Secure Softmax/Sigmoid for Machine-Learning Computation

Yu Zheng[*‡] Chinese U. of Hong Kong
Qizhi Zhang[*] Morse Team, Ant Group
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) with 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 ABY 2.0 (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, when compared with prior arts, shortening the private training process with much less communication than state-of-the-art frameworks, namely, CryptGPU (S&P '21), Piranha (Usenix Security '22), and quantized training from MP-SPDZ (ICML '22), while maintaining competitive accuracy.

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 *Proceedings of The 39th Applied Computer Security Applications Conference (ACSAC '2023).* ACM, New York, NY, USA, 17 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Advances in machine learning techniques, notably neural networks, attain great performance by training with 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 [22, 27, 28], private training poses unique challenges [14, 21, 32, 35]. It produces fluctuating computation results and crucially relies on non-linear layers. In general, cryptographic techniques such as secure multi-party computation (MPC) 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). MPC 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 VGG16 (138M parameters) [30]. However, large communication overhead persists as a major concern. Notably, in a WAN setting, over 94% of the training time of Piranha 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 (GC) leads to large circuit sizes, while using oblivious transfer (OT) [25–28] or function secret sharing (FSS) [11, 29] requires extensive communication. Existing works [15, 25, 33, 34] either approximate e^x, $1/x$, or $\sum e^x$ separately, or replace them with polynomial approximations. A recent SoK [23] 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

Our work proposes new secure protocols for softmax and sigmoid with more effective approximations by a holistic approach integrating scientific computing, machine learning, and cryptographic techniques. We explore a rarely-explored approach to numerical approximations through ordinary differential equations and Fourier series. In more detail, we define "quasi-softmax" to capture the essential characteristics regarding the probability distribution of softmax's outputs. We then employ an ordinary differential equation to solve quasi-softmax. Secure computation of its solution only requires addition and multiplication. Our secure protocol for sigmoid computes its approximation via the Fourier series. With the help of trigonometric identities, similar to our protocol for softmax, it only takes addition and multiplication. Both protocols avoid computing intermediate values resulting from exact computation of unbounded e^x and $1/x$ but directly approximate bounded

[*]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 Ant Group.

ACSAC '2023, December 04–08, 2023, Austin, Texas, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/23/12…$15.00
https://doi.org/XXXXXXX.XXXXXXX

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 [25]	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+

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.

softmax/sigmoid. We thus largely reduced the communication complexity as in Table 1.

1.1.1 ODE-Iterative Softmax. For softmax, exact computation [15] requires the interplay of various MPC protocols for computing maximum (i.e., comparison), exponentiation, and division. Approximated softmax (ASM) [6, 33, 34] replaces the exponential function in softmax with ReLU, which crucially relies on manual efforts in tuning the model towards a satisfying accuracy [14].

We consider the softmax computation in training as a whole with an accurate approximation from a numerical perspective. Specifically, we propose a new notion of quasi-softmax (QSMax) for crypto-friendly computation. QSMax solves the initial value problem (IVP) for the ordinary differential equation (ODE) via the Euler formula.

Our protocol Π_{QSMax} improves efficiency (Table 1) by avoiding running expensive secure comparison and division. Π_{QSMax} takes a constant round and is superior to solutions [33, 34] that directly replace softmax with ASM. Additionally, the Π_{QSMax} removes the CryptGPU [32]'s assumption of guessing initials, which may affect the accuracy of using the Newton-Raphson algorithm to approximate the value of $1/y$.

1.1.2 Fourier-Series-Approximated Sigmoid. Earlier secret-sharing frameworks resort to piecewise linear approximation [20, 25], which requires relatively heavy comparison to identify pieces. Approximating via Chebyshev polynomial [5, 6], on the other hand, may result in a gradient explosion that risks destroying the model.

We define the "local sigmoid" for expressing the significance of accurate sigmoid when the inputs are near to 0. We propose to approximate sigmoid by Fourier series (FS) in our secret-shared protocol Π_{LSig}. From trigonometric identities, Π_{LSig} only requires local computation of $\sin x$, $\cos x$, $\sin y$, and $\cos y$ with only secret-shared multiplication. Our approximation is more accurate than piecewise linear function [11, 21, 25]. Meanwhile, gradient explosion caused by unbounded errors of the polynomial fitting (faced by TFE [6] and Rosetta [5]) can be avoided.

Π_{LSig} only requires constant online communication costs with 1-round (Table 1), independent of the number of FS terms. Broadly, we extend MPC-friendly functions to composition rings (with trigonometric polynomials), i.e., approximating other bounded functions derived from trigonometric polynomials.

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-classification, 100-classification, and 1000-classification, respectively. Meanwhile, it demonstrates a 10× speed-up in running time. Previous attempts require 170 − 704 rounds of online communication, whereas Π_{QSMax} consumes constant rounds, i.e., 16 or 32 in experiments. For sigmoid, Π_{LSig} reduces online communication from $\gg 100$ bits to 36 bits and speeds up 570× computation. Currently, Π_{LSig} is the only lightweight (OT/FSS-free) solution that simultaneously achieves bounded error and 1-round communication.

1.2.2 Accuracy and Efficiency for Model Training. we systematically evaluate AlexNet [17], LeNet [18], VGG16 [30], ResNet [12], and four relatively small networks [33] over multiple standard benchmark datasets. As in Section 7.5, experimental results show a reduction of 57%-77% in communication compared with state of the arts [32, 34, 35]. For testing accuracy in Section 7.4, we consider the training-from-sketch setting of Piranha [35] from randomized initialization without any pertaining, and the pre-trained setting in CryptGPU [32] that trains with a pre-trained model. We attain higher accuracy than Piranha, e.g., 52.1% vs. 40.7% for AlexNet and 59.4% vs. 58.7% for VGG16. Our established model can reach an almost identical or slightly higher accuracy for a relatively small network than Keller and Sun's quantized training using MP-SPDZ (hereinafter referred to as "SPDZ-QT") [14] with fewer training epochs. Also, faster convergence means less communication and rounds, approximately reducing $10/15 \approx 67\%$ communication.

1.2.3 Practical Deployment in LAN and WAN. For realistic deployment, we simulate real-world data transmission (online and offline) under varying environmental conditions, i.e., LAN and WAN. In a LAN setting, our optimization in Piranha achieves a 10% − 60% speed-up, as elaborated in Section 7.6. Section 7.7 shows the training in the WAN with limited bandwidth and different time latency. For LeNet and AlexNet, our work trains 56% − 78% faster than Piranha in WAN.

1.2.4 Python & C++ Implementations. Currently, there are two series of implementations, one in Python (e.g., [15, 32]) while the other in C++ (e.g., [33, 35]).[1] We implemented our protocols in both languages, as detailed in Section 6.3. To our knowledge, our Python version is the first TensorFlow-style framework for cryptographic training.

[1]SecureML [21] is implemented in C++. CrypTen [15] is a popular framework built on PyTorch, without optimizing protocols. Two recent GPU frameworks, CryptGPU [32] and Piranha [35], are implemented in PyTorch and C++, respectively.

2 RELATED WORKS

Cryptographic Machine Learning. Compared with ML or AI flourish (e.g., ChatGPT) in plaintext, there is still a long run to achieve similar training performance in private learning. A recent SoK [23] dissects cryptographic techniques and "relationships" with machine learning alongside a genealogy highlighting noteworthy developments. For private training, SecureML [21] reaches an online training on MNIST at 93% accuracy. Later, with the assistance of plaintext pretraining, CryptGPU [32] integrates more advanced ML techniques, e.g., BatchNorm and softmax, and achieves 83.7% accuracy on CIFAR-10 with VGG-16. Piranha [35] provides a platform for accelerating secret sharing-based MPC protocols using GPUs. Piranha achieves a 58.7% accuracy over CIFAR-10 with VGG-16, by training from scratch, i.e., from a randomly initialized model without any pretraining. Nevertheless, the huge communication overhead poses a bottleneck. For non-linear computation, the computational complexity increases largely, finally affecting the training time and communication overhead. Finally, we remark that differential privacy [9, 36] is an orthogonal approach, which could be integrated [39] for mitigating advanced attacks.

Softmax. CrypTen [15] tries exact computation on softmax, which dedicates 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 offline 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 $\mathsf{Softmax}(\vec{x})_i = \exp(x_i - x_{\max})/(\sum_i \exp(x_i - x_{\max}))$. We call this variant ComDiExp for combining division and exponentiation. The major concern of ComDiExp 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. Simultaneously, Keller and Sun [14] question the efficiency gains of ASM and optimize exponentiation with base two. Unlike the frameworks above, our protocol for softmax aims for constant rounds and low communication costs.

Sigmoid. SecureML [21], ABY series [20, 25], and Squirrel [19] approximate sigmoid by a piecewise function, with bounded error and thus without experiencing exploding gradients. Yet, they dedicate $O(\log n)$ comparison rounds for locating pieces. Chebyshev polynomial [5, 6] could provide a satisfying approximation in $[-4, 4]$, but may risk gradient explosion (i.e., inaccurate approximation) and even destroy the training model. The root cause is 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 of the attempts above either require > 1 round of communication or are lost in 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 $\mathsf{P}_0, \mathsf{P}_1$ who perform secure computation, assisted by an MPC-service provider (or commodity server [31]) T. T only generates the required randomness in the offline phase, letting $\mathsf{P}_0, \mathsf{P}_1$ perform relatively cheap online computation. We consider semi-honest probabilistic polynomial time (PPT) adversaries (cf., [35]) 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 nor the program being executed by 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 p-bit fractional part. $\langle x\rangle_0$ and $\langle x\rangle_1$ denote the shares of fixed-pointed representation x such that $\langle x\rangle_0 + \langle x\rangle_1 \bmod 2^n = x \cdot 2^p$. Particularly, P_0 owns the shares in the form of $\langle\cdot\rangle_0$, while P_1 owns the shares in the form of $\langle\cdot\rangle_1$. For achieving secret-shared multiplication ($\Pi_\times$), we generate shares using pseudorandom function PRF (e.g., [24]). In offline phase, T, P_0 generate randomness $\langle u\rangle_0, \langle v\rangle_0, \langle z\rangle_0$ via $\mathsf{PRF}_0(\mathsf{key}_0)$, while T, P_1 generate random numbers $\langle u\rangle_1, \langle v\rangle_1$ using $\mathsf{PRF}_1(\mathsf{key}_1)$. Later, T computes $\langle z\rangle_1 = uv - \langle z\rangle_0$, and sends $\langle z\rangle_1$ to P_1. In the online phase, P_0 computes $\delta_{\langle x\rangle_0} = \langle x\rangle_0 - \langle u\rangle_0$ and $\delta_{\langle y\rangle_0} = \langle y\rangle_0 - \langle v\rangle_0$, later sending $\delta_{\langle x\rangle_0}, \delta_{\langle y\rangle_0}$ to party P_1. Similarly, P_1 computes $\delta_{\langle x\rangle_1}, \delta_{\langle y\rangle_1}$ and sends them to P_0. Then, both $\mathsf{P}_0, \mathsf{P}_1$ can construct δ_x and δ_y. Thus, P_0 and P_1 can collaboratively compute $xy = \delta_x(\langle y\rangle_0 + \langle y\rangle_1) + (\langle u\rangle_0 + \langle u\rangle_1)\delta_y + (\langle z\rangle_0 + \langle z\rangle_1)$. For mitigating overflow, we perform truncation operation [21] after every multiplication to maintain the p-bit fractional part of the result.

