Secure Softmax/Sigmoid for Machine-Learning Computation

Yu Zheng[*‡] Chinese U. of Hong Kong

Qizhi Zhang[*] Ant Group & ByteDance

Sherman S. M. Chow[†] Chinese U. of Hong Kong

Yuxiang Peng Northeastern U., China

Sijun Tan[‡] UC Berkeley

Lichun Li Morse Team, Ant Group

Shan Yin Morse Team, Ant Group

ABSTRACT

Softmax and sigmoid, composing exponential functions (e^x) and division ($1/x$), are activation functions often required in training. Secure computation on non-linear, unbounded $1/x$ and e^x is already challenging, let alone their composition. Prior works aim to compute softmax by its exact formula via iteration (CrypTen, NeurIPS '21) or with ASM approximation (Falcon, PoPETS '21). They fall short in efficiency and/or accuracy. For sigmoid, existing solutions such as ABY2.0 (Usenix Security '21) compute it via piecewise functions, incurring logarithmic communication rounds.

We study a rarely-explored approach to secure computation using ordinary differential equations and Fourier series for numerical approximation of rational/trigonometric polynomials over composition rings. Our results include 1) the first constant-round protocol for softmax and 2) the first 1-round error-bounded protocol for approximating sigmoid. They reduce communication by ~83% and ~95%, respectively, shortening the private training process of state-of-the-art frameworks or platforms, namely, CryptGPU (S&P '21), Piranha (Usenix Security '22), and quantized training from MP-SPDZ (ICML '22), while maintaining competitive accuracy.

CCS CONCEPTS

• **Security and privacy** → **Cryptography**; **Privacy-preserving protocols**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

Secure Computation, Machine Learning, Crypto, Softmax, Sigmoid

ACM Reference Format:

Yu Zheng, Qizhi Zhang, Sherman S. M. Chow, Yuxiang Peng, Sijun Tan, Lichun Li, and Shan Yin. 2023. Secure Softmax/Sigmoid for Machine-Learning Computation. In *Annual Computer Security Applications Conference (ACSAC '23), December 4–8, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3627106.3627175

[*]Equal contribution.

[†]Corresponding author is with Dept. of Information Engineering, CUHK, Hong Kong.

[‡]Part of the work was done when Yu and Sijun were at Morse Team, Ant Group.

ACSAC '23, December 4–8, 2023, Austin, TX, USA

ACM ISBN 979-8-4007-0886-2/23/12…$15.00
https://doi.org/10.1145/3627106.3627175

1 INTRODUCTION

Advances in machine learning techniques, notably neural networks, attain great performance by training on vast datasets. Privacy-preserving machine learning has garnered immense attention, especially in sensitive domains, *e.g.*, finance and healthcare. While most frameworks support simpler inference tasks [11, 20, 23, 28, 29], private training poses unique challenges [14, 22, 32, 35]. It produces fluctuating computation results and crucially relies on non-linear layers. In general, cryptographic techniques, such as secure multi-party computation, excel primarily with finite fields and linear functions. Expanding the finite field to cater to the fluctuating ranges increases the computational and communication overheads (for processing larger field elements) while confining it incurs accuracy issues. Earlier works were limited to polynomial approximations or simpler functions such as rectified linear units (ReLU, computing comparison). Secure protocols for the exact computation of non-linear functions have been known to be heavyweight.

Not until recently did we start to have frameworks such as CryptGPU [32] and CrypTen [15] that support more complex activation functions, including softmax and sigmoid. When deployed with GPU, they show good computational performance in training large networks such as AlexNet (60M parameters) [17] and VGG-16 (138M parameters) [31]. However, large communication overhead persists as a major concern. Prominently, in a wide-area network (WAN) setting, over 94% of the training time of Piranha [35], a GPU platform for secure computation, is consumed by communication [35].

$\mathsf{Softmax}(x) = e^{x_i}/\sum_j e^{x_j}$ and $\mathsf{Sigmoid}(x) = e^x/(e^x+1)$ involve unbounded continuous functions e^x and $1/x$ for private x. Securely computing them while retaining efficiency is very challenging. Using garbled circuits leads to large circuit sizes, while oblivious transfer [26–29] or function secret sharing [11, 30] requires extensive communication. Existing works [15, 26, 33, 34] either approximate e^x, $1/x$, or $\sum e^x$ separately or replace them with polynomial approximations. A recent SoK [24] explicitly highlights the challenge of efficiently and accurately approximating secure computation of $\mathsf{Softmax}(x)$ and $\mathsf{Sigmoid}(x)$.

1.1 Numerical Approximation and Protocols

We propose secure protocols for softmax and sigmoid with more effective approximations by a holistic approach integrating scientific computing, machine learning, and cryptographic techniques. We venture into a rarely-explored approach in numerical approximations, utilizing ordinary differential equations and Fourier series.

1.1.1 ODE-Iterative Softmax. Exactly computing softmax [15] involves various secure protocols for computing maximum (*i.e.*, comparison), exponentiation, and division. When the inputs get large, e^x and $\sum e^x$ easily overflow. One could remove the largest input

Table 1: Communication Comparison of Secure Computation of Softmax and Sigmoid

Frameworks	Softmax		Sigmoid		Party
	Communication	Round	Communication	Round	
SecureNN [33]	$8mn(n+1)\log p + 24mn(n+1)$	$11n$	n/a	n/a	3
ABY2.0 [26]	n/a	n/a	$(2x_1+15)\lambda + 2x_1 + 4x_2 + 18n + 2$	$\log_4 n + 2$	2
CrypTen [15]	$275(m-1)n\log n + 39mn + 2m + 106n - 2$	$39\log m + 41$	$96n$	16	2+
Ours	$(8mn + 2n)r$	$2r$	$<12n$	1	2+

The security parameter is $\lambda = 128$. r is a preset value representing the number of iterations. m is the number of classes. n is the bit-length of fixed-point numbers, usually $n = 64$. x_1, x_2 are parameters linear to the number of AND gates. p is the minimal prime greater than n.

by securely computing $\max(x_0, \ldots, x_{m-1})$, but it takes $O(\log m)$ rounds. Approximated softmax (ASM) [6, 33, 34] replaces the exponential function with ReLU, which crucially relies on manual efforts in tuning the model towards a satisfying accuracy [14].

We formulate *"quasi-softmax"* (QSMax), which captures the essential characteristics regarding the probability distribution of softmax's outputs. QSMax computes softmax with a high degree of accuracy by using the Euler formula to solve the initial value problem in ordinary differential equations (ODE). Our secure protocol Π_{QSMax} is tailored to avoid costly secure comparisons and divisions. It is constant-round and superior to solutions [33, 34] that directly replace softmax with ASM. Π_{QSMax} also avoids guessing initials as needed by CryptGPU [32], which uses the Newton-Raphson algorithm to approximate $1/y$ and may affect the accuracy.

1.1.2 Fourier-Series-Approximated Sigmoid. Earlier secret-sharing frameworks resort to piecewise linear approximation [21, 26], which involves relatively heavy comparison to identify pieces. Approximating by the Chebyshev polynomial [5, 6], on the other hand, may result in a gradient explosion that risks destroying the model.

We define *"local-sigmoid"* (LSig), emphasizing accuracy for inputs near 0. It approximates sigmoid using the Fourier series (FS). With the help of trigonometric identities, our secure protocol Π_{LSig} only takes local computations of $\sin x$, $\cos x$, $\sin y$, and $\cos y$ for secrets x and y and secret-shared multiplication. It incurs a 1-round constant online communication cost, independent of the number of FS terms. Our approximation is more accurate than piecewise linear function [11, 22, 26]. It also avoids gradient explosion caused by unbounded errors of the polynomial fitting (faced by tf-encrypted [6] and Rosetta [5]). Broadly, we extend secret-sharing-based protocols to composition rings, approximating bounded functions derived from trigonometric polynomials.

Both Π_{QSMax} and Π_{LSig} avoid computing intermediate values[1] resulting from the exact computation of unbounded e^x and $1/x$ but directly approximate bounded softmax/sigmoid. We thus largely reduced the communication complexity, as shown in Table 1.

1.2 Experimental Results

1.2.1 Protocol-level Improvement. We report protocol-level improvement in communication in Section 7.2 and running time in Section 7.3. Π_{QSMax} for softmax reduces the communication by 83%, 85%, and 85% for 10-, 100-, and 1000-classification, respectively. Moreover, it achieves a 10× speed-up in running time. Previous attempts require 170-704 rounds of online communication, whereas Π_{QSMax} is constant-round (16 or 32 in experiments). For sigmoid, Π_{LSig} reduces online communication from $\gg$100 bits to 36 bits and speeds up computation by 570×. Currently, Π_{LSig} is the only lightweight solution (free from oblivious transfer or functional secret sharing) that is 1-round and with bounded error.

1.2.2 Accuracy and Efficiency for Model Training. We evaluate our approach on various neural network architectures, including AlexNet [17], LeNet [18], VGG-16 [31], ResNet [12], and four relatively small networks [33] over multiple standard benchmark datasets. Experimental results in Section 7.5 show a 57%-77% reduction in communication compared to the state of the art [32, 34, 35]. Regarding accuracy, we consider in Section 7.4 the training-from-sketch setting of Piranha [35] from randomized initialization without any pretraining, and the pretrained setting in CryptGPU [32] that trains with a pretrained model. We attain higher accuracy than Piranha, *e.g.*, 52.1% vs. 40.7% for AlexNet and 59.4% vs. 58.7% for VGG-16. Meanwhile, our model can reach an almost identical or slightly higher accuracy for relatively small networks compared to quantized training using MP-SPDZ (hereinafter referred to as "SPDZ-QT") of Keller and Sun [14] with fewer training epochs. Moreover, faster convergence means less communication and rounds, roughly reducing $10/15 \approx 67\%$ communication.

1.2.3 Evaluation in LAN and WAN. We simulate real-world data transmission (for online and offline precomputation phases) under various settings. In a local-area-network (LAN) setting, our protocols surpass Piranha with a 10%-60% speed-up, as elaborated in Section 7.6. Section 7.7 shows training under WAN with limited bandwidth and different time latency. For LeNet and AlexNet, our protocols train 56%-78% faster than Piranha under WAN.

1.2.4 Python & C++ Implementations. We implemented our protocols in Python (*e.g.*, [15, 32]) and C++ (*e.g.*, [33, 35])[2], as detailed in Section 6.3. To our knowledge, our Python version is the first TensorFlow-style framework for cryptographic training.

2 RELATED WORKS

2.1 Cryptographic Machine Learning

A recent SoK [24] dissects the interaction of cryptographic techniques with machine-learning applications alongside a genealogy highlighting contributions of works from various sub-fields.[3] There

[1] Privacy of intermediate results should remain protected by cryptography. Revealing intermediate results, even in the context of inference, is known to be risky [36].

[2] SecureML [22] and Piranha [35] are implemented in C++. CrypTen [15] is built on PyTorch without optimizing protocols. CryptGPU [32] is implemented in PyTorch too.

[3] Differential privacy (*e.g.*, [9, 37]) is an orthogonal approach that could be integrated [40] for mitigating advanced attacks.

is still a long run for private training to achieve training performance similar to training in plaintext. SecureML [22], one of the earliest works, attains 93% accuracy for online training on MNIST. Four years later, CryptGPU [32] integrates more ML techniques, *e.g.*, softmax and batch normalization. With the help of plaintext pretraining, it attains 83.7% accuracy on CIFAR-10 with VGG-16. Piranha [35] provides a GPU platform for accelerating known secret-sharing-based protocols. It demonstrates a 58.7% accuracy over CIFAR-10 with VGG-16 by training from scratch (from a randomly initialized model without pretraining). Nevertheless, its huge communication overhead poses a bottleneck. For non-linear computation, the large computational complexity easily affects the training time and communication overhead.

2.2 Softmax

CrypTen [15] computes softmax exactly, with great effort to evaluate maximum, exponentiation, and division securely. For an m-dimensional vector of n-bit numbers, it takes $55(m-1)n\log n + 13mn + 8n$ bits in the offline precomputation phase and $220(m-1)n\log n + 26mn + 2m + 98n - 2$ bits online. Exponential computation in the softmax function may result in overflow when the input gets very large or very small. To avoid numeric imprecision, CryptGPU [32] subtracts the maximum entry $x_{\max}$ of input vector $\vec{x}$, and later, computes $\mathsf{Softmax}(\vec{x})_i = e^{(x_i - x_{\max})}/(\sum_i e^{(x_i - x_{\max})})$. We call this standard "normalization" technique *"ComDiExp"* for combining division and exponentiation. Its major issue is the increased communication with the number of training classes. SecureNN [33] and Falcon [34] replace the exponential function in softmax with the ReLU function, called ASM, which requires linear (in n) communication rounds. Keller and Sun [14] question the efficiency gains of ASM and optimize exponentiation with base two. Unlike the above frameworks, our protocol for softmax aims for constant rounds and low communication costs.

2.3 Sigmoid

SecureML [22], ABY^3 [21], ABY2.0 [26], and Squirrel [19] approximate sigmoid by piecewise function, with bounded error and thus without experiencing exploding gradients. However, it costs $O(\log n)$ comparison rounds for locating pieces. Chebyshev polynomial [5, 6] could provide a satisfying approximation in the $[-4, 4]$ range but may risk gradient explosion (and even destroy the training model) due to its unbounded error for very large or small inputs. CrypTen [15] uses Newton-Raphson iterations. With statistical masking, Boura *et al.* [2] require $2(3n + 40)$ bits in at most 3-round communication. All these attempts require >1 round of communication or are challenged by unbounded error, whereas we are looking for an accurate 1-round protocol for sigmoid.

3 SECURITY MODEL AND SECRET SHARING

3.1 Security Model in the Commodity Setting

Our system models include two computing parties, P_0 and P_1, who perform secure computation assisted by T, a crypto commodity server. T only generates the required randomness in the offline phase, letting P_0 and P_1 perform relatively cheap online computation. We consider semi-honest probabilistic polynomial time (PPT) adversaries who follow the protocol execution but attempt to extract more information about the data they receive and process. T does not participate in the online phase and receives nothing about the private inputs executed by the two parties.

