PoPETs

Proceedings on Privacy Enhancing Technologies ; 2022 (4):746–767

Nuttapong Attrapadung, Koki Hamada, Dai Ikarashi, Ryo Kikuchi*, Takahiro Matsuda, Ibuki Mishina, Hiraku Morita, and Jacob C. N. Schuldt

Adam in Private: Secure and Fast Training of Deep Neural Networks with Adaptive Moment Estimation

Abstract: Machine Learning (ML) algorithms, especially deep neural networks (DNN), have proven themselves to be extremely useful tools for data analysis, and are increasingly being deployed in systems operating on sensitive data, such as recommendation systems, banking fraud detection, and healthcare systems. This underscores the need for privacy-preserving ML (PPML) systems, and has inspired a line of research into how such systems can be constructed efficiently. However, most prior works on PPML achieve efficiency by requiring advanced ML algorithms to be simplified or substituted with approximated variants that are "MPC-friendly" before multi-party computation (MPC) techniques are applied to obtain a PPML systems. A drawback of this approach is that it requires careful fine-tuning of the combined ML and MPC algorithms, and might lead to less efficient algorithms or inferior quality ML (such as lower prediction accuracy). This is an issue for secure training of DNNs in particular, as this involves several arithmetic algorithms that are thought to be "MPC-unfriendly", namely, integer division, exponentiation, inversion, and square root extraction. In this work, we take a structurally different approach and propose a framework that allows efficient and secure evaluation of full-fledged state-of-the-art ML algorithms via secure multi-party computation. Specifically, we propose secure and efficient protocols for the above seemingly MPC-unfriendly computations (but which are essential to DNN). Our protocols are three-party protocols in the honest-majority setting, and we propose both passively secure and actively secure with abort variants. A notable feature of our protocols is that they simultaneously provide high accuracy and efficiency. This framework enables us to efficiently and securely compute modern ML algorithms such as Adam (Adaptive moment estimation) and the softmax function "as is", without resorting to approximations. As a result, we obtain secure DNN training that outperforms state-of-the-art three-party systems; our *full* training is up to 6.7 times faster than just the *online* phase of FALCON (Wagh et al. at PETS'21) and up to 4.2 times faster than Dalskov *et al.* (USENIX'21) on the standard benchmark network for secure training of DNNs. The potential advantage of our approach is even greater when considering more complex realistic networks. To demonstrate this, we perform measurements on real-world DNNs, AlexNet and VGG16, which are large networks containing millions of parameters. The performance of our framework for these networks is up to a factor of $26 \sim 33$ faster for AlexNet and $48 \sim 51$ faster for VGG16 to achieve an accuracy of 60% and 70%, respectively, when compared to FALCON. Even compared to CRYPTGPU (Tan *et al.* IEEE S&P'21), which is optimized for and runs on powerful GPUs, our framework achieves a factor of 2.1 and 4.1 faster performance, respectively, on these networks.

Keywords: MPC, fixed-point arithmetic, deep learning

DOI 10.56553/popets-2022-0131

Received 2022-02-28; revised 2022-06-15; accepted 2022-06-16.

1 Introduction

Secure multi-party computation (MPC) [13, 30, 67] enables function evaluation, while keeping the input data secret. An emerging application area of secure computation is privacy-preserving machine learning (ML), such as (secure) deep neural networks. Combining secure computation and deep neural networks, it is possible to gather, store, train, and derive predictions based on data, which is kept confidential. This provides data

***Corresponding Author: Ryo Kikuchi:** NTT, E-mail: kikuchi_ryo@fw.ipsj.or.jp
Nuttapong Attrapadung, Takahiro Matsuda, Jacob C. N. Schuldt: AIST, E-mail: {n.ttrapadung, t-matsuda, jacob.schuldt}@aist.go.jp
Koki Hamada, Dai Ikarashi, Ibuki Mishina: NTT, E-mail: {koki.hamada.rb, dai.ikarashi.rd, ibuki.mishina.br}@hco.ntt.co.jp
Hiraku Morita: University of St. Gallen, E-mail: hiraku.morita@unisg.ch

security and encourages data holders to share their confidential data for machine learning. As a consequence, it becomes possible to use a large amount of data for model training and obtain accurate predictions.

We first briefly review a typical *training* (or learning) process of a deep neural network in the clear (*i.e.,* without secure computation). A deep neural network (DNN) consists of several types of interconnected layers, and is evaluated on the training data sequentially in a layer-by-layer manner. Each layer might contain a set of neurons, which are activated by an activation functions such as ReLU in intermediate layers or the *softmax* function in the final layer. The strength of the connections of the neurons to the output of the previous layer are the parameters of the network. Then, the tentative output from the network is compared to the intended classification, and based on this, the parameters are updated via an optimization method. This process is iterated several times over the training data. A traditional optimization method is *stochastic gradient descent (SGD)*. As SGD tends to incur many repetitions (and hence slow convergence), more efficient approaches have been proposed; adaptive gradient methods such as *adaptive moment estimation (Adam)* [37] are popular optimization methods which improve upon SGD and are adopted in many real-world tool-kits, *e.g.,* [6].

A key challenge towards privacy-preserving ML, especially for DNN, is how to securely compute functions that are *not* "MPC-friendly". MPC-friendly functions refer to functions that are easy to securely compute in MPC, and for which very efficient protocols exist. However, unfortunately, functions required in DNN are often *MPC-unfriendly*, especially those used in more modern approaches to training. In particular, Adam [37] (and also the softmax function) consist of several MPC-unfriendly functions, namely, integer division, exponentiation, inversion, and square root computations.

To cope with this challenge, up to now, there have been two lines of research. First, many works (to name just a few, [14, 15, 18, 19, 28, 38, 39, 45, 53, 57, 58]) have focused mainly on secure protocols for the *prediction* (or inference) process only, which is significantly more lightweight compared to the *training*, as gradient optimization methods are not required for prediction. Second, and more recently, there have been a few works in the literature that can handle secure training. These are done mostly by replacing originally MPC-unfriendly functions with different ones that are *MPC-friendly* and approximate the original function on the domain of interest. These approximation approaches either can be done only for elementary optimization methods such as SGD, as in [20, 50, 51, 63] or require specific "fine-tuning" of the interaction between ML and MPC, as in [8], such that the replaced functions will not degrade the quality of ML architectures significantly (such as lowering prediction accuracy). In practice, however, this replacement is not easy. For example, Keller and Sun [34] reported that ASM, which is widely used as a replacement for the softmax function, reduces accuracy in training, sometimes significantly.

Due to the rapid advancements in ML, we believe that a more robust approach to privacy-preserving ML is to achieve efficient protocols for a set of functions that are often used in ML but might typically be thought of as MPC-unfriendly. In this way, the requirement for fine-tuning between ML and MPC would be only minimal, if any at all, and one would be able to plug-and-play new ML advancements into an existing MPC framework to obtain new privacy-preserving ML protocols, without having to worry about the degradation on the ML side.

1.1 Our Contributions

We present a framework that allows seamless implementation of secure training for DNNs using modern ML algorithms. Specifically, our contribution is twofold as follows.

New Elementary Three-party Protocols. We propose new secure and efficient protocols for a set of elementary functions that are useful for DNN but are normally deemed to be MPC-unfriendly. These include secure division, exponentiation, inversion, and square root extraction. Our protocols are three-party protocols in the honest-majority setting, and we propose both passively secure and actively secure with abort variants. A notable feature of our protocols is that they simultaneously provide high accuracy and efficiency. A key component to this is our new division protocol, which enables secure fixed-point arithmetic. Previous direct fixed-point arithmetic protocols [24, 50] has quadratic communication cost (in bits) in the ring/field size, and [50] and related protocols introduce errors with a certain probability which must be mitigated, typically resulting in an increased overhead or reduced accuracy. In contrast, our protocol has linear communication cost, and requires no error mitigation step. Combined with a range of optimizations suitable for each of the functionalities we consider, we obtain a set of protocols that are both efficient and accurate. In fact, our implementations of our protocols provide efficient 23-bit accuracy fixed-point arithmetic, which is comparable to single-precision real number operations *in the clear*. Reaching this level of accuracy is an important aspect of our approach to PPML systems as the commonly used standard for DNN training in the clear is single-precision computation (*e.g.,* TensorFlow [6], PyTorch [5]), and our aim is to implement these without degrading training accuracy. We further discuss about accuracy in Section 1.3. We describe our construction techniques in the next subsection.

New Applications to ML. We apply our new elementary MPC protocols to "seamlessly" instantiate secure computations for softmax and Adam. That is, due to our elementary MPC protocols, we can securely and efficiently compute softmax and Adam "as is", in particular, without approximation using (MPC-friendly) functions or sacrificing training accuracy. Consequently, due to the fast convergence of Adam, we obtain fast and secure training (and prediction) protocols for DNN. Using the DNN architecture and MNIST dataset typically used as a benchmark, our protocol achieved 95.64% accuracy within 117 seconds, improving upon the state-of-the-art such as ABY3 [50] (94% accuracy within 2700 seconds reported in [50]) and FALCON [64] (780 seconds for the online phase only) in the passive security setting[1]. In the active security setting, our protocol completed training within 570 seconds, improving upon the three-party variant of Fantastic Four [25] (95.43% accuracy within 1879 seconds). Furthermore, our protocol achieves the same accuracy as training over *plaintext* data, using TensorFlow [6]. We further perform measurement on real-world DNNs from the ML literature, AlexNet [41] and VGG16 [61], which contain millions of parameters. Comparing the total training time (i.e. time to reach a certain accuracy), the *full* running time of our framework outperforms the *online* phase of FALCON[2] with a factor of about $12 \sim 14$ for AlexNet and $46 \sim 48$ for VGG16 in the LAN setting. Our framework even outperforms CRYPTGPU [62], which draws upon the computational power of GPUs to implement efficient protocols, with a factor of 1.8 and 2.0, respectively, for these networks. A detailed performance evaluation and comparison considering different security and network settings, different datasets, and large DNNs, is given in Section 6.