4 SECURE SOFTMAX COMPUTATION

Achieving secret-shared softmax in the fixed-point setting is challenging since e^x and $1/x$ are easy to overflow and heavyweight in MPC computation. Our goal is to find an MPC-friendly softmax, which 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. We formulate the notion of Quasi-Softmax, capturing the essence of the probability distribution outputted by the softmax function.

Later, we establish an m-dimensional ordinary differential equation (ODE) with softmax-like outputs by transforming a multivariable function into a single-variable function. Atop it, the problem of approximating the softmax function is formulated as an initial value problem (IVP) whose solution evaluated at the terminal point gives the 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 MPC-efficient multiplication only. Each iteration's output gradually approaches the original distribution provided by the real softmax.

4.1 New Notion of Quasi-Softmax

Recall that the standard $\mathsf{Softmax}(\vec{x})_i = e^{x_i}/\sum_j e^{x_j} : \mathbb{R}^m \to (0,1)^m$, where m is the dimensionality of the input. 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. Concerning our observation, the probability distribution over the outputs dominates the model accuracy instead of absolute values. Accordingly, we propose a new

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

notion of quasi-softmax (QSMax) for acting like the real softmax from the probability perspective.

Definition 4.1 (Quasi-Softmax). We define a function $f : \mathbb{R}^m \to [0,1]^m$ to be a quasi-Softmax such that:
(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 $i, j = 1, \ldots, m$.

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 regarding $\vec{x}$. We consider the output domain to be $[0,1]^m$, which adds the boundary $\{0,1\}$ to the original domain $(0,1)^m$ of real Softmax. In Definition 4.1, property (i) captures the range defined over the output domain. To be specific, the value of $e^{x_i}/\sum_j e^{x_j}$ is always non-negative, and meantime the value of $\sum_i e^{x_i}/\sum_j e^{x_j}$ is always equal to 1. The property (ii) abstracts the input-output monotonous relation, informally, a larger input causing a larger output. Prior approximation resorts to ASM approximation, which is essentially an instantiation of QSMax. The $\mathsf{ASM}(\vec{x})_i = \mathsf{ReLU}(x_i)/\sum_j \mathsf{ReLU}(x_j)$ has the similar form of real Softmax. Property (i) and Property (ii) can be verified similarly.

The ideal instantiation of $f(\vec{x})$ should be reasonably represented via fixed-point numbers. We first transform the multi-variable function $f(\vec{x})$ to be a single-variable function $\vec{\mathrm{g}}(\frac{i}{r})$, which also outputs an m-dimensional vector $f(\vec{x})$. Here, i is the independent variable for $\vec{\mathrm{g}}(\frac{i}{r})$. We found the QSMax $\vec{\mathrm{g}}(\cdot)$, explained as the pseudo-code below. [2] The r, $\frac{1}{r}$ are the preset hyper-parameters, meanwhile the r represents the number of for-loop iterations.

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

[2]Section 4.4.1 discusses mathematical intuition for replacing $\mathsf{Softmax}(\vec{x})$ with $\vec{\mathrm{g}}(\cdot)$

At the start, we fix $\vec{\mathrm{g}}(0) = [1/m, \ldots, 1/m]$. Given the real softmax's input $\vec{x}$, we can get $\vec{\mathrm{g}}(\frac{1}{r}) = \vec{\mathrm{g}}(0) + (\vec{x} - \langle\vec{x}, \vec{\mathrm{g}}(0)\rangle\vec{1}) * \vec{\mathrm{g}}(0)\cdot\frac{1}{r}$. Here, $\langle\vec{x}, \vec{\mathrm{g}}(0)\rangle$ denotes the inner product of $\vec{x}$ and $\vec{\mathrm{g}}(0)$, while $*$ means the element-wise multiplication over vectors. Given $\vec{\mathrm{g}}(\frac{1}{r})$, we then can get $\vec{\mathrm{g}}(\frac{2}{r}), \vec{\mathrm{g}}(\frac{3}{r})$ and so on. After $i = r$ iteration steps, we have $\vec{\mathrm{g}}(1) = \vec{\mathrm{g}}(\frac{r}{r})$, which replaces the output of the real softmax.

4.2 Protocol for Quasi-Softmax

With the pseudo-code of computing $\vec{\mathrm{g}}(\frac{i}{r})$, we propose Π_{QSMax} (Protocol 1), in which party P_0 and party P_1 iteratively construct shares of $\vec{\mathrm{g}}(\frac{1}{r}), \vec{\mathrm{g}}(\frac{2}{r}), \ldots, \vec{\mathrm{g}}(1)$. After r iterations, Π_{QSMax} outputs shares of $\vec{\mathrm{g}}(1)$, used as outputs of the real softmax.

In Line 2 and Line 3, the P_0 and P_1 locally process their initial values before iterations. After representing $\frac{1}{r}$ to be the fixed-point number $\mathsf{FR}(\frac{1}{r})$, both of them multiply this public scalar to their vector-valued shares $\langle\vec{x}\rangle_0, \langle\vec{x}\rangle_1$, respectively. Processing $\langle\vec{x}\rangle_0, \langle\vec{x}\rangle_1$ corresponds to the rightest side of solving $\vec{\mathrm{g}}(\frac{i}{r})$, i.e., $\cdot\frac{1}{r}$ in Definition 4.1. Notably, we put the operation $\cdot\frac{1}{r}$ outside the loop due to saving computation, i.e., no need to multiply $\cdot\frac{1}{r}$ at each iteration. The initial value $\vec{\mathrm{g}}(0) = \vec{1}/m$ is fixed for any inputs $\vec{x}, r$; Thus, the $\mathsf{P}_0, \mathsf{P}_1$ can trivially set $\langle\vec{\mathrm{g}}(0)\rangle_0 = \vec{1}/m, \langle\vec{\mathrm{g}}(0)\rangle_1 = \vec{0}$.

From Line 6 to Line 8, secure inner product $\langle\vec{x}, \vec{\mathrm{g}}(\frac{i-1}{r})\rangle\cdot\vec{1}$ is achieved, where $\cdot\vec{1}$ is a vectorized transformation. The P_0 and P_1 execute secret-shared multiplication $\Pi_\times$, explained in Section 3.2, for element-wise multiplication between two vectors. Then, the P_0 adds all entries of $\langle\vec{o}\rangle_0$ and multiplies the result to $\vec{1}$ for transforming the scalar into a vector. Similarly, P_1 constructs the vector $\langle\vec{q}\rangle_1$.

From Line 9, P_0 and P_1 obtain the shares of $\langle\vec{x}, \vec{\mathrm{g}}(\frac{i-1}{r})\rangle\vec{1}) * \vec{\mathrm{g}}(\frac{i-1}{r})$. Combining with Line 10 and Line 11, P_0 and P_1 achieve secure computation of $(\vec{x} - \langle\vec{x}, \vec{\mathrm{g}}(\frac{i-1}{r})\rangle\vec{1}) * \vec{\mathrm{g}}(\frac{i-1}{r})$. Remember that multiplying $\frac{1}{r}$ is done before the loop. In Line 12 and Line 13, P_0 and P_1 locally compute the shares of $\vec{\mathrm{g}}(\frac{i}{r})$ at the i-th iteration. At last, P_0 and P_1 obtain the shares of $\vec{\mathrm{g}}(1)$. Theorem 4.2 asserts the semi-honest security of Π_{QSMax} with security proof deferred to Appendix C.1.

THEOREM 4.2 (SECURITY FOR Π_{QSMax}). *The protocol Π_{QSMax} realizes the functionality $\mathcal{F}_{\mathsf{QSMax}}(\vec{x})$ against semi-honest PPT adversaries with static corruption.*

4.3 Communication Complexity

Theorem 4.3 summarizes communication costs of Π_{QSMax}. Recall $\Pi_\times$ (Section 3.2) that P_0 sends the 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 every iteration. Thus, the number of communication rounds is $2r$ for r iterations. Notably, the number of communication rounds remains constant when the input dimension m is enlarged.

During the first call of $\Pi_\times$, P_0 sends two masked values for $\langle\vec{x}\rangle_0, \langle\vec{\mathrm{g}}(\frac{(i-1)}{r})\rangle_0$ and P_1 sends two masked values for $\langle\vec{x}\rangle_1, \langle\vec{\mathrm{g}}(\frac{i-1}{r})\rangle_1$. The communication costs of m-dimensional element-wise multiplication are $4mn$ bits for n-bit data. In the second call of $\Pi_\times$, masking $\langle\vec{\mathrm{g}}(\frac{(i-1)}{r})\rangle_0, \langle\vec{\mathrm{g}}(\frac{(i-1)}{r})\rangle_1$ 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 in the same scalar; thus, two parties can send the masked value of one entry only. Therefore, online communication costs for r iterations require $(6mn + 2n)r$ bits. In the offline phase, the communication costs take $2mn$ bits by following $\Pi_\times$ twice at every iteration. Taking r iterations into account, we get overall offline communication to be $2mnr$ bits.

Theorem 4.3. *Let x be the n-bit fixed-point number, m be the dimension of input vectors, and r be the number of iterations. Securely evaluating $\vec{\mathrm{g}}(\frac{i}{r})$ via protocol 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 $\vec{f}(t) = \mathrm{softmax}(t\vec{x})$, for transforming the multi-variable softmax to be a single-variable function $\vec{f}(t)$. Specifically, on the input of vector $\vec{x}$ of m dimensions, $\vec{f}$ outputs an m-dimensional vector element-wise to the input. When $t = 1$, we can get the real softmax $\vec{f}(1) = \mathrm{QSMax}(\vec{x})$.

We observe that $\vec{f}(t) = \mathrm{QSMax}(t\vec{x})$ satisfies an ordinary differential equation (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 $\vec{f}(t) = \mathrm{Softmax}(t\vec{x}) : [0, 1] \longrightarrow (0, 1)^m$. We have $\vec{f}(0) = \vec{1}/m$ and $\vec{f}'(t) = (\vec{x} - \langle\vec{x}, \vec{f}(t)\rangle\vec{1}) * \vec{f}(t)$.*

To be specific, $\vec{f}(t)$ builds on the initial value problem for the ordinary differential equation. To solve ODE, we can use the Euler formula for r iterations. Actually, $\vec{\mathrm{g}}(\frac{i}{r})$ is an iterative solution of $\vec{f}(t)$. We can see that the process of computing $\vec{\mathrm{g}}(\frac{i}{r})$ is MPC-friendly for obtaining communication efficiency only with online vectored multiplication and offline/local addition.

