the *softmax function* to map a high-dimensional vector into a list of prediction probabilities, $\boldsymbol{y}_n = e^{\boldsymbol{z}_n} / \sum_{j=1}^{l} e^{z_{nj}}$, where $\boldsymbol{z}_n = \boldsymbol{a}_{n-1}\boldsymbol{w}_n + \boldsymbol{b}_n$.

Backpropagation: Once the forward propagation derives the prediction probability $\boldsymbol{y}_n$, the distance between the prediction and the true label $\boldsymbol{t} = (t_1, t_2, \ldots, t_l)$ is calculated as the cost. SecureTrain adopts the widely used cross entropy cost function, $C = -\sum_{j=1}^{l}(t_j \ln y_j + (1-t_j)\ln(1-y_j))$ [1], [10], [11]. The weight and bias are then updated based on the backward error propagation as $\widehat{\boldsymbol{w}}_i = \boldsymbol{w}_i - \eta\Delta\boldsymbol{w}_i$ and $\widehat{\boldsymbol{b}}_i = \boldsymbol{b}_i - \eta\Delta\boldsymbol{b}_i$ where η is the learning rate. The gradients $\Delta\boldsymbol{w}_i$, $\Delta\boldsymbol{b}_i$ are calculated as follows,

$$\Delta\boldsymbol{w}_i = \boldsymbol{a}_{i-1}\boldsymbol{\delta}_i, \Delta\boldsymbol{b}_i = \boldsymbol{\delta}_i, \tag{1}$$

where $\boldsymbol{\delta}_n = \boldsymbol{y}_n - \boldsymbol{t}$ based on the cross-entropy cost, $\boldsymbol{\delta}_i = \boldsymbol{\delta}_{i+1}\boldsymbol{w}_{i+1} \odot \frac{\partial \boldsymbol{a}_i}{\partial \boldsymbol{z}_i}$, and $\boldsymbol{a}_0 = \boldsymbol{x}$. As for the ReLU function, it is straighforward to compute $\frac{\partial \boldsymbol{a}_i}{\partial z_{ij}} = 1$ if $\boldsymbol{z}_{ij} \geq 0$, and $\frac{\partial \boldsymbol{a}_i}{\partial z_{ij}} = 0$ otherwise.

Since the learning rate η is a constant pre-determined by the client and server, we simplify the notations for updating the weight and bias as follows, assuming η has been multiplied into $\boldsymbol{\delta}_i$,

$$\widehat{\boldsymbol{w}}_i = \boldsymbol{w}_i - \boldsymbol{a}_{i-1}\boldsymbol{\delta}_i, \quad \widehat{\boldsymbol{b}}_i = \boldsymbol{b}_i - \boldsymbol{\delta}_i. \tag{2}$$

B. Cryptographic Tools

(1) Packed Homomorphic Encryption. Homomorphic Encryption (HE) is a crypto system that supports certain computations on encrypted data to obtain encrypted results, which, after decryption, match the corresponding results computed on the plaintext. It has found increasing applications in data communications, storage and computation [57]. Traditional HE operates on individual ciphertexts one by one [48]. The *packed homomorphic encryption* (PHE) enables packing multiple values into a single ciphertext and performs component-wise homomorphic computation in a Single Instruction Multiple Data (SIMD) manner [37]. Among various PHE techniques, SecureTrain uses the popular Cheon-Kim-Kim-Song (CKKS) scheme [58]. The secure computation involves two parties at a time, i.e., a client $\mathcal{C}$ and a server $\mathcal{S}$. When multiple clients are involved in training, they interact with the server in sequence.

In CKKS, the encryption algorithm encrypts a plaintext vector x in $\mathbb{R}^n$ into a ciphertext $[x]$ with n slots. We denote $[x]_{\mathcal{C}}$ and $[x]_{\mathcal{S}}$ as the ciphertexts encrypted by the private keys of client $\mathcal{C}$ and server $\mathcal{S}$, respectively. The decryption algorithm returns the plaintext vector x from the ciphertext $[x]$. Certain computation can be performed on the ciphertext. In general, an evaluation algorithm operates on input ciphertexts $[x_1], [x_2], \cdots$ and outputs a ciphertext $[x'] = f([x_1], [x_2], \cdots)$. The function f is constructed by *homomorphic addition* (Add), *homomorphic multiplication* (Mult) and *homomorphic permutation* (Perm). For example, $\text{Add}([x], [y]) = [x] + [y]$ outputs a ciphertext $[x + y]$ which encrypts the elementwise sum of x and y. $\text{Mult}([x], u) = [x] \odot u$ outputs a ciphertext $[x \odot u]$ which encrypts the elementwise multiplication of x and plaintext u. It is worth pointing out that SecureTrain is designed to require only scalar multiplication between a ciphertext and a plaintext, but not the much more expensive multiplication between two ciphertexts. Specifically, the run-time for the ciphertext-ciphertext multiplication is more than 4 times slower than the ciphertext-plaintext multiplication. $\text{Perm}([x])$ permutes the n slots in $[x]$ into another ciphertext $[x_\pi]$, where $x_\pi = (x(\pi_0), x(\pi_1), \cdots)$ and $\pi_i \in \{0, 1, \ldots, n-1\}$.

TABLE II
RUN-TIME FOR ARITH. MULT

Vector Dim.	Perm (ms)	Mult+Add (ms)	Potential Speedup
256	148.56	0.6	247×
512	167	0.66	253×
1024	185	0.7	264×
2048	204.3	0.76	268×
4096	222	0.81	274×

The complexities of the Add and Mult are significantly lower than Perm. For instance, our experiments on the Microsoft Seal Library [59] show that Perm is 254 times slower than Add and 97 times slower than Mult. The permutation is mainly used to obtain the arithmetic multiplication (dot product) in PHE, which is needed to compute the weighted sum and convolution in neural networks. Table II shows the run-time of the arithmetic multiplication (denoted as "Arith. Mult") between two vectors with a dimension from 256 to 4096. There exists a significant gap of over $200\times$ between Perm and other HE operations. SecureTrain completely eliminates Perm in inference and training, thus substantially reducing the computation time.

(2) Homomorphic Secret Share. In the secret sharing protocol, a value is shared between two parties, such that combining the two secrets yields the true value. SecureTrain is developed with an efficient secret share mechanism based on the homomorphic secret share (HSS) [30]–[32]. Specifically, a two-party HSS scheme for a class of programs $\boldsymbol{P}$ consists of algorithms $(Gen, Enc, Eval)$ with the following syntax: 1) $Gen(1^\lambda)$: On input a security parameter 1^λ, the key generation algorithm outputs a public key pk and a pair of evaluation keys (ek_0, ek_1). 2) $Enc(pk, \boldsymbol{x})$: Given public key pk and secret input value $\boldsymbol{x}$, the encryption algorithm outputs a ciphertext $\boldsymbol{ct}$. 3) $Eval(b, ek_b, (\boldsymbol{ct}_1, \ldots, \boldsymbol{ct}_n), P)$: On input party index $b \in \{0, 1\}$, evaluation key ek_b, vector of n ciphertexts, a program $P \in \boldsymbol{P}$ with n inputs, the homomorphic evaluation algorithm outputs y_b, constituting party b's share of an output $y = P(\boldsymbol{x})$.

For different functions (i.e., different programs P) with different homomorphic encryption (i.e., the different encryption algorithms $Enc(pk, \boldsymbol{x})$), the $Eval$ function should be

[2] The convolution operation in Convolutional Neural Network (CNN) can also be transformed into weighted-sum operation [56]

specifically designed, which, in our case is to develop $Eval$ function for the linear and nonlinear computations in neural networks with the packed homomorphic encryption. Here creative designs are required to enable its effective application in practice. This is because in many applications the two parties need to securely obtain $\boldsymbol{x}$ and perform computation on their respective shares to produce correct results. How to compute $\boldsymbol{x}$ (i.e., construct a set of data $\boldsymbol{ct}_1, \ldots, \boldsymbol{ct}_n$) and reconstruct the results (i.e., get the y_b) by $Eval$ function at each party is nontrivial, particularly for encrypted nonlinear computations in neural networks.

IV. Threat Model

As described in Section I, SecureTrain enables multiple clients to collaboratively train a neural network model. Similar to [21], [25]–[27], [40], [48], we adopt the semi-honest model, in which both the clients and server try to learn additional information from the received message, while they have bounded computational capability and do not collude. Specifically, a client $\mathcal{C}$ and the server $\mathcal{S}$ follow the protocol. However, $\mathcal{C}$ may want to learn the data from other clients, or intermediate results during training, or model parameters after training. On the other hand, $\mathcal{S}$ may attempt to learn the data from $\mathcal{C}$, or intermediate results during training, or the model parameters after training. Hence, the goal is to keep each client blind with other clients' private data while making the server oblivious of the private data from all the clients. Meanwhile, the intermediate results during training and model parameters after training need to be confidential to both server and client as they can expose the user data [14], [48].

We will prove that the proposed SecureTrain framework is secure under semi-honest corruption using the ideal/real security [33]. Since most of neural network applications are built on well-known architectures such as AlexNet [10], VGG16/19 [11] and ResNet50 [1], the clients and server are assumed to negotiate the network architecture at the beginning of training. Our framework does not target at protecting the architecture (number of layers, kernel size, etc), but the model parameters which are valuable information and can be used to derive client training data.

There is an array of emerging attacks to the security and privacy of neural networks [13]–[15], [60]–[62] that can be classified by the targeted processes: training, inference (model), or input.

(1) Training. The attack in [60] attempts to steal hyperparameters during training based on the assumption that the training dataset is available to the public in plaintext. This attack is not applicable in our case since our training data is encrypted. The membership inference attack [13], [63] aims to find out whether an input belongs to the training set by exploring the similarities between a duplicated model and the targeted model. This attack is not applicable to our framework since the users' data format, training process and model parameters are fully protected.

(2) Model. The model extraction attack [14] exploits the linear transformation at the inference stage to extract the model parameters. The model inversion attack [15] attempts to deduce the training datasets by finding the input that maximizes the classification probability. These attacks require full knowledge of the probability vectors from softmax layer, which are protected in SecureTrain. The Generative Adversarial Networks (GAN)-based attacks [61] can recover the training data by accessing the model. For SecureTrain, since the model parameters are well protected during and after training, this attack can be defended effectively.

(3) Input. A plethora of attacks adopt adversarial examples by adding a small perturbation to the input, causing misclassification by the neural network [62], [64]–[68]. This work considers rational clients that aim to train a good neural network model. Thus, this type of attacks do not apply in our framework.

V. System Design

A key design challenge to enable secure and privacy-preserved training is to develop a secure training framework that is *accurate* and *efficient*. The accuracy is imperative to ensure the success of training, while the computation efficiency is critical for practical applicability. In this paper, we propose a novel secure training framework, *SecureTrain*, that features the following design principles. First, SecureTrain is developed based on the Homomorphic Secret Sharing (HSS) approach [30]–[32] that enables secure and approximation-free computation for linear and non-linear functions, in order to achieve stable neural network training without accuracy loss. Second, SecureTrain is carefully designed according to the neural network architecture, by piggybacking part of the computation of the backpropagation into the forward propagation, and by combining linear and non-linear computation in both the forward and back propagation to accelerate the overall computation and minimize the total communication cost.

A. System Overview

The proposed SecureTrain framework supports multiple clients to work with a server to collaboratively train a neural network. The server sequentially interacts with each client to complete the training process. In the following discussion, we will focus on one client and one server only. Once the training with one client is finished, the client passes its share of the neural network model parameters to the next client. Note that the randomness of the share does not reveal user data or model parameters to the next client.

To start the training process, the weight and bias (i.e., $\boldsymbol{w}$ and $\boldsymbol{b}$) of each layer are initialized randomly. Without loss of generality, we consider the operation of one layer in the neural network in the discussions since the operations of different layers are similar. Moreover, for the ease of description, we omit the subscript or superscript to denote the network layer, and simply refer to $\boldsymbol{w}$ and $\boldsymbol{b}$ unless specified otherwise.