1 The measurements for our protocol and FALCON were done in the environment described in Section 6, which is roughly comparable to the one in [50].

2 FALCON requires an offline phase to be executed before the online phase, whereas our framework consists of standard MPC protocols not requiring an offline phase. In the comparison, we do not include the execution time of the FALCON offline phase, which favors FALCON in the comparison.

1.2 Our Techniques

New Techniques for Secure Truncation. We first briefly describe the idea behind a common building block for all our protocols: division (which also implies truncation). Let p be the size of the underlying ring/-field, x be the secret and d is the divisor (so the desired output is $\frac{x}{d}$). Known efficient truncation protocols, *e.g.,* [50, 51], reconstruct a masked secret $x+r$ for a random r, divide this by d in the clear, and subtract $\frac{r}{d}$. However, in this approach, a large error, $-\frac{p}{d}$, sneaks into the output when $x+r > p$ because the reconstructed value becomes $x+r-p$. To avoid this, the message space has to be much smaller than p, which leads to reduced accuracy for a given value of p. Dalskov *et al.* [24] avoid this error by detecting if the reconstructed value is $x+r-p$ or not efficiently. However, preparing r and $\frac{r}{d}$ still requires quadratic communication to the ring/-field size. We employ a different approach. Let x_1 and x_2 be additive shares of x such that $x_1 + x_2 = x + qp$ for $q \in \{0, 1\}$. Our approach is essentially to securely compute q and eliminate qp (without exposing q to any parties), which makes the (local) division of sub-shares be the desired output. Through this approach, we do not need to prepare $r, \frac{r}{d}$ that causes quadratic communication, and we can embed a large value into a single share, which, in turn, enables accurate computation of functions such as exponentiation. We note that our approach has the technically interesting property that the ideal functionalities for truncation and division depend not only on the input/output, but also on the randomness of shares held by the computing parties. While this makes the corresponding theoretical analysis more involved, we formally establish correctness and security, and as a result obtain a truncation protocol which provides higher accuracy and better overall performance compared to previous works.

New Techniques for Elementary Protocols. For securely computing exponentiation, inversion, division with private divisor, square root, and inversion of square root, we utilize Taylor or Newton series expansions. A key challenge here is to ensure fast convergence that, in general, is only guaranteed for a narrow range of input values. We resolve this by constructing protocols that use a combination of private input pre-processing and partial evaluation of the pre-processed input. We

Table 1. Categorization of PPML systems for neural networks

Secure Prediction	Secure Training	Supporting Advance ML†	PPML systems		
			2-party	3-party	4-party
●	○	○	MiniONN [45], Chameleon [57], EzPC [18], Gazelle [33], XONN [56], Delphi [49]	CryptFlow [42], ASTRA [19], QuantizedNN [24], BLAZE [53]	FLASH [15]
●	○	◐	PrivEdge [60],	-	-
●	●	○	SecureML [51] FHE-based SGD [52] Glyph [46]	ABY3 [50], SecureNN [63], FALCON [64], CRYPTGPU [62], FantasticFour [25]	Trident [55] FantasticFour [25]
●	●	◐	-	Lu *et al.* [47]	-
●	●	●	Quotient [8]	**This work**	-

† Supporting Advance ML refers to systems that can support beyond the SGD optimization. In particular, our work supports Adam, while Quotient supports AMSGrad. ◐ means partial supports (PrivEdge and BAYHENN use Adam but in the clear, while [47] provided essentially only an inverted square-root protocol).

devise *private scaling* techniques, which allow inputs to be scaled to fit an optimal input range, and furthermore allow the protocol to make the most out of the available bit range in the internal computations. We also utilize what we call *hybrid table-lookup/series-expansion* techniques, which separate inputs into two parts and apply table-lookup and series-expansion to the respective parts. The details of how these techniques are used in our protocols differ depending on the functionality of the protocols. We provide the details in Section 4.

1.3 Comparison to Related Works

Various ML algorithms have been considered in connection with privacy preserving ML, include decision trees, linear regression, logistic regression, support-vector-machine classifications, and deep neural networks (DNN). Among these, deep neural networks are the most flexible and have yielded the most impressive results in the ML literature. However, at the same time, secure protocols for DNN are the most difficult to obtain, especially for the training process. We show a brief comparison among PPML systems supporting DNN in Table 1, and also provide a more comprehensive comparison table in Appendix A.

Secure DNN Training. Our work focuses on *secure training for deep neural networks* (secure inference can be obtained as a special case). There have been several works on secure DNN training such as SecureML [51], SecureNN [63], ABY3 [50], Quotient [8], FHE-based SGD [52], Glyph [46], Trident [20], FALCON [64], Fantastic Four [25], and CRYPTGPU [62]. All of these achieve efficiency by simplifying the underlying DNN training algorithms (e.g. replacing functionalities with less-accurate easier-to-compute alternatives), and optimizing the computation of these. As a consequence of this approach, they are restricted to simple SGD optimization, with the exception of Fantastic Four which implements Momentun (a SGD variant), and Quotient which implements an approximation to AMSGrad. These works furthermore implement training with reduced computation accuracy, and typically argue (*e.g.,* FALCON [64]) that this does not significantly degrade training accuracy. We note that these arguments are based on experiments on smaller networks and data sets, like 3DNN on MNIST, and do not imply that a similar conclusion holds for larger more complex networks such as AlexNet and VGG16 (which we experiment on in this paper). Furthermore, specific ML components might require more accuracy than others[3]. We emphasize that we take a structurally different approach by constructing protocols that allow unmodified advanced training to be implemented "as is", without simplifications or sacrificing computation accuracy. To the best of our knowledge, the only the work that suggests secure training with ADAM is by Lu *et al.* [47]. They provided an inverse square root protocol required by ADAM, but its accuracy is far from the 23-bit accuracy we guarantee in this paper. In addition, how MPC-unfriendly functions, such as softmax, can be implemented is not described, and in-depth experimental results with training accuracy is not provided. In the following, we highlight properties of the above related works.

Security. SecureML, Quotient, FHE-based SGD, Glyph, and SecureNN considered *semi-honest (passive)* security tolerating one corrupted party, while SecureNN can be extended to achieve so-called *privacy against malicious adversaries* (formalized by [12]). ABY3 improved security upon these by considering *malicious (active) security with abort* tolerating one corrupted party, which is also the security notion considered for Fantastic Four. Trident improved security in term of *fairness* (again, tolerating one corrupted party); this comes with the cost of reducing the tolerated corruption fraction from 33% to 25%. It should be noted that, unlike the other schemes, FALCON sacrifice full security to compute batch-normalization more efficiently (see Section 5.2).

Efficiency. For secure training over a basic 3-layer DNN on the MNIST dataset, ABY3 outperforms both SecureML/SecureNN and was state-of-the-art before Trident and FALCON. FHE-based SGD and Glyph use fully homomorphic encryption, which makes non-

3 For example softmax, which is not implemented in FALCON. In fact, in the existing mixed-precision libraries [4, 7], the softmax function requires 23-bit accuracy, while the other operations are done using half-precision.

interactive training possible. Glyph is the most efficient of the two, but is still far less efficient than ABY3 in terms of execution time. Trident improves the *online* phase of ABY3 but with the cost of adding a fourth party who only participates in offline phase. Most recently, FALCON improves upon the *online* phase of ABY3, and Fantastic Four improves upon FALCON in the active security setting. Finally, CRYPTGPU leverage the power of GPUs to construct very efficeint protocols. As highlighted above, our framework improves upon both FALCON and Fantastic Four, and can even outperform CRYPTGPU for large networks.

Additional Related Works. When confining to only secure *inference/prediction* (*i.e.,* without secure training) for DNN, there are works that uses Adam in the training phase (but *in the clear*) such as PrivEdge [60][4] and BAYHENN [66][5]. Also for secure inference, BLAZE [53] achieved a strong security called *fairness*. For more related works, we refer to a very recent comprehensive survey in [68].

2 Preliminaries and Settings

Notations for Division. For $a, b \in \mathbb{Z}$, we denote by $\frac{a}{b} \in \mathbb{R}$ real-valued division, and by $a/b \in \mathbb{Z}$ integer division that discards the remainder. That is, $a/b := \lfloor \frac{a}{b} \rfloor$.

Data Representation. Our protocols operate on binary values $\mathbb{Z}_2$, ℓ-bit unsigned and signed integers, denoted $\mathbb{Z}^+_{\langle \ell \rangle}$ and $\mathbb{Z}_{\langle \ell \rangle}$, as well as ℓ-bit fixed-point unsigned and signed rational numbers $\mathbb{Q}^+_{\langle \ell,u \rangle} = \{b \in \mathbb{Q} \mid b = \frac{a}{2^u}, a \in \mathbb{Z}^+_{\langle \ell \rangle}\}$ and $\mathbb{Q}_{\langle \ell,u \rangle} = \{b \in \mathbb{Q} \mid b = \frac{a}{2^u}, a \in \mathbb{Z}_{\langle \ell \rangle}\}$, respectively. We will represent integers and fixed-point values as elements of $\mathbb{F}_p$. In order to do so, the latter are scaled to become integers. Specifically, we will use a set of (unsigned) ℓ-bit integers $0 \le a \le 2^\ell - 1$, which we denote $\widehat{\mathbb{Q}}^+_{\langle \ell,\alpha \rangle}$, to represent the values $\{\frac{a}{2^\alpha} | 0 \le a \le 2^\ell - 1\}$, and will refer to α as the *offset* for these. For a fixed-point value a, we will use the notation $a_{\langle \alpha \rangle}$ to denote the integer representation with offset α i.e. $a_{\langle \alpha \rangle} = a \cdot 2^\alpha$. The integers in $\widehat{\mathbb{Q}}^+_{\langle \ell,\ell \rangle}$ are represented as elements of $\mathbb{F}_p$, and we denote the signed extension by $\widehat{\mathbb{Q}}_{\langle \ell,\ell \rangle}$. Note that the representation of fixed-point values requires the scaling factor to be taken into account for multiplication (and division). Specifically, for values $a_{\langle \alpha \rangle}$ and $b_{\langle \alpha \rangle}$, the correct representation of the product of a and b is $a_{\langle \alpha \rangle} \cdot b_{\langle \alpha \rangle} / 2^\alpha = (a \cdot b)_{\langle \alpha \rangle}$. For simplicity, we use $\times_\alpha$ to denote this operation, i.e. (ordinary) multiplication followed by division by 2^α.