4.4.2 Correctness of Approximation. Theorem 4.5 (Proof in Section A.2) shows the correctness of solving softmax. Property (a) and Property (b) are essentially indicating that $\vec{\mathrm{g}}(\frac{i}{r})$ belongs to quasi-softmax. Property (c) means $\vec{\mathrm{g}}(\frac{i}{r})$ is an approximation of $\vec{f}(t)$, thus $\vec{\mathrm{g}}(\frac{r}{r})$ is an approximation of $\mathrm{softmax}(\vec{x})$. The relation $\max(\vec{x}) - \min(\vec{x}) \le r$ serves as the reference for choosing hyper-parameter r in practice. More details are in Appendix D.

Theorem 4.5 (Correctness). *Given $\vec{\mathrm{g}}(\frac{i}{r})$, if $\max(\vec{x}) - \min(\vec{x}) \le r$ holds, then we have:*
(a). $\vec{\mathrm{g}}(\frac{i}{r}) \in [0, 1]$ is a probability distribution for $i = 0, \ldots, r$.
(b). $[\vec{\mathrm{g}}(\frac{i}{r})]_j \le [\vec{\mathrm{g}}(\frac{i}{r})]_k$ iff $x_j \le x_k$ for $\forall j, k = 1, \ldots, m$.
(c). $\lim_{r \to +\infty} \vec{\mathrm{g}}(\frac{i}{r}) = \vec{f}(1) = \mathrm{Softmax}(\vec{x})$.

5 SECURE SIGMOID COMPUTATION

Similar to softmax, securely computing the e^x, $1/x$ is expensive. Piecewise linear approximation such as ABY2.0 [25] removes e^x, $1/x$ and yet incurs comparison. That is, its approximation $\mathrm{Sig}_{\mathrm{ABY2.0}}(x) = x + 0.5$ if $|x| \le 0.5$; otherwise 0 for $x < -0.5$ or 1 for $x > 0.5$. We aim to securely compute sigmoid without computing e^x, $1/x$, and comparison for higher efficiency.

We approximate the sigmoid function within a finite range $\left[-2^{m-1}, 2^{m-1}\right]$ using the Fourier series (FS). The idea is to include the first few terms regarding approximation accuracy, and simultaneously, remove secure comparisons to reduce the communication complexity. In our FS approximation, there exist sine functions $\sin\frac{2k\pi x}{2^m}$ only for $k = 1, 2, \ldots$ Each term of $\sin\frac{2k\pi x}{2^m}$ can be securely computed in an MPC-efficient manner (as in Protocol 2).

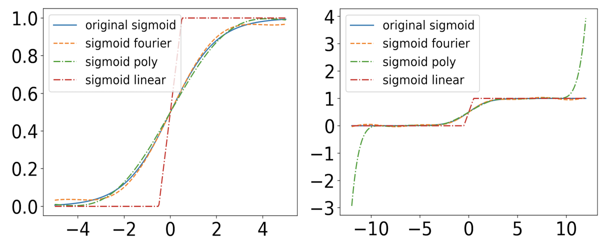
5.1 Local-Sigmoid

$\mathrm{Sigmoid}(x) = \frac{e^x}{e^x+1} : \mathbb{R} \longrightarrow (0, 1)$ 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[3], an effectively-trained model takes the inputs near to 0 in most cases, i.e., out-of-range inputs with very small probabilities. Given this fact, the natural idea is to make the approximation accurate in the range with very high probabilities. Accordingly, we define local-Sigmoid in Definition 5.1 to capture the characteristics.

Definition 5.1 (Local-Sigmoid). Let $0 < \mathrm{a}, \mathrm{b} < +\infty$ and Δ be a small value. We define a function $f : [-\mathrm{a}, \mathrm{a}] \longrightarrow [0, 1]$ to be a local-Sigmoid such that:
(i). $f(x)$ is continuous for any $x \in [-\mathrm{a}, \mathrm{a}]$.
(ii). $\|f(x) - \mathrm{Sigmoid}(x)\|_2 \le \Delta$, where $\|\cdot\|_2$ is L_2-norm.

Definition 5.1 defines an interval $[-\mathrm{a}, \mathrm{a}]$, which contains 0 and the points near to 0. The value of a determines the length of the interval. The property (ii) indicates that the $f(x)$ looks like $\mathrm{Sigmoid}(x)$ in the interval $[-\mathrm{a}, \mathrm{a}]$, where Δ measures the quantitative difference between them. Property (i) avoids a singularity in $f(x)$ for guaranteeing the similarity between $f(x)$ and $\mathrm{Sigmoid}(x)$.

Although $\mathrm{Sigmoid}(x)$ is both continuous and derivative in plaintext, we keep "continuous" only since we take the piecewise approximation into account. In particular, both piecewise linear approximation (PLA) and polynomial fitting (PT) are local-Sigmoid, as shown in Figure 1a and Figure 1b. To be specific, the Δ for PLA is relatively large, while the $[-\mathrm{a}, \mathrm{a}]$ for PT (vs. $[-\infty, \infty]$ for PLA) is smaller than $[-10, 10]$.



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

Figure 1: Comparison for different approximations

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

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

[3]Sigmoid's gradient vanishes (≈ 0) both when its inputs are large and when they are small. Moreover, gradients of the overall product may vanish, which may cut off gradients at some layer and even plague training [37].

The outputs of $\mathsf{LSig}(x)$ are used as the outputs of $\mathsf{sigmoid}(x)$ during the training.

We set periodic parameter $m = 5$ since we empirically find the period $[-16, 16]$ is large enough in Appendix E. We know that $\mathsf{Sigmoid}(x) - 0.5$ is an odd function; thus, we set $\alpha = 0.5$ to make the Fouries-series approximation to be more accurate and simplified. For approximating an odd function, we can use the items of the sine function only (no requirement of cosine functions). The Fourier series coefficients $\vec{\beta}$ can be computed by the integrals,

$$[\vec{\beta}]_k = \frac{1}{16}\int_{-16}^{+16}(\mathsf{sigmoid}(x) - 0.5)\sin(\frac{\pi k x}{16})dx \qquad (2)$$

Thus, we have $\alpha = 0.5$ and $\vec{\beta} = [0.61727893, -0.03416704, 0.16933091, -0.04596946, 0.08159136]$ for $k = 1, 2, 3, 4, 5$.

The $\mathsf{LSig}(x) : [-16, 16] \rightarrow [-\epsilon, 1+\epsilon]$ is very near[4] to Definition 5.1. Notably, the ϵ can be neglected (i.e., $\lim_{K\rightarrow\infty}\epsilon \rightarrow 0$) since it does not affect the results of binary classification at all. The area between two lines in Figure 1a essentially reflects the value of Δ. We can see that $\mathsf{LSig}(x)$ 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.

5.2 Protocols 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 $\mathsf{LSig}(x) = \alpha + \vec{\beta}\cdot\sin(x\vec{k})$. For masking private x, we introduce a random value t, and set $\delta_x = x - t$.

One may question that securely computing/approximating the sine function is also expensive. Notably, $\sin(x\vec{k}) = \sin(\delta_x\vec{k} + t\vec{k})$, which can be solved by using trigonometric identities, i.e., $\sin(\delta_x\vec{k}) * \cos(t\vec{k}) + \cos(\delta_x\vec{k}) * \sin(t\vec{k})$. Here, the $*$ is the element-wise multiplication over vectors. If the $\sin(\delta_x\vec{k}), \cos(t\vec{k}), \cos(\delta_x\vec{k}), \sin(t\vec{k})$ are computed locally, the online computation in Protocol 2 contains simple multiplication only.

In Line 7 and Line 8, 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 in offline phase, P_0 and P_1 compute the $\delta_{\langle x\rangle_0}$ and $\delta_{\langle x\rangle_1}$, respectively. Notably, an additional modular operation is performed for saving communication, since $\mathsf{LSig}(x)$ with $m = 5$ is periodic. Later, the P_0 sends the $\delta_{\langle x\rangle_0}$ in Line 9, while the P_1 sends the $\delta_{\langle x\rangle_1}$ in Line 10.

In line 11, P_0 and P_1 construct δ_x. Then, both parties compute $\sin(\delta_x\vec{k}), \cos(\delta_x\vec{k})$ and get their fixed-point representation $\vec{p}, \vec{q}$. In Line 13 and Line 14, the P_0 and P_1 compute the shares of $\alpha + \vec{\beta}\sin(x\vec{k})$ locally.

Regarding the online computation above, the offline phase mostly follows randomness generation in Section 3.2 except for the vectors $\langle\vec{u}\rangle_1, \langle\vec{v}\rangle_1$. Specifically, given 5-dimensional random 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 4. Theorem 5.2 gives semi-honest security of Π_{LSig}, proved in Section C.2.

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

[4]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	$\langle \mathsf{LSig}(x)\rangle_1$

```
1: Offline Phase
2: $P_0, T$: $\langle t\rangle_0, \langle\vec{u}\rangle_0, \langle\vec{v}\rangle_0 \leftarrow \mathsf{PRF}_0(\mathsf{key}_0)$
3: $P_1, T$: $\langle t\rangle_1 \leftarrow \mathsf{PRF}_1(\mathsf{key}_1)$
4: T: Compute $t = \langle t\rangle_0 + \langle t\rangle_1$, $\langle\vec{u}\rangle_1 = \mathsf{FR}(\sin(t\vec{k})) - \langle\vec{u}\rangle_0$, $\langle\vec{v}\rangle_1 = \mathsf{FR}(\cos(t\vec{k})) - \langle\vec{v}\rangle_0$
5: T: Send $\langle\vec{u}\rangle_1, \langle\vec{v}\rangle_1$ to $P_1$
6: Online Phase
7: $P_0$: Compute $\delta_{\langle x\rangle_0} = \langle x\rangle_0 - \langle t\rangle_0 \mod 32$
8: $P_1$: Compute $\delta_{\langle x\rangle_1} = \langle x\rangle_1 - \langle t\rangle_1 \mod 32$
9: $P_0$: Send $\delta_{\langle x\rangle_0}$ to $P_1$          ▷ 5 bits for integral part
10: $P_1$: Send $\delta_{\langle x\rangle_1}$ to $P_0$
11: $P_0, P_1$: Compute $\delta_x = \delta_{\langle x\rangle_0} + \delta_{\langle x\rangle_1}$
12: $P_0, P_1$: Compute $\vec{p} = \mathsf{FR}(\sin(\delta_x\vec{k}))$ and $\vec{q} = \mathsf{FR}(\cos(\delta_x\vec{k}))$
13: $P_0$: $\langle\mathsf{LSig}(x)\rangle_0 = \alpha + \vec{\beta}(\vec{p} * \langle\vec{v}\rangle_0 + \vec{q} * \langle\vec{u}\rangle_0)$
14: $P_1$: $\langle\mathsf{LSig}(x)\rangle_1 = \alpha + \vec{\beta}(\vec{p} * \langle\vec{v}\rangle_1 + \vec{q} * \langle\vec{u}\rangle_1)$
```

5.3 Communication Complexity

Recall that x is an n-bit decimal with p-bit fractional part. The 1-round online communication requires $2(m+p)$ bits in Line 9 and Line 10, for latter reconstruction of δ_x. With PRF, offline communication gets to $2Kn$ bits in Line 5, where K is $\vec{\beta}$'s dimension. Below, Theorem 5.3 summarizes communication costs of Π_{LSig}.