3.2 Secret-Shared Multiplication

Let $\mathsf{FR}(z)$ be the fixed-point representation of an arbitrary $z \in \mathbb{R}$ by n-bit decimals with a p-bit fractional part. $\langle x\rangle_0$ and $\langle x\rangle_1$ denote the shares of fixed-pointed representation x owned by P_0 and P_1, respectively. We denote the secret-recovery algorithm by $\mathsf{Rec}(\langle x\rangle_0, \langle x\rangle_1) := \langle x\rangle_0 + \langle x\rangle_1 \bmod 2^n$, which returns $x \cdot 2^p$.

For secret-shared multiplication, we use a standard protocol (*e.g.*, [25]) denoted by $\Pi_\times$, which generates shares using the pseudorandom function PRF. In its offline phase, T and P_0 generate randomness $\langle u\rangle_0, \langle v\rangle_0, \langle z\rangle_0$ using the same key with a synchronized monotonically-increasing counter, denoted by the abused notation $\mathsf{PRF}_0(\mathsf{key}_0)$. Likewise, T and P_1 generate randomness $\langle u\rangle_1, \langle v\rangle_1$ via $\mathsf{PRF}_1(\mathsf{key}_1)$. T then sends $\langle z\rangle_1 = uv - \langle z\rangle_0$ to P_1. In the online phase, P_0 computes $\delta_{\langle x\rangle_0} = \langle x\rangle_0 - \langle u\rangle_0$ and $\delta_{\langle y\rangle_0} = \langle y\rangle_0 - \langle v\rangle_0$, and sends $\delta_{\langle x\rangle_0}, \delta_{\langle y\rangle_0}$ to P_1. Similarly, P_1 computes $\delta_{\langle x\rangle_1}, \delta_{\langle y\rangle_1}$ and sends them to P_0. Both P_0 and P_1 can then construct δ_x and δ_y and collaboratively compute $xy = \delta_x(\langle y\rangle_0 + \langle y\rangle_1) + (\langle u\rangle_0 + \langle u\rangle_1)\delta_y + (\langle z\rangle_0 + \langle z\rangle_1)$. To mitigate overflow, we perform truncation after every multiplication to maintain the p-bit fractional part of the result.

4 SECURE SOFTMAX COMPUTATION

Secret-shared softmax computation is challenging since e^x and $1/x$ are easy to overflow and costly to compute securely. Our goal is to find a softmax formulation that does not directly compute e^x or $1/x$. Usually, $\mathsf{Softmax}(\vec{x})$ in the output layer is commonly used for the multi-classification model whose outputs look like a probability distribution. Our quasi-softmax formulation captures this probability distribution outputted by the softmax function.

We establish an m-dimensional ODE with softmax-like outputs by transforming a multi-variable function into a single-variable function. Atop it, the problem of approximating softmax is formulated as an initial value problem (IVP) whose solution evaluated at the terminal point gives an estimate. To solve the IVP of ODE, we employ the Euler formula to output the softmax-like estimates iteratively. In each iteration, the online computation involves simple multiplication only. Each iteration's output gradually approaches the distribution provided by the standard softmax.

4.1 New Notion of Quasi-Softmax

Given an input vector $\vec{x} = [x_1, \ldots, x_m]$, $\mathsf{Softmax}(\vec{x})$ transforms it into a score vector filled with normalized probabilities via $\mathsf{Softmax}(\vec{x})_i = e^{x_i}/\sum_j e^{x_j} : \mathbb{R}^m \to (0,1)^m$. We observe that the probability distribution over the outputs dominates the model accuracy instead of the absolute values. Accordingly, quasi-softmax aims to closely approximate the behavior of the standard softmax in terms of modeling the probability distribution.

Definition 4.1 (QSMax). $f : \mathbb{R}^m \to [0,1]^m$ is quasi-softmax s.t.:

(i) $f(\vec{x})$ is a probability distribution on $[0,1]^m$,
i.e., $\sum_{i=1}^m [f(\vec{x})]_i = 1$ and $[f(\vec{x})]_i \geq 0$ for any $i = 1, \ldots, m$.
(ii) $[f(\vec{x})]_i \leq [f(\vec{x})]_j$ iff $x_i < x_j$ for any $i, j = 1, \ldots, m$.

Protocol 1 Π_{QSMax}: Quasi-Softmax via ODE

P_0 Input	$\langle\vec{x}\rangle_0, r$	P_0 Output	$\langle\mathsf{QSMax}(\vec{x})\rangle_0$
P_1 Input	$\langle\vec{x}\rangle_1, r$	P_1 Output	$\langle\mathsf{QSMax}(\vec{x})\rangle_1$

1: P_0: Set $\langle\vec{x}\rangle_0 = \mathsf{FR}(\frac{1}{r}) \cdot \langle\vec{x}\rangle_0$, $\langle\vec{\mathsf{g}}(0)\rangle_0 = \vec{1}/m$ {▷ Local Processing
2: P_1: Set $\langle\vec{x}\rangle_1 = \mathsf{FR}(\frac{1}{r}) \cdot \langle\vec{x}\rangle_1$, $\langle\vec{\mathsf{g}}(0)\rangle_1 = \vec{0}$ of Initial Value}
3: **for** $i = 1, 2, \ldots, r$ **do** {▷ Iteration for $\vec{f}(\frac{i}{r})$ via Euler Formula}
4: $\mathsf{P}_0, \mathsf{P}_1$: $\langle\vec{o}\rangle_0, \langle\vec{o}\rangle_1 = \Pi_\times(\langle\vec{x}\rangle_0, \langle\vec{\mathsf{g}}(\frac{i-1}{r})\rangle_0; \langle\vec{x}\rangle_1, \langle\vec{\mathsf{g}}(\frac{i-1}{r})\rangle_1)$
5: P_0: Compute $\langle\vec{q}\rangle_0 = (\sum_{i=1}^{m} \langle[\vec{o}]_i\rangle_0) \cdot \vec{1}$
6: P_1: Compute $\langle\vec{q}\rangle_1 = (\sum_{i=1}^{m} \langle[\vec{o}]_i\rangle_1) \cdot \vec{1}$
7: $\mathsf{P}_0, \mathsf{P}_1$: $\langle\vec{t}\rangle_0, \langle\vec{t}\rangle_1 = \Pi_\times(\langle\vec{q}\rangle_0, \langle\vec{\mathsf{g}}(\frac{i-1}{r})\rangle_0; \langle\vec{q}\rangle_1, \langle\vec{\mathsf{g}}(\frac{i-1}{r})\rangle_1)$
8: P_0: Compute $\langle\vec{\delta}_{ot}\rangle_0 = \langle\vec{o}\rangle_0 - \langle\vec{t}\rangle_0$
9: P_1: Compute $\langle\vec{\delta}_{ot}\rangle_1 = \langle\vec{o}\rangle_1 - \langle\vec{t}\rangle_1$
10: P_0: Compute $\langle\vec{\mathsf{g}}(\frac{i}{r})\rangle_0 = \langle\vec{\mathsf{g}}(\frac{i-1}{r})\rangle_0 + \langle\vec{\delta}_{ot}\rangle_0$
11: P_1: Compute $\langle\vec{\mathsf{g}}(\frac{i}{r})\rangle_1 = \langle\vec{\mathsf{g}}(\frac{i-1}{r})\rangle_1 + \langle\vec{\delta}_{ot}\rangle_1$
12: **end for**
13: P_0: $\langle\mathsf{QSMax}(\vec{x})\rangle_0 = \langle\vec{\mathsf{g}}(1)\rangle_0$
14: P_1: $\langle\mathsf{QSMax}(\vec{x})\rangle_1 = \langle\vec{\mathsf{g}}(1)\rangle_1$

We use $[f(\vec{x})]_i$ to denote the i-th entry of the output of $f(\vec{x})$, where $f(\vec{x})$ outputs a vector. The output domain is $[0,1]^m$, which introduces the boundary $\{0,1\}$ to the original domain $(0,1)^m$ of the standard softmax. In Definition 4.1, property (i) captures the range defined over the output domain. Specifically, the value of $e^{x_i}/\sum_j e^{x_j}$ is always non-negative, and the value of $\sum_i e^{x_i}/\sum_j e^{x_j}$ always equals 1. Property (ii) captures the input-output monotonous relation. Informally, a larger input results in a larger output. Prior works resort to $\mathsf{ASM}(\vec{x})_i = \mathsf{ReLU}(x_i)/\sum_j \mathsf{ReLU}(x_j)$, which has a similar form and is essentially an instantiation of QSMax. Properties (i) and (ii) can be verified similarly.

An ideal instantiation of $f(\vec{x})$ should be reasonably represented by fixed-point numbers. We first transform the multi-variable function $f(\vec{x})$ into a single-variable function $\vec{\mathsf{g}}(\frac{i}{r})$, which also outputs an m-dimensional vector $f(\vec{x})$, where i is the independent variable for $\vec{\mathsf{g}}(\frac{i}{r})$ and r is a hyperparameter for the number of for-loop iterations. For QSMax, below describes the code of $\vec{\mathsf{g}}(\cdot)$.

1: $\vec{\mathsf{g}}(0) = [1/m, \ldots, 1/m]$
2: **for** $i = 1, 2, \ldots, r$ **do**
3: $\vec{\mathsf{g}}(\frac{i}{r}) = \vec{\mathsf{g}}(\frac{i-1}{r}) + (\vec{x} - \langle\vec{x}, \vec{\mathsf{g}}(\frac{i-1}{r})\rangle\vec{1}) * \vec{\mathsf{g}}(\frac{i-1}{r}) \cdot \frac{1}{r}$
4: **end for**

Section 4.4.1 will delve into the intuition of replacing $\mathsf{Softmax}(\vec{x})$ with $\vec{\mathsf{g}}(\cdot)$. At the start, we fix $\vec{\mathsf{g}}(0) = [1/m, \ldots, 1/m]$. Given an input $\vec{x}$, we can get $\vec{\mathsf{g}}(\frac{1}{r}) = \vec{\mathsf{g}}(0) + (\vec{x} - \langle\vec{x}, \vec{\mathsf{g}}(0)\rangle\vec{1}) * \vec{\mathsf{g}}(0) \cdot \frac{1}{r}$. Here, $\langle\vec{x}, \vec{\mathsf{g}}(0)\rangle$ denotes the inner product of $\vec{x}$ and $\vec{\mathsf{g}}(0)$, while $*$ represents the element-wise multiplication over vectors. Given $\vec{\mathsf{g}}(\frac{1}{r})$, we then can get $\vec{\mathsf{g}}(\frac{2}{r}), \vec{\mathsf{g}}(\frac{3}{r})$, and so on. After $i = r$ iteration steps, we have $\vec{\mathsf{g}}(1) = \vec{\mathsf{g}}(\frac{r}{r})$, which replaces the output of the standard softmax.

4.2 Protocol for Quasi-Softmax

With the pseudocode for computing $\vec{\mathsf{g}}(\frac{i}{r})$, Protocol 1 presents our proposed Π_{QSMax}, in which parties P_0 and P_1 iteratively construct shares of $\vec{\mathsf{g}}(\frac{1}{r}), \vec{\mathsf{g}}(\frac{2}{r}), \ldots, \vec{\mathsf{g}}(1)$. After r iterations, Π_{QSMax} outputs shares of $\vec{\mathsf{g}}(1)$, representing outputs of the standard softmax.

P_0 and P_1 locally process their initial values in Lines 1-2. Both multiply the fixed-point representation $\mathsf{FR}(\frac{1}{r})$ to their respective vector-valued shares $\langle\vec{x}\rangle_0, \langle\vec{x}\rangle_1$. Processing $\langle\vec{x}\rangle_0, \langle\vec{x}\rangle_1$ corresponds to the rightmost side of computing $\vec{\mathsf{g}}(\frac{i}{r})$, *i.e.*, $\cdot\frac{1}{r}$, which is outside the loop for saving computation. The initial value $\vec{\mathsf{g}}(0) = \vec{1}/m$ is fixed for any inputs $\vec{x}, r$. $\mathsf{P}_0, \mathsf{P}_1$ can thus set $\langle\vec{\mathsf{g}}(0)\rangle_0 = \vec{1}/m, \langle\vec{\mathsf{g}}(0)\rangle_1 = \vec{0}$.

Lines 4-6 perform secure inner product $\langle\vec{x}, \vec{\mathsf{g}}(\frac{i-1}{r})\rangle \cdot \vec{1}$, where $\cdot\vec{1}$ is a vectorized transformation. P_0 and P_1 execute $\Pi_\times$ (in Section 3.2) for element-wise multiplication between two vectors. Then, P_0 sums all entries of $\langle\vec{o}\rangle_0$ and multiplies the result by $\vec{1}$ to transform the scalar into a vector. Similarly, P_1 constructs the vector $\langle\vec{q}\rangle_1$.

In Line 7, P_0 and P_1 obtain the shares of $\langle\vec{x}, \vec{\mathsf{g}}(\frac{i-1}{r})\rangle\vec{1}) * \vec{\mathsf{g}}(\frac{i-1}{r})$. Combining with Lines 8 and 9, P_0 and P_1 securely compute $(\vec{x} - \langle\vec{x}, \vec{\mathsf{g}}(\frac{i-1}{r})\rangle\vec{1}) * \vec{\mathsf{g}}(\frac{i-1}{r})$. Note that multiplying $\frac{1}{r}$ is done before the loop. In Lines 10-11, P_0 and P_1 locally compute the shares of $\vec{\mathsf{g}}(\frac{i}{r})$ at the i-th iteration. At last, P_0 and P_1 obtain the shares of $\vec{\mathsf{g}}(1)$. Theorem 4.2 asserts the semi-honest security of Π_{QSMax} with its proof and the functionality $\mathcal{F}_{\mathsf{QSMax}}(\vec{x})$ deferred to Appendix C.1.