Multi-Party Computation Setting. We consider secret-sharing (SS)-based three-party computation secure against single corruption in the client/server model. In this model, both the input and output of the parties are in a secret-shared form, and our protocols are thus share-input and share-output protocols. In the context of PPML systems, any number of clients can contribute to the data set used for training by sharing their data to the parties. P_1, P_2, P_3 denote the three parties and treat the party index i as to refer to the i'-th party where $i' \equiv i \pmod 3$ and $i' \in \{1, 2, 3\}$. For example, $P_0 = P_3$ and $P_4 = P_1$. Regarding the adversarial behavior, we consider both passive and active adversaries with abort. We consider the standard security definition for these settings (e.g. see [35]) – due to space limitations, these are deferred to the full version.

Secret Sharing Schemes and Their Protocols. In this paper, we use three *replicated secret sharing schemes* [23, 32]. We consider the 2-out-of-3 threshold access structure for the first two schemes. For the third scheme, the minimal access structure is simply $\{\{1, 2\}\}$, meaning only P_1 and P_2 can together reconstruct the secret. We denote them as:

$$\begin{aligned} [\![\cdot]\!]\text{-sharing} \quad &: \text{2-out-of-3 replicated sharing in } \mathbb{Z}_p, \\ [\cdot]\text{-sharing} \quad &: \text{2-out-of-3 replicated sharing in } \mathbb{Z}_2, \\ \langle\!\langle \cdot \rangle\!\rangle\text{-sharing} \quad &: \text{simple additive sharing in } \mathbb{Z}_p. \end{aligned}$$

Addition and scalar multiplication can be done using only local operations for these schemes, whereas multiplication requires a dedicated protocol [22, 26, 27]. We denote the multiplication functionality as $\mathcal{F}_{\text{mult}}$.

Share Conversions. Our protocols will utilize conversions among share types. We provide a summary of conversion protocols in Table 2. Here, for $a \in \mathbb{Z}_p$ we let $(a_\ell, \ldots, a_1)$ be the bit representation of a; that is, $a = \sum_{i=1}^{\ell} 2^{i-1} a_i$. The given round complexity is for passively secure protocols. Lastly, note that unlike the original bit-composition protocol in [10], we use a protocol that can be applied to $\mathbb{Z}_p$. In Appendix F we highlight how this modification is achieved.

Conditional Assignment. We define a functionality of conditional assignment $[\![z]\!] \leftarrow \mathcal{F}_{\text{CondAssign}}(a, b, [\![c]\!])$ by setting $z := a$ if $c = 0$ and $z := b$ if $c = 1$. A protocol for this simply computes $[\![z]\!] := a \cdot (1 - [\![c]\!]) + [\![c]\!] \cdot b$.

4 PrivEdge [60] has a different configuration from other works mentioned here; it lets users independently train their own data locally (hence, in particular, training is done in plaintext).

5 This work has a vulnerability found by [65]

Conversion	Functionality name	Protocol	Round
$[\![a]\!] \to \langle\!\langle a \rangle\!\rangle$	ConvertToAdd	local operations	0
$\langle\!\langle a \rangle\!\rangle \to [\![a]\!]$	ConvertToRep	one $[\![\cdot]\!]$-sharing	1
$[\![a]\!] \to ([a_1], \ldots, [a_\ell])$	$\mathcal{F}_{\mathrm{BDC}}$ – Bit decomposition	[36] (*cf.* §F)	$\ell + 1$
$([a_1], \ldots, [a_\ell]) \to [\![a]\!]$	$\mathcal{F}_{\mathrm{BC}}$ – Bit composition	[10] (modified, *cf.* §F)	$\ell + 3$
$[a] \to [\![a]\!]$	$\mathcal{F}_{\mathrm{mod}}$ – Modulus conversion	[36] (*cf.* §F)	3

Table 2. Share Conversions

Quotient Transfer Protocol. Consider the reconstruction of shared secret, $[\![a]\!]$ or $\langle\!\langle a \rangle\!\rangle$, over $\mathbb{N}$ as opposed to $\mathbb{Z}_p$ for which $[\![\cdot]\!]$ and $\langle\!\langle \cdot \rangle\!\rangle$ sharings are defined. The resulting value would be of the form $a + qp$, where q is called the quotient. We define the functionality of quotient transfer $\langle\!\langle q \rangle\!\rangle \leftarrow \mathcal{F}_{\mathrm{QT}}(\langle\!\langle a \rangle\!\rangle)$ (resp. $[\![q]\!] \leftarrow \mathcal{F}_{\mathrm{QT}}([\![a]\!])$) that compute q in secret-shared form. These can be efficiently instantiated by using [36] when p is prime, especially a Mersenne prime, as described in Appendix E.

3 Secure Real Number Protocols

In this section, we present the division protocols that will allow us to do fixed-point arithmetic efficiently and securely. The key to achieve efficient fixed-point computations is the ability to perform *truncation* (or equivalently, integer division by 2^k), as this allows multiplication of scaled integer representations of fixed-point values, as introduced in Section 2.

We note that the commonly used ABY3 and Dalskov *et al.*'s division (truncation) and protocol used in most PPML systems requires a heavy offline phase, whereas it provides an efficient online phase. In addition, as we will see later, ABY3 division has an inherent possibility of introducing errors that is much larger than rounding errors, which must be mitigated. In contrast, our new division protocol is efficient in terms of its overall cost i.e. the total cost is comparable to just the online cost of the division protocols, while eliminating the possibility of introducing large errors.

3.1 Current Secure Division Protocols

In this section, we analyze the approach taken to division [50, 55, 63]. For simplicity, we consider *unsigned* integers shared over $\mathbb{Z}_p$ for a general p and a general divisor d, but similar observations holds for the signed integers and specific p, such as 2^{64}.

Let (a_1, a_2, a_3) be the sub-shares of a in the replicated secret sharing scheme, i.e., $a = a_1 + a_2 + a_3$, and $a = \alpha_a d + r_a$ for $0 \le r_a < d$, and let (b_1, b_2, b_3) be the sub-shares of the output of a division protocol. Here, the intention is that $b_1 + b_2 + b_3$ is a value close to $\frac{a}{d}$, such as α_a, or perhaps $\alpha_a \pm 1$.

The current approach proceeds as follows. The parties first prepare a shared correlated randomness $([\![s']\!], [\![s]\!])$, where $s' \leftarrow \mathbb{Z}_p$ and $s := s'/d$. (Note that d is public and known a priori.) The parties then reconstruct a masked secret $(a + s')$, and set $[\![b]\!] = (a + s')/d - [\![s]\!]$.

While this approach appears to work well, the output can in fact be far from the intended $\frac{a}{d}$. To see this, let $s' = sd + r_{s'}$ and $p = \alpha_p d + r_p$. Considering the reconstructed value $[\![a + s']\!]$ over $\mathbb{N}$, we see that the parties obtain $a + s' - qp$, where $q \in \{0, 1\}$. Hence, the computed shared secret corresponds to

$$-s + (a + s' - qp)/d = \alpha_a - q\alpha_p + (r_a + r_{s'} - qr_p)/d. \quad (1)$$

We easily confirm that the second term, $q\alpha_p$, is large, such as 2^{52}, if $q = 1$. Therefore, we must be made the probability of $q = 1$ occurs negligible. The approach taken in [50, 55, 63] to address this, is to adjust the input space (i.e. the parameter ℓ) to be sufficiently small enough to ensure $q = 0$ with overwhelming probability. This, however, leads to a larger reduction of the input space, which can negatively impact the computation being done, due to lower supported accuracy.

Another approach to avoid this error has been proposed by Catrina and de Hoogh [16] and Dalskov *et al.* [24]. The former provides only the statistical security, whereas the latter provides perfect security, as in the previous example. Intuitively, in [24], they compute the quotient q from the MSBs of s and $(a + s')$, assuming the MSB of a is 0. Although their protocol is an elegant solution, the communication complexity is linear to the bit-length of d for preparing s and s', and it is non-trivial to extend their protocol to a general d.

3.2 Protocol for Division by Public Value

Intuition. We first give the intuition behind our protocols. In our protocol for input $[\![a]\!]$, we locally convert $[\![a]\!]$ into $\langle\!\langle a \rangle\!\rangle$ before division. Hence, in the following, we assume the input is $\langle\!\langle a \rangle\!\rangle$ and a public divisor d.