THEOREM 5.3. *Let x be the n-bit fixed-point numbers with the p-bit fractional part and c_i, a_i, 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 in a different manner, i.e., approximating the sigmoid in the whole input domain instead of local sigmoid. Specifically, Squirrel uses piecewise functions with roughly $O(2n\log n)$ bits for choosing the piece, in which the second piece is a Fourier-series-based approximation. Squirrel transmits the shares of $\vec{\beta}\sin(x\vec{k})$ in the online phase, in which it takes $8Kn$ bits communication and is linear to K. Our idea optimizes 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[5] of $n = 64, p = 16, K = 5, m = 5$, Squirrel requires 2560 bits (or $8Kn$ bits), while Π_{LSig} takes 42 bits. Besides, if desired, the independence on K can benefit the approximation accuracy, although we empirically found $K = 5$ is accurate enough.

5.4 Correctness for Sigmoid-Like Protocol

Since we perform truncation over float numbers, we analyze the absolute difference between secret-shared values and real (or 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 in a small probability, are bounded in terms of the approximation. We get Theorem 5.4 (Proof in Section A.3), which showcases the correctness of Π_{LSig}.

[5]$n = 64, p = 16$ are the default configurations in many literature.

Theorem 5.4 (Correctness). *Let x be the n-bit fixed-point number with p-bit fractional part and $a_1, \ldots, a_K$ be scalars.* Π_{LSig} *uses $(K{+}1)$-term Fourier series. Then, 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* $x \in [-\mathrm{a}, \mathrm{a}]$*;*
(ii) $1 + 2\epsilon$ *for any* $x < -\mathrm{a}$ *or* $x > -\mathrm{a}$*, where* ϵ *is a small value.*

In practice, Figures 1a,1b show the accuracy of fitting with 6 terms. As in Figure 1a, using Fourier series is more accurate than piecewise linear functions. As shown in Figure 1b, our solution provides a more stable approximation (i.e., without the risk in gradient explosion caused by large/small inputs) in the whole range, error bounded by ≤ 1. Particularly, for the inputs $[-1.5, 1.5]$ (usually with high probability), local-sigmoid is more accurate than a polynomial approximation. Besides, ODE-based approximation is also applicable for Sigmoid. Yet, the iterative method makes it impossible to achieve a 1-round protocol.

6 END-TO-END MACHINE LEARNING

Achieving end-to-end private training is similar to deploying machine learning models in plaintexts, detailed in Section 6.1. We explicate backpropagation and gradients computation in Section 6.2.

Prior implementations involve two branches, one in C++ and the other in Python. We implemented the proposed protocols in both languages to provide further convenience. Section 6.3, Python implementation following TensorFlow's programming styles is a private training framework including secure multiplication, convolution, layers, and models. The C++ version in Section 6.4 refers to implementing multiple plug-in modules in the existing Piranha framework regarding the proposed protocols.

6.1 General Private Learning Pipeline

In the end-to-end private training, P_0 and P_1 collaboratively execute a sequential of protocols tailored by the predefined model. A machine learning model can be decomposed into multiple layers sequentially. The predefined layers include the dense, ReLU, convolution, average pooling, max pooling, and loss layers. For each layer, forward and backward propagation are required to be computed.

Define x as input data and w as model parameters. Forward propagation is to compute the output $y = f(x, w)$ given x, w and the activation function f at the corresponding layer. Protocol 1 and Protocol 2 showcase the forward pass when the softmax/sigmoid functions input the private data. Backward propagation performs a backward pass via chain rule to adjust the model's parameters w by computing $\frac{\partial f}{\partial y}, \frac{\partial f}{\partial x}, \frac{\partial f}{\partial w}$. Notably, all computations mentioned should be performed in a secret-shared manner.

6.2 Backpropagation and Auto-differentiation

Let y be the model output and $\hat{y}$ be its label. Auto-differentiation is essentially computing the differentiation of the loss functions over y. In our implementation, we use `CrossEntropyWithSoftmax` (SoftmaxCE) and `BinaryCrossEntropyWithSigmoid` (SigmoidBCE) as loss functions for softmax and sigmoid, respectively. Below, we elaborate on the details of their backpropagation. More details can be found in Appendix B.2.

6.2.1 Softmax. The SoftmaxCE over $\mathsf{Softmax}(\vec{y})$ is defined to be $\mathsf{SoftmaxCE} = -\sum_i \log(\mathsf{Softmax}(\vec{y})_i)\hat{y}_i$. To perform backpropagation, the gradients over $\vec{y}$ gets $\mathsf{Softmax}(\vec{y}) - \hat{y}$ (details in Appendix B.2.2). Given secret-shared $\langle y\rangle_0, \langle y\rangle_1$, the P_0 and P_1 can jointly call Protocol 1 to obtain secret-shared outputs. Then, the P_0 and P_1 respectively subtract local shares of $\hat{y}$ to obtain secret-shared differentiation over $\vec{y}$.

6.2.2 Sigmoid. The SigmoidBCE is defined to be $\mathsf{SigmoidBCE} = -\log(\mathsf{Sigmoid}(y))\hat{y} - \log(1 - \mathsf{Sigmoid}(y))(1 - \hat{y})$ The partial differentiation over y (see Appendix B.2.3) can be computed by $\mathsf{Sigmoid}(y) - \hat{y}$. Given secret-shared $\langle y\rangle_0, \langle y\rangle_1$, the P_0 and P_1 can jointly execute Protocol 2 and local subtraction to obtain the secret-shared differentiation over y.

6.3 Framework Implementation in Python

The Python version has been implemented in the TensorFlow Framework. Its defined operations, layers, and models follow TensorFlow programming styles for providing user-oriented APIs. To support efficient model training, we implemented various protocols for the cryptographic supports regarding the ideas [28, 35] of ReLU, pooling, convolution, etc. The truncation mentioned in Section 3.2 has been implemented following SecureML's probabilistic truncation [21] and Cheetah's faithful truncation [13].

In the offline phase, we realize a secure deterministic random bit generator. Our implementation is the first to achieve a standard and secure online/offline phase. With our implementation, researchers can construct whatever networks they expect by integrating the necessary operators and layers. To our knowledge, our implementation is the first to achieve efficient training in TensorFlow Framework.

6.4 C++ Implementation in Piranha

6.4.1 Plug-in Modules. The C++ implementation contains the plug-in blocks in the Piranha [35], which provides state-of-the-art GPU implementations of cryptographic training protocols. We pick Piranha as our infrastructure since Piranha is a general-purpose solution for supporting latter-inserting blocks. Piranha provides the cryptography backend (e.g., implemented protocols) and neural network library (e.g., activation, pooling). We added the blocks of Sigmoid protocol, Softmax protocol, their backpropagation, and truncation [21] by following Piranha's programming interface. We simply use Piranha+ to represent improved Piranha's library with our protocols.

6.4.2 Private Training in Piranha+. We provide an example of private training by replacing Piranha's softmax with Protocol 1. Originally, Piranha's private training consisted of the implementation of backpropagation via CrossEntropyWithSoftmax, called `_adhoc_softmax_grad`. Using Piranha's programming template, we follow Protocol 1 and Section 6.2 to implement the function `ode_softmax` and its backpropagation `ode_softmax_grad(labels, deltas)`, respectively. We replace `_adhoc_softmax_grad(labels, deltas)` with `ode_softmax_grad(labels, deltas)` for private training through Piranha+. Similarly, researchers can implement the proposed protocols as modules in other private training frameworks (e.g., CrypTen) for improved training performance.

7 EXPERIMENTAL EVALUATION

CPU-based experiments are conducted on three CentOS servers, each with an Intel(R) Xeon(R) Platinum 8163 CPU@2.50GHz and 16GB RAM. GPU-based experiments are run on the Ubuntu servers, with 8-core Intel(R) Xeon(R) Platinum 8163 2.50GHz CPUs of 64GB RAM and NVIDIA-T4 GPU of 16GB RAM. To simulate the real-world network condition, we set network bandwidth to be 12.5MB/s or 30.1MB/s under the WAN condition. Our experiments include evaluations of the proposed protocol and a comparison of training performance. We aim to answer the four questions below:

Q1: *How much the proposed protocols have improved in communication and running time?* (Section 7.2 and Section 7.3 for protocol-level comparison)

Q2: *Compared with the state of the art, how much training performance has been improved in model accuracy and training time?* (Section 7.4 and Section 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 different WAN setting impact the latency?* (Section 7.7)

7.1 Baselines and Setup

7.1.1 Protocol-Level Comparison. In Table 2 and Table 3, we compare the communication for particular protocols atop distinctive computing methods. Accordingly, Table 4 and Table 5 display the running time. In most tests, we use fixed-point numbers of 64-bit decimals with 14-bit fractional parts.

For softmax, we test Quasi-Softmax with ASM and ComDiExp. For sigmoid, we compare Local-Sigmoid with Newton-Raphson (NR) method, polynomial approximation, and squirrel's approximation. The accuracy of the protocol is meaningful with respect to the model accuracy, which is omitted here.

7.1.2 Configuration for Model-Level Comparison. In Tables 6-11, we compare the model accuracy, communication, and training time for both large models and small models. We evaluate model training using the following standard datasets:

- MNIST dataset [8] contains handwritten digits of $0 \sim 9$, totally training set of $60,000$ examples, and a test set of $10,000$ examples.

- CIFAR10 [16] dataset contains $60,000$ RGB images of size 32×32, splitting evenly into 10 classes.

7.1.3 Large-Model Testing. We conduct the tests of running large models mainly targeting GPU frameworks, i.e., Piranha [35] and CryptGPU [32]. We use the learning rate ranging from 0.001 to 0.02 for testing model accuracy for tuning. CryptGPU, mainly implementing CrypTen, is the first framework for providing cryptographic training with the pretraining phase. Piranha implements three main-stream cryptographic protocols and runs them with GPUs. Following Piranha's experiments, we test the classical models including AlexNet [17], LeNet [18], VGG16 [30], and ResNet [12].