A novel secret share scheme is carefully crafted to protect both the user data and the neural network model parameters. More specifically, the client $\mathcal{C}$ and server $\mathcal{S}$ respectively keep their weight and bias shares $\boldsymbol{w}^{\mathcal{C}}$, $\boldsymbol{w}^{\mathcal{S}}$, $\boldsymbol{b}^{\mathcal{C}}$, and $\boldsymbol{b}^{\mathcal{S}}$, subject to $\boldsymbol{w} =$

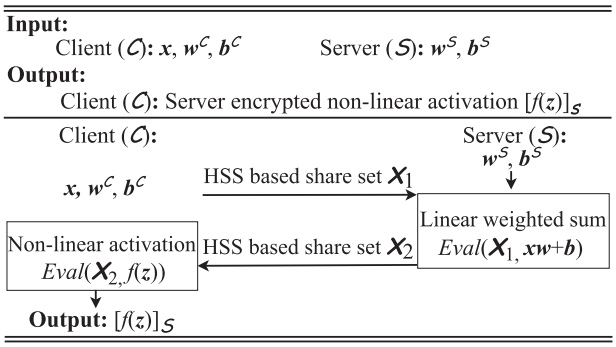

Fig. 3. Forward propagation.

$\boldsymbol{w}^{\mathcal{C}} + \boldsymbol{w}^{\mathcal{S}}$ and $\boldsymbol{b} = \boldsymbol{b}^{\mathcal{C}} + \boldsymbol{b}^{\mathcal{S}}$. Initially, the client $\mathcal{C}$ has the input data and the random share of weights and bias, while the server $\mathcal{S}$ has the other share of weights and bias. During training, the client and server update their shares of the weights and bias, respectively. Each training round consists of three stages, namely forward propagation, softmax calculation, and backpropagation.

B. Forward Propagation

During the forward propagation, the input data is fed in the forward direction through the network layers, as introduced in Sec. III-A. Each layer takes a vector of data, $\boldsymbol{x}$, as input to compute a linear transformation (i.e., the weighted sum, $\boldsymbol{z} = \boldsymbol{x}\boldsymbol{w} + \boldsymbol{b}$) followed by the nonlinear activations (i.e., $\boldsymbol{a} = f(z)$). The output (i.e., the activations $\boldsymbol{a}$) is then fed to the next layer, serving as the input to continue the forward propagation. The challenge is how to perform such computations in a secure and privacy-preserved manner based on the shares owned by $\mathcal{C}$ and $\mathcal{S}$.

The overall design principle of SecureTrain is based on the Homomorphic Secret Sharing (HSS). In this research, we apply HSS to develop an efficient approach to enable secure linear and non-linear computation. In particular, the key contribution is to devise innovative evaluation algorithms, i.e., $Eval$, based on the HSS masked paring scheme to ensure $P(\boldsymbol{\chi})$, i.e., the linear and non-linear functions in each layer, can be efficiently reconstructed from $Eval(\boldsymbol{\chi}_1, P)$ and $Eval(\boldsymbol{\chi}_2, P)$. The computation in all layers is essentially similar, but the treatment for the first layer and the rest layers is slightly different. In the following discussion, we introduce Layer 1 first and then highlight the difference in computing Layer k when k 2.

1) Calculation for the First Layer: As illustrated in Fig. 3, the input of the first layer is the client's data $\boldsymbol{x}$. In order to protect $\boldsymbol{x}$, $\mathcal{C}$ generates two random vectors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ where $\boldsymbol{x} = \boldsymbol{x}_1 + \boldsymbol{x}_2$. It also generates two random numbers r_1 and r_2. $\mathcal{C}$ sends a tuple $(r_1\boldsymbol{x}_1, r_2\boldsymbol{x}_2)$ to $\mathcal{S}$. This design is based on the (t, w)-threshold scheme [69] tailored by data splintering [70]. The detailed security analysis is given in Appendix A. Furthermore, $\mathcal{C}$ computes and sends $[r_2]_{\mathcal{C}}$, $[\frac{r_2}{r_1}]_{\mathcal{C}}$ and $[r_2(\boldsymbol{x}\boldsymbol{w}^{\mathcal{C}} + \boldsymbol{b}^{\mathcal{C}})]_{\mathcal{C}}$ to $\mathcal{S}$. Hereafter, the subscription $[\cdot]_{\mathcal{C}}$ denotes ciphertext encrypted by the client's private key, while $[\cdot]_{\mathcal{S}}$ denotes ciphertext encrypted by the private key of the server. All encryptions, unless specified otherwise, are realized by packed HE (e.g., CKKS).

Here $(r_1\boldsymbol{x}_1, r_2\boldsymbol{x}_2)$, $[r_2]_{\mathcal{C}}$, $[\frac{r_2}{r_1}]_{\mathcal{C}}$, and $[r_2(\boldsymbol{x}\boldsymbol{w}^{\mathcal{C}} + \boldsymbol{b}^{\mathcal{C}})]_{\mathcal{C}}$ form the HSS based share set $\boldsymbol{\chi}_1$, which will be used by $\mathcal{S}$ for calculating the linear weighted sum $Eval(\boldsymbol{\chi}_1, \boldsymbol{x}\boldsymbol{w} + \boldsymbol{b})$ as to be introduced next.

Calculation of Linear Weighted Sum at Server. The server $\mathcal{S}$ has its share of the neural network model parameters, i.e., $\boldsymbol{w}^{\mathcal{S}}$ and $\boldsymbol{b}^{\mathcal{S}}$. Upon receiving $(r_1\boldsymbol{x}_1, r_2\boldsymbol{x}_2)$, $[r_2(\boldsymbol{x}\boldsymbol{w}^{\mathcal{C}} + \boldsymbol{b}^{\mathcal{C}})]_{\mathcal{C}}$, $[r_2]_{\mathcal{C}}$ and $[\frac{r_2}{r_1}]_{\mathcal{C}}$ from $\mathcal{C}$, $\mathcal{S}$ computes the following:

$$\begin{cases} r_1\boldsymbol{x}_1\boldsymbol{w}^{\mathcal{S}} \odot [\frac{r_2}{r_1}]_{\mathcal{C}} = [r_2\boldsymbol{x}_1\boldsymbol{w}^{\mathcal{S}}]_{\mathcal{C}}, \\ \boldsymbol{b}^{\mathcal{S}} \odot [r_2]_{\mathcal{C}} = [r_2\boldsymbol{b}^{\mathcal{S}}]_{\mathcal{C}}, \end{cases} \tag{3}$$

where '$\odot$' denotes element-wise multiplication, which is based on the packed HE if it involves ciphertext.[3] Then $\mathcal{S}$ computes $[r_2\boldsymbol{x}_1\boldsymbol{w}^{\mathcal{S}}]_{\mathcal{C}} + r_2\boldsymbol{x}_2\boldsymbol{w}^{\mathcal{S}} + [r_2\boldsymbol{b}^{\mathcal{S}}]_{\mathcal{C}} = [r_2(\boldsymbol{x}\boldsymbol{w}^{\mathcal{S}} + \boldsymbol{b}^{\mathcal{S}})]_{\mathcal{C}}$, and finally obtains the following which is essentially the weighted sum, but scrambled by r_2 and encrypted by $\mathcal{C}$:

$$[r_2(\boldsymbol{x}\boldsymbol{w}^{\mathcal{S}} + \boldsymbol{b}^{\mathcal{S}})]_{\mathcal{C}} + [r_2(\boldsymbol{x}\boldsymbol{w}^{\mathcal{C}} + \boldsymbol{b}^{\mathcal{C}})]_{\mathcal{C}} = [r_2\boldsymbol{z}]_{\mathcal{C}}, \tag{4}$$

Calculation of Non-Linear ReLU Activation at Client. The next step is to calculate the activation. Here, we focus on ReLU, which is predominantly used in state-of-the-art deep neural networks due to its superior performance [10]. Similar design can be readily extended to other activation functions (e.g. *sigmoid* and *tanh* functions), as shown in Appendix B.

$\mathcal{S}$ cannot perform the activation calculation directly as discussed in Sec. II. A naive approach is to let $\mathcal{S}$ send $[r_2 z]_{\mathcal{C}}$ to $\mathcal{C}$, which then recovers z and calculates the ReLu function. However, releasing the weighted sum z to $\mathcal{C}$ can leak the model parameters as shown in [14], [48].

To securely perform the activation calculation, $\mathcal{S}$ scrambles each element in $r_2 z$ by a random vector $\boldsymbol{v}^{\mathcal{S}}$:

$$[r_2 z]_{\mathcal{C}} \odot \boldsymbol{v}^{\mathcal{S}} = [r_2\boldsymbol{z} \odot \boldsymbol{v}^{\mathcal{S}}]_{\mathcal{C}}. \tag{5}$$

Meanwhile, $\mathcal{S}$ generates a vector $\boldsymbol{u}^{\mathcal{S}}$ satisfying $\boldsymbol{u}^{\mathcal{S}} \odot \boldsymbol{v}^{\mathcal{S}} = \{1\}$, and constructs two vectors $\boldsymbol{g}_1$ and $\boldsymbol{g}_2$ with the same dimension as $\boldsymbol{z}$:

$$\boldsymbol{g}_1 = [g_{11}, g_{12}, \ldots, g_{1j}, \cdots],$$
$$\boldsymbol{g}_2 = [g_{21}, g_{22}, \ldots, g_{2j}, \cdots],$$

where (g_{1j}, g_{2j}) is a pair of *polar indicators*, given below:

$$(g_{1j}, g_{2j}) = \begin{cases} (0, u_j^{\mathcal{S}}), \ if \ v_j^{\mathcal{S}} > 0, \\ (u_j^{\mathcal{S}}, -u_j^{\mathcal{S}}), \ if \ v_j^{\mathcal{S}} < 0. \end{cases} \tag{6}$$

$\mathcal{S}$ encrypts $\boldsymbol{g}_1$ and $\boldsymbol{g}_2$ into $[\boldsymbol{g}_1]_{\mathcal{S}}$ and $[\boldsymbol{g}_2]_{\mathcal{S}}$, and sends them along with $[r_2\boldsymbol{z} \odot \boldsymbol{v}^{\mathcal{S}}]_{\mathcal{C}}$ to $\mathcal{C}$. These three items form the HSS based share set $\boldsymbol{\chi}_2$, which will be used by $\mathcal{C}$ for calculating the non-linear ReLU activation $Eval(\boldsymbol{\chi}_2, f(\boldsymbol{z}))$. Note that, $[\boldsymbol{g}_1]_{\mathcal{S}}$ and $[\boldsymbol{g}_2]_{\mathcal{S}}$ can be transmitted offline since $\boldsymbol{g}_1$ and $\boldsymbol{g}_2$ are

[3] The addition between two ciphertext (or between one ciphertext and one plaintext) is also in element-wise manner.

pre-generated by $\mathcal{S}$. Upon receiving the inputs from $\mathcal{S}$, $\mathcal{C}$ first obtains $\boldsymbol{y} = \boldsymbol{z} \odot \boldsymbol{v}^{\mathcal{S}}$ by decrypting $[r_2 \boldsymbol{z} \odot \boldsymbol{v}^{\mathcal{S}}]_{\mathcal{C}}$ and canceling the r_2 term. We now show how the client $\mathcal{C}$ can compute ReLU based on $\boldsymbol{y}$, $[\boldsymbol{g}_1]_{\mathcal{S}}$ and $[\boldsymbol{g}_2]_{\mathcal{S}}$.

Lemma 1: $[\boldsymbol{g}_1]_{\mathcal{S}} \odot \boldsymbol{y} + [\boldsymbol{g}_2]_{\mathcal{S}} \odot f(\boldsymbol{y})$ recovers the server-encrypted true ReLu function outcome, i.e., $[f(\boldsymbol{z})]_{\mathcal{S}}$.

Proof: If $\mathcal{C}$ had the true weighted sum outcome, i.e., $\boldsymbol{z}$, the corresponding ReLu function would be calculated as follows:

$$f(z_j) = \begin{cases} z_j, \ if \ z_j \geq 0 \\ 0, \ if \ z_j < 0, \end{cases} \tag{7}$$

for each element z_j in $\boldsymbol{z}$, as introduced in Sec. III-A.