First, let us analyze what happens when we simply divide each share by d. Let $\langle\!\langle a \rangle\!\rangle_1 + \langle\!\langle a \rangle\!\rangle_2 = qp + a$ in $\mathbb{N}$, where $q \in \{0, 1\}$. Here, suppose that $\langle\!\langle a \rangle\!\rangle_j = \alpha_j d + r_j$, $a = \alpha_a d + r_a$, and $p = \alpha_p d + r_p$ for $j = 1, 2$ If each party divide its share $\langle\!\langle a \rangle\!\rangle_j$ by d, the new share is $\alpha_j = \langle\!\langle a \rangle\!\rangle_j / d$. Then, the reconstruction of (α_1, α_2) will be

$$\alpha_1 + \alpha_2 = \alpha_a + q\alpha_p + \frac{r_a + qr_p - (r_1 + r_2)}{d}, \quad (2)$$

Protocol 1 Secure Division by Public Value in $\langle\!\langle\cdot\rangle\!\rangle$

Functionality: $\langle\!\langle c\rangle\!\rangle \leftarrow \mathsf{Div}_{(2,2)}(\langle\!\langle a\rangle\!\rangle, d)$
Input: Share of dividend $\langle\!\langle a\rangle\!\rangle$ and (public) divisor d, where a and d are even numbers.
Output: $\langle\!\langle c\rangle\!\rangle$, where $c \approx \frac{a}{d}$.
1: Let α_p and r_p be $p = \alpha_p d + r_p$, where $0 \le r_p < d$.
2: $\langle\!\langle q\rangle\!\rangle \leftarrow \mathcal{F}_{\mathrm{QT}}(\langle\!\langle a\rangle\!\rangle)$
3: P_1 computes $\langle\!\langle b\rangle\!\rangle_1 \leftarrow (\langle\!\langle a\rangle\!\rangle_1 + d - 1 - r_p)/d$. ▷ "in $\mathbb{N}$" means no reduction $\mod p$
4: P_2 computes $\langle\!\langle b\rangle\!\rangle_2 \leftarrow \langle\!\langle a\rangle\!\rangle_2/d$ in $\mathbb{N}$
5: $\langle\!\langle c\rangle\!\rangle := \langle\!\langle b\rangle\!\rangle - (\alpha_p + 1)\langle\!\langle q\rangle\!\rangle + 1$

which contains extra terms, $q\alpha_p$ and $\frac{(r_a+qr_p-(r_1+r_2))}{d}$.

The insight behind our protocols is that the $q\alpha_p$ term can be eliminated, which we essentially achieve via the quotient transfer protocol [36], that allow us to obtain $\langle\!\langle q\rangle\!\rangle$ efficiently. This protocol suits our setting since it requires a prime p, and prefers a Mersenne prime. The quotient transfer protocol furthermore requires a to be a multiple of 2, but this is easily achieved by locally multiplying a and d with 2, and performing the division using $a' = 2a$ and $d' = 2d$. Note that the output of the division remains unchanged by this.

For the remaining error term $e = \frac{r_a+qr_p-(r_1+r_2)}{d}$, each value r_a, r_p and r_j is less than d, and hence, $-1 \le e \le 2$. In our protocols, we reduce this error to $0 \le e \le 2$ by adding a combination of $\langle\!\langle q\rangle\!\rangle$ and appropriate constants to the output.

Passively Secure Protocols. We propose passively secure division protocols in Protocol 1 and 2. The first protocol works for input $\langle\!\langle a\rangle\!\rangle$, where a is a multiple of 2, and the second for $[\![a]\!]$ by extending the first protocol.

Both Protocol 1 and 2 have probabilistic rounding that outputs a/d, $a/d+1$, or $a/d+2$. In other words, our protocols guarantees that there is only a small difference between $\frac{a}{d}$ and the output of our protocols, as known probabilistic divisions.

Note, our protocols have the technically interesting property that the distribution of the output depends not only on the input value, but also the randomness of the shares of the computing parties. Specifically, if p is a Mersenne prime, the output of Protocol 1 is a/d if

$$((\langle\!\langle a\rangle\!\rangle_1 \le a) \wedge (r_a < r_1)) \vee ((a < \langle\!\langle a\rangle\!\rangle_1) \wedge (r_a - 1 < r_1) \quad (3)$$

and $a/d+1$ otherwise. We define the functionality $\mathcal{F}_{\mathrm{div}}$ that computes the division as according to Eq. (3). The following theorem establishes security of Protocol 1.

Theorem 1. *Protocol 1 securely computes the division functionality $\mathcal{F}_{div}$ in the $\mathcal{F}_{QT}$-hybrid model in the presence of a passive adversary.*

Protocol 2 Secure Division by Public Value in $[\![\cdot]\!]$

Functionality: $[\![c]\!] \leftarrow \mathsf{Div}_{(2,3)}([\![a]\!], d)$
Input: Share of dividend $[\![a]\!]$ and public divisor d, where $0 \le a \le 2^{|p|-1} - 1$
Output: $[\![c]\!]$, where $c \approx \frac{a}{d}$.
1: $\langle\!\langle a\rangle\!\rangle \leftarrow \mathsf{ConvertToAdd}([\![a]\!])$
2: $\langle\!\langle a'\rangle\!\rangle := \langle\!\langle 2a\rangle\!\rangle, d' := 2d$
3: $\langle\!\langle c\rangle\!\rangle \leftarrow \mathsf{Div}_{(2,2)}(\langle\!\langle a'\rangle\!\rangle, d')$
4: $[\![c]\!] \leftarrow \mathsf{ConvertToRep}(\langle\!\langle c\rangle\!\rangle)$
5: Output $[\![c]\!]$

Proof (sketch). Privacy of Protocol 1 is trivially obtained since calling $\mathcal{F}_{\mathrm{QT}}$ is the only step that requires communication, and we consider the $\mathcal{F}_{\mathrm{QT}}$-hybrid model.

Regarding correctness, we sketch a proof when p is a Mersenne prime. The output of Protocol 1 is

$$\alpha_a - q + 1 + \frac{r_a + q(d-1) - r_1 - r_2}{d}. \quad (4)$$

We observe that $r_a + q(d-1) - r_1 - r_2$ must be a multiple of d and $0 \le r_a, r_1, r_2 < d$. Therefore, in the case of $q = 0$, the output is α_a if $r_a < r_1$ and $\alpha_a + 1$ otherwise. In the case of $q = 1$, the output is α_a if $r_a - 1 < r_1$ and $\alpha_a + 1$ otherwise. Furthermore, $q = 0$ can be rewritten as $\langle\!\langle a\rangle\!\rangle_1 \le a$. Combining all the above, we obtain the proof.

Regarding Protocol 2, step 3 and 4 require communication. We have already shown that step 3 is secure and step 4, ConvertToRep, can likewise be seen to be secure. Hence, we conclude that Protocol 2 is secure if Protocol 1 is secure.

Actively Secure Protocols. We show how we construct a division protocol satisfying active security with abort. The difference between our actively and passively secure division protocols is the use of an actively secure quotient transfer protocol. Specifically, employing [36], which takes input $[\![a]\!]$, where a is required to be a multiple of 4, our actively secure division protocol works directly on $[\![a]\!]$ and essentially executes the following steps:

1. $[\![a']\!] := 4[\![a]\!]$ and $d' := 4d$
2. $[\![q]\!] \leftarrow \mathcal{F}_{\mathrm{QT}}([\![a]\!])$
3. Locally divide sub-shares of $[\![a']\!]$ by d' and let them be $[\![b]\!]$
4. $[\![c]\!] := [\![b]\!] - \alpha_p[\![q]\!]$

In addition to the above steps, we add constants to make the difference between $\frac{a}{d}$ and the output small. The actual division protocol is shown in Appendix B.

Comparison. We compare Protocol 2 with the ABY3 [50] (used in FALCON) and Dalskov *et al.*'s [24] truncation protocols (see Appendix C for a detailed analysis of the efficiency of our protocol). The results are shown in Table 3 and 4 for passive and active security, respectively. The ABY3 and Dalskov *et al.*'s proto-

		Communication [bit]	Rounds
ABY3 [50] (and FALCON [64])	offline	$6(2\|p\| - \|d\| - 1)$	$\|p\| - 1$
	online	$3\|p\|$	1
	total	$15\|p\| - 6\|d\| - 6$	$\|p\|$
Dalskov *et al.*[24]	offline	$8\|p\|$	1
	online	$2\|p\|$	1
	total	$10\|p\|$	2
Ours		$5\|p\| + 5$	4

Table 3. Truncation protocols in the passive setting

		Communication [bit]	Rounds
Dalskov *et al.* [24]	offline	$8\|p\|^2 + 6(\kappa\|p\| + 2\|p\| + \kappa)$	2
	online	$6\|p\|$	1
	total	$8\|p\|^2 + 6(\kappa\|p\| + 3\|p\| + \kappa)$	3
Ours		$30\,\|p\| + 21\kappa + 15$	5

Table 4. Truncation protocols in the active setting

cols can be separated into online and offline phases, so we show the cost of each phase as well as the total cost.

In the passive setting, our protocol performs much better than the ABY3 protocol, and has about half the communication cost compared to Dalskov *et al.*'s protocol in the passive setting albeit requiring 4 rounds as opposed to 2. Dalskov *et al.*'s division is close in communication complexity to ours even though preprocessed randomnesses is required, since this can be prepared by a single party in the passive setting.

In the active setting, the gap in communication complexity between ours and Dalskov *et al.*'s protocol gets larger as shown in Table 4 since the above optimization cannot be used. Note that Dalskov *et al.*'s division requires communication proportional to $|p|^2$, whereas ours is $|p|$. In a low latency environment, such as a LAN, this leads to an advantage for our protocol although 2 additional rounds are required.

Another aspect is that our protocol accepts any d as a divisor while the others require $d = 2^k$. This direct division has an advantage in terms of accuracy compared to using the reciprocal of the divisor followed by multiplication. Direct division appears in many computations, such as batch normalization.

4 Elementary Functions for ML

In the following, we will present efficient and high-accuracy protocols for arithmetic functions suitable for advanced machine learning algorithms, such as inversion, square root extraction, and exponential function evaluation. We note that almost all of the related works on PPML systems rely on simplifications that avoid these functionalities, and hence do not implement corresponding protocols.

Our protocols rely on fixed-point arithmetic, and will make use of the division protocols introduced above to implement this. To ease notation in the protocols, we use $[\![a]\!] \cdot [\![b]\!]$ to denote $\mathcal{F}_{\mathrm{mult}}([\![a]\!], [\![b]\!])$, and $[\![a]\!] \times_\ell [\![b]\!]$ to denote $\mathcal{F}_{\mathrm{div}}([\![a]\!] \cdot [\![b]\!], 2^\ell)$.