- AlexNet with 61M parameters has convolution layers, max-pooling layers, fully connected layers, ReLU, and Softmax.

- VGG-16 with 138M parameters uses 16 layers of convolution, ReLU, max pooling, fully-connected layers, and Softmax.

- ResNet family has been widely adopted in the computer vision community. They consist of convolution, max pooling, average pooling, batch normalization, fully connected layers, and Softmax.

Table 2: Communication Comparison for Softmax

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

In our experiments, we choose ResNet-18 with 11M parameters for evaluation.

7.1.4 Small-Network Testing. For small networks, we adopt Networks A, B, C, and D described in SecureNN [33, 34]. We compare our work with SecureNN [33], CrypTen [15], Falcon [34], and SPDZ-QT [14] using only CPUs. For accuracy, we compare with SPDZ-QT [14], which has the highest accuracy among the above. Besides, we benchmark running time with SecureNN, CrypTen, and Falcon since the three works are open-sourced.

7.2 Communication for Proposed Protocols

The communication costs affect the training time, especially under limited bandwidth and time latency. We first evaluate the communication of our proposed protocols in Table 2 and Table 3. Summarily, ASM approximation uses the largest number of rounds, whereas ComDiExp and Π_{QSMax} consume relatively fewer communication rounds. When the number of classes increases, unlike ComDiExp's increasing number of communication rounds, Π_{QSMax} requires constant rounds of communication. When the number of classes is the same for all baselines [15, 32, 33], Π_{QSMax} requires the lowest communication costs. As for sigmoid, both LLAMA [11] and Π_{LSig} achieve round-optimal communication. Moreover, Π_{QSMax} accomplishes the lowest overall/online communication costs.

7.2.1 Softmax. Recall that two cryptographic approaches for approximating Softmax are ASM and ComDiExp (Section 2). We take SecureNN for ASM and CrypTen/CryptGPU for ComDiExp. SecureNN did not differentiate between online and offline phases. Table 2 compares online and offline communication and rounds. For 64-bit shared data, we take $m \in \{10, 100, 1000\}$ classes. We list

Table 3: Communication Comparison for Sigmoid

Protocol	Offline (bit)	Online (bit)	Overall (bit)	Rd
Piece. Linear Approx.	-	≈ 800	-	5
Poly. Approx. ($K = 5$)	320	1280	1600	1
Poly. Approx. ($K = 8$)	512	2048	2560	1
Squirrel ($K = 5$)	-	$2560 + 1792$	-	3
Squirrel ($K = 8$)	-	$4096 + 1792$	-	3
LLAMA	-	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

Table 4: Running Time (10 Epochs) for Softmax over GPU

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

the communication of Π_{QSMax} in the setting of different iterations, i.e., $8, 16, 32, 64$. ComDiExp is better than ASM already, so we focus on comparing it with ComDiExp. For almost all cases, the Π_{QSMax} has much lower communication than ComDiExp. Empirically, we found that either $r = 16$ or $r = 32$ is large enough for training a model. In the case of $r = 32$, the Π_{QSMax} reduces the communication by 83%, 85%, and 85% for 10 classes, 100 classes, and 1000 classes, respectively.

7.2.2 Sigmoid. Table 3 compares the communication for Sigmoid, given 64-bit fixed-pointed numbers with 14 bits fractional parts. We take ABY2.0 [25] as an instantiation building atop piecewise linear approximation (PLA). The protocols of ABY2.0, LLAMA [11], and Squirrel rely on extra communication-heavy OT and garbled circuits. For fairness, we compare online communication only with them since their offline phase involves extra costs of other cryptographic tools. For Squirrel, the left part indicates the communication for Fourier-series-based approximation, while the right part 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 rounds. In particular, PolyA, Squirrel, and the Π_{LSig} refer to an additional parameter K, indicating the $K+1$ terms for approximation. Squirrel takes 8 in practice, and 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. The Π_{LSig} requires much lower communication costs than prior literature.

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

Time (s)	Batchsize			
	32	64	128	256
Piranha	40.05117	40.08663	40.08994	40.14356
Piranha+	0.077250	0.077017	0.076789	0.076450

Table 6: Training Accuracy Compared with Prior Arts

Model	Framework	Acc-5	Acc-10	Pretrain
AlexNet (CIFAR10)	Piranha	-	40.7%[\$]	No
	Ours	52.1%	**56.3%**	No
	CryptGPU	-	59.6%	Yes
	Ours	51.2%	**59.6%**	Yes
VGG16 (CIFAR10)	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

$f = 24$, batchsize= 32. Acc-1 and Acc-5/15 are accuracy at epoch= 1 and epoch=5 (or epoch= 15).
[\$] The best accuracy reported in Piranha ([35, Figure 5]) is 98.1% for LeNet, 40.7% for AlexNet, and 58.7% for VGG16.
SPDZ-QT's results are from [14, Table 3].

Table 7: Communication for Large Models (GB/batch)

Framework	AlexNet (CIFAR10)		VGG16 (CIFAR10)	
CryptGPU[\$]	1.37	[$\downarrow$ 57.7%]	7.55	[$\downarrow$ 61.3%]
Falcon[\$]	0.62×3	[$\downarrow$ 68.8%]	1.78×3	[$\downarrow$ 45.3%]
Piranha[\$]	0.58×3	[$\downarrow$ 66.7%]	4.26×3	[$\downarrow$ 77.2%]
Ours	0.39 ($b = 5$), 0.58 ($b = 7$)		2.925 (batch of 2^5)	

[\$] From Table 4 in [35], which reports per-party communication.
We use the same pooling and free truncation as in Piranha for a fair comparison.

7.3 GPU Acceleration for Proposed Protocols

We perform running-time tests in the aligning environments and standards, i.e., comparing the speed of executing protocols in Piranha. The goal is to report acceleration for each protocol explicitly, Table 4 for softmax and Table 5 for sigmoid. Generally, Π_{QSMax}'s running time remains relatively stable and very short due to the GPU parallelism. Round-optimal Π_{LSig} is much faster than the polynomial approximation, which unavoidably spends multiple rounds for computing multi-order multiplication and truncation followed by each multiplication.

7.3.1 Softmax. In Piranha, three protocols of softmax have been supported, including "Reveal", "ASM", and "Adhoc". The "Reveal", as a template, starts with reconstructing the private inputs and later computes plain-text softmax. We implemented the Π_{QSMax} with 16 iterations and 32 iterations.

Table 4 shows the running time of different protocols with 10, 100, or 1000 input classes. As for 16 iterations and 32 iterations, the Π_{QSMax} speeds up ASM-based protocol by 20× and 10×, respectively. The "Adhoc" can support 10 classes and fail the running with large-scale classes. Besides, the running time of Π_{QSMax} keeps nearly unvaried thanks to the massive parallel computation of GPU. The Π_{QSMax} can highly improve the training speed given vectorized inputs.

7.3.2 Sigmoid. In Piranha, we implemented Π_{LSig} and compared it with the polynomial-approximated sigmoid one. Recall that PLA, Squirrel, and LLAMA rely on additional cryptographic primitives, which are not supported in Piranha. Given the different batch sizes, the running time remains almost unvaried. Intuitively, GPU instantiation has a great advantage in parallel computation. Notably, the Π_{LSig} speeds up $\approx 570\times$, so running sigmoid replaced by Π_{LSig} can accelerate training largely in Piranha.

7.4 Training Accuracy for Models

We report training accuracy, both for large and small models. Table 6 shows the training accuracy compared with recent breakthroughs. In particular, Piranha and CryptGPU are the newest works for testing large models, while SPDZ-QTmostly aims at relatively small networks. Piranha supports training large models from scratch; meanwhile, CryptGPU trains a model with the pretraining phase (i.e., assisted by plaintext training over public datasets). Accordingly, we tested accuracy with and without the pretraining phase.

Via training from sketch, we record the accuracy at each epoch. Later, the accuracy at epoch=10 is used to be the highest one. To be specific, our training accuracy is higher than Piranha's accuracy, e.g., 56.3% versus 40.7% for AlexNet, meanwhile 59.4% versus 58.7% for VGG16. The small networks can reach the comparative accuracy as SPDZ-QTwith fewer training epochs, from 15 to 5. The accuracy results imply that our protocols have superior approximation/convergence. Overall, private training can achieve reasonable accuracy for relatively small models, whereas attaining high accuracy for large models still requires the assistance of pretraining. Notably, the overall learning performance is still unsatisfying compared to normally learned models, motivating further research of improving accuracy.

7.5 Communication for Model Training

We benchmark training with large and small models, i.e., testing communication and accuracy. To clarify, we remove the effect of environmental conditions (i.e., time latency, bandwidth) and then conduct the experiments in the ideal LAN settings. For communication, we show results of relatively large models (with more non-linear computations) in Section 7.5.1 and mainstream networks (of cryptographic training) in Section 7.5.2. We count the online and offline communication costs of all parties (2 computing parties and 1 commodity server), and the head size of data frames (gRPC format[6]). No matter for training with large models in Table 7 (batch size $b \in \{5, 7\}$) or small models in Table 8 (batch size is 128, with 32 iterations for computing softmax), our work achieves the lowest communication costs.

Table 8: Communication for Small Models (GB/batch)

Framework	Model (MNIST)			
	Network			
	A	B	C	D
SecureNN$^+$	0.047	1.31	2.11	0.37
Piranha-2$^\$$	-	-	1.25	-
SPDZ-QT$^!$	0.055	0.43	0.75	0.09
CryptGPU*	-	-	1.14	-
Ours	0.021	0.41	0.64	0.05

* from [32, Table IV]
$^\$$ from [35, Table 4], which reports per-party communication.
$^!$ We estimate 1-batch communication by $x/60000 \cdot 128$, where 1-epoch communication x is from [14, Table 3].

Table 9: Training Time (s/batch) in LAN via GPU

Work	LeNet (MT)	AlexNet (CIFAR10)	VGG-16 (CIFAR10)	ResNet-18 (CIFAR10)
Piranha	2.574691	6.506521	30.345877	29.879927
Piranha+	2.208990	2.569884	26.390739	25.876978

7.5.1 Communication for AlexNet&VGG16. Currently, only a small set of works (i.e., Falcon [34], CryptGPU[32], and Piranha [35]) can successfully benchmark training with large models, such as AlexNet and VGG16. As in Table 7, we show the communication for training AlexNet over the dataset CIFAR10, VGG16 over CIFAR10, and VGG16 over Tiny-ImageNet. We conduct experiments by following CryptGPU and Piranha, i.e., testing one-batch communication and using identical model architectures. Table 7 shows that our work significantly reduces communication costs for large model training. In particular, our work reduces approximately 57%-77% communication compared with prior arts. The superiority of our non-linear units can be sufficiently reflected by improved large-model (with more non-linear computation) training. For a more specific explanation, we conduct protocol-level comparison in Section 7.2.