However, $\mathcal{C}$ only has $y_j = v_j^{\mathcal{S}} \times z_j$. Since $v_j^{\mathcal{S}}$ is a random number that could be positive or negative, it is infeasible to obtain the correct activation directly. Instead, $\mathcal{C}$ computes

$$[\boldsymbol{g}_1]_{\mathcal{S}} \odot \boldsymbol{y} + [\boldsymbol{g}_2]_{\mathcal{S}} \odot f(\boldsymbol{y}). \tag{8}$$

Since $y_j = v_j^{\mathcal{S}} \times z_j$, $f(y_j)$ may yield four possible outputs, depending on the signs of $v_j^{\mathcal{S}}$ and z_j.

$$f(y_j) = \begin{cases} y_j, \ if \ v_j^{\mathcal{S}} > 0 \ \& \ z_j > 0 \\ y_j, \ if \ v_j^{\mathcal{S}} < 0 \ \& \ z_j < 0 \\ 0, \ if \ v_j^{\mathcal{S}} > 0 \ \& \ z_j \leq 0 \\ 0, \ if \ v_j^{\mathcal{S}} < 0 \ \& \ z_j \geq 0. \end{cases} \tag{9}$$

For example, when $v_j^{\mathcal{S}} > 0$ and $z_j > 0$, we have $g_{1j} = 0$ and $g_{2j} = u_j^{\mathcal{S}}$ according to Eq. (6). Therefore,

$$[g_{1j}]_{\mathcal{S}} \odot y_j = [0]_{\mathcal{S}}, [g_{2j}]_{\mathcal{S}} \odot f(y_j) = [u_j^{\mathcal{S}} \times v_j^{\mathcal{S}} \times z_j]_{\mathcal{S}}.$$

Note that we have chosen $v_j^{\mathcal{S}} u_j^{\mathcal{S}} = 1$. Therefore, Eq. (8) should yield $[z_j]_{\mathcal{S}}$. This is clearly the server-encrypted ReLu output, i.e., the correct result of $[f(z_j)]_{\mathcal{S}}$. Similarly, it can be shown that Eq. (8) always produce the server-encrypted ReLu outcome for other cases of $v_j^{\mathcal{S}}$ and z_j in Eq. (9). The lemma is thus proven. ■

By Lemma 1, $\mathcal{C}$ has successfully obtained $[f(z)]_{\mathcal{S}}$. This ends the computation in the first layer.

2) Calculation for the k-Th Layer $(2 \leq k \leq n)$*:* In a neural network, the activations will be fed into the next layer as the input to continue the forward propagation. So, we essentially want to let $\boldsymbol{x} = f(z)$ and repeat the calculations discussed above for all layers.

However, we are facing a new challenge because $\mathcal{C}$ only has the encrypted $[f(z)]_{\mathcal{S}}$, but not the plaintext data as in the first layer. $\mathcal{C}$ could still let $\boldsymbol{x} = [f(z)]_{\mathcal{S}}$. As discussed in the first layer, it is not an option to provide $\boldsymbol{x}$ directly to $\mathcal{S}$, since $\mathcal{S}$ would recover $f(z)$ and accordingly derive the user data [14], [15], [48]. As a result, $\mathcal{C}$ generates two shares, $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ where $\boldsymbol{x}_1$ is a random vector and $\boldsymbol{x}_2 = \boldsymbol{x} - \boldsymbol{x}_1$, as discussed before. Note that, $\boldsymbol{x}_2$ is essentially encrypted by $\mathcal{S}$ since $\boldsymbol{x}$ is in the PHE domain. This leads to a fundamental challenge in calculating $r_2(\boldsymbol{x}\boldsymbol{w}^{\mathcal{C}} + \boldsymbol{b}^{\mathcal{C}})$, because it would require a vector multiplication namely the dot product, which is computationally expensive in the PHE domain as discussed in Sec. III-B. This renders it impractical to implement the envisioned secure training framework for modern neural networks.

We take a different approach by again adopting the (t, w)-threshold splintering strategy. More specifically, $\mathcal{C}$ constructs the tuple $(h_1 \boldsymbol{w}_1^{\mathcal{C}}, h_2 \boldsymbol{w}_2^{\mathcal{C}})$, where $\boldsymbol{w}_1^{\mathcal{C}} + \boldsymbol{w}_2^{\mathcal{C}} = \boldsymbol{w}^{\mathcal{C}}$, and h_1 and h_2 are two random numbers. It sends the tuple along with $[\frac{1}{h_1}]_{\mathcal{C}}$ and $[\frac{1}{h_2}]_{\mathcal{C}}$ to $\mathcal{S}$. Similar to the discussion for the first layer, $\mathcal{C}$ also sends $(r_1 \boldsymbol{x}_1, r_2 \boldsymbol{x}_2)$, $[r_2(\boldsymbol{x}_1 \boldsymbol{w}^{\mathcal{C}} + \boldsymbol{b}^{\mathcal{C}})]_{\mathcal{C}}$, $[r_2]_{\mathcal{C}}$ and $[\frac{r_2}{r_1}]_{\mathcal{C}}$ to $\mathcal{S}$. The above two tuples and five ciphertexts form the HSS based share set $\boldsymbol{\chi}_1$, which will be used by $\mathcal{S}$ for calculating the linear weighted sum $Eval(\boldsymbol{\chi}_1, \boldsymbol{x}\boldsymbol{w} + \boldsymbol{b})$.

Upon receiving the inputs from $\mathcal{C}$, $\mathcal{S}$ performs the following calculation using the received two tuples and five ciphertexts:

(1) Similar to the discussion in the first layer, $\mathcal{S}$ computes $r_1 \boldsymbol{x}_1 \boldsymbol{w}^{\mathcal{S}} \odot [\frac{r_2}{r_1}]_{\mathcal{C}} = [r_2 \boldsymbol{x}_1 \boldsymbol{w}^{\mathcal{S}}]_{\mathcal{C}}$.
(2) Similar to the first layer, $\mathcal{S}$ computes $r_2 \boldsymbol{x}_2 \boldsymbol{w}^{\mathcal{S}}$.
(3) Similar to the first layer, $\mathcal{S}$ computes $\boldsymbol{b}^{\mathcal{S}} \odot [r_2]_{\mathcal{C}} = [r_2 \boldsymbol{b}^{\mathcal{S}}]_{\mathcal{C}}$.
(4) $\mathcal{S}$ computes $r_2 \boldsymbol{x}_2 h_1 \boldsymbol{w}_1^{\mathcal{C}} \odot [\frac{1}{h_1}]_{\mathcal{C}} = [r_2 \boldsymbol{x}_2 \boldsymbol{w}_1^{\mathcal{C}}]_{\mathcal{C}}$.
(5) $\mathcal{S}$ computes $r_2 \boldsymbol{x}_2 h_2 \boldsymbol{w}_2^{\mathcal{C}} \odot [\frac{1}{h_2}]_{\mathcal{C}} = [r_2 \boldsymbol{x}_2 \boldsymbol{w}_2^{\mathcal{C}}]_{\mathcal{C}}$.

Summing up the above five terms along with $[r_2(\boldsymbol{x}_1 \boldsymbol{w}^{\mathcal{C}} + \boldsymbol{b}^{\mathcal{C}})]_{\mathcal{C}}$ received from $\mathcal{C}$, $\mathcal{S}$ obtains the following:

$$\begin{aligned} &[r_2(\boldsymbol{x}_1 \boldsymbol{w}^{\mathcal{C}} + \boldsymbol{b}^{\mathcal{C}})]_{\mathcal{C}} + [r_2 \boldsymbol{x}_1 \boldsymbol{w}^{\mathcal{S}}]_{\mathcal{C}} + r_2 \boldsymbol{x}_2 \boldsymbol{w}^{\mathcal{S}} \\ &\quad + [r_2 \boldsymbol{x}_2 \boldsymbol{w}_1^{\mathcal{C}}]_{\mathcal{C}} + [r_2 \boldsymbol{x}_2 \boldsymbol{w}_2^{\mathcal{C}}]_{\mathcal{C}} + [r_2 \boldsymbol{b}^{\mathcal{S}}]_{\mathcal{C}} \\ &\quad = [r_2(\boldsymbol{x}\boldsymbol{w} + \boldsymbol{b})]_{\mathcal{C}} = [r_2 \boldsymbol{z}]_{\mathcal{C}}. \end{aligned} \tag{10}$$

Thus, $\mathcal{S}$ has obtained the weighted sum but scrambled by r_2 and encrypted by $\mathcal{C}$. This is the same as Eq. (4) introduced in the first layer. The same method can be applied to continue the calculation of the non-linear activation. The process repeats until it reaches the last layer, which is followed by softmax to be discussed next.

C. Softmax Calculation

After $\mathcal{S}$ obtains the masked weighted sum $[r_2 z]_{\mathcal{C}}$ for the last layer, it starts to calculate the non-linear softmax function for backpropagation. As discussed in Sec. II, softmax is critical to the training process.

It is fundamentally challenging to efficiently calculate softmax under the secure training framework because the two mainstream approaches for secure computation (HE and GC) have limitations to calculate non-linear functions as discussed in Sec. II. Fig. 4 illustrates the softmax calculation under the secure training framework. The goal is to let $\mathcal{C}$ and $\mathcal{S}$ each obtain a secret share of the true softmax value, i.e., $\frac{e^z}{\sum_{j=1}^{l} e^{z_j}}$.

In order to precisely and securely calculate the softmax shares, random vectors are introduced at three occasions to protect the true value of $\boldsymbol{z}$. First, $\mathcal{S}$ generates a random vector $\boldsymbol{d}_{\mathcal{S}}$ with the same dimension as $\boldsymbol{z}$, and constructs $[e^{-\boldsymbol{d}_{\mathcal{S}}}]_{\mathcal{S}}$ which will be used for noise cancellation later. Recall that $\mathcal{S}$ has obtained $[r_2]_{\mathcal{C}}$ from $\mathcal{C}$ in the forward propagation, so $\mathcal{S}$ can compute

$$[r_2 \boldsymbol{z}]_{\mathcal{C}} + \boldsymbol{d}_{\mathcal{S}} \odot [r_2]_{\mathcal{C}} = [r_2(\boldsymbol{z} + \boldsymbol{d}_{\mathcal{S}})]_{\mathcal{C}}, \tag{11}$$

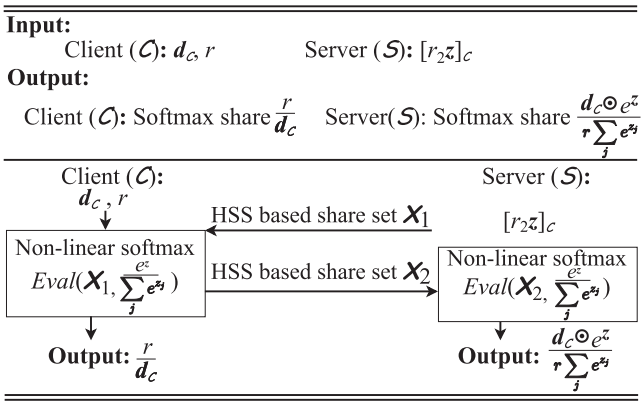

Fig. 4. Softmax calculation.

where z is scrambled by $d_{\mathcal{S}}$, and thus even $\mathcal{C}$ decrypts the above, it would not know z. $\mathcal{S}$ sends both $[r_2(z + d_{\mathcal{S}})]_{\mathcal{C}}$ and $[e^{-d_{\mathcal{S}}}]_{\mathcal{S}}$ to $\mathcal{C}$. The above two ciphertexts form the HSS based share set χ_1, which will be used by $\mathcal{C}$ for calculating the non-linear softmax $Eval(\chi_1, \frac{e^z}{\sum_{j=1}^{l} e^{z_j}})$. Upon receiving them, $\mathcal{C}$ decrypts the former and cancels r_2 to obtain $z + d_{\mathcal{S}}$, and then computes the following:

$$re^{(z+d_{\mathcal{S}})} \odot [e^{-d_{\mathcal{S}}}]_{\mathcal{S}} + o = [re^z + o]_{\mathcal{S}}, \tag{12}$$

where r is a random number and o is a random zero-sum vector with $\sum_{j=1}^{l} o_j = 0$. r and o are introduced here to protect z. $\mathcal{C}$ further generates a random vector $d_{\mathcal{C}}$ and computes:

$$d_{\mathcal{C}} \odot e^{(z+d_{\mathcal{S}})}, \tag{13}$$

where $d_{\mathcal{C}}$ is introduced to protect z. $\mathcal{C}$ sends the results of Eqs. (12) and (13) to $\mathcal{S}$, which form the HSS based share set χ_2 that will be used by $\mathcal{S}$ for calculating the non-linear softmax $Eval(\chi_2, \frac{e^z}{\sum_{j=1}^{l} e^{z_j}})$.