All protocols will explicitly have as parameters the offset of both input and output values, often denoted α and δ, and in particular will allow these to be different. This can be exploited to obtain more accurate computations when reasonable bounds for the input and output are publicly known. For example, consider the softmax function, often used in neural networks, defined as $\frac{e^{u_i}}{\sum_{j=0}^{k-1} e^{u_j}} = \frac{1}{\sum_{j=0}^{k-1} e^{u_j - u_i}}$ for input $(u_0, \ldots, u_{k-1})$. The output is a value between 0 and 1, and to maintain high accuracy, the offset should be large, e.g. 23 to maintain 23 bits of accuracy below the decimal point. However, the computation of $\sum_{j=0}^{k-1} e^{u_j - u_i}$ is often expected to be a large value in comparison, and a much smaller offset can be used to prevent overflow e.g. -4. Furthermore, we highlight that internally, some of the protocols will switch to using an offset different from α and δ to obtain more accurate numerical computations. By fine-tuning and tailoring the offsets to the computations being done, the most accurate computation with the available ℓ bits for shared values, can be obtained.

In the following, we show protocols for unsigned inputs. These can relatively easily be extended to signed inputs by appropriate input conversion before running the protocol. The details are defered to the full version.

4.1 Inversion

In the following, we introduce a protocol for computing the inverse of a shared fixed-point value. However, before presenting the inversion protocol itself, we introduce a specialized bit-level functionality of *private scaling* that will compute a representation of the input $[\![a]\!]$ which allows us to make full use of the available bit range for shared values. Specifically, the representation of $[\![a]\!]$ is $[\![b]\!] = [\![a]\!] \cdot [\![c]\!]$, where $2^{\ell-1} \leq b \leq 2^\ell - 1$ and c is a power of 2 (recall that shared values are ℓ bit integers). This functionality corresponds to a left-shift of the shared value $[\![a]\!]$ such that the most significant *non-zero* bit becomes the most significant bit, where c represents the required shift to obtain this. We will denote this operation MSNZBFit (MSNZB denoting Most Significant Non-Zero Bit) and the corresponding functionality $\mathcal{F}_{\text{msnzbfit}}$. The protocol presented in Protocol 3 implements this functionality. Recall that $\mathcal{F}_{\text{BC}}$ and $\mathcal{F}_{\text{BDC}}$ are the functionalities of bit-(de)composition.

Theorem 2. *Protocol 3 securely implements $\mathcal{F}_{msnzbfit}$ in the ($\mathcal{F}_{BDC}$, $\mathcal{F}_{BC}$, $\mathcal{F}_{mult}$)-hybrid model in the presence of a passive adversaries.*

Protocol 3 MSNZB Fitting

Functionality: $(\llbracket b\rrbracket, \llbracket c\rrbracket) \leftarrow \mathsf{MSNZBFit}(\llbracket a\rrbracket)$
Input: $\llbracket a\rrbracket$
Output: $\llbracket b\rrbracket, \llbracket c\rrbracket$, where $\llbracket b\rrbracket = \llbracket a\rrbracket \cdot \llbracket c\rrbracket$, $2^{\ell-1} \leq b \leq 2^{\ell} - 1$, and $c = 2^e$ for some $e \in \mathbb{N}$.
Parameter: ℓ

$([a_1], \ldots, [a_\ell]) \leftarrow \mathcal{F}_{\text{BDC}}(\llbracket a\rrbracket)$
$[f_\ell] := [a_\ell]$
for $i = \ell - 1$ to 1 **do**
  $[f_i] := [f_{i+1}] \vee [a_i]$ ▷ $f_i = 1$ for all i corresponding to MSNZB of a or smaller
$[x_\ell] := [a_\ell]$
for $i = \ell - 1$ to 1 **do**
  $[x_i] := [f_i] \oplus [f_{i+1}]$ ▷ $x_i = 1$ only for i corresponding to MSNZB of a
$\llbracket c\rrbracket \leftarrow \mathcal{F}_{\text{BC}}([x_\ell], \ldots, [x_1])$ ▷ Bit-compose $[x_i]$ *in the reverse order* to obtain $c = 2^{\ell-1-\lfloor \log_2 a\rfloor}$
$\llbracket b\rrbracket = \llbracket a\rrbracket \cdot \llbracket c\rrbracket$
Output $\llbracket b\rrbracket$ and $\llbracket c\rrbracket$

Using MSNZBFit as a building block, we now construct our inversion protocol. The protocol is based on the Taylor series for $(1 - x_0)^{-1}$ centered around 0, where $x \in [0; \frac{1}{2})$:

$$\frac{1}{1 - x_1} = \sum_{i=0}^{\infty} x_1^i = 1 + x_1 + x_1^2 + \cdots . \tag{5}$$

Continuing this Taylor series until the n-degree, yields the remainder term $\frac{x^{n+1}}{1-x} \leq \frac{1}{2^n}$, which implies that the approximation has n bits accuracy.

Firstly, we use MSNZBFit to left-shift input $\llbracket a\rrbracket$ to obtain $\llbracket b\rrbracket = \llbracket a\rrbracket \cdot \llbracket c\rrbracket$. Interpreting the resulting value $\llbracket b\rrbracket$ as being a fixed-point value with offset ℓ implies that $b \in [\frac{1}{2}; 1)$. This representation forms the basis of our computation. (Note that since $b \in [\frac{1}{2}; 1)$, we have that $\frac{1}{b} \leq 2$, ensuring that $\frac{1}{b}$ can be represented using $\ell + 1$ bits.)

For the computation of $\frac{1}{b}$, instead of using Eq. (5) directly, which requires r multiplications for $(r + 1)$-th degree terms, we use the following product requiring only $\log r$ multiplications.

$$\prod_{j=0}^{\infty}(1 + x_1^{2^j}) = (1 + x_1)(1 + x_1^2)(1 + x_1^4) \cdots . \tag{6}$$

Letting $b = 1 - x_1$ (which ensures $x_1 \in [0; \frac{1}{2})$), our inversion protocol shown in Protocol 4 iteratively computes Eq. (6) by first setting $x_1 = 1 - b$ and $y_1 = 1 + x_1 = 2 - b$ (Step 2-3), and in each iteration computing $(1 + x_1^i)$ and multiplying this with y_i (Step 4-6). The number of iterations is specified via the parameter I. Finally, to obtain (an approximation to) $\llbracket \frac{1}{a}\rrbracket$, we essentially only need to scale the computed $\llbracket y_I\rrbracket = \llbracket \frac{1}{b}\rrbracket$ with $\llbracket c\rrbracket$ (as $\frac{1}{b} \cdot c = \frac{1}{ac} \cdot c = \frac{1}{a}$). Note, however, that the output has to be scaled taking into account the input and output offsets, as well as the offset used in the internal computation. To see that the correct scaling factor is $2^{\alpha+\delta-2\ell}$, note that for input $a = a'_{\langle\alpha\rangle}$ and output $b = a \cdot c = b'_{\langle\ell\rangle}$ of MSNZBFit, we have

$$y_I = \frac{2^\ell}{b'} = \frac{2^\ell}{a' \cdot 2^\alpha \cdot c \cdot 2^{-\ell}} = \frac{1}{a' \cdot c} \cdot 2^{2\ell-\alpha}$$

and that the output should be scaled with 2^δ.

We define the corresponding functionality $\mathcal{F}_{\text{Inv}}$ in which on input of shares computes the above Taylor series expansion and output shares of that output.

Protocol 4 Inversion

Functionality: $\llbracket d\rrbracket \leftarrow \mathsf{Inv}(\llbracket a\rrbracket)$
Input: $\llbracket a\rrbracket$, where $a \in \widehat{\mathbb{Q}}_{\langle\ell,\alpha\rangle}$
Output: $\llbracket d\rrbracket$, where $d \approx \left(\frac{1}{a}\right)_{\langle\delta\rangle}$
Parameter: $(\ell, I, \alpha, \delta)$, where I is the number of iterations (say, $I = \lceil \log \ell\rceil$) used in the computation

1: $(\llbracket b\rrbracket, \llbracket c\rrbracket) \leftarrow \mathsf{MSNZBFit}(\llbracket a\rrbracket)$ ▷ $b = b' \cdot 2^\ell$ where $b' \in [\frac{1}{2}, 1)$
2: $\llbracket x_1\rrbracket := 1_{\langle\ell\rangle} - \llbracket b\rrbracket$
3: $\llbracket y_1\rrbracket := 2_{\langle\ell\rangle} - \llbracket b\rrbracket$
4: **for** $i = 2$ to I **do**
5:   $\llbracket x_i\rrbracket := \llbracket x_{i-1}\rrbracket \times_\ell \llbracket x_{i-1}\rrbracket$
6:   $\llbracket y_i\rrbracket := \llbracket y_{i-1}\rrbracket \times_\ell (1_{\langle\ell\rangle} + \llbracket x_{i-1}\rrbracket)$
7: Output $\llbracket y_I\rrbracket \cdot \llbracket c\rrbracket \cdot 2^{\alpha+\delta-2\ell}$

Theorem 3. *The protocol* Inv *securely computes inversion functionality $\mathcal{F}_{Inv}$ in the ($\mathcal{F}_{msnzbfit}, \mathcal{F}_{div}, \mathcal{F}_{mult}$)-hybrid model in the presence of a passive adversary.*

Division with private division is a relatively simple extension of the inversion protocol. Due to space limitations, we defer the details to the full version.

4.2 Square Root and Inverse Square Root

Computing the inverse of the square root of an input value, is a useful operation for many computations, *e.g.,* normalization of a vector, and is likewise used in Adam. Hence, having an efficient protocol for directly computing this, is beneficial.