Theorem 4.2. *Protocol* Π_{QSMax} *realizes functionality* $\mathcal{F}_{\mathsf{QSMax}}(\vec{x})$ *against semi-honest PPT adversaries with static corruption.*

4.3 Communication Complexity

Recall that in $\Pi_\times$, P_0 sends masked values $\delta_{\langle x\rangle_0}, \delta_{\langle y\rangle_0}$ and P_1 sends $\delta_{\langle x\rangle_1}, \delta_{\langle y\rangle_1}$ for one multiplication. In Π_{QSMax}, online communication comes from calling $\Pi_\times$ twice in each iteration. Thus, the number of communication rounds is $2r$ for r iterations. Notably, it remains constant regardless of the input dimension m.

For the first call of $\Pi_\times$, P_0 and P_1 send two masked values for $\langle\vec{x}\rangle_0, \langle\vec{\mathsf{g}}(\frac{(i-1)}{r})\rangle_0$ and $\langle\vec{x}\rangle_1, \langle\vec{\mathsf{g}}(\frac{i-1}{r})\rangle_1$, respectively. The communication costs of m-dimensional element-wise multiplication are $4mn$ bits for n-bit data. For the second call of $\Pi_\times$, masking $\langle\vec{\mathsf{g}}(\frac{(i-1)}{r})\rangle_0, \langle\vec{\mathsf{g}}(\frac{(i-1)}{r})\rangle_1$ requires $2mn$ bits online, whereas masking $\langle q\rangle_0, \langle q\rangle_1$ requires $2n$ bits online. Each entry in $\vec{q}$ is filled with the same scalar; thus, two parties can only send the masked value of one entry. The online communication costs for r iterations require $(6mn + 2n)r$ bits. In the offline phase, each iteration takes $2mn$ bits by running $\Pi_\times$ twice. For r iterations, the overall offline communication is $2mnr$ bits. Theorem 4.3 summarizes the communication.

Theorem 4.3. *Let x be an n-bit fixed-point number, m be the dimension of input vectors, and r be the number of iterations.* Π_{QSMax} *requires $6mnr + 2nr$ bits online and $2mnr$ bits offline in $2r$ rounds.*

4.4 Mathematical Intuition for Quasi-Softmax

4.4.1 Softmax Solved by Ordinary Differential Equation. We first define a vector-valued function $\vec{f}(t) = \mathsf{Softmax}(t\vec{x})$ for transforming the multi-variable softmax into a single-variable function $\vec{f}(t)$. Specifically, upon the input of an m-dimensional vector $\vec{x}$, $\vec{f}$ outputs element-wise an m-dimensional vector. When $t = 1$, we can get the standard softmax $\vec{f}(1) = \mathsf{QSMax}(\vec{x})$.

We observe that $\vec{f}(t) = \mathsf{QSMax}(t\vec{x})$ satisfies an ODE [3, 10], *i.e.*, a solution of an m-dimensional ODE. Theorem 4.4 presents the ODE form of representation.

Theorem 4.4. *Define* $\vec{f}(t) = \mathsf{Softmax}(t\vec{x}) : [0,1] \to (0,1)^m$. *We have* $\vec{f}(0) = \vec{1}/m$ *and* $\vec{f}'(t) = (\vec{x} - \langle\vec{x}, \vec{f}(t)\rangle\vec{1}) * \vec{f}(t)$.

Table 2: Kullback-Leibler Divergence

Protocol	Classes			
	10	**100**	**1000**	**10000**
ASM	6.3×10^{-2}	6.8×10^{-2}	7.0×10^{-2}	7.0×10^{-2}
ComDiExp	8.6×10^{-6}	2.3×10^{-5}	n/a	n/a
QSMax	3.0×10^{-4}	1.0×10^{-3}	1.5×10^{-3}	6.5×10^{-3}

$\vec{f}(t)$ builds on the IVP for the ODE. We can use the Euler formula for r iterations to solve ODE. Actually, $\vec{g}(\frac{i}{r})$ is an iterative solution of $\vec{f}(t)$. We can see that the computation of $\vec{g}(\frac{i}{r})$ only needs online vector multiplication and offline/local addition.

Theorem 4.5 shows the correctness of solving softmax. Its proof is given in Appendix B.2. Its properties (i) and (ii) indicate that $\vec{g}(\frac{i}{r})$ belongs to quasi-softmax. Property (iii) means $\vec{g}(\frac{i}{r})$ is an approximation of $\vec{f}(t)$, thus $\vec{g}(\frac{r}{r})$ is an approximation of $\text{Softmax}(\vec{x})$. The relation $\max(\vec{x}) - \min(\vec{x}) \leq r$ serves as the reference for choosing hyperparameter r in practice.

Theorem 4.5 (Correctness). *Given $\vec{g}(\frac{i}{r})$, if $\max(\vec{x}) - \min(\vec{x}) \leq r$ holds, then we have:*

(i) $\vec{g}(\frac{i}{r}) \in [0,1]$ *is a probability distribution for* $i = 0, \ldots, r$.
(ii) $[\vec{g}(\frac{i}{r})]_j \leq [\vec{g}(\frac{i}{r})]_k$ *iff* $x_j \leq x_k$ *for* $\forall j, k = 1, \ldots, m$.
(iii) $\lim_{r \to +\infty} \vec{g}(\frac{i}{r}) = \vec{f}(1) = \text{Softmax}(\vec{x})$.

4.4.2 Measurement of Probability Distribution. Kullback-Leibler divergence calculates a score that measures the (asymmetric) distance of one probability distribution from another. Table 2 records such divergence between the standard softmax function and different approximations. Inputs are sampled from a Gaussian distribution $\mathcal{N}(0,1)$. ASM does not produce an accurate approximation for softmax and, therefore, has a much higher Kullback-Leibler divergence. Although ComDiExp [32] has lower Kullback-Leibler divergence at 10 and 100 classes (#Class), it produces off-the-chart results when #Class scales to 1000 and 10000. In contrast, QMax could achieve accurate approximation when #Class is very large (say, 10000).

5 SECURE SIGMOID COMPUTATION

Securely computing e^x and $1/x$ in sigmoid is expensive. Piecewise linear approximation avoids computing them. For example, ABY2.0 [26] outputs $x + 0.5$ if $|x| \leq 0.5$; otherwise, 0 for $x < -0.5$, or 1 for $x > 0.5$. Instead, our Protocol 2 approximates it within a finite range $\left[-2^{m-1}, 2^{m-1}\right]$ using the Fourier series, which computes $\sin \frac{2k\pi x}{2^m}$ only for a few k terms. Each term can be securely computed efficiently, without comparison, e^x, or $1/x$.

5.1 Local-Sigmoid

$\text{Sigmoid}(x) = \frac{e^x}{e^x+1} : \mathbb{R} \to (0,1)$ is mostly for binary classification. It squashes any input in the range $(-\infty, +\infty)$ to some value in the range $(0,1)$. If the model parameters are properly initialized[4], an effectively trained model takes the inputs near 0 in most cases, *i.e.*, out-of-range inputs occur with very small probabilities. Given this fact, a natural idea is to make the approximation accurate within the range of high probability. Accordingly, we define local-sigmoid in Definition 5.1 to capture these characteristics.

Definition 5.1 (Local-Sigmoid). Let $0 < \mathrm{a} < +\infty$ and Δ be a small value. $f : [-\mathrm{a}, \mathrm{a}] \to [0,1]$ is local-sigmoid if it satisfies:

(i) $f(x)$ is continuous for any $x \in [-\mathrm{a}, \mathrm{a}]$.
(ii) $\|f(x) - \text{Sigmoid}(x)\|_2 \leq \Delta$, where $\|\cdot\|_2$ is the L_2-norm.

Definition 5.1 defines an interval $[-\mathrm{a}, \mathrm{a}]$, which contains 0 and points near 0. The value of a determines the length of this interval. Property (ii) signifies that $f(x)$ resembles $\text{Sigmoid}(x)$ within the interval $[-\mathrm{a}, \mathrm{a}]$, with Δ quantifying the degree of difference. Property (i) avoids singularity in $f(x)$ to guarantee the similarity between $f(x)$ and $\text{Sigmoid}(x)$.

Although $\text{Sigmoid}(x)$ in plaintext is both continuous and differentiable, we keep "continuous" in the definition only for also taking the piecewise approximation into account. In particular, both piecewise linear approximation (PLA) and polynomial fitting (PT) are local-sigmoid, as shown in Figures 1a and 1b. Typically, Δ for PLA is relatively large, while $[-\mathrm{a}, \mathrm{a}]$ for PT (vs. $[-\infty, \infty]$ for PLA) is smaller than $[-10, 10]$.

Concretely, we expect to find a local-sigmoid function that: 1) has a sufficient interval $[-\mathrm{a}, \mathrm{a}]$ with a small Δ; and 2) is not very far away from $\text{Sigmoid}(x)$ for the out-of-interval inputs. Recall that the Fourier series [4] starts with a small frequency to be increased, which meets the expectation. We employ $(K+1)$-term ($K = 5$ below) Fourier series for approximating sigmoid:

$$\begin{aligned} \text{LSig}(x) = \alpha &+ \beta_1 \sin(2\pi x/2^m) + \beta_2 \sin(4\pi x/2^m) \\ &+ \beta_3 \sin(6\pi x/2^m) + \beta_4 \sin(8\pi x/2^m) + \beta_5 \sin(10\pi x/2^m) \end{aligned} \quad (1)$$

$\text{LSig}(x)$ are used as replacements for $\text{Sigmoid}(x)$ during training.

We set the periodic parameter $m = 5$ since we find empirically that $[-16, 16]$ is large enough. Given that $(\text{Sigmoid}(x) - 0.5)$ is an odd function, we set $\alpha = 0.5$ for the accuracy of the Fourier-series approximation. For an odd function, only the sine terms are required (cosine terms are unnecessary). The Fourier series coefficients $\vec{\beta}$ can be computed by the integrals

$$[\vec{\beta}]_k = \frac{1}{16} \int_{-16}^{+16} (\text{Sigmoid}(x) - 0.5) \sin(\frac{\pi k x}{16}) dx$$

i.e., $[0.61727893, -0.03416704, 0.16933091, -0.04596946, 0.08159136]$.

$\text{LSig}(x) : [-16, 16] \to [-\epsilon, 1+\epsilon]$ is very near[5] to Definition 5.1. Notably, ϵ can be neglected (*i.e.*, $\lim_{K \to \infty} \epsilon \to 0$) since it does not affect the results of binary classification.

5.2 Protocol for Local-Sigmoid

Equation 1 requires no e^x, $1/x$, or comparison. For simplification, we fix $m = 5$ and define a public vector $\vec{k} = [\pi/16, \pi/8, 3\pi/16, \pi/4, 5\pi/16]^\top$. We get $\text{LSig}(x) = \alpha + \vec{\beta} \cdot \sin(x\vec{k})$. For masking private x, we introduce a random value t and set $\delta_x = x - t$.

Protocol 2 presents our proposed Π_{LSig} for computing local-sigmoid via Fourier series. For the secure computation of the sine function, note that $\sin(x\vec{k}) = \sin(\delta_x\vec{k} + t\vec{k})$, which can be computed by using trigonometric identity $\sin(\delta_x\vec{k}) * \cos(t\vec{k}) + \cos(\delta_x\vec{k}) * \sin(t\vec{k})$. Here, $*$ denotes element-wise multiplication over vectors.

[4] Sigmoid's gradient vanishes (≈ 0) when its inputs are either very large or small. Moreover, gradients of the overall product may vanish, which may cut off gradients at certain layers and even plague training [38].

[5] The strict interval satisfying Definition 5.1 depends on the value of K. The mentioned $[-10, 10]$ for PT is also not strict.

Protocol 2 Π_{LSig}: Local-Sigmoid via Fourier Series

P_0 Input	$\langle x\rangle_0, \mathsf{key}_0$	P_0 Output	$\langle \mathsf{LSig}(x)\rangle_0$
P_1 Input	$\langle x\rangle_1, \mathsf{key}_1$	P_1 Output	$\langle \mathsf{LSig}(x)\rangle_1$

Offline Phase

1: P_0, T: $\langle t\rangle_0, \langle \vec{u}\rangle_0, \langle \vec{v}\rangle_0 \leftarrow \mathsf{PRF}_0(\mathsf{key}_0)$
2: P_1, T: $\langle t\rangle_1 \leftarrow \mathsf{PRF}_1(\mathsf{key}_1)$
3: T: Compute $t = \langle t\rangle_0 + \langle t\rangle_1$,
$\langle \vec{u}\rangle_1 = \mathsf{FR}(\sin(t\vec{k})) - \langle \vec{u}\rangle_0$, $\langle \vec{v}\rangle_1 = \mathsf{FR}(\cos(t\vec{k})) - \langle \vec{v}\rangle_0$
4: T: Send $\langle \vec{u}\rangle_1, \langle \vec{v}\rangle_1$ to P_1

Online Phase

5: P_0: Compute $\delta_{\langle x\rangle_0} = \langle x\rangle_0 - \langle t\rangle_0 \bmod 32$
6: P_1: Compute $\delta_{\langle x\rangle_1} = \langle x\rangle_1 - \langle t\rangle_1 \bmod 32$
7: P_0: Send $\delta_{\langle x\rangle_0}$ to P_1 {▷ 5 bits for Integral Part}
8: P_1: Send $\delta_{\langle x\rangle_1}$ to P_0
9: $\mathsf{P}_0, \mathsf{P}_1$: Compute $\delta_x = \delta_{\langle x\rangle_0} + \delta_{\langle x\rangle_1}$
10: $\mathsf{P}_0, \mathsf{P}_1$: Compute $\vec{p} = \mathsf{FR}(\sin(\delta_x\vec{k}))$, $\vec{q} = \mathsf{FR}(\cos(\delta_x\vec{k}))$
11: P_0: $\langle \mathsf{LSig}(x)\rangle_0 = \alpha + \vec{\beta}(\vec{p} * \langle \vec{v}\rangle_0 + \vec{q} * \langle \vec{u}\rangle_0)$
12: P_1: $\langle \mathsf{LSig}(x)\rangle_1 = \alpha + \vec{\beta}(\vec{p} * \langle \vec{v}\rangle_1 + \vec{q} * \langle \vec{u}\rangle_1)$

If $\sin(\delta_x\vec{k})$, $\cos(t\vec{k})$, $\cos(\delta_x\vec{k})$, and $\sin(t\vec{k})$ are computed locally, online computation in Protocol 2 contains simple multiplication only.