7.5.2 Communication for Network A/B/C/D. We follow SecureNN's benchmarking[7] and train networks A/B/C/D over MNIST to show more results. In Table 8, we compare communication with SecureNN [33], Piranha [35], "SPDZ-QT" [14] and CryptGPU [32]. Our work greatly reduces communication compared with them.

[6]https://grpc.io
[7]SecureNN's Network-C is Falcon's Network-D.

7.6 Training Acceleration with GPU in LAN

To show the acceleration with our protocols, we test the training time per batch in LAN. Table 9 presents the training time (seconds), where Piranha+ represents Piranha with the proposed plug-in protocols. Here, we perform the tests on relatively large models since GPU benefits parallel and vectorized computation for scalability. In LeNet, max-pooling is used, while AlexNet uses average pooling. For each training step, the batch size is set to be 128. We count time for 10 batches and get the average. In summary, Piranha+ optimizes the training in Piranha at about 10% ∼ 60%.

7.7 Impact of Bandwidth and Latency in WAN

For testing the training time, we take latency into account and later count the time for both communication and computation in Table 10. The "B, T" represents training with a basic/tree-structured version of ReLU protocols, and MT denotes the MNIST dataset. Under different WAN conditions, we should pick different types of protocols for higher training performance. Specifically, if the model size is large, we tend to choose the basic version; meanwhile, if time latency is high, we tend to pick a tree-structured version.

Table 10: Training Time in WAN via CPU

Work	Latency	Network (s/batch)			
		A (MT)	B (MT)	C (MT)	D (MT)
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**

Bandwidth=12.5MB/s for 5/50ms latency
We re-run SecureNN, Falcon, and CrypTen.

Table 11: Training Time for LeNet and AlexNet in WAN

Work	Latency	LeNet (MNIST)		AlexNet (CIFAR10)	
Piranha	60ms	5680 min	-	9192 min	-
Ours-B	60ms	1780 min	[↓ 69%]	4029 min	[↓ 56%]
Ours-T	60ms	**1232 min**	[↓ 78%]	**2462 min**	[↓ 73%]

Bandwidth=40MB/s for 60ms latency

In Table 11, we adjust the time latency to be 60ms for comparing training time with Piranha. Our work reduces training time by 50% ∼ 80% for LeNet and AlexNet. The root cause is that communication consumes a large percentage of the overall running time (Figure 6 in [35]). Largely-reduced training time confirms that "minimizing communication benefits in a WAN setting" [35].

8 CONCLUDING REMARKS

Our work proposes two approximation approaches to realize softmax and sigmoid in a cryptography-friendly manner, leading to faster training with much lower communication. The new approximation may boost secure training for up-to-date fancy ML models, such as transformers [38] and recognition [1]. Essentially, we shed new light on designing MPC protocols for bounded non-linear functions, i.e., avoiding computing unbounded functions (e^x, $1/x$) in the middle computation. Exacting the property of non-linear functions could help to simplify MPC-protocol design, i.e., good at accuracy and communication/computation compared with directly combining arithmetic operations. In practice, our protocol designs are modular, which could be easily integrated into different private learning frameworks, such as CrypTen. We propose our secure secret-shared protocol in the basic 2-party setting. It could be extended to other settings using replicated shares and homomorphic encryption for other features, such as malicious security.

We are the first to explore IVP-ODE-based approximation (for non-linear functions) with rational polynomials and trigonometric functions. In essence, the Π_{QSMax} transforms non-linear approximation into an iterative solution of ODE solved by Euler formula. In MPC, non-linear approximations are replaced by multi-order multiplications. Our Π_{QSMax} showcases a new route of decreasing required multiplication orders for non-linear/polynomial approximation. This is equal to reducing computational or communication complexity from $O(n^c)$ accuracy-limitation (e.g., Taylor expansion) to $O(cn)$ accuracy-limitation (i.e., c rounds of iteration). Broadly, we extend the MPC-friendly functions to the solutions of differential equations whose coefficients are rational polynomials or trigonometric functions. Fourier series approximation can be extended to securely compute other single-variable non-linear functions, such as step function. Notably, online communication of Π_{LSig} is independent of the number of Fourier series terms, always resulting in $< 2n$ bits. Ideally, the Π_{LSig} supports negligible-error approximation for any square-integrable function if we choose a sufficiently large number of Fourier series terms.

Finally, we hope our Python implementation could benefit further development such as combining with other frameworks, say, for graph neural networks.

ACKNOWLEDGEMENT

The authors wholeheartedly appreciate the invaluable feedback from the anonymous shepherd, reviewers, and artifacts evaluation committee. We thank Yuan Zhao, Yashun Zhou, Dong Yin, and Jiaofu Zhang at Ant Group for their insightful discussions and endeavors on coding, and Jiafan Wang for his help and guidance to Yu. 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. Sherman Chow is supported in part by the General Research Fund (CUHK 14210319) of 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*.

[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*.

[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, Vol. 1. In *Reviews in Group Representation Theory, Part A (Pure and Applied Mathematics Series, Vol. 7).*

[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.

[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.

[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.*

[8] Li Deng. 2012. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. In *IEEE Signal Process. Mag.*

[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 *CCS*.

[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. In *Proc. Priv. Enhancing Technol.*

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

[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.*

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

[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.*

[16] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images.

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

[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. In *Neural Comput.*

[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.*

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

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

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

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

[24] 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.

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

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

[27] 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.*

[28] 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.*

[29] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis R. Bach. 2022. AriaNN: Low-Interaction Privacy-Preserving Deep Learning via Function Secret Sharing. In *Proc. Priv. Enhancing Technol.*

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

[31] Nigel P. Smart and Titouan Tanguy. 2019. TaaS: Commodity MPC via Triples-as-a-Service. In *CCSW@CCS.*

[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.*

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

[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. In *Proc. Priv. Enhancing Technol.*

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

[36] 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.*

[37] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. 2021. Dive Into Deep Learning.

[38] Mengxin Zheng, Qian Lou, and Lei Jiang. 2023. Primer: A Privacy-preserving Transformer on Encrypted Data. In *DAC.*

[39] 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.*

A THEORETICAL DERIVATION

This part includes proofs of all theorems, excluding security-related ones.

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

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

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

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

where r is the number of iterations. Define the function Softmax_r, which value on the input x is defined by y_r in the above. Now, we can see that Softmax_r is quasi-Softmax. Moreover, Softmax_r converges to real Softmax as r tends to be infinity (see Theorem 4.5). We thus obtain iterative Softmax y_{t+1}. When computing $\text{Softmax}_r(x)$, we can see that only addition and multiplication are required using Euler formula. For model accuracy, we verify it by experimental results since we observe that the theoretical bound is very loose, which is useless for practical training. □

A.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 just need to prove $y_t^i \geq 0$ and $\sum_i y_t^i = 1$ for $t = 0, \ldots, r$. Firstly, we have $y_0^i \geq 0$ and $\sum_i y_0^i = 1$ obviously. Secondly, by mathematical induction, we can suppose $y_t^i \geq 0$ and

$\sum_i y_t^i = 1$ for some t. Then, we have, $\sum_i y_{t+1}^i = \sum_i y_t^i + (\sum_i (x^i * y_t^i) - \langle x, y_t \rangle \sum_i y_t^i)/r = \sum_i y_t^i + \langle x, y_t \rangle (1 - \sum_i (y_t^i))/r = 1$. We reformat the equation (ii) as $y_{t+1} = y_t * (\mathbf{1}_m + (x - \langle x, y_t \rangle \mathbf{1}_m)/r)$, where $\vec{1}$ denotes the vector $(1, \dots, 1)^\top$ of dimension m, $\langle x, y \rangle$ denoted the inner product of vector x and y, and $*$ denotes element-wise multiplication. Let $u := (x - \langle x, y_t \rangle \vec{1})/r$. Then, 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, $(\max_i(x_i) - \min_i(x_i))/r \le r/r = 1$. On the other hand, we have $-1 < \min(u) \le \mathbb{E}_{y_t}(u) \le \max(u) < 1$. Hence, we get $\vec{1} + u \ge 0 \Rightarrow y_{t+1} = y_t * (\vec{1} + u) \ge 0$.

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

(c) Let $f : [0, 1] \to R^m$ defined by $f(t) = \text{softmax}(tx)$. We could know that, $f(0) = 1/m, f'(t) = (x - \langle x, f(t) \rangle \mathbf{1}_m) * f(t)$. By Euler formula, we get that $f(1) = \text{Softmax}(x)$. □

A.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, we have that,

$$\begin{aligned}
&|\text{Dec}(\cos \tfrac{2k\pi t}{2^m}) - \cos \tfrac{2k\pi t}{2^m}| \le 2^{-f-1},\\
&|\text{Dec}(\sin \tfrac{2k\pi t}{2^m}) - \sin \tfrac{2k\pi t}{2^m}| \le 2^{-f-1},\\
&|\text{Dec}(\sin \tfrac{2k\pi \delta_x}{2^m}) - \sin \tfrac{2k\pi \delta_x}{2^m}| \le 2^{-f-1},\\
&|\text{Dec}(\cos \tfrac{2k\pi \delta_x}{2^m}) - \cos \tfrac{2k\pi \delta_x}{2^m}| \le 2^{-f-1}
\end{aligned}$$

Let $\delta_x = x - t$. By triangle identities, we know $\sin \frac{2k\pi(x-t)}{2^m} = \sin \frac{2k\pi\delta_x}{2^m} \cos \frac{2k\pi t}{2^m} + \cos \frac{2k\pi\delta_x}{2^m} \sin \frac{2k\pi t}{2^m}$.