Our protocol for computing the inverse of a square root is shown in Protocol 6, and is based on Newton's method for the function $f(y) = \frac{1}{y^2} - x$ for input value

x (note that $f(y') = 0$ implies $y' = \frac{1}{\sqrt{x}}$). This involves iteratively computing approximations

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)} = \frac{y_n(3 - x \cdot y^2)}{2}$$

for an appropriate initial guess y_1 (Step 5-6 performs this iteration). To ensure fast convergence for a large range of input values, we represent the input $x = b \cdot 2^e$ for $b \in [\frac{1}{2}; 1)$, which implies

$$\frac{1}{\sqrt{x}} = \begin{cases} \left(\frac{1}{\sqrt{b}}\right) \cdot 2^{-e/2} & \text{if } e \text{ is even} \\ \left(\frac{\sqrt{2}}{\sqrt{b}}\right) \cdot 2^{-(e+1)/2} & \text{if } e \text{ is odd} \end{cases}.$$

Hence, we only need to compute $\frac{1}{\sqrt{b}}$ for $b \in [\frac{1}{2}; 1)$, in which case using 1 as the initial guess provides fast convergence. However, the parties should not learn which of the above two cases the input falls into. We introduce a sub-protocol, MSNZBFitExt shown in Protocol 5, that computes values r and c', where $(r, c') = (0, 2^{e/2})$ if x falls into the first case, and $(r, c') = (1, 2^{(e+1)/2})$ otherwise. Note that like MSNZBFit, the extended MSNZBFitExt right-shifts the input a to obtain an ℓ-bit value $b = a \cdot c$, that when interpreted as an element of $\widehat{\mathbb{Q}}_{\langle \ell, \ell \rangle}$, represents $b \in [\frac{1}{2}; 1)$, and that $c' = \sqrt{c}$. Having r and c' allows us to compute $\frac{1}{\sqrt{x}}$ as $\frac{1}{\sqrt{b}} \cdot (1 + r \cdot (\sqrt{2} - 1)) \cdot c'$. Finally, note that the output has to be scaled, taking into account the input and output offsets, as well as the offset used for the internal computation. To see that the correct scaling factor is $2^{\delta - \frac{3}{2}\ell + \frac{\alpha}{2}}$, note that for input $a = a'_{\langle \alpha \rangle}$ and output $b = a \cdot (c')^2 = b' * 2^\ell$ of MSNZBFitExt, we have

$$y_I \approx \frac{2^\ell}{\sqrt{b'}} = \frac{2^\ell}{\sqrt{a' \cdot 2^\alpha \cdot (c')^2 \cdot 2^{-\ell}}} = \frac{1}{\sqrt{a'}} \cdot \frac{1}{c'} \cdot 2^{\ell/2 - \alpha/2}$$

and that the output should be scaled with 2^δ. We define the functionality $\mathcal{F}_{\text{InvSqrt}}$ that computes $\frac{1}{\sqrt{x}}$ as done in the above using the Newton's method.

Theorem 4. *The protocol* InvSqrt *securely computes the inverse of the square root functionality $\mathcal{F}_{InvSqrt}$ in the $(\mathcal{F}_{msnzbfit}, \mathcal{F}_{mod}, \mathcal{F}_{div}, \mathcal{F}_{mult})$-hybrid model in the presence of a passive adversary.*

Given InvSqrt for computing $\frac{1}{\sqrt{x}}$, and noting that $\sqrt{x} = \frac{x}{\sqrt{x}}$, we can easily construct a protocol for computing $\sqrt{x}$, simply by running InvSqrt and multiplying the result with x. The resulting protocol, Sqrt, is shown in Protocol 7. Let $\mathcal{F}_{\text{sqrt}}$ be the functionality that on input $[\![a]\!]$ outputs $[\![\sqrt{a}]\!]$ in which $[\![\frac{1}{\sqrt{a}}]\!]$ is obtained by $\mathcal{F}_{\text{InvSqrt}}$.

Theorem 5. *The protocol* Sqrt *securely computes the square root functionality $\mathcal{F}_{sqrt}$ in the $(\mathcal{F}_{InvSqrt}, \mathcal{F}_{div}, \mathcal{F}_{mult})$-hybrid model in the presence of a passive adversary.*

Protocol 5 MSNZBFitExt Sub-protocol for InvSqrt

Functionality: $([\![b]\!], [\![c']\!], [\![r]\!]) \leftarrow \mathsf{MSNZBFitExt}([\![a]\!])$
Input: $[\![a]\!]$
Output: $([\![b]\!], [\![c']\!], [\![r]\!])$, where $b = b'_{\langle \ell \rangle} \in \widehat{\mathbb{Q}}_{\langle \ell, \ell \rangle}$ and $b' \in [\frac{1}{2}; 1)$, $x = b' \cdot 2^e$, and $r = 0$ if e is even, and $r = 1$ otherwise.
Parameter: ℓ

1: Parties jointly execute steps 1-9 of protocol MSNZBFit.
2: $\ell' := \left\lfloor \frac{\ell}{2} \right\rfloor$
3: $[x'_i] := [x_{\ell+1-i}]$ for $1 \le i \le \ell$
4: **for** $i = 1$ to $\ell' - 1$ **do**
5: $\quad [y_i] := [x'_{2i}] \oplus [x'_{2i+1}]$
6: **if** ℓ is an even number **then**
7: $\quad [y_{\ell'}] := [x'_{2\ell'}] \oplus [x'_{2\ell'+1}]$
8: **else**
9: $\quad [y_{\ell'}] := [x'_{2\ell'}]$
10: $[r] := [x'_2] \oplus [x'_4] \oplus \cdots \oplus [x'_{2\lfloor \frac{\ell'}{2} \rfloor}]$
11: $[\![r]\!] \leftarrow \mathcal{F}_{\text{mod}}([r])$
12: $[\![c']\!] \leftarrow \mathcal{F}_{\text{BC}}([y_1], \ldots, [y_{\ell'}])$
13: Output $([\![b]\!], [\![c']\!], [\![r]\!])$

Protocol 6 Inversion of Square Root

Functionality: $[\![z]\!] \leftarrow \mathsf{InvSqrt}([\![a]\!])$
Input: $[\![a]\!]$, where $a = a'_{\langle \alpha \rangle} \in \widehat{\mathbb{Q}}_{\langle \ell, \alpha \rangle}$
Output: $[\![z]\!]$, where $z \approx \left(\frac{1}{\sqrt{a'}}\right)_{\langle \delta \rangle}$
Parameter: $(\ell, I, \alpha, \delta)$, where I is the number of iteration (say, $I = \lceil \log \ell \rceil$) in the computation.

1: $([\![b]\!], [\![c']\!], [\![r]\!]) \leftarrow \mathsf{MSNZBFitExt}([\![a]\!])$
2: $[\![x_1]\!] := 3_{\langle \ell \rangle} - [\![b]\!]$
3: $[\![y_1]\!] := [\![x_1]\!]/2$
4: **for** $i = 2$ to I **do**
5: $\quad [\![x_i]\!] := 3_{\langle \ell \rangle} - ([\![y_{i-1}]\!] \times_\ell [\![y_{i-1}]\!]) \times_\ell [\![b]\!]$
6: $\quad [\![y_i]\!] := [\![x_{i-1}]\!] \times_{\ell+1} [\![y_{i-1}]\!]$ ▷ Implicit scaling by $\frac{1}{2}$
7: Output $[\![y_I]\!] \cdot (1 + [\![r]\!] \cdot (\sqrt{2} - 1)) \cdot [\![c']\!] \cdot 2^{\delta - \frac{3}{2}\ell + \frac{\alpha}{2}}$

4.3 Exponential Function

To obtain a fast and highly accurate protocol for the exponential function, we adopt what we call *hybrid table-lookup/series-expansion* technique. It utilizes the table lookup approach for the large-value part of the input, in combination with the Taylor series evaluation for its small-value counterpart. We first recall that the Taylor series of the exponential function:

$$\exp x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \cdots .$$

which converges fast for small values of x. To minimize the value for which we use the above Taylor series, we separate the input a into three parts:

1. μ: a lower bound for the input

Protocol 7 Square Root

Functionality: $[\![z]\!] \leftarrow \mathsf{Sqrt}([\![a]\!])$
Input: $[\![a]\!]$, where $a = a'_{\langle\alpha\rangle} \in \widehat{\mathbb{Q}}_{\langle\ell,\alpha\rangle}$
Output: $[\![z]\!]$, where $z \approx (\sqrt{a'})_{\langle\delta\rangle}$
Parameter: $(\ell, I, \alpha, \delta)$ where I is the number of iterations used in the computation.

$[\![z']\!] \leftarrow \mathsf{InvSqrt}_{(\ell,I,\alpha,\delta)}([\![a]\!])$ ▷ $z' = \left(\frac{1}{\sqrt{a'}}\right)_{\langle\delta\rangle}$
Output $[\![a]\!] \cdot [\![z]\!] \cdot 2^{-\alpha}$

2. $b_\ell, \dots, b_{\ell-t}$: bit representation of the t most significant bits of $b := a - \mu$
3. b_σ: integer representation of $(a-\mu) - \sum_{\ell-t \le i \le \ell} 2^{i-\alpha} b_i$

which means that we can compute $\exp(a)$ as

$$\exp a = \left(\prod_{i=\ell-t}^{\ell} \exp(b_i \cdot 2^{i-\alpha})\right) \cdot \exp(b_\sigma) \cdot \exp(\mu). \qquad (7)$$

Here, α is the input offset and t is a parameter of our protocol that determines which part of the input we will evaluate using table lookups, and which part we will evaluate using a Taylor series. In this product, we evaluate $\exp(\mu)$ locally (as μ is public), $\prod_{i=\ell-t}^{\ell} \exp(b_i \cdot 2^i)$ using table lookups, and $\exp(b_\sigma)$ using the Taylor series. The taylor series rapidly converge since b_σ is made small due to the subtraction of μ and the value of the t most significant bits of a.

More specifically, for the table lookup computation, note that the binary value b_i determines whether the factor $\exp 2^i$ will included. Hence, by combining bit decomposition, that allows parties to compute $[b_i]$ from $[\![b]\!]$, with the $\mathsf{CondAssign}$ protocol using $[b_i]$ as the condition, and the values 1 and $\exp 2^i$, which are public and can be precomputed, we obtain an efficient mechanism for computing $\prod_{i=\ell-t}^{\ell} \exp(b_i \cdot 2^i)$. However, to maintain high accuracy, the parties will not use $\exp 2^i$ directly, but precompute a mantissa f_i and exponent 2^{ϵ_i} such that $f_i \cdot 2^{\epsilon_i} = \exp 2^{i-\alpha}$. This allows the parties to compute the product of f_i values separately from the product of 2^{ϵ_i} values, and only combine these in the final step constructing the output, thereby avoiding many of the rounding errors that potentially occur in large products of increasingly larger values.