$\mathcal{S}$ decrypts $[re^z + o]_{\mathcal{S}}$ to obtain $re^z + o$, and subsequently sums up all elements of the vector to compute $r\sum_{j=1}^{l} e^{z_j}$. At the same time, since $\mathcal{S}$ has $d_{\mathcal{S}}$, it obtains $d_{\mathcal{C}} \odot e^z$ by cancelling $e^{d_{\mathcal{S}}}$ in Eq. (13). Therefore, $\mathcal{S}$ computes its softmax share as follows:

$$\frac{d_{\mathcal{C}} \odot e^z}{r\sum_{j=1}^{l} e^{z_j}}. \tag{14}$$

Meanwhile, $\mathcal{C}$ constructs $\frac{r}{d_{\mathcal{C}}}$ as its share of softmax. Clearly, the true softmax value can be recovered by multiplying the two shares:

$$\frac{d_{\mathcal{C}} \odot e^z}{r\sum_{j=1}^{l} e^{z_j}} \odot \frac{r}{d_{\mathcal{C}}} = \frac{e^z}{\sum_{j=1}^{l} e^{z_j}}.$$

The shares at $\mathcal{C}$ and $\mathcal{S}$ serve as the input for backpropagation.

D. Backpropagation

As introduced in Sec. III-A, the backpropagation begins from the last layer to recursively update the network parameters. The weights and bias in the i-th layer are updated as follows where x is the activation from the previous layer, and w, b, and δ are the weight, bias and error in current layer.

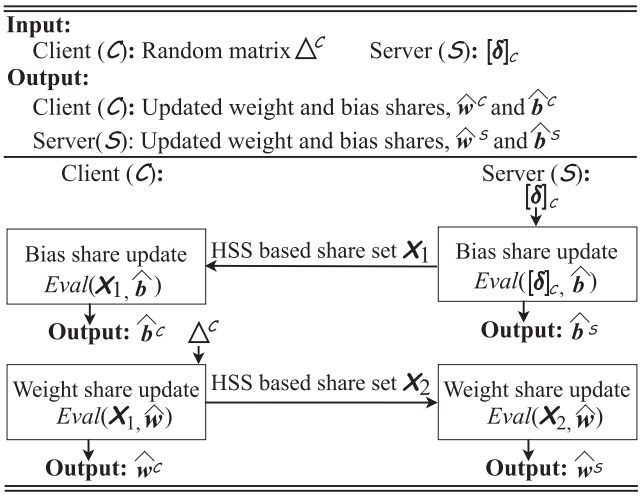

Fig. 5. Backpropagation diagram.

$$\widehat{w} = w - x\delta, \ \widehat{b} = b - \delta, \tag{15}$$

Fig. 5 shows the backpropagation. According to Eq. (15), the weight w, bias b and error δ in current layer, as well as non-linear activation x in previous layer are needed to update the weight and bias for the current layer. As discussed earlier, w, b and x are shared between $\mathcal{C}$ and $\mathcal{S}$ as $w^{\mathcal{C}}$, $b^{\mathcal{C}}$, $r_1 x_1$ and $w^{\mathcal{S}}$, $b^{\mathcal{S}}$, $r_2 x_2$, respectively.

1) Update of Weight/Bias in the Last Layer: In order to update the weights and bias for the last layer, we first introduce how to calculate δ. Recall that, after the softmax calculation, $\mathcal{C}$ and $\mathcal{S}$ respectively have the shares $\frac{r}{d_{\mathcal{C}}}$ and $\frac{d_{\mathcal{C}} \odot e^z}{r\sum_{j=1}^{l} e^{z_j}}$. For backpropagation, $\mathcal{C}$ sends $[\frac{r}{d_{\mathcal{C}}}]_{\mathcal{C}}$ and $[t]_{\mathcal{C}}$ to $\mathcal{S}$ (piggybacked to the transmission of the results of Eqs. (12) and (13)), where $[t]_{\mathcal{C}}$ is the $\mathcal{C}$-encrypted label vector. $\mathcal{S}$ computes the following $\mathcal{C}$-encrypted ciphertext, i.e., $[\delta]_{\mathcal{C}}$, which essentially shows the difference between the output of softmax and the label vector:

$$\frac{d_{\mathcal{C}} \odot e^z}{r\sum_{j=1}^{l} e^{z_j}} \odot \left[\frac{r}{d_{\mathcal{C}}}\right]_{\mathcal{C}} - [t]_{\mathcal{C}} = \left[\frac{e^z}{\sum_{j=1}^{l} e^{z_j}} - t\right]_{\mathcal{C}} = [\delta]_{\mathcal{C}}. \tag{16}$$

Next, three steps are followed within one communication round to update the weights and bias at $\mathcal{C}$ and $\mathcal{S}$.

Step 1: Bias Share Update at $\mathcal{S}$. Once $\mathcal{S}$ obtains $[\delta]_{\mathcal{C}}$ by Eq. (16), it generates its share of δ as a random vector $\delta^{\mathcal{S}}$ and updates its bias share by $\widehat{b}^{\mathcal{S}} = b^{\mathcal{S}} - \delta^{\mathcal{S}}$. Note that this update involves no communication as $\delta^{\mathcal{S}}$ is self-generated by $\mathcal{S}$. Meanwhile, four ciphertexts are created by $\mathcal{S}$, which form the HSS based share set χ_1 and will be used to update weights and bias shares at $\mathcal{C}$. The first ciphertext is the other share of δ for $\mathcal{C}$: $[\delta]_{\mathcal{C}} - \delta^{\mathcal{S}} = [\delta - \delta^{\mathcal{S}}]_{\mathcal{C}} = [\delta^{\mathcal{C}}]_{\mathcal{C}}$. The second ciphertext is generated by masking δ element-wisely with a random noise vector $l^{\mathcal{S}}$: $[\delta]_{\mathcal{C}} \odot l^{\mathcal{S}} = [\delta \odot l^{\mathcal{S}}]_{\mathcal{C}}$.

The third ciphertext, $[\overrightarrow{r_2 x_2}]_{\mathcal{S}}$, is transformed and encrypted by $\mathcal{S}$ based on $r_2 x_2$, which is the share of $\mathcal{S}$ for the activation

function in the previous layer. As to be introduced in Step 2, $\mathcal{C}$ will need to compute the arithmetic multiplication (dot product) between $r_2\boldsymbol{x}_2$ and $\boldsymbol{\delta} \odot \boldsymbol{l}^{\mathcal{S}}$. However, the arithmetic multiplication is computationally expensive if directly done in HE as discussed in Sec. III-B. The transformation converts it into element-wise multiplication, which is significantly more efficient for HE computation. As illustrated in the figure below, the transformation essentially expands the original vector $r_2\boldsymbol{x}_2$ to a matrix ($\overrightarrow{r_2\boldsymbol{x}_2}$) by row filling, i.e., duplicating the element of each row, such that the dot product can be realized by element-wise multiplication. The fourth ciphertext, $[\overrightarrow{\boldsymbol{l}^{\mathcal{S}}}]_{\mathcal{S}}$, is transformed and encrypted by $\mathcal{S}$ according to $\boldsymbol{l}^{\mathcal{S}}$. It is generated in a way similar to the third ciphertext, but by column filling, i.e., duplicating the element of each column of $\boldsymbol{l}^{\mathcal{S}}$.

Step 2: Weight/Bias Share Update at $\mathcal{C}$. Upon receiving the four ciphertexts generated by $\mathcal{S}$, $\mathcal{C}$ decrypts $[\boldsymbol{\delta}^{\mathcal{C}}]_{\mathcal{C}}$ and $[\boldsymbol{\delta} \odot \boldsymbol{l}^{\mathcal{S}}]_{\mathcal{C}}$. Note that since $\boldsymbol{\delta}$ is perturbed by $\boldsymbol{l}^{\mathcal{S}}$, $\mathcal{C}$ cannot deduce $\boldsymbol{\delta}$. Then, $\mathcal{C}$ updates its bias share as:

$$\widehat{\boldsymbol{b}}^{\mathcal{C}} = \boldsymbol{b}^{\mathcal{C}} - \boldsymbol{\delta}^{\mathcal{C}}. \tag{17}$$

$\mathcal{C}$ generates a random matrix $\Delta^{\mathcal{C}}$ and updates its weight share as

$$\widehat{\boldsymbol{w}}^{\mathcal{C}} = \boldsymbol{w}^{\mathcal{C}} - \Delta^{\mathcal{C}}. \tag{18}$$

We will show later that the shares at $\mathcal{C}$ and $\mathcal{S}$ together result in a correct update of the weights and bias of the neural network.

$\mathcal{C}$ then calculates two terms that will enable $\mathcal{S}$ to update its weight share in Step 3. Specifically, the first term is

$$\boldsymbol{x}_1(\boldsymbol{\delta} \odot \boldsymbol{l}^{\mathcal{S}}). \tag{19}$$

The second term is

$$[\overrightarrow{r_2\boldsymbol{x}_2}]_{\mathcal{S}} \odot \overrightarrow{\boldsymbol{\delta} \odot \boldsymbol{l}^{\mathcal{S}}}, \tag{20}$$

where $\overrightarrow{\boldsymbol{\delta} \odot \boldsymbol{l}^{\mathcal{S}}}$ is transformed from $\boldsymbol{\delta} \odot \boldsymbol{l}^{\mathcal{S}}$ by column filling as illustrated in former figure. As $\mathcal{C}$ has r_2, it can cancel r_2 to obtain

$$[\overrightarrow{\boldsymbol{x}_2} \odot \overrightarrow{\boldsymbol{\delta} \odot \boldsymbol{l}^{\mathcal{S}}}]_{\mathcal{S}}. \tag{21}$$

Adding Eq. (19) and Eq. (21) results in

$$\boldsymbol{x}_1(\boldsymbol{\delta} \odot \boldsymbol{l}^{\mathcal{S}}) + [\overrightarrow{\boldsymbol{x}_2} \odot \overrightarrow{\boldsymbol{\delta} \odot \boldsymbol{l}^{\mathcal{S}}}]_{\mathcal{S}} = [\boldsymbol{x}(\boldsymbol{\delta} \odot \boldsymbol{l}^{\mathcal{S}})]_{\mathcal{S}},$$

where $\boldsymbol{x} = \boldsymbol{x}_1 + \boldsymbol{x}_2$ as discussed in Sec. V-B.

Finally, $\mathcal{C}$ calculates the corresponding weight share for $\mathcal{S}$ as:

$$[\boldsymbol{x}(\boldsymbol{\delta} \odot \boldsymbol{l}^{\mathcal{S}})]_{\mathcal{S}} - \Delta^{\mathcal{C}} \odot [\overrightarrow{\boldsymbol{l}^{\mathcal{S}}}]_{\mathcal{S}} = [\boldsymbol{x}(\boldsymbol{\delta} \odot \boldsymbol{l}^{\mathcal{S}}) - \Delta^{\mathcal{C}} \odot \overrightarrow{\boldsymbol{l}^{\mathcal{S}}}]_{\mathcal{S}}, \tag{22}$$

which forms the HSS based share set $\boldsymbol{\chi}_2$ and is then sent to $\mathcal{S}$ for weight update in Step 3.