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

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

$$\begin{aligned}
= \; & |\text{Dec}(\sin \tfrac{2k\pi\delta_x}{2^m})(\text{Dec}(\cos \tfrac{2k\pi t}{2^m}) - \cos \tfrac{2k\pi t}{2^m} + \cos \tfrac{2k\pi t}{2^m})\\
& +\text{Dec}(\cos \tfrac{2k\pi\delta_x}{2^m})(\text{Dec}(\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})|\\
= \; & |\text{Dec}(\sin \tfrac{2k\pi\delta_x}{2^m})(\text{Dec}(\cos \tfrac{2k\pi t}{2^m}) - \cos \tfrac{2k\pi t}{2^m})\\
& +\text{Dec}(\sin \tfrac{2k\pi\delta_x}{2^m}) \cos \tfrac{2k\pi t}{2^m}\\
& +\text{Dec}(\cos \tfrac{2k\pi\delta_x}{2^m})(\text{Dec}(\sin \tfrac{2k\pi t}{2^m}) - \sin \tfrac{2k\pi t}{2^m})\\
& +\text{Dec}(\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})|\\
= \; & |\text{Dec}(\sin \tfrac{2k\pi\delta_x}{2^m})(\text{Dec}(\cos \tfrac{2k\pi t}{2^m}) - \cos \tfrac{2k\pi t}{2^m})\\
& +\text{Dec}(\cos \tfrac{2k\pi\delta_x}{2^m})(\text{Dec}(\sin \tfrac{2k\pi t}{2^m}) - \sin \tfrac{2k\pi t}{2^m})\\
& +\text{Dec}(\sin \tfrac{2k\pi\delta_x}{2^m}) \cos \tfrac{2k\pi t}{2^m} + \text{Dec}(\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}|\\
= \; & |\text{Dec}(\sin \tfrac{2k\pi\delta_x}{2^m})(\text{Dec}(\cos \tfrac{2k\pi t}{2^m}) - \cos \tfrac{2k\pi t}{2^m})\\
& +\text{Dec}(\cos \tfrac{2k\pi\delta_x}{2^m})(\text{Dec}(\sin \tfrac{2k\pi t}{2^m}) - \sin \tfrac{2k\pi t}{2^m})\\
& + \cos \tfrac{2k\pi t}{2^m}(\text{Dec}(\sin \tfrac{2k\pi\delta_x}{2^m}) - \sin \tfrac{2k\pi\delta_x}{2^m})\\
& + \sin \tfrac{2k\pi t}{2^m}(\text{Dec}(\cos \tfrac{2k\pi\delta_x}{2^m}) - \cos \tfrac{2k\pi\delta_x}{2^m})|\\
\le \; & |\text{Dec}(\sin \tfrac{2k\pi\delta_x}{2^m})(\text{Dec}(\cos \tfrac{2k\pi t}{2^m}) - \cos \tfrac{2k\pi t}{2^m})|\\
& +|\text{Dec}(\cos \tfrac{2k\pi\delta_x}{2^m})(\text{Dec}(\sin \tfrac{2k\pi t}{2^m}) - \sin \tfrac{2k\pi t}{2^m})|\\
& +|\cos \tfrac{2k\pi t}{2^m}(\text{Dec}(\sin \tfrac{2k\pi\delta_x}{2^m}) - \sin \tfrac{2k\pi\delta_x}{2^m})|\\
& +|\sin \tfrac{2k\pi t}{2^m}(\text{Dec}(\cos \tfrac{2k\pi\delta_x}{2^m}) - \cos \tfrac{2k\pi\delta_x}{2^m})|\\
\le \; & 1 \cdot |\text{Dec}(\cos \tfrac{2k\pi t}{2^m}) - \cos \tfrac{2k\pi t}{2^m}| + 1 \cdot |\text{Dec}(\sin \tfrac{2k\pi t}{2^m}) - \sin \tfrac{2k\pi t}{2^m}|\\
& +1 \cdot |\text{Dec}(\sin \tfrac{2k\pi\delta_x}{2^m}) - \sin \tfrac{2k\pi\delta_x}{2^m}|\\
& +1 \cdot |\text{Dec}(\cos \tfrac{2k\pi\delta_x}{2^m}) - \cos \tfrac{2k\pi\delta_x}{2^m}|\\
\le \; & 2^{-f-1} + 2^{-f-1} + 2^{-f-1} + 2^{-f-1}\\
\le \; & 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 true Sigmoid is not bounded at all. 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 for independent and modular analysis. Assuming the correct truncation with ideal probability 100%, the difference $|\text{Rec}(\langle \text{Sig}(x) \rangle_0, \langle \text{Sig}(x) \rangle_1) - \text{Sigmoid}(x)|$ is bounded if and only if $|\text{Rec}(\langle \sin \frac{2k\pi x}{2^m} \rangle_0, \langle \sin \frac{2k\pi x}{2^m} \rangle_1) - \sin \frac{2k\pi x}{2^m}|$ is bounded. For brevity, we take the bounding difference 2^{-f+1}.

The bound for difference of $\text{Rec}(\langle \text{Sig}(x) \rangle_0, \langle \text{Sig}(x) \rangle_1)$ and real Sigmoid is,

$$\begin{aligned}
& |\text{Rec}(\langle \text{Sig}(x) \rangle_0, \langle \text{Sig}(x) \rangle_1) - \text{Sigmoid}(x)|\\
\le \; & |\textstyle\sum_{k=1}^K a_k \cdot \text{Rec}(\langle \sin \tfrac{2k\pi x}{2^m} \rangle_0, \langle \sin \tfrac{2k\pi x}{2^m} \rangle_1) - \sum_{k=1}^K a_k \sin \tfrac{2k\pi x}{2^m}|\\
\le \; & |\textstyle\sum_{k=1}^K a_k (\text{Rec}(\langle \sin \tfrac{2k\pi x}{2^m} \rangle_0, \langle \sin \tfrac{2k\pi x}{2^m} \rangle_1) - \sum_{k=1}^K a_k \sin \tfrac{2k\pi x}{2^m})|\\
\le \; & \textstyle\sum_{k=1}^K |a_k| |\text{Rec}(\langle \sin \tfrac{2k\pi x}{2^m} \rangle_0, \langle \sin \tfrac{2k\pi x}{2^m} \rangle_1) - \sin \tfrac{2k\pi x}{2^m}|\\
\le \; & \textstyle\sum_{k=1}^K |a_k| \cdot 2^{-f+1}
\end{aligned}$$

This concludes the proof of (i). When K is sufficiently large, the $\epsilon \to 0$. 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 [21], where p is the probability for error truncation. In our case, we can just take $l = n$ and $l_x = \lceil \log 2(|x|) \rceil + f$ into [21] to get probability p.

B BACKPROPAGATION AND AUTO-DIFFERENTIATION

B.1 A Training Example

We illustrate the training process using Network A [33]. Network A contains three dense layers and ReLU activations, followed by the loss function `CrossEntropyWithSoftmax` (SoftmaxCE). Network A's structure is as follows,

$$\begin{tikzcd} x \arrow[r] & \boxed{\begin{array}{c}\text{Dense}\\ \boxed{w_1}\end{array}} \arrow[r] & \boxed{\text{ReLU}} \arrow[r] & \boxed{\begin{array}{c}\text{Dense}\\ \boxed{w_2}\end{array}} \arrow[r] & \boxed{\text{ReLU}} \arrow[r] & \boxed{\begin{array}{c}\text{Dense}\\ \boxed{w_3}\end{array}} \arrow[r] & \boxed{\text{Loss}} \end{tikzcd}$$

Following Network A's architecture, we build `class NETWORKA` in the following code snippet.

```
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 loss are imported as if using TensorFlow. At each training iteration, forward propagation and backward propagation are performed layer by layer in a secret-shared manner.

B.2 Derivative for Auto-differentiation

B.2.1 Softmax. Let $g_i = \text{Softmax}(\vec{y})_i = e^{y_i}/\sum_j e^{y_j}$ and meantime $g = \text{Softmax}(\vec{y})$. 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 $\text{diag}(g)$ 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 phase of backpropagation, the secret-shared g can be directly obtained from the forward propagation computed by Π_{QSMax}. Given secret-shared g, the $\text{diag}(g)$ and $g^\top$ are obtained by local transformation, i.e., diagonalization and matrix transpose. The $\frac{\partial \text{loss}}{\partial g}$ 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 protocol of secure multiplication $\Pi_\times$. The right part $gg^\top \frac{\partial \text{loss}}{\partial g}$ can be securely computed by secure multiplication $\Pi_\times$ from right to left. In our implementation, we use the loss of `CrossEntropyWithSoftmax`, which could be simplified (see below).

B.2.2 CrossEntropyWithSoftmax. Let $g_i = \text{Softmax}(\vec{y})_i = e^{y_i}/\sum_j e^{y_j}$ and $g = \text{Softmax}(\vec{y})$. According to `CrossEntropyWithSoftmax` (SoftmaxCE), we get,

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

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

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

B.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 (see below).

B.2.4 BinaryCrossEntropyWithSigmoid. According to 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,

$$\frac{\partial \text{SigmoidBCE}}{\partial g} = -\frac{\hat{y}}{g} + \frac{1-\hat{y}}{1-g}$$

Applying the chain rule,

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

C SECURITY ANALYSIS AND PROOF

We provide a theoretical analysis of the proposed protocols and their security proofs. The security proof models the real-world execution and ideal-world simulation paradigm. Consider an adversary $\mathcal{A}$, the environment $\mathcal{Z}$, and the simulator $\mathcal{S}$. In the real-world execution of protocols, the parties execute the protocol in the presence of an adversary $\mathcal{A}$ and the environment $\mathcal{Z}$. In the ideal world, the parties send their inputs to the simulator $\mathcal{S}$ that computes the functionality $\mathcal{F}$ truthfully.

The security requires that for every $\mathcal{A}$ in the real interaction, a simulator $\mathcal{S}$ exists in the ideal interaction such that the environment $\mathcal{Z}$ cannot distinguish reality and unreality. That is, whatever knowledge the $\mathcal{A}$ can extract in the real world, the $\mathcal{S}$ can also extract it in the ideal world. Below, Algorithm 3 and Algorithm 4 define the ideal functionality solved by Π_{QSMax} and Π_{LSig}, respectively.

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=1,2,3,...,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
```