Lastly, the result is computed as $\prod_i f_i \cdot \prod_i 2^{\epsilon_i} \cdot \exp a_\sigma$. Here, the input and output offsets have to be taken into account, and the output is adjusted appropriately. We define the functionality $\mathcal{F}_{\exp}$ such that on input $[\![a]\!]$, e^a is computed as done in the above and output $[\![e^a]\!]$.

Protocol 8 Exponential Function

Functionality: $[\![z]\!] \leftarrow \mathsf{Exponent}([\![a]\!])$
Input: $[\![a]\!]$, where $a = a'_{\langle\alpha\rangle} \in \widehat{\mathbb{Q}}_{\langle\ell,\alpha\rangle}$
Output: $[\![z]\!]$, where $z \approx (\exp a')_{\langle\delta\rangle}$
Parameter: $(\ell, I, \alpha, \beta, \delta, \mu, t)$ where I is the number of iterations used in the computation, β is the offset of the lookup table values, t indicates the lookup table vs Taylor series threshold, and μ is a lower bound for the input.

$[\![b]\!] := [\![a]\!] - \mu_{\langle\alpha\rangle}$
$[b_\ell], \dots, [b_{\ell-t}] \leftarrow \mathcal{F}_{\mathrm{BDC}}([\![b]\!])$ ▷ We use only $t+1$ MSBs while $\mathcal{F}_{\mathrm{BDC}}$ outputs ℓ bits.
$[\![b_i]\!] \leftarrow \mathcal{F}_{\mathrm{mod}}([b_i])$ for $i = \ell, \dots \ell - t$.
$[\![b_\sigma]\!] := [\![b]\!] - \sum_{\ell-t \le i \le \ell} 2^i [\![b_i]\!]$ ▷ Value of t LSBs of $[\![b]\!]$
Parties define f_i, ϵ_i such that $\exp 2^{i-\alpha} = f_i \cdot 2^{\epsilon_i}$ ▷ Precomputed lookup table values
Using $[\![f'_i]\!] \leftarrow \mathcal{F}_{\mathrm{CondAssign}}(1_{\langle\beta\rangle}, (f_i)_{\langle\beta\rangle}, [\![b_i]\!])$, the parties obtain $[\![f'_i]\!] = \begin{cases} (f_i)_{\langle\beta\rangle} & \text{if } b_i = 1 \\ 1_{\langle\beta\rangle} & \text{otherwise} \end{cases}$ for $i = \ell, \dots, \ell - t$
Using $[\![\epsilon'_i]\!] \leftarrow \mathcal{F}_{\mathrm{CondAssign}}(1, 2^{\epsilon_i}, [\![b_i]\!])$, the parties obtain $[\![\epsilon'_i]\!] = \begin{cases} 2^{\epsilon_i} & \text{if } b_i = 1 \\ 1 & \text{otherwise} \end{cases}$ for $i = \ell, \dots, \ell - t$
$[\![f']\!] := [\![f'_\ell]\!] \times_\beta \dots \times_\beta [\![f'_{\ell-t}]\!]$
$[\![\epsilon']\!] := [\![\epsilon'_\ell]\!] \cdot \dots \cdot [\![\epsilon'_{\ell-t}]\!]$
$b_{\sigma,0} := 1$
for $i = 1$ to $I - 1$ **do**
    $[\![b_{\sigma,i}]\!] \leftarrow [\![b_{\sigma,i-1}]\!] \times_\alpha [\![b_\sigma]\!]$
$[\![b'_\sigma]\!] := \sum_{0 \le i < I} \frac{[\![b_{\sigma,i}]\!]}{i!}$ ▷ Division using Div
Output $[\![f']\!] \cdot [\![\epsilon']\!] \cdot [\![b'_\sigma]\!] \cdot \exp\mu \cdot 2^{\delta-\beta-\alpha}$

Theorem 6. *The protocol* $\mathsf{Exponent}$ *securely computes the exponential function functionality $\mathcal{F}_{exp}$ in the $(\mathcal{F}_{CondAssign}, \mathcal{F}_{mod}, \mathcal{F}_{BDC}, \mathcal{F}_{div}, \mathcal{F}_{mult})$-hybrid model in the presence of a passive adversary.*

Satisfying Active Security with Abort. There are known compilers that convert a passively secure protocol to an actively secure one (with abort). The compiler [21] and its extension [35] can be applied to Binary/arithmetic circuit computation, and each step of our proposed protocols except $\mathcal{F}_{\mathrm{BDC}}$, $\mathcal{F}_{\mathrm{mod}}$, and $\mathcal{F}_{\mathrm{div}}$ is circuit computation over modulus 2 and p. Therefore, we can obtain actively secure versions of our protocols computing elementary functions by applying that compiler on modulus 2 and p in parallel.

4.4 Comparison to Related Works

We compare our invesion/division, Inverse Square-root, and Exponentiation protocols with the division from Catrina-Saxena [17], the inverse square root protocol of Lu *et al.* [47], and the exponentiation from Aly-Smart[9], respectively, in Appendix H. In addition, we list the detailed round complexity and communication cost of all our protocols in Table 14 in Appendix D.

5 Secure Deep Neural Networks

5.1 Neural Networks

In this paper, we deal with feedforward and convolutional neural networks. A network with two or more hidden layers is called a deep neural network, and learning in such a network is called deep learning.

A layer contains neurons and the strength of the coupling between neurons and the output of the previous layer are described by parameters w_i. Learning is a process that iteratively updates the parameters to obtain the appropriate output.

Layers. There are several types of layers. A fully connected layer is computing the inner product of the input with the parameters. A convolutional layer divides the input into certain units and processes these in an overlapping manner. A max-pooling layer computes the maximum value of (parts of) the input, and discards the remaining values. A batch normalization layer performs normalization and an affine transformation of the input. To normalize an input $\overrightarrow{x} = (x_1, \ldots, x_n)$, we compute

$$x_i \leftarrow \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \tag{8}$$

where μ and σ are mean and variance of $\overrightarrow{x}$, and ϵ is a small constant.

Activation Function. In a neural network, the activation functions of the hidden layer and the output layer are selected according to purpose.

ReLU Function. A popular activation function for hidden layers is the ReLU function defined as $\mathbf{ReLU}(u) = \max(0, u)$. Note that this function is not differentiable (which is typically required for training), and the function $\mathbf{ReLU}'(u)$, outputting 0 if $u \leq 0$ and 1 otherwise, is often used as a substitute for the differentiated ReLU function.

Softmax Function Classifications for image identification commonly use the softmax function $\mathbf{softmax}(u_i)$ at the output layer. The softmax function for classification into k classes is as follows:

$$\mathbf{softmax}(u_i) = \frac{e^{u_i}}{\sum_{j=0}^{k-1} e^{u_j}} = \frac{1}{\sum_{j=0}^{k-1} e^{u_j - u_i}}. \tag{9}$$

Optimization. A basic method of parameter update is stochastic gradient descent (SGD). This method is relatively easy to implement but has drawbacks such as slow convergence. and the potential for becoming stuck at local maxima. To address these drawbacks, optimized algorithm have been introduced. [59] analyzed eight representative algorithms, and Adam [37] was found to be providing particularly good performance. In fact, Adam is now used in several machine learning framework [3, 6].

The process of Adam includes computing the iterative value

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{\hat{V}_{t+1}} + \epsilon} \circ \hat{M}_{t+1}, \tag{10}$$

where t indicates the iteration number of training, $W, \hat{V}$, and $\hat{M}$ are matrices, $\circ$ denotes the element-wise multiplication, and η and ϵ are parameters.

5.2 Secure Protocols for DNNs

The softmax function, Adam, and batch-normalization are quite common and popular algorithms for deep neural network due to their superior performance compared to alternatives. However, efficient secure protocols have been elusive due to intractability of computing the elementary functions they depend on. The softmax function requires exponentiation and inversion, as shown in Eq. (9), and Adam and batch-normalization require the inverse of square roots, as shown in Eq. (10) and (8). Thus, the softmax function has often been approximated by a different function [51], which always reduces accuracy, sometimes significantly [34], and only SGD, an elemental optimization method is used. Although FALCON realized secure batch-normalization [64], it is not fully secure because it leaks the magnitude of $b = \sigma^2 + \epsilon$, i.e., α such that $2^\alpha \leq b < 2^{\alpha+1}$, to compute $\frac{1}{\sqrt{b}}$.

In contrast to related works, our efficient protocols presented in Section 4 allow us to implement secure deep neural network using softmax, Adam, and batch-normalization, as opposed to resorting to approximations and less efficient learning algorithms. Note that to ensure stability of softmax, we implement *clipping* (see Appendix J).

Besides softmax, Adam, and batch-normalization, we also require more building blocks to fully implement training. For matrix multiplication, we apply [21] to compute the inner products with the same communication cost of a single multiplication. The ReLU′ function extracts the sign of the input, and the ReLU function is obtained by simply multiplying the input with the output of ReLU′. We repeatedly apply the comparison protocol [36] to implement secure max-pooling, which computes the maximum value and argmax (which is required in backpropagation). We defer the details to the full version. See the communication costs in Table 14.

6 Experimental Evaluation

Environment. We implemented our protocols using $p = 2^{61} - 1$, and instantiated $\mathcal{F}_{\mathrm{BC}}$, $\mathcal{F}_{\mathrm{mod}}$, and $\mathcal{F}_{\mathrm{QT}}$ with the bit-composition, modulus-conversion, and quotient transfer protocols from [36], respectively. We set the statistical security parameter for active-with-abort security to $\kappa = 8$.[6] The parameters used for the elementary functions as well as Adam are listed in Appendix G. All of the following experiments were run on a machines with dual Intel Xeon CPUs (3.50GHz 8 core/ 16 thread) and 756 GB of memory, connected via a Intel X710/X557-AT 10G Ring network. We artificially limited the network speed to 320Mbps and latency to 40ms when simulating a WAN setting.