Step 3: Weight Share Update at $\mathcal{S}$. By decrypting the ciphertext from Eq. (22), $\mathcal{S}$ gets $\boldsymbol{x}(\boldsymbol{\delta} \odot \boldsymbol{l}^{\mathcal{S}}) - \Delta^{\mathcal{C}} \odot \overrightarrow{\boldsymbol{l}^{\mathcal{S}}}$. As $\boldsymbol{l}^{\mathcal{S}}$ is known by $\mathcal{S}$, it can be cancelled, yielding $\boldsymbol{x}\boldsymbol{\delta} - \Delta^{\mathcal{C}}$.$\mathcal{S}$ finally updates its weight share by

$$\widehat{\boldsymbol{w}}^{\mathcal{S}} = \boldsymbol{w}^{\mathcal{S}} - (\boldsymbol{x}\boldsymbol{\delta} - \Delta^{\mathcal{C}}). \tag{23}$$

It is easy to verify that the sum of updated weight and bias at $\mathcal{C}$ and $\mathcal{S}$ are

$$\widehat{\boldsymbol{b}}^{\mathcal{C}} + \widehat{\boldsymbol{b}}^{\mathcal{S}} = \boldsymbol{b} - \boldsymbol{\delta} \text{ and } \widehat{\boldsymbol{w}}^{\mathcal{C}} + \widehat{\boldsymbol{w}}^{\mathcal{S}} = \boldsymbol{w} - \boldsymbol{x}\boldsymbol{\delta},$$

which are exactly the updated weights and bias as shown in Eq. (15). By now the update of weights and bias at $\mathcal{C}$ and $\mathcal{S}$ is completed for the last layer. The communication is within one round.

2) Update of Weight/Bias in the k-Th Layer ($k \leq (n-1)$): The backpropagation in the k-th layer is very similar to that in the last layer as introduced above. The only difference is the calculation of $[\boldsymbol{\delta}]_{\mathcal{C}}$. In the last layer, $[\boldsymbol{\delta}]_{\mathcal{C}} = [\frac{e^{\boldsymbol{z}}}{\sum_{j=1}^{l} e^{z_j}} - \boldsymbol{t}]_{\mathcal{C}}$ as shown in Eq. (16), which is simply the difference between the output of softmax and the label vector. In the k-th layer, $[\boldsymbol{\delta}]_{\mathcal{C}}$ depends on the derivative of the activation function, i.e., $\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{z}}$. More specifically, $\boldsymbol{\delta}$ in the k-th layer should be computed as:

$$\boldsymbol{\delta} = \boldsymbol{\delta}\boldsymbol{w} \odot \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{z}} = (\boldsymbol{\delta}^{\mathcal{C}} + \boldsymbol{\delta}^{\mathcal{S}})(\boldsymbol{w}^{\mathcal{C}} + \boldsymbol{w}^{\mathcal{S}}) \odot \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{z}},$$

where $\boldsymbol{w}$ and $\boldsymbol{\delta}$ are the weight and error of the $(k+1)$-th layer, while $\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{z}}$ is the derivative of the current layer's activation function.

The key challenge is to securely compute the derivative. This can be achieved by embedding the computation into the forward propagation. Recall that $[\boldsymbol{g}_1]_{\mathcal{S}}$ and $[\boldsymbol{g}_2]_{\mathcal{S}}$ have been introduced in the forward propagation to enable $\mathcal{C}$ to obtain the $\mathcal{S}$-encrypted ReLU by Eq. (8). To compute the derivative, $\mathcal{S}$ introduces another vector $\boldsymbol{g}_3$ and sends $[\boldsymbol{g}_3]_{\mathcal{S}}$ to $\mathcal{C}$, where

$$g_{3j} = \begin{cases} 0, \text{if } v_j^{\mathcal{S}} > 0 \\ 1, \text{if } v_j^{\mathcal{S}} < 0, \end{cases} \tag{24}$$

Accordingly, while $\mathcal{C}$ calculates the $\mathcal{S}$-encrypted ReLU by Eq. (8), it also computes the $\mathcal{S}$-encrypted ReLU derivative as follows:

$$f'_R(\boldsymbol{y}) + (1 - 2f'_R(\boldsymbol{y})) \odot [\boldsymbol{g}_3]_{\mathcal{S}}, \tag{25}$$

where $\boldsymbol{y}$ is the masked weighted sum as discussed in Sec. V-B and $f'_R(y_j)$ denotes the derivative of ReLU, which is 1 if $y_j > 0$ or 0 otherwise as introduced in Sec. III-A.

Lemma 2: Eq. (25) yields the $\mathcal{S}$-encrypted ReLU derivative, i.e., $[\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{z}}]_{\mathcal{S}}$.

Proof: If $v_j^{\mathcal{S}} > 0$, then $g_{3j} = 0$ and accordingly Eq. (25) results in $[f'_R(y_j)]_{\mathcal{S}}$. Since $y_j = v_j^{\mathcal{S}} \times z_j$ and $v_j^{\mathcal{S}} > 0$, it is straightforward to show that $f'_R(y_j) = f'_R(z_j)$. Therefore, Eq. (25) yields the $\mathcal{S}$-encrypted ReLU derivative. On the other hand, if $v_j^{\mathcal{S}} < 0$, we have $g_{3j} = 1$, and thus Eq. (25) results in $[1 - f'_R(y_j)]_{\mathcal{S}}$. It is again easy to show that $f'_R(y_j) = 1 - f'_R(z_j)$ when $v_j^{\mathcal{S}} < 0$. Therefore, Eq. (25) still yields the $\mathcal{S}$-encrypted ReLU derivative. ■

Till now, $\mathcal{C}$ has obtained the ReLU derivative in a way piggybacked to the forward propagation with marginal computation cost. A secret share approach can then follow to compute the backpropagation using a method similar to the last layer as

TABLE III
COMPUTATION COMPLEXITY

Methodology	Perm	Mult	Add
Halevi-Shoup [71]	$O(n_i)$	$O(n_i)$	$O(n_i)$
GAZELLE [50]	$O(\log \frac{n_s}{n_o} + \frac{n_i n_o}{n_s})$	$O(\frac{n_i n_o}{n_s})$	$O(\log \frac{n_s}{n_o} + \frac{n_i n_o}{n_s})$
SecureTrain(Inf.)	0	$O(1)$	$O(1)$
SecureTrain(Sof.)	0	$O(1)$	$O(1)$
SecureTrain(Bac.)	0	$O(\frac{n_i n_o}{n_s})$	$O(\frac{n_i n_o}{n_s})$

TABLE IV
COMMUNICATION COMPLEXITY IN EACH PART

Comput. part	Commu. cost in bit	Commu. round
Forward Prop.	$8s_c + 2n_i s_p(1 + n_o)$	1
Softmax	$3s_c + n_o s_p$	1
Backprop.	$4n_o s_p + (11 + \frac{3n_i n_o}{n_s})s_c$	1

discussed in Sec. V-D1. The detailed design is presented in Appendix C.

E. Complexity Analysis

The computation and communication complexities in a layer of SecureTrain are summarized in Tables III and IV, where n_i is the input dimension at a layer; n_o is the output dimension; n_s is the number of slots in a CKKS ciphertext; s_c is the size of a CKKS ciphertext in bit; and s_p is the size of a plaintext value in bit. We assume $n_s \gg n_i, n_o$, which is also adopted in [50].

The detailed analysis can be found in Appendix D. Table III summarizes the computation complexity of SecureTrain in the forward propagation (i.e., inference), softmax, and backpropagation. It also compares with classic methodology in [71] and the state-of-the-art approach in GAZELLE [50]. Note that [71] and [50] focus on inference only. SecureTrain reduces the layer-wise forward calculation to constant complexity by integrating secret-share-based plaintext calculation and the HE-based non-permutation computation. It is worth pointing out that SecureTrain finishes the linear and non-linear calculation for each layer with above complexity while [71] and [50] only compute the linear part. Meanwhile, we give the analytical communication complexity of SecureTrain in Table IV. The quantitative performance comparison is given in Sec. VII.

VI. SECURITY

We prove the security of SecureTrain using the simulation approach [33]. As discussed in Sec. IV, the semi-honest adversary A can compromise any one of the client or server, but not both (i.e., the client and server do not collude). Here, security means that the adversary only learns the inputs from the party that it has compromised, but nothing else beyond that. It is modeled by two interactions. The first is an interaction in the real world that parties follow the protocol in the presence of an adversary A and the environment machine Z which chooses the inputs to the parties; the second is an ideal interaction that parties forward their inputs to a trusted functionality machine F. To prove security, we demonstrate that no environment Z can distinguish the real and ideal interactions. In other words, we want to show that the real-world simulator achieves the same effect in the ideal interaction.

(1) Security against a semi-honest client. We define a simulator sim that simulates an admissible adversary A which has compromised the client in the real world. As for forward propagation (see Figure 3), sim conducts the following: 1) receives from Z the HSS based share set $\boldsymbol{\chi}_1$; 2) sends $\boldsymbol{\chi}_1$ to F and receives the HSS based share set $\boldsymbol{\chi}_2$, including three ciphertexts (see from Eq. (5)); 3) constructs another HSS based share set $\widetilde{\boldsymbol{\chi}}_1$, which has the same data structure as $\boldsymbol{\chi}_1$; and 4) sends $\widetilde{\boldsymbol{\chi}}_1$ to $\mathcal{S}$ and receives the HSS based share set $\widetilde{\boldsymbol{\chi}}_2$. Here, $\widetilde{\boldsymbol{\chi}}_2$ is indistinguishable from $\boldsymbol{\chi}_2$ due to the randomness of $v^{\mathcal{S}}$ in Eq. (5) and the security of CKKS. Thus the forward propagation is secure against a semi-honest client.

In softmax calculation as shown in Figure 4, sim conducts as follows: 1) receives from Z a random number r and a random vector $\boldsymbol{d}_{\mathcal{C}}$; 2) sends r and $\boldsymbol{d}_{\mathcal{C}}$ to F and receives the HSS based share set $\boldsymbol{\chi}_1$, including two ciphertexts (see from Eq. (11)); 3) constructs another random number $\widetilde{r}$ and random vector $\widetilde{\boldsymbol{d}}_{\mathcal{C}}$, which have the same structure as r and $\boldsymbol{d}_{\mathcal{C}}$; and 4) receives from $\mathcal{S}$ the HSS based share set $\widetilde{\boldsymbol{\chi}}_1$. Here $\boldsymbol{\chi}_1$ is indistinguishable from $\widetilde{\boldsymbol{\chi}}_1$ due to the randomness of $\boldsymbol{d}_{\mathcal{S}}$ in Eq. (11) and the security of CKKS. Thus the softmax calculation is secure against a semi-honest client.

In backpropagation as shown in Figure 5, sim conducts as follows: 1) receives from Z the random matrix $\Delta^{\mathcal{C}}$; 2) sends $\Delta^{\mathcal{C}}$ to F and gets HSS based share set $\boldsymbol{\chi}_1$; 3) constructs another random matrix $\widetilde{\Delta}^{\mathcal{C}}$; and 4) receives from $\mathcal{S}$ the HSS based share set $\widetilde{\boldsymbol{\chi}}_1$. Here $\widetilde{\boldsymbol{\chi}}_1$ is indistinguishable from $\boldsymbol{\chi}_1$ due to randomness of $\boldsymbol{\delta}^{\mathcal{S}}$ and $l^{\mathcal{S}}$, and the security of CKKS.

Furthermore, the calculation for $[\boldsymbol{\delta}]_{\mathcal{C}}$ is piggybacked in the weight/bias update for previous layer to enable the next round of weight/bias update. In such case, sim conducts as follows: 1) receives from Z a random vector $\boldsymbol{p}^{\mathcal{C}}$; 2) sends $\boldsymbol{p}^{\mathcal{C}}$ to F and gets a ciphertext according to Eq. (32) and a plaintext tuple $(r_3\boldsymbol{\delta}_1^{\mathcal{S}}, r_4\boldsymbol{\delta}_2^{\mathcal{S}})$; 3) constructs another random vector $\widetilde{\boldsymbol{p}}^{\mathcal{C}}$; 4) receives from $\mathcal{S}$ the ciphertext $[\boldsymbol{g}_3]_{\mathcal{S}}$ and calculates the ReLU derivative by Eq. (25); 5) calculates a ciphertext $[\widetilde{\boldsymbol{p}}^{\mathcal{S}}]_{\mathcal{S}}$ by Eq. (31) and sends $[\widetilde{\boldsymbol{p}}^{\mathcal{C}}]_{\mathcal{C}}$ and $[\widetilde{\boldsymbol{p}}^{\mathcal{S}}]_{\mathcal{S}}$ to $\mathcal{S}$; and 6) receives from $\mathcal{S}$ the ciphertext of Eq. (32) and the plaintext tuple with the same structure as $(r_3\boldsymbol{\delta}_1^{\mathcal{S}}, r_4\boldsymbol{\delta}_2^{\mathcal{S}})$. Here the ciphertext in step 2) are indistinguishable from these in step 6) due to the randomness of $q^{\mathcal{S}}$. The plaintext tuple in step 2) is also indistinguishable from the one in step 6) due to randomness of $\boldsymbol{\delta}^{\mathcal{S}}$. Thus, the backpropagation is secure against a semi-honest client.