In Lines 5-6, P_0 and P_1 mask their private inputs $\langle x\rangle_0$ and $\langle x\rangle_1$, respectively. Given two random values $\langle t\rangle_0, \langle t\rangle_1$ generated offline, P_0 and P_1 compute $\delta_{\langle x\rangle_0}$ and $\delta_{\langle x\rangle_1}$, respectively. An additional modular operation is performed to save on communication since $\mathsf{LSig}(x)$ with $m = 5$ is periodic. P_0 sends $\delta_{\langle x\rangle_0}$ in Line 7, while P_1 sends $\delta_{\langle x\rangle_1}$ in Line 8. In Line 9, P_0 and P_1 construct δ_x.

Both parties then compute $\sin(\delta_x\vec{k})$, $\cos(\delta_x\vec{k})$ and get their fixed-point representation $\vec{p}, \vec{q}$. In Lines 11-12, P_0 and P_1 compute the shares of $\alpha + \vec{\beta}\sin(x\vec{k})$ locally.

The offline phase mostly follows randomness generation in Section 3.2 except for vectors $\langle \vec{u}\rangle_1, \langle \vec{v}\rangle_1$. Given random 5-dimensional vectors $\langle \vec{u}\rangle_0, \langle \vec{v}\rangle_0$ generated by PRF, T generates $\langle \vec{u}\rangle_1, \langle \vec{v}\rangle_1$ according to trigonometric identities in Line 3. Theorem 5.2 says that Π_{LSig} securely realizes $\mathcal{F}_{\mathsf{QSMax}}(\vec{x})$, to be discussed in Appendix C.2.

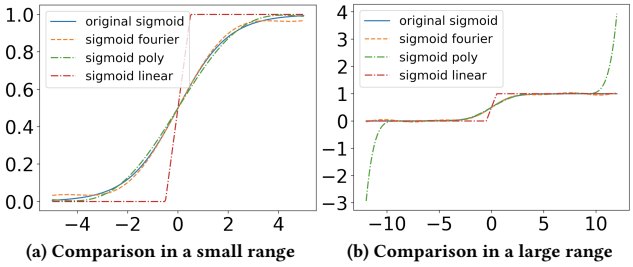
Theorem 5.2. *Protocol Π_{LSig} realizes functionality $\mathcal{F}_{\mathsf{LSig}}(x)$ in the presence of semi-honest PPT adversaries with static corruption.*

5.3 Communication Complexity

Recall that x is an n-bit decimal with a p-bit fractional part. The 1-round online communication in Lines 7-8 requires $2(m+p)$ bits for the reconstruction of δ_x. With PRF, Line 4 takes $2Kn$ bits of offline communication, where K is the dimension of $\vec{\beta}$. Theorem 5.3 summarizes the communication costs of Π_{LSig}.

Theorem 5.3. *Let x be the n-bit fixed-point numbers with p-bit fractional part and c_i, a_i, and K, m be public parameters. If $g(x) = a_0 + \sum_{k=1}^{K} a_k \sin\frac{2k\pi x}{2^m}$, securely evaluating $g(x)$ via Π_{LSig} requires 1-round $2(m+p)$ bits online and $2Kn$ bits offline.*

Squirrel [19] uses the Fourier series differently. It approximates sigmoid across the whole input domain, unlike our local-sigmoid. Specifically, it uses piecewise functions, requiring roughly $O(2n\log n)$ bits for choosing the piece, in which the second piece is



(a) Comparison in a small range (b) Comparison in a large range

Figure 1: Comparison of Different Approximations

a Fourier-series-based approximation. Squirrel transmits shares of $\vec{\beta}\sin(x\vec{k})$ in the online phase, taking $8Kn$ bits communication and is linear in K. We instead save online communication by sending the additive shares of δ_x. It thus requires $2(m+p)$ bits and is independent of K. Concretely, for the configuration of $n = 64$, $p = 16$ (default configurations in many related works), $K = 5$, $m = 5$, Squirrel requires 2560 bits (or $8Kn$ bits), while our Π_{LSig} takes 42 bits. The independence from K can benefit the approximation accuracy. While we found that $K = 5$ is accurate enough, it can still be tuned.

5.4 Correctness for Sigmoid-Like Protocol

Since we perform truncation over floating-point numbers, we analyze the absolute difference between shared values and floating values of $\sin\frac{2k\pi x}{2^m}$. As for the inputs in the interval, the error comes from the truncation. The out-of-interval inputs, which exist with a small probability, are bounded in terms of the approximation.

Theorem 5.4 (Correctness). *Let x be an n-bit fixed-point number with a p-bit fractional part and $a_1, \ldots, a_K$ be scalars. Π_{LSig} uses $(K+1)$-term Fourier series. The difference of replacing $\mathsf{Sigmoid}(x)$ with Π_{LSig} is bounded by:*
(i). $\sum_{k=1}^{K} |a_k| \cdot 2^{-f+1}$ for any $x \in [-\mathsf{a}, \mathsf{a}]$;
(ii). $1 + 2\epsilon$ for any $x < -\mathsf{a}$ or $x > -\mathsf{a}$, where ϵ is a small value.

The proof is in Appendix B.3.

Figures 1a-1b show the accuracy of fitting with 6 terms. The area between the two curves in Figure 1a essentially reflects the value of Δ. As in Figure 1a, using the Fourier series is more accurate than piecewise linear functions, *i.e.*, $\mathsf{LSig}(x)$ has a much smaller value of Δ than PLA. For out-of-interval inputs with low probability, the outputs of $\mathsf{LSig}(x)$ are bounded by 1, not like unbounded PT in Figure 1b. Our solution provides a more stable approximation (*i.e.*, without the risk of gradient explosion caused by large/small inputs) in the whole range, with the error bounded by ≤ 1. Particularly, for inputs in $[-1.5, 1.5]$ (occur with high probability), local-sigmoid is more accurate than polynomial approximation. Besides, ODE-based approximation is also applicable to sigmoid. Nevertheless, the iterative method makes it impossible to achieve a 1-round protocol.

Figure 2 shows the approximation when $m = 4, 5, 6, 7$ with periodic intervals of $[16, 32, 64, 128]$. For $K = 5$, we can see that $m = 4$ generally provides a better approximation. For secret-shared inputs, the computing parties do not know whether the input value is positive/negative if the underlying interval is $[-\mathsf{a}, \mathsf{a}]$. Thus, to avoid additional secure comparisons, we use $[0, \mathsf{a}]$ instead of $[-\mathsf{a}, \mathsf{a}]$.

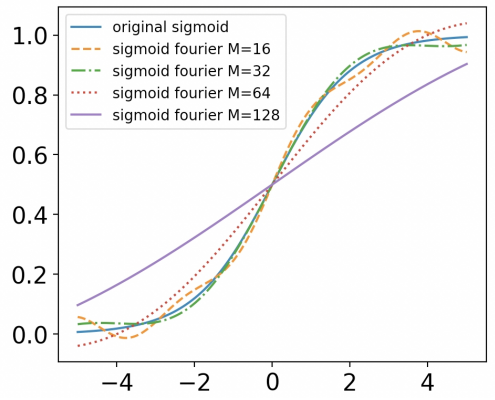


Figure 2: Approximation Accuracy with Different m

6 END-TO-END MACHINE LEARNING

6.1 General Private Learning Pipeline

A machine learning model can be decomposed into sequential multiple layers. The predefined layers include the dense, ReLU, convolution, average pooling, max pooling, and loss layers. Each layer computes both forward and backward propagation. In end-to-end private training, P_0 and P_1 collaboratively execute a sequence of protocols based on the predefined model.

Let x be the input data and w be the model parameter. Forward propagation is to compute the output of the activation function $y = f(x, w)$ at the corresponding layer. Protocols 1 and 2 are for the forward pass using softmax and sigmoid functions. Backward propagation performs a backward pass to adjust the model's parameters w via the chain rule by computing $\frac{\partial f}{\partial y}$, $\frac{\partial f}{\partial x}$, and $\frac{\partial f}{\partial w}$. All these computations should be performed over secret shares.

6.2 Backpropagation and Auto-differentiation

Let y be the model output and $\hat{y}$ its label. Auto-differentiation is essentially computing the differentiation of the loss functions over y. As loss functions for softmax and sigmoid, our implementation respectively uses `CrossEntropyWithSoftmax` (SoftmaxCE) and `BinaryCrossEntropyWithSigmoid` (SigmoidBCE). Below, we elaborate on backpropagation. Appendix A provides more details.

6.2.1 Softmax. We define $\text{SoftmaxCE} = -\sum_i \log(\text{Softmax}(\vec{y})_i)\hat{y}_i$ upon $\text{Softmax}(\vec{y})$. To perform backpropagation, we compute the gradients over $\vec{y}$ as $\text{Softmax}(\vec{y}) - \hat{y}$ (details in Appendix A.2.2). Given secret shares $\langle y\rangle_0$ and $\langle y\rangle_1$, P_0 and P_1 jointly call Protocol 1 to obtain secret-shared outputs. Then, P_0 and P_1 respectively subtract local shares of $\hat{y}$ to obtain secret-shared derivatives over $\vec{y}$.

6.2.2 Sigmoid. We define $\text{SigmoidBCE} = -\log(\text{Sigmoid}(y))\hat{y} - \log(1 - \text{Sigmoid}(y))(1 - \hat{y})$. The partial differentiation over y (in Appendix A.2.3) can be computed by $\text{Sigmoid}(y) - \hat{y}$. Given secret shares $\langle y\rangle_0$ and $\langle y\rangle_1$, P_0 and P_1 jointly execute Protocol 2 and local subtraction to obtain the secret-shared derivatives over y.

6.3 Our Framework Implementation in Python

Our Python version is implemented in the TensorFlow framework. For training, we implemented various cryptographic protocols [29, 35] for ReLU, pooling, convolution, and more. The truncation in Section 3.2 has been implemented following the probabilistic truncation of SecureML [22] and Cheetah's faithful truncation [13].

To our knowledge, our implementation is the first to achieve efficient cryptographic training in TensorFlow with online/offline computation using a secure deterministic random-bit generator. Researchers and practitioners can use it to realize other networks by integrating the necessary operators and layers.

6.4 Our C++ Implementation for Piranha

6.4.1 Plug-in Modules. Our C++ implementation contains the plug-in modules for Piranha [35], a general-purpose platform providing the cryptography backend and neural network library (*e.g.*, activation, pooling). We added implementations of sigmoid and softmax protocols, their backpropagation, and truncation [22]. We use "QSLS" to denote implementations of our protocols fitting Piranha.

6.4.2 Private Training with QSLS. We provide an example of private training by replacing Piranha's softmax with Protocol 1. Piranha originally consists of `_adhoc_softmax_grad`, an implementation of backpropagation via CrossEntropyWithSoftmax. We follow Section 6.2 to implement `ode_softmax` and its backpropagation `ode_softmax_grad` for private training using QSLS instead of `_adhoc_softmax_grad`. Other modules could also be plugged in.

7 EXPERIMENTAL EVALUATION

CPU-based experiments are run on three CentOS servers, each with an Intel® Xeon® Platinum 8163 2.50GHz CPU and 16GB RAM. GPU-based experiments are conducted on Ubuntu servers, running 8-core Intel® Xeon® Platinum 8163 2.50GHz CPUs with 64GB RAM and NVIDIA-T4 GPU of 16GB RAM. To simulate real-world network conditions, we set the network bandwidth to 12.5MB/s or 30.1MB/s for the WAN condition. Our experiments include evaluations of the proposed protocols and a comparison of training performance, addressing the following questions:

Q1: *How much the proposed protocols have improved the state of the art in communication and running time?* (Sections 7.2 and 7.3)

Q2: *Compared with the state of the art, how much has training performance improved in terms of model accuracy and training time?* (Sections 7.4 and 7.6)

Q3: *Compared with the state of the art, how much communication has been reduced for training models?* (Section 7.5)

Q4: *How do WAN settings impact training time?* (Section 7.7)

7.1 Baselines and Setup

7.1.1 Protocol-Level Comparison. Tables 3 and 4 compare the communication of specific protocols atop different computing methods. Tables 5 and 6 show the running time (in seconds). Most of our tests use 64-bit fixed-point numbers with 14-bit fractional parts. For softmax, we test our quasi-softmax with ASM and ComDiExp. For sigmoid, we compare our local-sigmoid with the Newton-Raphson method, polynomial approximation, and Squirrel's approximation.