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 $\delta_x = (x - t) \mod 32$
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
```

C.1 Proof of Theorem 4.2

Proof. Recall that Π_{QSMax} calls the $\Pi_\times$ [21] twice, which are semi-honest secure. From the view of security proof, Π_{QSMax} sequentially integrates $\Pi_\times$ with non-interactive local computations. Consider an adversary $\mathcal{A}$, the environment $\mathcal{Z}$, and the simulator $\mathcal{S}$. In the real-world execution of protocols, the parties execute Π_{QSMax} and communicate with adversary $\mathcal{A}$ in environment $\mathcal{Z}$. In the ideal world, the simulator $\mathcal{S}$ computes the functionality $\mathcal{F}$ truthfully by the inputs forwarded by the parties.

Π_{QSMax} sequentially composes the $\Pi_\times$ twice. We can regard them as sequential usage since each subroutine calling is independent. We have known that $\Pi_\times$ is against semi-honest PPT adversaries with static corruption. If a semi-honest adversary $\mathcal{A}$ can attack Π_{QSMax}, $\Pi_\times$ can be attacked successfully. However, this statement contradicts the knowledge that $\Pi_\times$ [21] provides the semi-honest security. Thus, for every $\mathcal{A}$ in the real interaction, the environment $\mathcal{Z}$ cannot distinguish the execution between a simulator $\mathcal{S}$ and an adversary $\mathcal{A}$. □

C.2 Proof of Theorem 5.2

Proof. We construct Π_{LSig} from sketch, thus requiring slightly different offline randomness from prior literature. In the ideal world, the parties communicate with the ideal functionality $\mathcal{F}_{\mathsf{LSig}}$. Our goal is to prove that no PPT environment $\mathcal{Z}$ can distinguish between the real execution Π_{Sin} and the ideal execution.

To be specific, two parties input $\langle x\rangle_0, \langle x\rangle_1$ to $\mathcal{F}_{\mathsf{LSig}}$. Upon receiving the input, $\mathcal{F}_{\mathsf{Sin}}$ reconstructs $x = \langle x\rangle_0 + \langle x\rangle_1$ and compute δ_x. Given $t, \vec{k}, \delta_x$, $\mathcal{F}_{\mathsf{Sin}}$ computes $\sin(\delta_x\vec{k}), \cos(t\vec{k}), \cos(\delta_x\vec{k})$, and $\sin(t\vec{k})$ At last, the $\mathcal{F}_{\mathsf{Sin}}$ constructs the y and generates the shares as the functionality outputs.

With the symmetry of P_0 and P_1), we consider P_0 is corrupted 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 behave of P_1.

Table 12: Kullback-Leibler Divergence

#Class	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}

Game G_0: This is equivalent to the real protocol execution. The corrupted parties are controlled by the adversary $\mathcal{A}$ who simply forwards messages from/to $\mathcal{Z}$.

Game G_1: G_1 is the equivalent to G_0 except that P_1 sends a random $\tilde{\delta}_{\langle x\rangle_1}$ to P_0 instead of $\delta_{\langle x\rangle_1}$. Since $\tilde{\delta}_{\langle x\rangle_1}$ is computationally indistinguishable from true random, the distribution of $\delta_{\langle x\rangle_1}$ is also computationally indistinguishable from true random regardless of $\langle x\rangle_1$. Therefore, $G_1 \approx G_0$ follows.

□

D MEASUREMENT OF PROBABILITY DISTRIBUTION

Kullback-Leibler (KL) divergence calculates a score that measures the (asymmetry) distance of one probability distribution from another. Table 12 records the KL divergence between the pure Softmax function and the Softmax approximators. 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 has lower Kullback-Leibler divergence at 10 and 100 classes, it produces off-the-chart results when the class size scales to 1000 and 10000. Instead, QMax could achieve accurate approximation when the class size is very large (say, 10, 000).

E CHOICE OF PARAMETERS

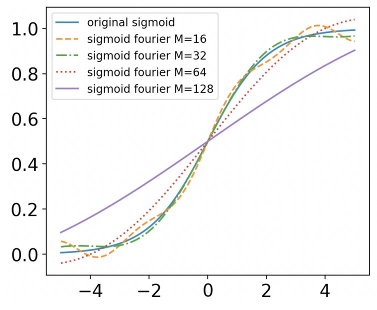


Figure 2: Approximation accuracy with different m

Figure 2 shows the approximation when $m = 4, 5, 6, 7$ with periodic intervals of $[16, 32, 64, 128]$. For $K = 5$ with sufficient accuracy, we can see that the generally better approximation is the case of $m = 4$. We consider the interval of $[-a, a]$ not $[0, a]$ for removing the requirement of secure comparison. The interval $[0, a]$ works in plaintext. For secret-shared inputs, the computing parties do not know whether the input value is positive/negative. Thus, additional secure comparison is desired yet with more computational and communication overhead.

F ARTIFACT EVALUATION

F.1 Abstract

Our open-source code is available at https://github.com/alipay/Antchain-MPC/tree/sec_softmoid. The artifact consists of CPU and GPU implementation prototypes in Python and C++, respectively. Besides, we provide scripts for reproducing experimental results in Section 7. Artifact evaluation is to reproduce the results of replicating Tables 3-12 in the paper by executing the protocol-level and end-to-end training benchmarks. The evaluation consists of communication, running time, and training accuracy. As for GPU implementation in Piranha [35], participating parties require a machine equipped with a GPU and access to the NVIDIA CUDA toolkit.

The Python implementation (Apache license) follows the TensorFlow programming styles for providing user-oriented APIs. It supports both simulation by a local server and distributed training among three servers. In the Python prototype, we adopted the ideas of [13, 21, 28, 35] regarding our security model. In the offline phase, we realize a secure deterministic random bit generator (DRBG) conforming to CTR_DRBG standardized in NIST Special Publication 800-90A. The C++ implementation (MIT license) contains the plug-in blocks in the Piranha [35], known as a general-purpose solution for supporting later-inserting blocks. We added the blocks of Sigmoid protocol and Softmax protocol and truncation [21] by following Piranha's programming interface.

F.2 Artifact Checklist

- Code license and public available link.
- Python implementation.
- C++ implementation.
- Experimental scripts for Tables 3-12.
- README.md files for installation and running scripts.

F.3 Python Implementation

Figure 3 displays implementation architecture, providing user-oriented APIs and extensive bottom-level engines.

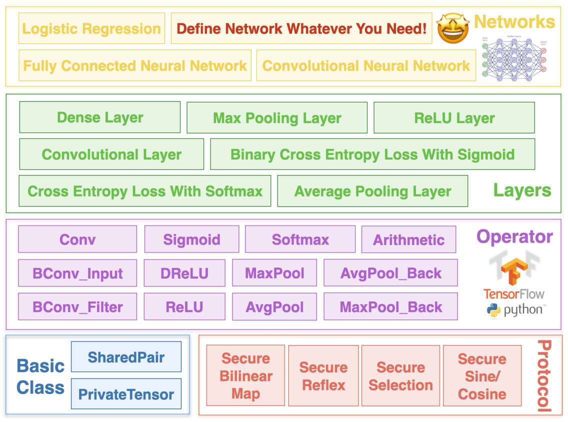


Figure 3: Framework Architecture

F.3.1 Bottom-level Engine. Figure 3 includes basic classes, basic protocols, operators, layers, and networks. Common layers and sufficient operators are built on basic classes and protocols for providing user-oriented APIs.

Data types. We present two basic classes, defined to be fixed-point numbers of INT64. PRIVATETENSOR represents private data owned by one party, while SHAREDPAIR represents secret shares owned by two parties. SecureML's truncation [21] and Cheetah's truncation [13] are supported.

PRF-instantiated randomness. We realize a secure deterministic random bit generator (DRBG) conforming to CTR_DRBG standardized in NIST Special Publication 800-90A. We use AES-NI and generate multiple random number streams simultaneously in one DRBG to fill up the pipelines of AES-NI execution units.

Basic protocols. We define four basic protocols, including SECURE BILINEAR MAP, SECURE SELECTION, SECURE REFLEX, and SECURE SINE/COSINE, which are called by operators. Variables in basic protocols belong to basic classes or other common classes.

Operators. We define operators to represent common training functions, arithmetic operations (add, sub, matrix multiplication, etc.), and forward/backward propagation. Operators can be regarded as a lite version of cryptographic NumPy[8], including Sigmoid, Softmax, convolution, convolutional backpropagation, etc. Also, users could create new operators by calling basic classes and protocols.

F.3.2 User-Oriented APIs and Usage. Following TensorFlow's programming style, users could build layers and neural networks they expect. For the non-developing adoption, we publish Python packages PyPI link (to be added in the final version after AE) in PyPI for simple installation.

Load data and start servers. Python implementation supports loading training/test data from a local server (i.e., a light simulation with different ports on a MacBook) or distributed servers. It relies on TensorFlow 2 and Python 3.6. Before training, users are required to install requirements.txt. Then, the users require to configure and start three servers for the distributed training.

Training/inference. For easy usage and convenient reference, Python implementation shows multiple training/inference examples, saying, AlexNet [17], LeNet [18], VGG16 [30], etc.

F.4 C++ Implementation

F.4.1 The Architecture of Piranha. Piranha [35] is a general-purpose, modular platform for accelerating secret sharing-based MPC protocols using GPUs. It contains three state-of-the-art linear secret sharing MPC protocols for secure NN training: 2-party SecureML [21], 3-party Falcon[34], and 4-party FantasticFour [7].

Piranha devises a three-layer architecture – device, modular protocol, and application layers.

- Device layer – independently accelerating secret-sharing protocols by providing integer-based kernels.
- Modular protocol layer – maximize the utility of limited GPU memory with in-place computation and iterator-based support for non-standard memory access patterns.

[8]https://numpy.org

- Application layer – allow applications to remain completely agnostic to their underlying protocols.

F.4.2 New Plug-in Protocols. We added plug-in protocols for securely computing Softmax and Sigmoid in the application layer. In particular, Protocol 1 and Protocol 2 have been implemented as functions in the codes. By replacing new functions with previous ones, we can benchmark the training in Piranha.

F.5 Evaluation and Experimental Scripts

F.5.1 Preparation and Description.

- Program: Python and C++ languages, Docker, TensorFlow 2, and CUDA 11.6.
- Metric: communication, training time, and model accuracy.
- Datasets: MNIST, CIFAR-10.
- Outputs: Results replicate Tables 3-12. Running time would be different on different servers.
- Hardware dependence: NVIDIA GPUs are required for micro-benchmarks and macro-benchmarks in Piranha.
- Time: It roughly takes 1 $\sim$ 2 hours to set up and 2 hour to run most experiments (except for model accuracy). The time may vary depending on the power of the server.
- Server reference: We have tested our codes on three types of servers, including (1) Alibaba cloud servers equipped with 8-core 2.50GHz CPU of 64GB RAM and NVIDIA-T4 GPU of 16GB RAM, (2) the commodity server equipped with two 24-core 2.10GHz CPUs and two NVIDIA-A40 GPUs of 48GB RAM, and (3) MacBook Pro (CPU only).
- Public availability: The code is available at https://github.com/alipay/Antchain-MPC/tree/sec_softmoid. Since the code repository is owned by anonymous company. and not under the management of the authors permanently, the authors maintain a mirror repository at https://github.com/MathCrypt0/softmoid-public by forking the original repository for permanent availability.
- Code licenses: The Python implementation is under Apache license according to code-disclosure rules in Ant Group. The C++ implementation follows Piranha’s license – MIT license.

F.5.2 Protocol-level Communication. In the cpu folder, run the script of experiment_table_3&4.py to obtain experimental results of Tables 3,4.

F.5.3 CPU-based Experiments. Follow the README.md instruction in cpu folder for installation. Then, run the script run.sh to obtain results of Tables 7,8,9,11,12. Since training for model accuracy costs a relatively long time (i.e., several hours), we put the relevant experiments at last.

F.5.4 GPU-based Experiments. Follow the README.md instruction in gpu folder to install C++ implementation. Then, download the MNIST and CIFAR-10 datasets. At last, execute the script run.sh in the Docker container to get results of Tables 5,6,10.

F.6 Remarks

To make artifact evaluation easier, we write experimental scripts for benchmarks using the mode of local server, i.e., using a single IP with multiple hosts. For the distributed training, three different servers are required to be initialized to communicate with each other. For user convenience, we attach the record of parameter tuning in the file ./cpu/artifacts/record.csv.

Disclaimer. Intellectual properties have been protected by Chinese patents and are free for academic usage. For business usage in the Chinese market, please get in touch with Morse team[9] at Ant Group.

[9]https://antdigital.com/products/morse