(2) Security against a semi-honest server. The proof is similar to the security against a semi-honest client, as detailed in Appendix E.

VII. EVALUATION

We implement SecureTrain using a C++ backend. The source code is published at GitHub.[4] Both the client and server run

[4] https://github.com/ChiaoThon/SecureTrain

TABLE V
INFERENCE PERFORMANCE COMPARISON

Framework	Runtime (s)	Communication	Accuracy (%)
SecureML [21]	4.88	-	93.1
Minionn [22]	1.04	15.8MB	97.6
EzPC [23]	0.7	76MB	97.6
Xonn [24]	0.13	4.29MB	97.6
SecureTrain	**0.1**	**1.89MB**	**97.6**

Ubuntu and have an Intel i7-8700 3.2 GHz CPU with 12 cores and 16 GB RAM. The network link between them is a Gigabit Ethernet. This experiment setting is similar to the ones adopted in existing works such as [24]. The Microsoft SEAL package is used for HE computations [59]. The CKKS scheme is adopted in SecureTrain. Note that CKKS directly supports floating point encryption/decryption/operations. That is, it does not need to encode floating point numbers in NN compuation into integers for encryption/decryption as many other encryption schemes. The five parameters of CKKS, i.e., the polynomial modulus degree, coefficient modulus size, noise standard deviation, number of slots in the ciphertext, and the precision of floating point in bits, are set as 8192, 200, 3.2, 4096, and 40, respectively. Such parameter selection can guarantee the correct decryption of the 0-multiplicative-depth ciphertexts in SecureTrain (e.g., the randomized ciphertexts from server in Eq. (5)) on the crypto domain, as demonstrated in SEAL library [59]. We test SecureTrain on the widely used MNIST dataset [54] with 60 K training images and 10 K testing images. We adopt a classic neural network model that has been widely used in previous works including SecureML [21], GELU-Net [48], CryptoDL [27], SecureNN [29] and ABY [28]. It includes four layers including two hidden layers. The input dimension is 784 which corresponds to the total number of pixels in a MNIST image. The dimension for two hidden layers is 128. The output dimension is 10 which corresponds to 10 digit classes of MNIST dataset.

A. Performance in Inference

SecureTrain is able to conduct both inference and training. We first look at the inference performance compared with four state-of-the-art privacy-preserved inference frameworks. Table V illustrates their inference performance including the runtime, communication cost, and the inference accuracy. The runtime is the duration from the moment when the client sends an image to the server, to the moment when the client receives the inference result from the server. SecureTrain achieves an inference speedup of 48×, 10×, 7×, and 1.3×, respectively, compared with SecureML [21], MiniONN [22], EzPC [23], and Xonn [24]. SecureTrain achieves the same inference accuracy, 97.6%, as Minionn, EzPC and Xonn. With regard to the communication cost, SecureTrain outperforms other schemes by 2× to 40×. This is because all those schemes adopted GC for non-linear activation calculations, while SecureTrain uses a highly efficient approach based on HSS.

Next, as shown in Figure 6, we illustrate the performance gain of SecureTrain in non-linear computation by comparing

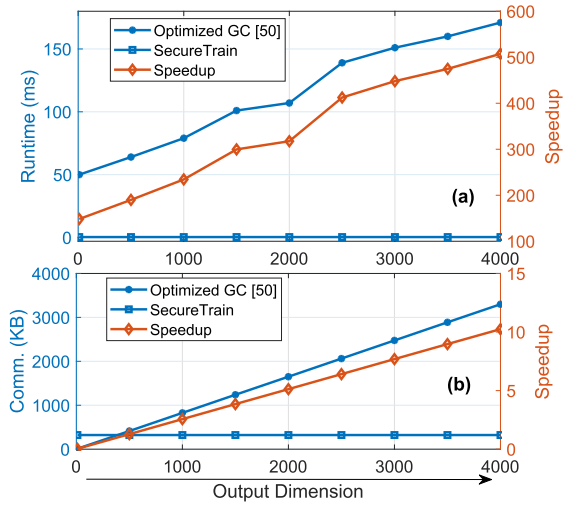

Fig. 6. Performance comparison for ReLU calculation: (a) Runtime and (b) Communication cost with different output dimensions.

the performance of ReLU calculation using GC and the scheme used by SecureTrain. Specifically, the SecureTrain scheme saves time about two orders of magnitude compared with GC. In these cases, the runtime of GC ranges from 50 to 171 milliseconds while the SecureTrain scheme can complete it around 0.4 ms. The communication cost reduction reaches up to one order of magnitude, thanks to both significantly reduced computation complexity and the scalability on data processing of the packed HE technique.

B. Performance in Training

In neural network training, a critical step is the computation of softmax, which is needed by the backpropagation. It is more difficult than other nonlinear functions, due to the specific form of the function which involves the exponential normalization of the input. The existing schemes all use approximation (see Fig. 2), e.g., the piecewise linear sigmoid approximation [25], Maclaurin sigmoid approximation [26], polynomial sigmoid approximation by CryptoDL [27], ReLU based sigmoid approximation by ABY [28], and ReLU based approximation by SecureML [21] and SecureNN [29]. Note that the first four schemes did not directly approximate the softmax function, but used an approximated sigmoid to substitute the softmax function.

Figs. 7(a)–(f) illustrate the training output over 10 epochs by the above five approximation approaches compared with SecureTrain that implements the original softmax function. Each row in a subfigure is the output vector corresponding to the 10 digit classes at a given epoch by the corresponding approach, while each column is the output value for a given class over 10 epochs. The training image is a digit '7'. Fig. 7(f) indicates that the training using SecureTrain that implements the original softmax function efficiently learns the input feature even at the early epochs, with a dominant value in the 8-th column (which corresponds to the class '7') and much smaller values in other columns. In contrast, all five approximation approaches (Fig. 7(a)–(e)) have poor performance. Among them, the piecewise linear

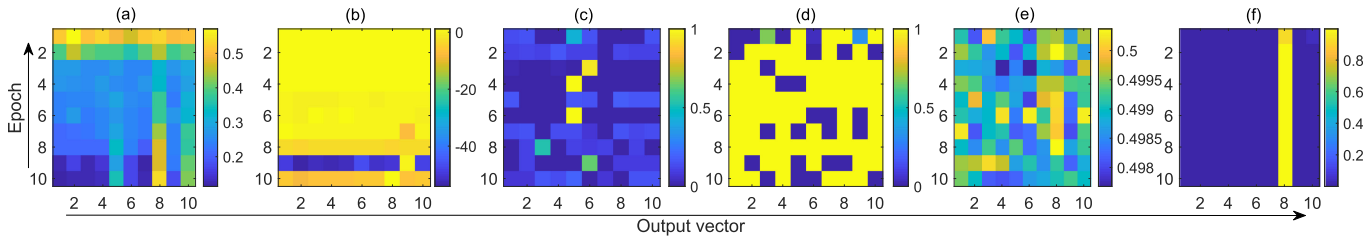

Fig. 7. Comparison of the last layer output of different approximations over 10 epochs: (a) Piecewise linear approximation [25]; (b) Maclaurin approximation [26]; (c) ReLU based softmax approximation [21], [29]; (d) ReLU based sigmoid approximation [28]; (e) Polynomial based sigmoid approximation [27]; (f) Non approximation in SecureTrain.

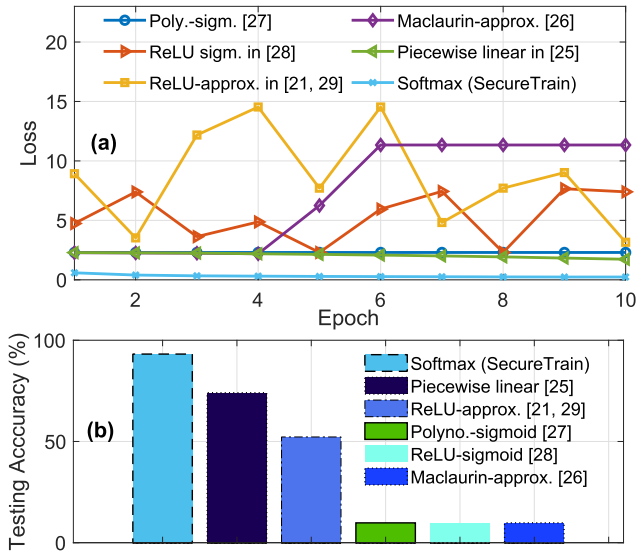

Fig. 8. Training loss and testing accuracy: (a) Loss during training and (b) Testing accuracy with different approximation approaches.

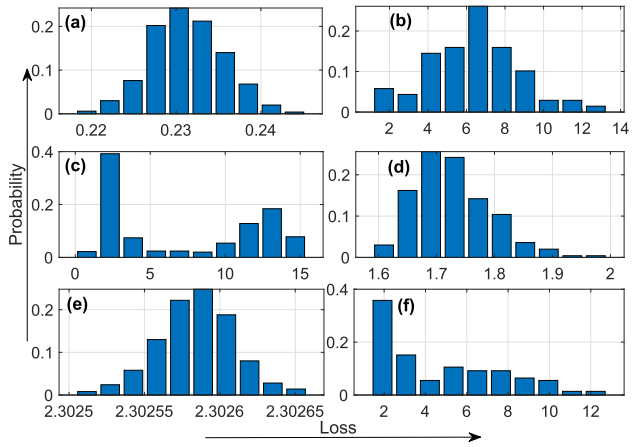

Fig. 9. Probability density distribution of training loss with different approximations: (a) Non approximation in SecureTrain; (b) ReLU based softmax approximation [21], [29]; (c) Maclaurin approximation [26]; (d) Piecewise linear approximation [25]; (e) Polynomial based sigmoid approximation [27]; (f) ReLU based sigmoid approximation [28].

approximation (Fig. 7(a)) performs better and converges, while other approximation approaches cannot learn the input feature '7' well and do not converge.

Next we examine the overall training loss and testing accuracy, as illustrated in Fig. 8. Among the approximation approaches, the piecewise linear approximation has a converged training loss. However, there is still a significant performance gap compared with SecureTrain, which converges significantly faster and achieves a 93.17% testing accuracy after 10 epochs. Other approximation approaches cannot even converge and have a poor testing accuracy. Fig. 8(a) illustrates that these approximation approaches have an unstable loss. ReLU based sigmoid approximation and Maclaurin approximation have about 10% accuracy, which is equivalent to a random guess, and indicates the trained models actually have not learned the features effectively. The polynomial based sigmoid approximation has an almost flat loss curve and a poor 10% accuracy, which indicates the model does not become better with the training. This happens when the input has a relatively wide range, which results in the polynomial approximation significantly deviating from the original softmax function.

To examine the training stability of each approximation approach, we train the network 500 times using each approach, and record the loss and testing accuracy for each experiment. Fig. 9 plots the *probability density distribution* (PDF) of the loss from the 500 experiments. While SecureTrain keeps the training loss around 0.23, the approximation based approaches have large loss. This indicates the poor training stability of those approaches. The polynomial based sigmoid approximation has a loss around 2.3. While it is relatively small, the problem of this approach is that the loss does not reduce or converge throughout the training process.

Fig. 10 plots the PDF of the testing accuracy over the 500 trainings for each approach. The testing accuracies of SecureTrain are all very close, centering around 93%. The piecewise linear approximation has an accuracy of around 73% to 77%. In contrast, other approximations have highly diverse accuracies among different experiments. This again indicates the poor training stability. The polynomial based sigmoid has a consistently poor accuracy around 10%, as the training process does not really converge. In summary, the training by SecureTrain is consistently stable, and the accuracy is much better than the approximation based approaches.