Table 3: Communication Comparison of Softmax

Protocol	Offline (bit)	Online (bit)	Overall (bit)	Round
($m = 10$)				
ASM	-	-	3017195	704
ComDiExp [32]	198912	783250	982162	171
Π_{QSMax} ($r = 8$)	10240	31744	41984	16
Π_{QSMax} ($r = 16$)	20480	63488	83968	32
Π_{QSMax} ($r = 32$)	40960	126976	167936	64
Π_{QSMax} ($r = 64$)	81920	253952	335872	128
($m = 100$)				
ASM	-	-	30171944	704
ComDiExp [32]	2174592	8536390	10710982	300
Π_{QSMax} ($r = 8$)	102400	308224	410624	16
Π_{QSMax} ($r = 16$)	204800	616448	821248	32
Π_{QSMax} ($r = 32$)	409600	1232896	1642496	64
Π_{QSMax} ($r = 64$)	819200	2465792	3284992	128
($m = 1000$)				
ASM	-	-	301719448	704
ComDiExp [32]	21931392	86067790	107999182	430
Π_{QSMax} ($r = 8$)	1024000	3073024	4097024	16
Π_{QSMax} ($r = 16$)	2048000	6146048	8194048	32
Π_{QSMax} ($r = 32$)	4096000	12292096	16388096	64
Π_{QSMax} ($r = 64$)	8192000	24584192	32776192	128

Table 4: Communication Comparison of Sigmoid

Protocol	Offline (bit)	Online (bit)	Overall (bit)	Round
Piece. Linear Approx.	-	≈ 800	-	5
Poly. Approx. ($K = 5$)	320	1280	1600	1
Poly. Approx. ($K = 8$)	512	2048	2560	1
Squirrel [19] ($K = 5$)	-	2560 + 1792	-	3
Squirrel [19] ($K = 8$)	-	4096 + 1792	-	3
LLAMA [11]	-	128	-	1
Π_{LSig} ($m = 4, K = 5$)	640	36	676	1
Π_{LSig} ($m = 4, K = 8$)	1024	36	1060	1
Π_{LSig} ($m = 5, K = 5$)	640	38	678	1
Π_{LSig} ($m = 5, K = 8$)	1024	38	1062	1

7.1.2 Configuration for Model-Level Comparison. Tables 7-12 compare the model accuracy, communication, and training time of both large and small models. We evaluate model training using:

- MNIST [8]: a handwritten-digit (0-9) dataset with 60000 training examples and 10000 test examples,
- CIFAR-10 [16]: an RGB-image dataset with 60000 32×32 images evenly distributed across 10 classes.

7.1.3 Large-Model Testing. We conduct the evaluations of running large models using Piranha [35] (which implements 2-party SecureML [22], 3-party Falcon [34], and 4-party Fantastic Four [7] with GPUs) and CryptGPU [32] (mainly implementing CrypTen and training based on pre-trained initialization). For testing model accuracy and tuning, we use learning rates ranging from 0.001 to 0.02. We test three classical models following Piranha:

- AlexNet [17]: 61M parameters, using convolution, ReLU, max pooling, fully-connected layers, and softmax.
- LeNet [18]: 60K parameters, using convolution, ReLU, max pooling, and softmax.
- VGG-16 [31]: 138M parameters, using 16 layers of convolution, ReLU, max pooling, fully-connected layers, and softmax.
- ResNet [12]: we choose ResNet-18 with 11M parameters, consisting of convolution, max pooling, average pooling, batch normalization, fully-connected layers, and softmax.

7.1.4 Small-Network Testing. We adopt Networks A, B, C, and D as described in SecureNN [33]. We benchmark running time using only CPUs with SecureNN [33], CrypTen [15], and Falcon [34] as they are open-source. For accuracy, we compare with SPDZ-QT [14], which has the highest accuracy among all of the above.

7.2 Communication of Proposed Protocols

Communication costs significantly impact training time, especially under limited bandwidth and high network latency. We first evaluate the communication of our proposed protocols in Table 3 and Table 4. In summary, ASM approximation requires the most rounds, whereas ComDiExp [32] and Π_{QSMax} consume relatively fewer communication rounds. When the number of classes increases, unlike ComDiExp, which requires more communication rounds, Π_{QSMax} maintains a constant number of rounds. When the number of classes is the same for all baselines [15, 32, 33], Π_{QSMax} incurs the lowest communication costs. As for sigmoid, both LLAMA [11] and Π_{LSig} achieve round-optimal communication. Moreover, Π_{QSMax} boasts the lowest overall and online communication costs.

7.2.1 Softmax. Recall that two cryptographic approaches for approximating softmax are ASM and ComDiExp [32] (Section 2.2). We take SecureNN for ASM and CrypTen/CryptGPU for ComDiExp. SecureNN did not differentiate between the online and offline phases. Table 3 compares online and offline communication and rounds. For 64-bit shared data, we explore $m \in \{10, 100, 1000\}$ classes. We list the communication of Π_{QSMax} under 8, 16, 32, or 64 iterations. ComDiExp performs better than ASM. For almost all cases, Π_{QSMax} has much lower communication than ComDiExp. Empirically, we found that either $r = 16$ or $r = 32$ is sufficient for model training. For $r = 32$, Π_{QSMax} reduces the communication by 83%, 85%, and 85% for 10, 100, and 1000 classes, respectively.

7.2.2 Sigmoid. Table 4 compares the communication for sigmoid, given 64-bit fixed-pointed numbers with 14-bit fractional parts. We take ABY2.0 [26] as an instantiation built upon piecewise linear approximation. The protocols of ABY2.0, LLAMA [11], and Squirrel rely on extra communication-heavy oblivious transfer, function secret sharing, or garbled circuits. We focus on comparing online communication. For Squirrel, the left value indicates the communication for Fourier-series-based approximation, while the right value (1792) comes from the secure comparison (similar to PLA).

We can see that LLAMA, polynomial approximation (PolyA), and our Π_{LSig} require one-round communication. The online communication costs of PLA are relatively low at the expense of more

Table 5: Running Time (10 Epochs) for Softmax over GPU

Protocol	Classes			
	10	100	1000	10000
Piranha-Reveal	0.059568	0.411686	3.930131	38.979216
Piranha-Adhoc	3.461582	-	-	-
Piranha-ASM	40.168154	40.179480	40.216057	40.378034
QSLS ($r = 16$)	1.898646	1.911498	1.942481	2.130517
QSLS ($r = 32$)	3.807372	3.819943	3.895476	4.277393

Table 6: Running Time (10 Epochs) for Sigmoid over GPU

Protocol	Batch Size			
	32	64	128	256
Piranha	40.05117	40.08663	40.08994	40.14356
QSLS	0.077250	0.077017	0.076789	0.076450

rounds. PolyA, Squirrel, and Π_{LSig} refer to an additional parameter K, indicating the $K+1$ terms for approximation. Squirrel takes 8. We pick 5. The communication of the former two increases as K enlarges, whereas our Π_{LSig} keeps constant, *i.e.*, 36 bits or 38 bits. Π_{LSig} incurs much lower communication costs than prior arts.

7.3 GPU Acceleration of Proposed Protocols

We assess the running time (in seconds) of Piranha and QSLS (our Π_{QSMax} and Π_{LSig} fitting Piranha). Table 5 is for softmax, and Table 6 is for sigmoid. Generally, Π_{QSMax}'s running time remains relatively stable and very short due to the GPU parallelism. Round-optimal Π_{LSig} significantly outpaces polynomial approximation, which unavoidably spends multiple rounds for computing multi-order multiplication and truncation followed by each multiplication.

7.3.1 Softmax. Piranha supports three softmax protocols: "Reveal," "ASM," and "Adhoc." "Reveal," as a template, starts with reconstructing the private inputs before computing plaintext softmax. We implemented Π_{QSMax} with 16 and 32 iterations.

Table 5 shows the running time of different protocols for 10, 100, or 1000 input classes. With 16 and 32 iterations, Π_{QSMax} outperforms ASM-based protocol by 20× and 10×, respectively. "Adhoc" can only support 10 classes but fails for 100 classes or more. Besides, the running time of Π_{QSMax} remains nearly unvaried thanks to the parallelization of GPU. This results in a significant improvement in training speed, particularly when handling vectorized inputs.

7.3.2 Sigmoid. In Piranha, we implemented Π_{LSig} and compared it with the polynomial-approximated sigmoid. Piecewise linear approximation, Squirrel, and LLAMA rely on additional cryptographic primitives, which are not supported in Piranha. We found that the running time remains almost unvaried across different batch sizes. Intuitively, GPU instantiations offer a great advantage in parallel computation. Notably, Π_{LSig} speeds up by ~570×, so running sigmoid replaced by Π_{LSig} can greatly accelerate training in Piranha.

7.4 Training Accuracy of Models

We report training accuracy of large and small models. Piranha supports training large models from scratch. In contrast, CryptGPU trains large models with a pretraining phase, aided by plaintext training over public datasets. Accordingly, we evaluated accuracy with and without the pretraining phase. SPDZ-QT mostly aims at relatively small networks. We configure $f = 24$ and the batch size to 32. Acc-1 and Acc-5/15 are accuracy at epoch=1 and epoch=5 (or 15). The best accuracy reported in Piranha ([35, Figure 5]) is 98.1% for LeNet, 40.7% for AlexNet, and 58.7% for VGG-16. SPDZ-QT's results are from [14, Table 3]. Table 7 shows the comparison.

Table 7: Training Accuracy Compared with Prior Arts

Model	Framework	Acc-5	Acc-10	Pretrain
AlexNet (CIFAR-10)	Piranha	-	40.7%	No
	Ours	52.1%	**56.3%**	No
	CryptGPU	-	59.6%	Yes
	Ours	51.2%	**59.6%**	Yes
VGG-16 (CIFAR-10)	Piranha	-	58.7%	No
	Ours	41.5%	**59.4%**	No
	Ours	69.7%	**77.6%**	Yes
LeNet (MNIST)	Piranha	-	98.1%	No
	Ours	98.5%	**98.5%**	No
	CryptGPU	-	93.9%	Yes
	Ours	98.6%	**99.4%**	Yes
Model	**Framework**	**Acc-1**	**Acc-5/15**	**Pretrain**
Network A (MNIST)	SPDZ-QT	-	$97.8\%_{(15)}$	No
	Ours	94.7%	$97.2\%_{(5)}$	No
	Ours	96.8%	$98.2\%_{(5)}$	Yes
Network B (MNIST)	SPDZ-QT	-	$98.0\%_{(15)}$	No
	Ours	96.4%	$98.1\%_{(5)}$	No
	Ours	96.7%	$98.7\%_{(5)}$	Yes
Network C (MNIST)	SPDZ-QT	-	$98.5\%_{(5)}$	No
	Ours	97.8%	$98.5\%_{(5)}$	No
	Ours	98.1%	$98.7\%_{(5)}$	Yes
Network D (MNIST)	SPDZ-QT	-	$98.1\%_{(15)}$	No
	Ours	95.5%	$98.2\%_{(5)}$	No
	Ours	96.5%	$98.2\%_{(5)}$	Yes

For training from sketch, we record the accuracy at each epoch. Accuracy at epoch=10 is then used as the highest one. Our training accuracy surpasses that of Piranha, *e.g.*, 56.3% versus 40.7% for AlexNet or 59.4% versus 58.7% for VGG-16. The small networks can reach an accuracy comparable with SPDZ-QT by fewer training epochs, from 15 to 5. These results underscore the superior approximation and convergence capabilities of our protocols.

Overall, private training can deliver reasonable accuracy for relatively small models, whereas attaining high accuracy for large models still necessitates pretraining assistance.

7.5 Communication of Model Training

We benchmark training with large and small models for evaluating communication and accuracy in the ideal LAN settings without being affected by high latency and low bandwidth. For communication, we show the results of relatively large models (with more non-linear computations) in Section 7.5.1 and smaller models in Section 7.5.2. We count the online and offline communication costs of the two computing parties and the commodity server and the

Table 8: Communication (GB/batch) for Small Model Training

Framework	Network			
	A	B	C	D
SecureNN	0.047	1.31	2.11	0.37
Piranha	-	-	1.25	-
SPDZ-QT	0.055	0.43	0.75	0.09
CryptGPU	-	-	1.14	-
Ours	0.021	0.41	0.64	0.05

Table 9: Communication (GB/batch) for Large Model Training

Framework	AlexNet (CIFAR-10)	VGG-16 (CIFAR-10)
CryptGPU	1.37 [↓ 57.7%]	7.55 [↓ 61.3%]
Falcon	0.62×3 [↓ 68.8%]	1.78×3 [↓ 45.3%]
Piranha	0.58×3 [↓ 66.7%]	4.26×3 [↓ 77.2%]
Ours	0.39 ($b = 5$), 0.58 ($b = 7$)	2.925 (batch of 2^5)

data frame size (gRPC format[6]). Whether training with large models in Table 9 (batch size $b \in \{5, 7\}$) or small models in Table 8 (batch size is 128, with 32 iterations for computing softmax), our approach achieves the lowest communication costs.

7.5.1 Communication for AlexNet/VGG-16. Only a small set of works (Falcon [34], CryptGPU [32], and Piranha [35]) can successfully benchmark training of large models like AlexNet and VGG-16. Table 9 shows the communication for training AlexNet and VGG-16 over the CIFAR-10 dataset. We conduct experiments following CryptGPU and Piranha in testing one-batch communication and using identical model architecture. We use the same pooling and free truncation as in Piranha and set the batch size to be $2^b, b \in \{5, 7\}$. Table 9 quotes the per-party communication figures from [35, Table 4]. Our approach reduces approximately 57%-77% communication compared with prior arts. The efficiency of our non-linear units can be reflected by large-model training with more non-linear computations. For a more specific explanation, Section 7.2 conducts protocol-level comparisons.

7.5.2 Communication for Network A/B/C/D. We follow SecureNN's benchmarking and train networks A/B/C/D over MNIST to show more results. Table 8 compares communication with SecureNN [33], Piranha (quoted from [35, Table 4], which reports per-party communication), SPDZ-QT [14], and CryptGPU ([32, Table IV]). For SPDZ-QT, we estimate 1-batch communication by $x/60000 \cdot 128$, where 1-epoch communication x is quoted from [14, Table 3]. Our approach greatly reduces the communication cost across the board.

7.6 Training Acceleration with GPU in LAN

To show the acceleration brought by our protocols, we count the average training time in LAN per batch over 10 batches. We perform the tests on relatively large models since GPU benefits parallel and vectorized computation. LeNet uses max pooling, and AlexNet uses average pooling. For each training step, the batch size is set to 128. Table 10 presents the training time in seconds, where QSLS represents using our proposed protocols as plug-ins in Piranha. In short, QSLS speeds up the training in Piranha by ~10%-60%.

[6]https://grpc.io

Table 10: GPU Training Time (s/batch) under LAN

Work	LeNet (MNIST)	AlexNet	VGG-16	ResNet-18
	(CIFAR-10)			
Piranha	2.574691	6.506521	30.345877	29.879927
QSLS	2.208990	2.569884	26.390739	25.876978

Table 11: CPU Training Time over MNIST under WAN

Work		Network (s/batch)			
	Latency	A	B	C	D
SecureNN	5ms	3.311	88.95	139.4	23.55
	50ms	7.024	97.54	148.7	28.93
Falcon	5ms	0.778	12.10	26.67	1.514
	50ms	3.311	17.20	31.76	3.977
CrypTen	5ms	6.192	73.90	108.8	5.282
	50ms	32.98	104.0	144.2	30.58
Ours-B	5ms	1.952	**21.15**	**33.25**	3.340
	50ms	21.62	36.11	47.41	12.52
Ours-T	5ms	**1.513**	22.93	35.16	**2.870**
	50ms	**4.665**	**26.70**	**43.86**	**6.210**

Table 12: Training Time for LeNet and AlexNet under WAN

Work	Latency	LeNet (MNIST)		AlexNet (CIFAR-10)	
Piranha	60ms	5680min	-	9192min	-
Ours-B	60ms	1780min	[↓ 69%]	4029min	[↓ 56%]
Ours-T	60ms	**1232min**	[↓ 78%]	**2462min**	[↓ 73%]

7.7 Impact of Bandwidth and Latency in WAN

For testing the training time, we take network latency into account in measuring the time for both communication and computation.

We compare with SecureNN, Falcon, and CrypTen for training over MNIST. Table 11 reports the result under a bandwidth of 12.5MB/s and a latency of either 5 or 50ms. Falcon reports the online time only. "B/T" denotes training with a basic/tree-structured version of ReLU protocols. Our protocols are faster than prior works. We suggest picking different types of protocols for different scenarios. If the model size is large, we suggest the basic version. If network latency is high, we suggest the tree-structured version.

In Table 12, we adjust the latency to 60ms with a bandwidth of 40MB/s to align with Piranha's configuration. Piranha's communication consumes a large part of the overall running time ([35, Figure 6]). Our work reduces training time by ~50%-80% for LeNet and AlexNet, confirming the benefit of minimizing communication.

8 CONCLUDING REMARKS

We propose two cryptography-friendly approximation approaches to secure computation of softmax and sigmoid, leading to expedited private training with much lower communication. We shed light

on the secret-sharing-based multi-party computation (SMC) protocol design for bounded non-linear functions, avoiding unbounded functions ($e^x, 1/x$) during intermediate computations. A pivotal contribution lies in our approximations for non-linear functions as an initial value problem. Broadly, we extend the realm of SMC protocols to encompass solutions for differential equations with rational polynomials or trigonometric function coefficients. The Fourier series approximation can be adapted for securely computing other single-variable non-linear functions, *e.g.*, step functions.

Our softmax protocol Π_{QSMax} transforms non-linear approximation into an iterative solution of ordinary differential equations solved by the Euler formula. In SMC, non-linear approximations are replaced by multi-order multiplications. Π_{QSMax} shows a new way to reduce the required multiplication orders for non-linear and polynomial approximations and reduce the computational and communication complexities from $O(n^c)$ accuracy-limitation (*e.g.*, Taylor expansion) to $O(cn)$ accuracy-limitation for c iterative rounds.

Our Π_{LSig} protocol approximates sigmoid by Fourier series. Notably, its online communication is independent of the number of Fourier series terms and less than $2n$ bits. Ideally, Π_{LSig} facilitates negligible-error approximation for any square-integrable function by choosing a sufficiently large number of Fourier series terms.

Both Π_{QSMax} and Π_{LSig} are secret-sharing-based. They can be extended using replicated shares and homomorphic encryption for other features, such as malicious security. They can also be building blocks for other private-learning frameworks, such as CrypTen. We hope our Python implementation could benefit further development, say, transformers [39] and recognition models [1].

ACKNOWLEDGMENTS

The authors wholeheartedly appreciate the invaluable feedback from the anonymous shepherd, reviewers, and artifacts evaluation committee. The authors with Ant Group thank Yuan Zhao, Yashun Zhou, Dong Yin, and Jiaofu Zhang at Ant Group for their insightful discussions and endeavors in coding. Yu is thankful for the help and guidance of Jiafan Wang. Special thanks go to David Wu, Florian Kerschbaum, and Jean-Pierre Hubaux for their constructive suggestions for Yu's poster at EPFL Summer Research Institute. Finally, Sherman Chow is supported in part by the General Research Fund (CUHK 14210319) of Hong Kong Research Grants Council (RGC).

REFERENCES

[1] Jianli Bai, Xiaowu Zhang, Xiangfu Song, Hang Shao, Qifan Wang, Shujie Cui, and Giovanni Russello. 2023. CryptoMask: Privacy-preserving Face Recognition. In *ICICS*. 333–350.

[2] Christina Boura, Ilaria Chillotti, Nicolas Gama, Dimitar Jetchev, Stanislav Peceny, and Alexander Petric. 2018. High-Precision Privacy-Preserving Real-Valued Function Evaluation. In *FC*. 183–202.

[3] John Charles Butcher. 2016. *Numerical Methods for Ordinary Differential Equations*. John Wiley & Sons, New Zealand.

[4] Paul L. Butzer and Rolf J. Nessel. 1971. *Fourier Analysis and Approximation: One Dimensional Theory*. Birkhäuser Basel, Switzerland.

[5] Yuanfeng Chen, Gaofeng Huang, Junjie Shi, Xiang Xie, and Yilin Yan. 2020. Rosetta: A Privacy-Preserving Framework Based on TensorFlow. https://github.com/LatticeX-Foundation/Rosetta. Also presented at the Privacy Preserving Machine Learning Workshop at ACM CCS 2021.

[6] Morten Dahl, Jason Mancuso, Yann Dupis, Ben Decoste, Morgan Giraud, Ian Livingstone, Justin Patriquin, and Gavin Uhma. 2018. Private Machine Learning in TensorFlow using Secure Computation. arXiV 1810.08130. Also presented at the Privacy Preserving Machine Learning Workshop at NeurIPS 2018.

[7] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. 2021. Fantastic Four: Honest-Majority Four-Party Secure Computation with Malicious Security. In *Usenix Security*. 2183–2200.

[8] Li Deng. 2012. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. *IEEE Signal Process. Mag.* 29, 6 (2012), 141–142.

[9] Minxin Du, Xiang Yue, Sherman S. M. Chow, Tianhao Wang, Chenyu Huang, and Huan Sun. 2023. DP-Forward: Fine-tuning and Inference on Language Models with Differential Privacy in Forward Pass. In *ACM CCS*. 18 pages. To appear, also available at arXiV 2309.06746.

[10] Simeon Ola Fatunla. 1988. *Numerical Methods for Initial Value Problems in Ordinary Differential Equations*. Elsevier, Boston.

[11] Kanav Gupta, Deepak Kumaraswamy, Nishanth Chandran, and Divya Gupta. 2022. LLAMA: A Low Latency Math Library for Secure Inference. *Proc. Priv. Enhancing Technol.* 2022, 4 (2022), 274–294.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778.

[13] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. 2022. Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference. In *Usenix Security*. 809–826.

[14] Marcel Keller and Ke Sun. 2022. Secure Quantized Training for Deep Learning. In *ICML*. 10912–10938.

[15] Brian Knott, Shobha Venkataraman, Awni Y. Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. 2021. CrypTen: Secure Multi-Party Computation Meets Machine Learning. In *NeurIPS*. 4961–4973.

[16] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning Multiple Layers of Features from Tiny Images. https://www.cs.toronto.edu/~kriz/cifar.html.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*. 1106–1114.

[18] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Comput.* 1, 4 (1989), 541–551.

[19] Wen-jie Lu, Zhicong Huang, Qizhi Zhang, Yuchen Wang, and Cheng Hong. 2023. Squirrel: A Scalable Secure Two-Party Computation Framework for Training Gradient Boosting Decision Tree. In *Usenix Security*. 6435–6451.

[20] Jack P. K. Ma, Raymond K. H. Tai, Yongjun Zhao, and Sherman S. M. Chow. 2021. Let's Stride Blindfolded in a Forest: Sublinear Multi-Client Decision Trees Evaluation. In *NDSS*. 18 pages.

[21] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *ACM CCS*. 35–52.

[22] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*. 19–38.

[23] Lucien K. L. Ng and Sherman S. M. Chow. 2021. GForce: GPU-Friendly Oblivious and Rapid Neural Network Inference. In *Usenix Security*. 2147–2164.

[24] Lucien K. L. Ng and Sherman S. M. Chow. 2023. SoK: Cryptographic Neural-Network Computation. In *IEEE S&P*. 497–514.

[25] Lucien K. L. Ng, Sherman S. M. Chow, Anna P. Y. Woo, Donald P. H. Wong, and Yongjun Zhao. 2021. Goten: GPU-Outsourcing Trusted Execution of Neural Network Training. In *AAAI*. 14876–14883.

[26] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *Usenix Security*. 2165–2182.

[27] Deevashwer Rathee, Anwesh Bhattacharya, Rahul Sharma, Divya Gupta, Nishanth Chandran, and Aseem Rastogi. 2022. SecFloat: Accurate Floating-Point meets Secure 2-Party Computation. In *IEEE S&P*. 576–595.

[28] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. 2021. SiRnn: A Math Library for Secure RNN Inference. In *IEEE S&P*. 1003–1020.

[29] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow2: Practical 2-Party Secure Inference. In *ACM CCS*. 325–342.

[30] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis R. Bach. 2022. AriaNN: Low-Interaction Privacy-Preserving Deep Learning via Function Secret Sharing. *Proc. Priv. Enhancing Technol.* 2022, 1 (2022), 291–316.

[31] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*. 14 pages.

[32] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. 2021. CryptGPU: Fast Privacy-Preserving Machine Learning on the GPU. In *IEEE S&P*. 1021–1038.

[33] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 26–49.

[34] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. 2021. Falcon: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *Proc. Priv. Enhancing Technol.* 2021, 1 (2021), 188–208.

[35] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. 2022. Piranha: A GPU Platform for Secure Computation. In *Usenix Security*. 827–844.

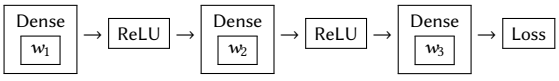
[36] Harry W. H. Wong, Jack P. K. Ma, Donald P. H. Wong, Lucien K. L. Ng, and Sherman S. M. Chow. 2020. Learning Model with Error - Exposing the Hidden

Model of BAYHENN. In *IJCAI*. 3529–3535.
[37] Zhiqin Yang, Yonggang Zhang, Yu Zheng, Xinmei Tian, Peng Hao, Tongliang Liu, and Bo Han. 2023. FedFed: Feature Distillation against Data Heterogeneity in Federated Learning. In *NeurIPS*. 32 pages.
[38] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. 2021. Dive into Deep Learning. arXiv:2106.11342.
[39] Mengxin Zheng, Qian Lou, and Lei Jiang. 2023. Primer: A Privacy-preserving Transformer on Encrypted Data. In *DAC*. 6 pages.
[40] Yu Zheng, Wei Song, Minxin Du, Sherman S. M. Chow Qian Lou, Yongjun Zhao, and Xiuhua Wang. 2023. Cryptography-Inspired Federated Learning for Generative Adversarial Networks and Meta Learning. In *ADMA*. 393–407.

A SUPPLEMENTARY FOR LEARNING

A.1 A Training Example

We illustrate the training process using Network A [22, 33]. It contains three dense layers and ReLU activations, followed by the loss function `CrossEntropyWithSoftmax` (SoftmaxCE). Its structure is:



The code snippet below builds Network A.

```
from stensorflow.ml.nn.networks.NN import NN
from stensorflow.ml.nn.layers.input import Input
from stensorflow.ml.nn.layers.relu import ReLU
from stensorflow.ml.nn.layers.loss import SoftmaxCE
from stensorflow.ml.nn.layers.dense import Dense
class NETWORKA(NN):
  def __init__(self, feature, label):
    super(NETWORKA, self).__init__()
    layer = Input(dim=28×28, x=feature)
    self.addLayer(layer)
    layer = Dense(output_dim=128, fathers=[layer])
    self.addLayer(layer)
    layer = ReLU(output_dim=128, fathers=[layer])
    self.addLayer(layer)
    layer = Dense(output_dim=128, fathers=[layer])
    self.addLayer(layer)
    layer = ReLU(output_dim=128, fathers=[layer])
    self.addLayer(layer)
    layer = Dense(output_dim=10, fathers=[layer])
    self.addLayer(layer)
    layer_label = Input(dim=10, x=label)
    self.addLayer(ly=layer_label)
    layer_loss = SoftmaxCE(layer_score=layer,
                           layer_label=layer_label)
    self.addLayer(ly=layer_loss)
```

In the code snippet, the cryptographic versions of dense and ReLU layers and the loss function are imported as if using TensorFlow. At each training iteration, forward propagation and backward propagation are performed layer by layer over secret shares.