Next, we explore the performance under different network structures. We change the number of hidden neurons in each hidden layer from as small as 8 to 1024, where 1024 is widely used in modern neural network structures. All network structures are trained for 10 epochs. Fig. 11 illustrates the output distribution of SecureTrain and the five approximation

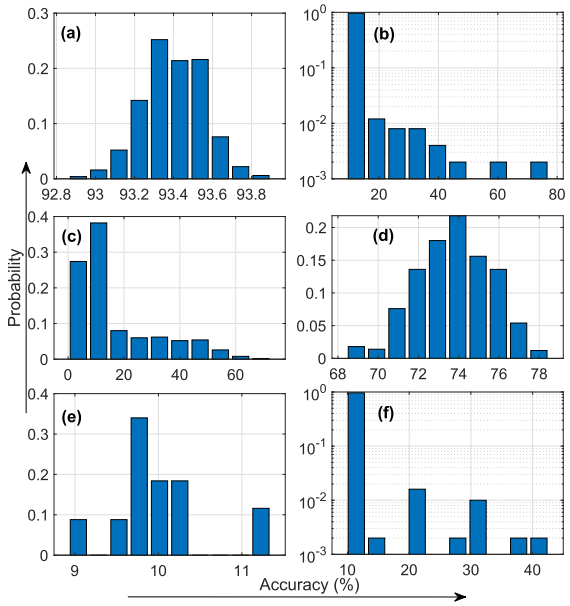

Fig. 10. Probability density distribution of accuracy with different approximations: (a) Non approximation in SecureTrain; (b) ReLU based softmax approximation [21], [29]; (c) Maclaurin approximation [26]; (d) Piecewise linear approximation [25]; (e) Polynomial based sigmoid approximation [27]; (f) ReLU based sigmoid approximation [28].

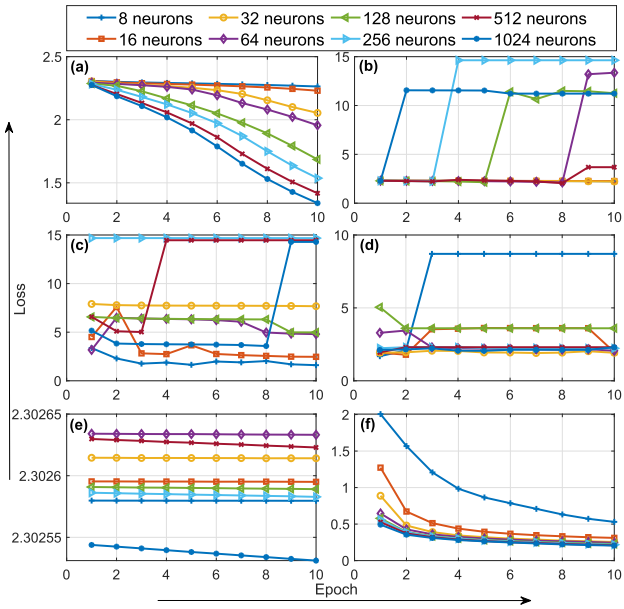

Fig. 12. Training loss of different network structures under different approximations: (a) Piecewise linear approximation [25]; (b) Maclaurin approximation [26]; (c) ReLU based softmax approximation [21], [29]; (d) ReLU based sigmoid approximation [28]; (e) Polynomial based sigmoid approximation [27]; (f) Non approximation in SecureTrain.

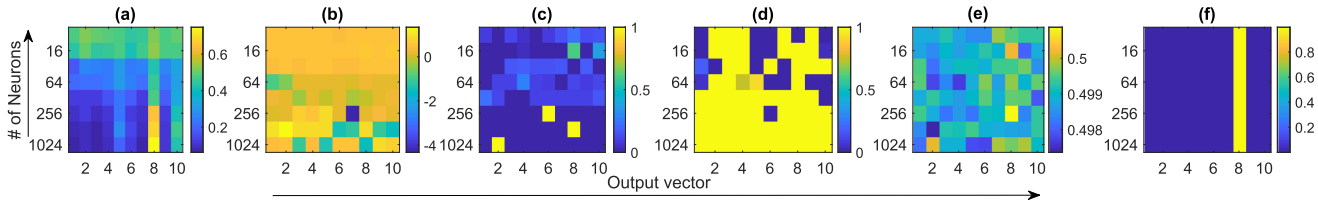

Fig. 11. Output probability distribution of each approach with different network structures in terms of the number of hidden neurons: (a) Piecewise linear approximation [25]; (b) Maclaurin approximation [26]; (c) ReLU based softmax approximation [21], [29]; (d) ReLU based sigmoid approximation [28]; (e) Polynomial based sigmoid approximation [27]; (f) Non approximation in SecureTrain.

approaches, as a function of the number of neurons in the hidden layer. Each row in a subfigure is the 10-dimension output vector for a network with the given number of hidden neurons. As can be seen, SecureTrain effectively learns the data feature under different network structures, which has a large output value for class '7' (corresponding to the 8-th column in the subfigure). Among the approximation schemes, the piecewise linear approximation performs relatively better than others. Nevertheless, it still has a significant performance gap compared with SecureTrain. It needs 128 or more hidden neurons to effectively recognize the input image. The remaining approximation approaches show unstable output distributions under different network structures. This illustrates the instability of these approximations as they work for certain network structures, but do not achieve the consistent stability for general larger networks.

Fig. 12 plots the training loss over 10 epochs under different network structures. We have the following observations. 1) The piecewise linear approximation approach performs better with a larger network (more hidden neurons); 2) the loss of the Maclaurin based approximation increases under all network settings; 3) the ReLU based approximation approach has loss decreasing for the network with 128 hidden neurons, while the loss bumps up and down or stays flat for all other networks; 4) the ReLU based sigmoid approximation approach performs similarly as the ReLU based approximation approach; 5) the loss of the polynomial based sigmoid approximation stays flat or decreases slightly (by about 10^{-5} for 10 epochs), which is technically not trainable. In contrast, SecureTrain converges for all network structures at a fast pace, thanks to its novel implementation of the original softamax.

The testing accuracies under different network structures for each approach are illustrated in Fig. 13. SecureTrain significantly outperforms all other approaches. Besides the low accuracy, another serious issue for the approximation based approaches (except the piecewise linear approximation) is that their accuracy decreases under a larger, more sophisticated network structures. The piecewise linear approximation approach is better than other approximation based approaches in that its accuracy increases under a larger network and reaches around 80% accuracy for the network with 1024 hidden neurons. However, it is still significantly lower than the 94% accuracy of SecureTrain.

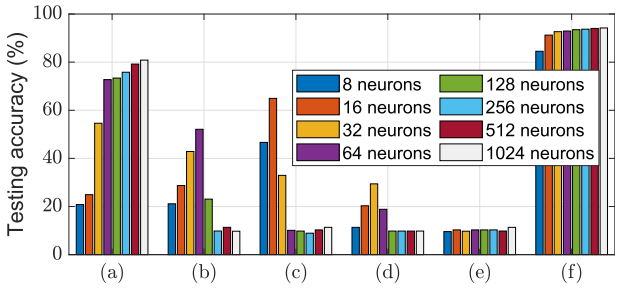

Fig. 13. Testing accuracy of different network structures under different approximations: (a) Piecewise linear approximation [25]; (b) Maclaurin approximation [26]; (c) ReLU based softmax approximation [21], [29]; (d) ReLU based sigmoid approximation [28]; (e) Polynomial based sigmoid approximation [27]; (f) Non approximation in SecureTrain.

TABLE VI
COMPARISON OF TRAINING TIME

Framework	Accuracy (%)	Time per batch (s)	TTP
SecureNN [29]	73.8	263.6	✓
SecureTrain	92.9	25.6	✗

Furthermore, Table VI shows the time cost of the state-of-the-art scheme and SecureTrain in training phase with batch size 128. We can see that SecureTrain keeps over 19% higher accuracy and a 10× training speedup. Meanwhile, SecureTrain dose not need a Trust Third Party (TTP), which is another sharp contrast in terms of practical usability as the TTP is not preferred in practice.

In summary, the softmax function in the last layer, is critical for the backpropagation in training. All existing approaches for privacy preserved neural network training must reply on approximation. However, most of them result in unstable training, which finally leads to an unusable model. The proposed SecureTrain uses a creative design to enable the secure implementation of the original softmax function as well as the updating process in a cost-efficient manner. Therefore, it maintains all the good features as the plaintext version of training, including the same model accuracy, the convergence of training, and better accuracy with a larger, more sophisticated model structure. SecureTrain also significantly outperforms all existing approaches in training speed, testing accuracy, and convergence speed.

VIII. Conclusion and Future Work

In this paper, we have proposed a novel privacy-preserved DL framework, *SecureTrain*, which address the two fundamental challenges faced by privacy-preserved DL model training: (1) model accuracy loss and training instability due to use of function approximation, and (2) computation efficiency. The overarching goal is to eliminate the use of function approximation to *carry out training without accuracy loss and instability*, and reduce the use of Perm operation to improve *computation efficiency*. SecureTrain features an innovative design that enables joint linear and non-linear computation based on the Homomorphic Secret Share (HSS) [30]–[32] to achieve approximation free of non-polynomial functions. On the other hand, it eliminates the time consuming Homomorphic *permutation operation* (Perm) by carefully designing the share set to substantially reduce the computation time. Moreover, SecureTrain exploits the data flow in both forward propagation and backpropagation to enable an efficient piggybacking, thus accelerating the overall computation and reducing the communication cost. We have analyzed the computation and communication complexity of SecureTrain and proven its security using the standard simulation approach. We have implemented SecureTrain and benchmarked its performance with well-known datasets. Our results have shown that SecureTrain not only ensures privacy-preserved inference and training, but also supports a significant speedup and training stability comparable to plaintext learning. Overall, SecureTrain is an accurate, efficient and privacy-preserved framework. To the best of knowledge, this is the first work that addresses both accuracy and efficiency in privacy-preserved deep neural network learning.

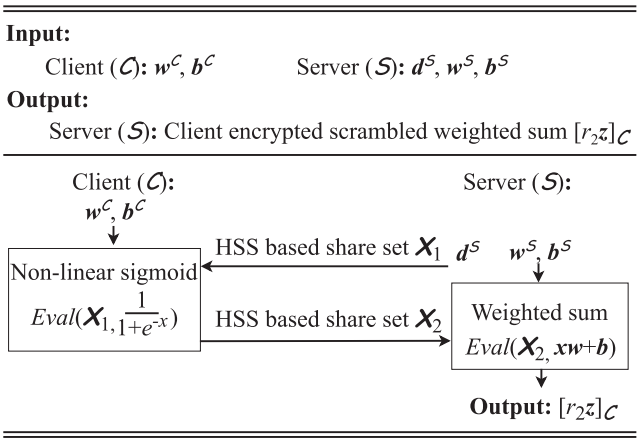

Fig. 14. Diagram of sigmoid calculation.

The privacy-preserved neural network training is at its early stage and dramatically developing. SecureTrain takes its initial effort in this forfront research. As future directions, we plan to make the system more adaptive to state-of-art network architectures such as support for changing learning rates, optimizers other than SGD that are required to achieve state-of-the-art performance, and hyperaparameter tuning on a validation set. Moreover, we plan to strengthen the security level to address malicious model, e.g., using the Zero-Knowledge proof [72].

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Patt. Recognit.*, 2016, pp. 770–778.
[2] O. M. Parkhi *et al.*, "Deep face recognition." in *Brit. Mach. Vis. Assoc.*, vol. 1, no. 3, 2015, p. 6.
[3] A. I. Maqueda, A. Loquercio, G. Gallego, N. García, and D. Scaramuzza, "Event-based vision meets deep learning on steering prediction for self-driving cars," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 5419–5427.
[4] C. Zhang, P. Patras, and H. Haddadi, "Deep learning in mobile and wireless networking: A survey," *IEEE Commun. Surv. Tut.*, vol. 21, no. 3, pp. 2224–2287, 2019.
[5] Q. Mao, F. Hu, and Q. Hao, "Deep learning for intelligent wireless networks: A comprehensive survey," *IEEE Commun. Surv. Tut.*, vol. 20, no. 4, pp. 2595–2621, 2018.

[6] X. Song, H. Kanasugi, and R. Shibasaki, "Deeptransport: Prediction and simulation of human mobility and transportation mode at a citywide level," in *Proc. 25th Int. Joint Conf. Artif. Intell.*, 2016, pp. 2618–2624.

[7] B. Mao *et al.*, "Routing or computing? the paradigm shift towards intelligent computer network packet transmission based on deep learning," *IEEE Trans. Comput.*, vol. 66, no. 11, pp. 1946–1960, Nov. 2017.