A.2 Derivative for Auto-differentiation

A.2.1 Softmax. Let $g_i = \text{Softmax}(\vec{y})_i = e^{y_i}/\sum_j e^{y_j}$ and also $g = \text{Softmax}(\vec{y})$. Then, we have $\frac{\partial g_i}{\partial \vec{y}_i} = \frac{e^{y_i}(\sum_j e^{y_j}) - (e^{y_i})^2}{(\sum_j e^{y_j})^2} = g_i - g_i^2$ and $\frac{\partial g_i}{\partial \vec{y}_j} = \frac{0*(\sum_j e^{y_j}) - e^{y_i}e^{y_j}}{(\sum_j e^{y_j})^2} = -g_i g_j$ for $j \neq i$. Hence, we have,

$$D_g = (\frac{\partial g_i}{\partial y_j})_{i,j} = \text{diag}(g) - gg^\top,$$

where $\text{diag}(g)$ is a diagonal matrix in which the $()_{i,i}$ entry contains g_i. Hence, the backpropagation of softmax is

$$\frac{\partial \text{loss}}{\partial g} \mapsto \frac{\partial \text{loss}}{\partial y} = D_g \frac{\partial \text{loss}}{\partial g} = \text{diag}(g)\frac{\partial \text{loss}}{\partial g} - gg^\top \frac{\partial \text{loss}}{\partial g}.$$

In the backpropagation phase, secret-shared g can be directly obtained from the forward propagation computed by Π_{QSMax}. Given secret-shared g, $\text{diag}(g)$ and $g^\top$ are obtained by local transformation, *i.e.*, diagonalization and matrix transpose. $\frac{\partial \text{loss}}{\partial g}$ are the inputs depending on the loss function. For $\text{diag}(g) \cdot \frac{\partial \text{loss}}{\partial g}$, the parties can jointly invoke the secure multiplication protocol $\Pi_\times$. The right part $gg^\top \frac{\partial \text{loss}}{\partial g}$ can be securely computed by secure multiplication $\Pi_\times$ from right to left.

Recall from Section 6.2 that our implementation uses the loss function of `CrossEntropyWithSoftmax`, which could be simplified.

A.2.2 CrossEntropyWithSoftmax. Let $g = \text{Softmax}(\vec{y})$ and $g_i = e^{y_i}/\sum_j e^{y_j}$. From our definition of SoftmaxCE, we have

$$\text{SoftmaxCE} = -\sum_i \log(\text{Softmax}(\vec{y})_i)\hat{y}_i = -\sum_i \log(g_i)\hat{y}_i.$$

So, $\frac{\partial \text{SoftmaxCE}}{\partial g_i} = -\frac{\hat{y}_i}{g_i}$. That is, we get $\frac{\partial \text{SoftmaxCE}}{\partial g} = -\frac{\hat{y}}{g}$. Hence,

$$\begin{aligned}\frac{\partial \text{SoftmaxCE}}{\partial \vec{y}} &= (\text{diag}(g) - gg^\top)\frac{\partial \text{SoftmaxCE}}{\partial g} \\ &= -\text{diag}(g)\frac{\hat{y}}{g} + gg^\top\frac{\hat{y}}{g} \\ &= -\hat{y} + g\sum_i \hat{y}_i = g - \hat{y} \\ &= \text{Softmax}(\vec{y}) - \hat{y}.\end{aligned}$$

A.2.3 Sigmoid. Let $g = \text{Sigmoid}(x) = \frac{e^x}{e^x+1}$. Then, we compute the derivative $\partial g/\partial x$,

$$\frac{\partial g}{\partial x} = \frac{e^x(e^x+1) - e^x e^x}{(e^x+1)^2} = g - g^2.$$

Similar to softmax, secure computation of $g - g^2$ can be achieved by combining Π_{LSig} and $\Pi_\times$. In our implementation, we use the loss of `BinaryCrossEntropyWithSigmoid`, which could be simplified.

A.2.4 BinaryCrossEntropyWithSigmoid. From the definition of `BinaryCrossEntropyWithSigmoid` (SigmoidBCE), we have

$$\begin{aligned}\text{SigmoidBCE} &= -\log(\text{Sigmoid}(y))\hat{y} - \log(1 - \text{Sigmoid}(y))(1-\hat{y}) \\ &= -\log(g)\hat{y} - \log(1-g)(1-\hat{y}).\end{aligned}$$

Taking derivative over g and then applying the chain rule,

$$\begin{aligned}\frac{\partial \text{SigmoidBCE}}{\partial g} &= -\frac{\hat{y}}{g} + \frac{1-\hat{y}}{1-g} \\ \frac{\partial \text{SigmoidBCE}}{\partial x} &= \frac{\partial \text{SigmoidBCE}}{\partial g} \cdot \frac{\partial g}{\partial x} = [-\frac{\hat{y}}{g} + \frac{1-\hat{y}}{1-g}](g - g^2) \\ &= (g-1)\hat{y} + (1-\hat{y})g = g - \hat{y} \\ &= \text{Sigmoid}(y) - \hat{y}.\end{aligned}$$

B THEORETICAL DERIVATION

Below are proofs of all theorems except security-related ones.

B.1 Proof of Theorem 4.4

Proof. The property $\vec{f}(0) = \vec{1}/m$ is trivial. We only prove that the function $\vec{f}_{\text{ode}}(t)$ satisfies the differential equation. Let $f_i(t)$ be

the i-th components of $\vec{f}(t)$, *i.e.*, $f_i(t) = \frac{e^{tx_i}}{\sum_j e^{tx_j}}$, we have

$$\begin{aligned} f_i'(t) &= \frac{d}{dt}\frac{e^{tx_i}}{\sum_j e^{tx_j}} = \frac{x_i e^{tx_i}(\sum_j e^{tx_j}) - e^{tx_i}\sum_j x_j e^{tx_j}}{(\sum_j e^{tx_j})^2} \\ &= x_i \frac{e^{tx_i}}{\sum_j e^{tx_j}} - \frac{e^{tx_i}}{\sum_j e^{tx_j}} \sum_j x_j \frac{e^{tx_j}}{\sum_k e^{tx_k}} \\ &= x_i f_i(t) - f_i(t)\langle x, \vec{f}(t)\rangle. \end{aligned}$$

Hence, we have $\vec{f}'_{\mathsf{ode}}(t) = (\vec{x} - \langle \vec{x}, \vec{f}(t)\rangle \vec{1}) * \vec{f}(t)$. For solving the ODE above, it is natural to use the Euler formula

$$y_0 = \vec{1}/m, \qquad y_{i+1} = y_i + (x - \langle x, y_i\rangle \vec{1}) * y_i / r,$$

where r is the number of iterations. Let the function $\mathsf{Softmax}_r$ be defined by y_r as above. We can see that $\mathsf{Softmax}_r$ is quasi-softmax. Moreover, $\mathsf{Softmax}_r$ converges to standard softmax as r tends to be infinity (see Theorem 4.5). We thus obtain iterative Softmax y_{t+1}. When computing $\mathsf{Softmax}_r(x)$, only addition and multiplication are required using the Euler formula. For model accuracy, we verify it by experimental results since we observe that the theoretical bound is too loose to capture practical performance. □

B.2 Proof of Theorem 4.5

Proof of Theorem 4.5. (a) For $t = 0, \ldots, r$, let y_t^i be the i-th component of y_t. We will prove $y_t^i \geq 0$ and $\sum_i y_t^i = 1$ for $t = 0, \ldots, r$. Obviously, $y_0^i \geq 0$ and $\sum_i y_0^i = 1$. Suppose $y_t^i \geq 0$ and $\sum_i y_t^i = 1$ for some t. $\sum_i y_{t+1}^i = \sum_i y_t^i + (\sum_i (x^i * y_t^i) - \langle x, y_t\rangle \sum_i y_t^i)/r = \sum_i y_t^i + \langle x, y_t\rangle(1 - \sum_i (y_t^i))/r = 1$, which can be written as $y_{t+1} = y_t * (\mathbf{1}_m + (x - \langle x, y_t\rangle \mathbf{1}_m)/r)$, where $\vec{1}$ denotes the m-dimensional vector $(1, \ldots, 1)^\top$, $\langle x, y\rangle$ denotes the inner product of vector x and y, and $*$ denotes element-wise multiplication. Let $u := (x - \langle x, y_t\rangle \vec{1})/r$. We have $\max_i u_i - \min_i u_i = ((\max_i (x_i) - \langle x, y_t\rangle) - (\min_i (x_i) - \langle x, y_t\rangle))/r$. That is, $(\max_i (x_i) - \min_i (x_i))/r \leq r/r = 1$. On the other hand, we have $-1 < \min(u) \leq \mathbb{E}_{y_t}(u) \leq \max(u) < 1$. Hence, $\vec{1} + u \geq 0 \Rightarrow y_{t+1} = y_t * (\vec{1} + u) \geq 0$.

(b) It is true for $t = 0$ because $y_0^i = \frac{1}{m}$ for all i. For mathematical induction, suppose $y_t^i \leq y_t^j$ if $x_i \leq x_j$ holds for any i, j. For any i, j s.t. $x_i \leq x_j$, we adopt symbols in the proof of (a). We have $u_i - u_j = (x_i - x_j)/r \leq 0$, hence $u_i \leq u_j$. From the proof of (a), $(\mathbf{1}_m + u) \geq 0$ and hence $0 \leq 1 + u_i \leq 1 + u_j$. On the other hand, $y_i \geq 0$ by conclusion (a) and $y_i \leq y_j$ by induction hypothesis. Therefore, $y_{t+1}^i = y_t^i(1 + u_i) \leq y_t^j(1 + u_j) = y_{t+1}^j$.

(c) Let $f : [0, 1] \to \mathbb{R}^m$ defined by $f(t) = \mathsf{Softmax}(tx)$. $f(0) = 1/m$, $f'(t) = (x - \langle x, f(t)\rangle \mathbf{1}_m) * f(t)$. By the Euler formula, $f(1) = \mathsf{Softmax}(x)$. □

B.3 Proof of Theorem 5.4

Proof of Theorem 5.4. In the fixed-point setting, for any n-bit value with f-bit fractional part, all values below are less than 2^{-f-1},

$$|\mathsf{FR}(\cos\frac{2k\pi t}{2^m}) - \cos\frac{2k\pi t}{2^m}|, |\mathsf{FR}(\sin\frac{2k\pi t}{2^m}) - \sin\frac{2k\pi t}{2^m}|,$$
$$|\mathsf{FR}(\sin\frac{2k\pi\delta_x}{2^m}) - \sin\frac{2k\pi\delta_x}{2^m}|, |\mathsf{FR}(\cos\frac{2k\pi\delta_x}{2^m}) - \cos\frac{2k\pi\delta_x}{2^m}|,$$

where $\delta_x = x - t$. By trigonometric (angle sum) identities, we have $\sin\frac{2k\pi(x-t)}{2^m} = \sin\frac{2k\pi\delta_x}{2^m}\cos\frac{2k\pi t}{2^m} + \cos\frac{2k\pi\delta_x}{2^m}\sin\frac{2k\pi t}{2^m}$.

Recall that $\langle u\rangle_1 = \mathsf{FR}(\sin\frac{2k\pi t}{2^m}) - \langle u\rangle_l$, $\langle v\rangle_1 = \mathsf{FR}(\cos\frac{2k\pi t}{2^m}) - \langle v\rangle_0$. For brevity, we drop the vector representation by the property of trigonometric functions for $|\sin x| \leq 1, |\cos x| \leq 1$, and also $|\mathsf{FR}(\cos x)| \leq 1 + 2^{-f-1}, |\mathsf{FR}(\sin x)| \leq 1 + 2^{-f-1}$. Since $\mathsf{FR}(\cdot)$ takes the floor function, we have a slightly tighter bound that $|\mathsf{FR}(\cos x)| \leq 1, |\mathsf{FR}(\sin x)| \leq 1$. Given the sine computation in protocol Π_{LSig}, we have,

$$\begin{aligned} & |\mathsf{Rec}(\langle\sin\tfrac{2k\pi x}{2^m}\rangle_0, \langle\sin\tfrac{2k\pi x}{2^m}\rangle_1) - \sin\tfrac{2k\pi x}{2^m}| \\ = \; & |(\langle\sin\tfrac{2k\pi x}{2^m}\rangle_0 + \langle\sin\tfrac{2k\pi x}{2^m}\rangle_1) - \sin\tfrac{2k\pi x}{2^m}| \\ = \; & |(\langle\sin\tfrac{2k\pi x}{2^m}\rangle_0 + \langle\sin\tfrac{2k\pi x}{2^m}\rangle_1) \\ & -(\sin\tfrac{2k\pi\delta_x}{2^m}\cos\tfrac{2k\pi t}{2^m} + \cos\tfrac{2k\pi\delta_x}{2^m}\sin\tfrac{2k\pi t}{2^m})| \\ = \; & |(\mathsf{FR}(\sin\tfrac{2k\pi\delta_x}{2^m})(\langle v\rangle_0 + \langle v\rangle_1) \\ & +\mathsf{FR}(\cos\tfrac{2k\pi\delta_x}{2^m})(\langle v\rangle_0 + \langle v\rangle_1)) \\ & -(\sin\tfrac{2k\pi\delta_x}{2^m}\cos\tfrac{2k\pi t}{2^m} + \cos\tfrac{2k\pi\delta_x}{2^m}\sin\tfrac{2k\pi t}{2^m})| \\ = \; & |(\mathsf{FR}(\sin\tfrac{2k\pi\delta_x}{2^m})(\langle v\rangle_0 + \mathsf{FR}(\cos\tfrac{2k\pi t}{2^m}) - \langle v\rangle_0)) \\ & +\mathsf{FR}(\cos\tfrac{2k\pi\delta_x}{2^m})(\langle u\rangle_0 + \mathsf{FR}(\sin\tfrac{2k\pi t}{2^m}) - \langle u\rangle_0) \\ & -(\sin\tfrac{2k\pi\delta_x}{2^m}\cos\tfrac{2k\pi t}{2^m} + \cos\tfrac{2k\pi\delta_x}{2^m}\sin\tfrac{2k\pi t}{2^m})| \\ = \; & |\mathsf{FR}(\sin\tfrac{2k\pi\delta_x}{2^m})\mathsf{FR}(\cos\tfrac{2k\pi t}{2^m}) \\ & +\mathsf{FR}(\cos\tfrac{2k\pi\delta_x}{2^m})\mathsf{FR}(\sin\tfrac{2k\pi t}{2^m}) \\ & -(\sin\tfrac{2k\pi\delta_x}{2^m}\cos\tfrac{2k\pi t}{2^m} + \cos\tfrac{2k\pi\delta_x}{2^m}\sin\tfrac{2k\pi t}{2^m})| \\ = \; & |\mathsf{FR}(\sin\tfrac{2k\pi\delta_x}{2^m})(\mathsf{FR}(\cos\tfrac{2k\pi t}{2^m}) - \cos\tfrac{2k\pi t}{2^m} + \cos\tfrac{2k\pi t}{2^m}) \\ & +\mathsf{FR}(\cos\tfrac{2k\pi\delta_x}{2^m})(\mathsf{FR}(\sin\tfrac{2k\pi t}{2^m}) - \sin\tfrac{2k\pi t}{2^m} + \sin\tfrac{2k\pi t}{2^m}) \\ & -(\sin\tfrac{2k\pi\delta_x}{2^m}\cos\tfrac{2k\pi t}{2^m} + \cos\tfrac{2k\pi\delta_x}{2^m}\sin\tfrac{2k\pi t}{2^m})| \\ = \; & |\mathsf{FR}(\sin\tfrac{2k\pi\delta_x}{2^m})(\mathsf{FR}(\cos\tfrac{2k\pi t}{2^m}) - \cos\tfrac{2k\pi t}{2^m}) \\ & +\mathsf{FR}(\sin\tfrac{2k\pi\delta_x}{2^m})\cos\tfrac{2k\pi t}{2^m} \\ & +\mathsf{FR}(\cos\tfrac{2k\pi\delta_x}{2^m})(\mathsf{FR}(\sin\tfrac{2k\pi t}{2^m}) - \sin\tfrac{2k\pi t}{2^m}) \\ & +\mathsf{FR}(\cos\tfrac{2k\pi\delta_x}{2^m})\sin\tfrac{2k\pi t}{2^m} \\ & -(\sin\tfrac{2k\pi\delta_x}{2^m}\cos\tfrac{2k\pi t}{2^m} + \cos\tfrac{2k\pi\delta_x}{2^m}\sin\tfrac{2k\pi t}{2^m})| \\ = \; & |\mathsf{FR}(\sin\tfrac{2k\pi\delta_x}{2^m})(\mathsf{FR}(\cos\tfrac{2k\pi t}{2^m}) - \cos\tfrac{2k\pi t}{2^m}) \\ & +\mathsf{FR}(\cos\tfrac{2k\pi\delta_x}{2^m})(\mathsf{FR}(\sin\tfrac{2k\pi t}{2^m}) - \sin\tfrac{2k\pi t}{2^m}) \\ & +\mathsf{FR}(\sin\tfrac{2k\pi\delta_x}{2^m})\cos\tfrac{2k\pi t}{2^m} + \mathsf{FR}(\cos\tfrac{2k\pi\delta_x}{2^m})\sin\tfrac{2k\pi t}{2^m} \\ & -\sin\tfrac{2k\pi\delta_x}{2^m}\cos\tfrac{2k\pi t}{2^m} - \cos\tfrac{2k\pi\delta_x}{2^m}\sin\tfrac{2k\pi t}{2^m}| \\ = \; & |\mathsf{FR}(\sin\tfrac{2k\pi\delta_x}{2^m})(\mathsf{FR}(\cos\tfrac{2k\pi t}{2^m}) - \cos\tfrac{2k\pi t}{2^m}) \\ & +\mathsf{FR}(\cos\tfrac{2k\pi\delta_x}{2^m})(\mathsf{FR}(\sin\tfrac{2k\pi t}{2^m}) - \sin\tfrac{2k\pi t}{2^m}) \\ & +\cos\tfrac{2k\pi t}{2^m}(\mathsf{FR}(\sin\tfrac{2k\pi\delta_x}{2^m}) - \sin\tfrac{2k\pi\delta_x}{2^m}) \\ & +\sin\tfrac{2k\pi t}{2^m}(\mathsf{FR}(\cos\tfrac{2k\pi\delta_x}{2^m}) - \cos\tfrac{2k\pi\delta_x}{2^m})|. \end{aligned}$$

The above can be bounded by:

$$\begin{aligned} \leq \; & |\mathsf{FR}(\sin\tfrac{2k\pi\delta_x}{2^m})(\mathsf{FR}(\cos\tfrac{2k\pi t}{2^m}) - \cos\tfrac{2k\pi t}{2^m})| \\ & +|\mathsf{FR}(\cos\tfrac{2k\pi\delta_x}{2^m})(\mathsf{FR}(\sin\tfrac{2k\pi t}{2^m}) - \sin\tfrac{2k\pi t}{2^m})| \\ & +|\cos\tfrac{2k\pi t}{2^m}(\mathsf{FR}(\sin\tfrac{2k\pi\delta_x}{2^m}) - \sin\tfrac{2k\pi\delta_x}{2^m})| \\ & +|\sin\tfrac{2k\pi t}{2^m}(\mathsf{FR}(\cos\tfrac{2k\pi\delta_x}{2^m}) - \cos\tfrac{2k\pi\delta_x}{2^m})| \\ \leq \; & 1|\mathsf{FR}(\cos\tfrac{2k\pi t}{2^m}) - \cos\tfrac{2k\pi t}{2^m}| + 1|\mathsf{FR}(\sin\tfrac{2k\pi t}{2^m}) - \sin\tfrac{2k\pi t}{2^m}| \\ & +1 \cdot |\mathsf{FR}(\sin\tfrac{2k\pi\delta_x}{2^m}) - \sin\tfrac{2k\pi\delta_x}{2^m}| \\ & +1 \cdot |\mathsf{FR}(\cos\tfrac{2k\pi\delta_x}{2^m}) - \cos\tfrac{2k\pi\delta_x}{2^m}| \\ \leq \; & 2^{-f-1} + 2^{-f-1} + 2^{-f-1} + 2^{-f-1} \\ \leq \; & 2^{-f+1}. \end{aligned}$$

The bound of correctness for Π_{LSig} holds under the assumption of correct truncation. If more than zero error happens for truncation, the difference between Π_{LSig} and standard sigmoid is not

Algorithm 4 $\mathcal{F}_{\mathsf{LSig}}$: Ideal Functionality of Π_{LSig}

Require: The functionality receives $\langle x\rangle_0, \langle x\rangle_1, t, \vec{k}, \alpha, \vec{\beta}$
Ensure: $\langle \mathsf{LSig}(x)\rangle_0, \langle \mathsf{LSig}(x)\rangle_1$
1: Reconstruct x
2: Compute $\delta_x = (x - t) \bmod 32$
3: Compute $y = \alpha + \vec{\beta}(\sin(\delta_x\vec{k}) * \cos(t\vec{k}) + \cos(\delta_x\vec{k}) * \sin(t\vec{k}))$
4: Generate random shares of y as the functionality outputs

Algorithm 3 $\mathcal{F}_{\mathsf{QSMax}}$: Ideal Functionality of Π_{QSMax}

Require: The functionality receives $\langle \vec{x}\rangle_0, \langle \vec{x}\rangle_1, r, \vec{g}(0) = \vec{1}/m$.
Ensure: $\langle \mathsf{QSMax}(\vec{x})\rangle_0, \langle \mathsf{QSMax}(\vec{x})\rangle_1$
1: Reconstruct x and compute x/r
2: **for** $i = 1, 2, \ldots, r$ **do**
3: Compute $q = \langle \vec{x}, \vec{g}(\frac{i-1}{r})\rangle$ and set $\vec{q} = q \cdot \vec{1}$
4: Compute $\vec{g}(\frac{i}{r}) = \vec{g}(\frac{i-1}{r}) + (\vec{x} - \vec{q}) * \vec{g}(\frac{i-1}{r})$
5: **end for**
6: Generate random shares of $\vec{g}(1)$ as the functionality outputs

bounded. Π_{LSig} utilizes probabilistic truncation in a black-box manner, which can be easily substituted by any faithful truncation, so we do not consider this probability in our analysis. Assuming the correct truncation with an ideal probability of 100%, the difference $|\mathsf{Rec}(\langle \mathsf{LSig}(x)\rangle_0, \langle \mathsf{LSig}(x)\rangle_1) - \mathsf{Sigmoid}(x)|$ is bounded if and only if $|\mathsf{Rec}(\langle \sin\frac{2k\pi x}{2^m}\rangle_0, \langle \sin\frac{2k\pi x}{2^m}\rangle_1) - \sin\frac{2k\pi x}{2^m}|$ is bounded. For brevity, we take the bounding difference 2^{-f+1}. For the difference of $\mathsf{Rec}(\langle \mathsf{LSig}(x)\rangle_0, \langle \mathsf{LSig}(x)\rangle_1)$ and standard sigmoid:

$$
\begin{aligned}
&|\mathsf{Rec}(\langle \mathsf{LSig}(x)\rangle_0, \langle \mathsf{LSig}(x)\rangle_1) - \mathsf{Sigmoid}(x)| \\
\le\ & |\sum_{k=1}^{K} a_k \mathsf{Rec}(\langle \sin\frac{2k\pi x}{2^m}\rangle_0, \langle \sin\frac{2k\pi x}{2^m}\rangle_1) - \sum_{k=1}^{K} a_k \sin\frac{2k\pi x}{2^m}| \\
\le\ & |\sum_{k=1}^{K} a_k (\mathsf{Rec}(\langle \sin\frac{2k\pi x}{2^m}\rangle_0, \langle \sin\frac{2k\pi x}{2^m}\rangle_1) - \sum_{k=1}^{K} a_k \sin\frac{2k\pi x}{2^m})| \\
\le\ & \sum_{k=1}^{K} |a_k| |\mathsf{Rec}(\langle \sin\frac{2k\pi x}{2^m}\rangle_0, \langle \sin\frac{2k\pi x}{2^m}\rangle_1) - \sin\frac{2k\pi x}{2^m}| \\
\le\ & \sum_{k=1}^{K} |a_k| \cdot 2^{-f+1}.
\end{aligned}
$$

This concludes the proof of (i). When K is sufficiently large, $\epsilon \to 0$. The proof of (ii) is straightforward. □

Remark. The probability that the difference for sigmoid is bounded is $(1-p)^4$ if Π_{LSig} adopts SecureML's truncation [22], where p is the probability for error truncation. In our case, we can just take $l = n$ and $l_x = \lceil \log 2(|x|)\rceil + f$ to get probability p.

C SECURITY ANALYSIS

Our proof is under the real-world execution and ideal-world simulation paradigm. Consider adversary $\mathcal{A}$, environment $\mathcal{Z}$, and simulator $\mathcal{S}$. For real-world execution, the parties execute the protocol in the presence of adversary $\mathcal{A}$ and environment $\mathcal{Z}$. In the ideal world, the parties send their inputs to simulator $\mathcal{S}$ that computes the functionality $\mathcal{F}$ truthfully. Security requires that for every $\mathcal{A}$ in the real interaction, $\mathcal{S}$ exists in the ideal interaction such that environment $\mathcal{Z}$ cannot distinguish the real world from the ideal world. That is, whatever knowledge $\mathcal{A}$ can extract in the real world, $\mathcal{S}$ can also extract it in the ideal world. Algorithms 3 and 4 define the ideal functionality of Π_{QSMax} and Π_{LSig}, respectively.

C.1 Proof of Theorem 4.2

Proof. Recall that Π_{QSMax} calls $\Pi_\times$ twice. Π_{QSMax} sequentially integrates $\Pi_\times$ with local computations. Consider adversary $\mathcal{A}$, environment $\mathcal{Z}$, and simulator $\mathcal{S}$. In the real-world execution, the parties execute Π_{QSMax} and communicate with adversary $\mathcal{A}$ in environment $\mathcal{Z}$. In the ideal world, simulator $\mathcal{S}$ computes functionality $\mathcal{F}$ truthfully by the inputs forwarded by the parties.

Π_{QSMax} sequentially composes two invocations of $\Pi_\times$, and each call is independent. If $\Pi_\times$ is secure against semi-honest PPT adversaries with static corruption, and a semi-honest adversary $\mathcal{A}$ can attack Π_{QSMax}, $\Pi_\times$ can be attacked successfully. However, this contradicts the semi-honest security of $\Pi_\times$. Thus, for every adversary $\mathcal{A}$ in the real interaction, environment $\mathcal{Z}$ cannot distinguish the execution between simulator $\mathcal{S}$ and $\mathcal{A}$. □

C.2 Proof of Theorem 5.2

Proof. We construct Π_{LSig} from sketch, requiring slightly different offline randomness from prior literature. In the ideal world, the parties communicate with the ideal functionality $\mathcal{F}_{\mathsf{LSig}}$. We aim to prove that no PPT environment $\mathcal{Z}$ can distinguish between the real execution Π_{LSig} and the ideal execution. The two parties input $\langle x\rangle_0, \langle x\rangle_1$ to $\mathcal{F}_{\mathsf{LSig}}$. Upon receiving the input, Π_{LSig} reconstructs $x = \langle x\rangle_0 + \langle x\rangle_1$ and computes δ_x. Given $t, \vec{k}, \delta_x$, Π_{LSig} computes $\sin(\delta_x\vec{k}), \cos(t\vec{k}), \cos(\delta_x\vec{k})$, and $\sin(t\vec{k})$. $\mathcal{F}_{\mathsf{Sin}}$ constructs y and generates the shares as the functionality outputs.

P_0 and P_1 play symmetric roles. Suppose P_0 is corrupt and P_1 is honest. $\mathcal{S}$ simulates the interface of $\mathcal{F}_{\mathsf{LSig}}$ as well as honest P_1. $\mathcal{S}$ picks random $\delta_{\langle x\rangle_1}$ and sends $\delta_{\langle x\rangle_1}$ to P_0 on behalf of P_1.

We consider game G_1, which is equivalent to real protocol execution G_0, except that P_1 sends a random $\tilde{\delta}_{\langle x\rangle_1}$ to P_0 instead of $\delta_{\langle x\rangle_1}$. Without $\langle x\rangle_0$, $\delta_{\langle x\rangle_1}$ follows a uniformly random distribution. So, $G_1 \approx G_0$. As the value P_0 sent in G_1 is truly random, the adversary cannot have any non-negligible advantage. □