[8] V. L. Thing, "IEEE 802.11 network anomaly detection and attack classification: A deep learning approach," in *Proc. IEEE Wireless Commun. Netw. Conf., WCNC*. 2017, pp. 1–6.

[9] M. A. Wijaya, K. Fukawa, and H. Suzuki, "Intercell-interference cancellation and neural network transmit power optimization for mimo channels," in *Proc. IEEE 82nd Veh. Technol. Conf., VTC2015-Fall*, 2015, pp. 1–5.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[12] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," in *Proc. IEEE*, vol. 108, no. 4, pp. 485–532, Apr. 2020.

[13] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *Proc. Secur. Privacy (SP), IEEE Symp*. 2017, pp. 3–18.

[14] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," in *Proc. 25th USENIX Conf. Secur. Symp.*, 2016, pp. 601–618.

[15] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 1322–1333.

[16] G. J. Annas *et al.*, "Hipaa regulations-a new era of medical-record privacy?," *New Engl. J. Med.*, vol. 348, no. 15, pp. 1486–1490, 2003.

[17] P. Voigt and A. Von dem Bussche, "The eu general data protection regulation (GDPR)," *A Practical Guide, 1st Ed., Cham: Springer*, 2017.

[18] A. A. Abd EL-Latif, B. Abd-El-Atty, S. E. Venegas-Andraca, and W. Mazurczyk, "Efficient quantum-based security protocols for information sharing and data protection in 5G networks," *Future Gener. Comput. Syst.*, vol. 100, pp. 893–906, 2019.

[19] H. Qiu, K. Kapusta, Z. Lu, M. Qiu, and G. Memmi, "All-or-nothing data protection for ubiquitous communication: Challenges and perspectives," *Inf. Sci.*, vol. 502, pp. 434–445, 2019.

[20] 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 *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 201–210.

[21] P. Mohassel and Y. Zhang, "SecureMLl: A system for scalable privacy-preserving machine learning," in *Proc. 38th IEEE Symp. Secur. Privacy, SP*, 2017, pp. 19–38.

[22] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via minionn transformations," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 619–631.

[23] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, "Ezpc: programmable, efficient, and scalable secure two-party computation for machine learning," *Cryptol. ePrint Arch.*, Tech. Rep. 2017/1109, 2017.

[24] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. E. Lauter, and F. Koushanfar, "XONN: XNOR-based oblivious deep neural network inference," *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 171, 2019.

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

[26] J. Yuan and S. Yu, "Privacy preserving back-propagation neural network learning made practical with cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 1, pp. 212–221, Jan. 2014.

[27] E. Hesamifard, H. Takabi, M. Ghasemi, and R. N. Wright, "Privacy-preserving machine learning as a service," in *Proc. Privacy Enhancing Technol.*, vol. 2018, no. 3, pp. 123–142, 2018.

[28] P. Mohassel and P. Rindal, "Aby 3: a mixed protocol framework for machine learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 35–52.

[29] S. Wagh, D. Gupta, and N. Chandran, "Securenn: 3-party secure computation for neural network training," *Proc. Privacy Enhancing Technol.*, vol. 1, no. 3, pp. 26–49, Jul. 2019.

[30] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, and M. Orrù, "Homomorphic secret sharing: optimizations and applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2105–2122.

[31] E. Boyle, N. Gilboa, and Y. Ishai, "Breaking the circuit size barrier for secure computation under ddh," in *Proc. Annu. Int. Cryptol. Conf.*. Berlin, Germany: Springer, 2016, pp. 509–539.

[32] E. Boyle, N. Gilboa, and Y. Ishai, "Group-based secure computation: optimizing rounds, communication, and computation," in *Proc. Annu. Int. Conf. Theory Appl. Cryptogr. Techn.*, Berlin, Germany: Springer, 2017, pp. 163–193.

[33] O. Goldreich, *Foundations Of Cryptography: Volume 2, Basic Applications, Chapter 7*. Cambridge, U.K.: Cambridge University Press, 2009.

[34] O. Goldreich, "Secure multi-party computation," *Manuscript. Preliminary Version*, vol. 78, 1998.

[35] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei, "Faster cryptonets: Leveraging sparsity for real-world encrypted inference," 2018, *arXiv:1811.09953*.

[36] X. Jiang, M. Kim, K. Lauter, and Y. Song, "Secure outsourced matrix computation and application to neural networks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1209–1222.

[37] Z. Brakerski, C. Gentry, and S. Halevi, "Packed ciphertexts in lwe-based homomorphic encryption," in *Public-Key Cryptogr.–PKC 2013*. Berlin, Germany: Springer, 2013, pp. 1–13.

[38] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "Deepsecure: Scalable provably-secure deep learning," in *Proc. 55th ACM/ESDA/IEEE Des. Automat. Conf., DAC*, 2018, pp. 1–6.

[39] A. C.-C. Yao, "How to generate and exchange secrets," in *Proc. 27th Annu. Symp. Found. Comput. Sci., SFCS 1986*, 1986, pp. 162–167.

[40] L. Wan, W. K. Ng, S. Han, and V. Lee, "Privacy-preservation for gradient descent methods," in *Proc. 13th ACM SIGKDD Conf. Knowl. Discov. Data Mining,* 2007, pp. 775–783.

[41] F. McKeen *et al.*, "Innovative instructions and software model for isolated execution," in *Proc. 2nd Int. Workshop Hardware Architect. Support Secur. Privacy*, Jun. 2013.

[42] O. Ohrimenko *et al.*, "Oblivious multi-party machine learning on trusted processors," in *Proc. 25th USENIX Secur. Symp. (USENIX Security 16)*, 2016, pp. 619–636.

[43] S. P. Bayerl *et al.*, "Offline model guard: Secure and private ml on mobile devices," *Design, Automat. & Test Europe Conf. & Exhib., DATE*, 2020, pp. 460–465.

[44] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation, NSDI 17*, 2017, pp. 283–298.

[45] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 1310–1321.

[46] M. Abadi *et al.*, "Deep learning with differential privacy," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 308–318.

[47] N. Phan, Y. Wang, X. Wu, and D. Dou, "Differential privacy preservation for deep auto-encoders: an application of human behavior prediction," in *Proc. 30th AAAI Conf. Artif. Intell.*, Feb. 2016, pp. 1309–1316.

[48] Q. Zhang, C. Wang, H. Wu, C. Xin, and T. V. Phuong, "Gelu-net: A globally encrypted, locally unencrypted deep neural network for privacy-preserved learning," in *Proc. Int. Joint Conf. Artif. Intell.*, 2018, pp. 3933–3939.

[49] S. Li *et al.*, "Falcon: A fourier transform based approach for fast and secure convolutional neural network predictions," 2018, *arXiv:1811.08257*.

[50] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "Gazelle: A low latency framework for secure neural network inference," in *Proc. 27th USENIX Secur. Symp.*, *USENIX Assoc.*, Aug. 2018, pp. 1651–1668.

[51] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A hybrid secure computation framework for machine learning applications," in *Proc. Asia Conf. Comput. Commun. Secur.*, 2018, pp. 707–721.

[52] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Helen: Maliciously secure coopetitive learning for linear models," 2019, *arXiv:1907.07212*.

[53] R. Xu, J. B. Joshi, and C. Li, "Cryptonn: Training neural networks over encrypted data," 2019, *arXiv:1904.07303*.

[54] Y. LeCun *et al.*, "Learning algorithms for classification: A comparison on handwritten digit recognition," *Neural Netw.: The Statist. Mechanics Perspective*, vol. 261, 1995, Art. no. 276.

[55] O. Russakovsky *et al.*, "Imagenet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.

[56] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*. 2014, pp. 675–678.

[57] H. Takabi, J. B. Joshi, and G.-J. Ahn, "Security and privacy challenges in cloud computing environments," *IEEE Secur. Privacy*, vol. 8, no. 6, pp. 24–31, Nov.–Dec. 2010.

[58] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Proc. Int. Conf. Theory Appl. Cryptology Inf. Secur.* Berlin, Germany: Springer, 2017, pp. 409–437.
[59] "Microsoft seal (release 3.2)," Microsoft Research, Redmond, WA, Feb. 2019. [Online]. Available: https://github.com/Microsoft/SEAL.
[60] B. Wang and N. Z. Gong, "Stealing hyperparameters in machine learning," 2018, *arXiv:1802.05351*.
[61] Y. Liu *et al.*, "Trojaning attack on neural networks," in *Proc. Conf.: Netw. Distrib. Syst. Secur. Symp.,* Jan. 2018.
[62] W. Hu and Y. Tan, "Generating adversarial malware examples for black-box attacks based on gan," 2017, *arXiv:1702.05983*.
[63] M. Nasr, R. Shokri, and A. Houmansadr, "Comprehensive privacy analysis of deep learning: Stand-alone and federated learning under passive and active white-box inference attacks," 2018, *arXiv:1812.00910*.
[64] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," 2016, *arXiv:1607.02533*.
[65] N. Narodytska and S. Kasiviswanathan, "Simple black-box adversarial attacks on deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Patt. Recognit. Workshops, CVPRW*, 2017, pp. 1310–1318.
[66] S. Gulshad, J. H. Metzen, A. Smeulders, and Z. Akata, "Interpreting adversarial examples with attributes," 2019, *arXiv:1904.08279*.
[67] A. Athalye, L. Engstrom, A. Ilyas, and K. Kwok, "Synthesizing robust adversarial examples," 2017, *arXiv:1707.07397*.
[68] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial machine learning at scale," 2016, *arXiv:1611.01236*.
[69] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979.
[70] P. Vepakomma, J. Balla, and R. Raskar, "Splintering with distributions: A stochastic decoy scheme for private computation," 2020, *arXiv:2007.02719*.
[71] S. Halevi and V. Shoup, "Algorithms in helib," in *Int. Cryptol. Conf.*. Berlin, Germany: Springer, 2014, pp. 554–571.
[72] O. Goldreich, *Foundations of Cryptography: Volume 1, Basic Tools, Chapter 4*. Berlin, Germany: Cambridge University Press, 2007.
[73] S. C. Kothari, "Generalized linear threshold scheme," in *Proc. Workshop Theory Appl. Cryptogr. Techn.*, Berlin, Germany: Springer, 1984, pp. 231–241.

Qiao Zhang received the B.S. and M.S. degrees from the School of Communication and Information Engineering, Chongqing University of Posts and Telecommunications, Chongqing, China in 2014 and 2017, respectively. She is currently the Ph.D. candidate with the Department of Electrical and Computer Engineering, Old Dominion University (ODU), Norfolk, VA, USA. Her current research focuses on privacy preserving machine learning.

Chunsheng Xin (Senior Member, IEEE) received the Ph.D. degree in computer science and engineering from the State University of New York, Buffalo in 2002. He is a Professor with the Center for Cybersecurity Education and Research, and the Department of Electrical and Computer Engineering, Old Dominion University. His research interests include cybersecurity, machine learning, wireless communications and networking, cyber-physical systems, and Internet of Things. His research has been supported by almost 20 NSF and other federal grants, and results in more than 100 papers in leading journals and conferences, including three best paper awards, as well as books, book chapters, and patent. He was Co-Editor-in-Chief/Associate Editors of multiple international journals, and symposium/track chairs of multiple international conferences including IEEE Globecom and ICCCN. .

Hongyi Wu (Fellow, IEEE) received the B.S. degree in scientific instruments from Zhejiang University, Hangzhou, China, in 1996, and the M.S. degree in electrical engineering and the Ph.D. degree in computer science from the State University of New York, Buffalo in 2000 and 2002, respectively. He is the Batten Chair of Cybersecurity and the Director with the Center for Cybersecurity Education and Research, Old Dominion University (ODU). He is also a Professor with the Department of Electrical and Computer Engineering and holds joint appointment in Department of Computer Science. Before joining ODU, he was an Alfred and Helen Lamson Endowed Professor with the Center for Advanced Computer Studies (CACS), University of Louisiana at Lafayette (UL Lafayette). His research interests include networked cyber-physical systems for security, safety, and emergency management applications, where the devices are often light-weight, with extremely limited computing power, storage space, communication bandwidth, and battery supply. He was the recipient of the NSF CAREER Award in 2004 and UL Lafayette Distinguished Professor Award in 2011.