Accuracy. We measured the accuracy of our division protocol and elementary functions. Note that accuracy might differ significantly from the used bit precision (*i.e.,* the bit-length of the input, intermediate, and output values) and in lieu of an implementation specific theoretical analysis, must be measured.

Firstly, we measured the L1-norm error of our division protocol using inputs ranging from 1 to 10,000, obtaining an average-case L1-norm error of $\frac{0.335}{2^t}$ for input with offset t. This shows the obtained output is close to real-valued division. Secondly, we measured the logarithm of L1-norm error for our elementary functions, which corresponds to the number of most significant bits equal to the real-valued output. All our protocols except exponentiation achieved $26 \sim 29$-bit accuracy, while exponentiation achieved $23 \sim 25$-bit accuracy. Hence, our implementations achieve at least 23-bit accuracy. The full details of this is deferred to the full version.

This result highlights that our implementations achieve accuracy corresponding to single-precision real number operations, which is the commonly used standard for ML algorithms in the clear *e.g.,* TensorFlow [6] and PyTorch [5] are all based on single-precision real number computations, and especially recommended for softmax in the output layer to avoid numeric issues [4]. That is, our protocols allows ML algorithms to be securely implemented without degrading training accuracy compared to plaintext performance.

6 While this parameter is relatively small compared to a somewhat more standard parameter like $\kappa = 40$ [11], an active adversary will have only a single chance to cheat for the implemented techniques [21, 35] and an honest party can detect this with probability over 99%.

	Security	Setting	Time [s]	Accuracy [%]
TensorFlow	-	-	1.19	95.54
Ours	Passive	LAN	117	95.64
Ours	Active	LAN	570	95.37
Ours	Passive	WAN	4,537	95.58
Ours	Active	WAN	11,516	95.61

Table 5. Measured running time for training 3DNN on MNIST : TensorFlow (which is in plaintext) vs. Our secure training.

6.1 Secure Training of DNNs

Network Architectures. We consider three networks in our experiments: (1) 3DNN, a simple 3-layer fully-connected network introduced in SecureML [51] and used as a benchmark for privacy preserving ML, (2) AlexNet, the famous winner of the 2012 ImageNet ILSVRC-2012 competition [41] and a network with more than 60 million parameters, and (3) VGG16, the runner-up of the ILSVRC-2014 competition [61] and a network with more than 138 million parameters. While the first network is typically used as a performance benchmark for privacy preserving ML, measurements with the latter two networks provide insight into the performance when using larger more realistic networks.

Datasets. We use two datasets: (1) MNIST [44], a collection of 28 x 28 pixel images of hand-written digits typically used for benchmarks, and (2) CIFAR-10 [40], a collection of colored 32 x 32 pixel images picturing dogs, cats, etc. We used MNIST in combination with 3DNN for benchmarking, and CIFAR-10 in combination with the larger networks AlexNet and VGG16.

Comparison to Plaintext. As a baseline, we measured the performance of TensorFlow when training 3DNN on MNIST using ADAM in the clear, and compared the running time and training accuracy with our framework. The result is shown in Table 5.

As shown in Table 5, the obtained training accuracy is essentially identical, which is expected as our framework allows training to be implemented "as is" without any simplifications (the minor difference in training accuracy is due to different initial randomness). For completeness, we additionally compare convergence of TensorFlow and our training in Appendix I. The running time of training with our framework is still two orders of magnitude slower in the LAN setting and four orders

	AlexNet					VGG16				
#epochs	1	2	3	4	5	1	2	3	4	5
TensorFlow	56.49	62.46	64.70	66.43	66.09	34.40	50.82	59.00	66.99	71.67
Ours	55.51	61.25	64.83	67.23	65.71	26.60	50.68	59.59	64.77	72.29

Table 6. Measured accuracy of AlexNet and VGG16 on CIFAR-10: TensorFlow (plaintext) vs. Our secure training.

of magnitude slower in the WAN setting, compared to training in the clear.

Additionally, to verify the performance of our framework on larger networks, we measured the obtained accuracy when training AlexNet and VGG16 on CIFAR-10 after 1 to 5 epochs, and compared this to training in the clear with TensorFlow. The result is shown in Table 6. As can be seen from the table, our framework achieves performance consistent with training in the clear. We highlight that when considering only one epoch, the accuracy of large networks such as VGG16, which has more than 100 million parameters, becomes highly dependent on the initial randomized state. The difference between TensorFlow and our framework in the corresponding table entry is consistent with this, and we note that after just 2 epochs, the accuracy converges towards similar values. For a visual representation of the variance of accuracy for training in the clear for AlexNet and VGG16, see Figure 1a and Figure 1b.

Comparison of ADAM and SGD. The main contribution of our framework is to enable advanced ML algorithms, such as ADAM, to be implemented securely and efficiently as part of a PPML system, without resorting to simplifications of the underlying ML algorithms or compromising on training accuracy. To illustrate the advantage of this approach, we measure the running time required to achieve roughly 95% accuracy of the 3DNN network on the MNIST dataset, when using our framework to implement both ADAM and SGD, respectively. Note that in comparison to ADAM, SGD is a much simpler optimization algorithm not requiring most of the functionalities implemented by our protocols in Section 4 (exponentiation being an exception, assuming a proper softmax function is implemented). For this reason, almost all of the related works in the PPML literature is based on SGD, as developing efficient MPC algorithms for the required functionalities is a much simpler task, and some works (*e.g.,* [50, 64]) even further simplifies SGD (*e.g.,* replaces softmax with a ReLu-based alternative in a bid to further increase efficiency. Note that our measurements are for the standard SGD with full 23-bit accuracy. The measured running times and obtained accuracy is shown in Table 7.

The results show an advantage of a factor of approximately $7 \sim 10$ in terms of running time when using ADAM, while ADAM still obtains a higher training accuracy. This clearly illustrates that the combination of our framework that allows efficient evaluation of the more complex functionalities required by ADAM, and the superior training obtained by using ADAM, has the potential to significantly outperform the approach of obtaining an efficient PPML system by simplifying the ML component and using less complex but very efficient MPC protocols. For more complex networks than 3DNN, performance measurements are available below.

	Security	Setting	Epochs	Time [s]	Accuracy [%]
SGD	Passive	LAN	15	855	94.04
ADAM	Passive	LAN	1	117	95.64
SGD	Active	LAN	15	4,725	92.28
ADAM	Active	LAN	1	570	95.37
SGD	Passive	WAN	15	39,310	93.82
ADAM	Passive	WAN	1	4,537	95.58
SGD	Active	WAN	15	86,953	92.09
ADAM	Active	WAN	1	11,516	95.61

Table 7. Measured running time for training 3DNN on MNIST (ADAM vs SGD) using our protocols.

Comparison to Related Works. For experimental comparison with related work, we will focus on the state-of-the-art three-party protocols: FALCON [64] and the three-party Fantastic Four (3FF) [25]. We additionally include the MP-SPDZ implementation (SPDZ) from [25] of the SGD variant Momentum. Lastly, we will discuss our results in relation to CRYPTGPU [62], which is implemented on and optimized for GPUs.

Concretely, in our experiments, we ran the FALCON, 3FF and SPDZ code in the same environment and measured the execution time. While FALCON is implemented for 3DNN and the larger networks AlexNet and VGG16, [25] does not consider the latter two, and we hence only compare against 3FF and SPDZ on 3DNN. Note that 3FF provides active security, whereas SPDZ is only passively secure, and we hence only compare with these in the relevant settings. We note that the FALCON code implements *the online phase only*, and hence, the measurements do not include the corresponding offline phase, which would make a significant contribution to the total running time. Lastly, the FALCON code does not update the parameters of the model, which makes the accuracy of the obtained model unclear. In contrast, the measurements for 3FF, SPDZ and our protocols are for the *total* running time and a fully trained model.

Results for 3DNN. We measured the running time and accuracy for training 3DNN on the MNIST dataset for passive and active security, in the LAN and WAN settings. The results for our protocols, FALCON, SPDZ,

	Security/NW	Methods	Epochs	Time [s]	Accuracy [%]
FALCON [64]		SGD	15	780	-
SPDZ [25]	Passive/LAN	Mnt	2	177	95.17
Ours		Adam	1	117	95.64
FALCON [64]		SGD	15	2,355	-
3FF [25]	Active/LAN	Mnt	2	1,879	95.43
Ours		Adam	1	570	95.37
FALCON [64]		SGD	15	16,110	-
SPDZ [25]	Passive/WAN	Mnt	2	8,479	95.17
Ours		Adam	1	4,537	95.58
FALCON [64]		SGD	15	37,185	-
3FF [25]	Active/WAN	Mnt	2	48,328	94.76
Ours		Adam	1	11,516	95.61

Table 8. Comparison of training time of 3DNN over the MNIST dataset for FALCON, 3FF, SPDZ and Ours.

	Security	MNIST		
		Per epoch [GB]	Epochs	Total [GB]
FALCON [64]	Passive	3.28	15	49.15
SPDZ [25]	Passive	26.59	2	53.18
Ours	Passive	92.17	1	92.18
FALCON [64]	Active	18.02	15	270.34
3FF [25]	Active	422.20	2	844.40
Ours	Active	387.69	1	387.69

Table 9. Communication cost for training of 3DNN on MNIST. (Note that, for [64], this is only the online cost.)

and 3FF can be seen in Table 8. The corresponding communication cost is shown in Table 9 (the cost for FALCON is obtained from [64], whereas the cost for SPDZ, 3FF and ours is measured in our environment). The number of epochs for FALCON, 3FF and SPDZ were chosen to obtain similar accuracy of the trained network[7]. Compared to FALCON, ours is between 3.2 to 6.7 times faster, depending on the setting. We again highlight that these results are achieved despite the advantages provided to FALCON in this comparison (e.g. measuring online time only, etc.). Compared to 3FF and SPDZ, which both are based on the more advanced SGD variant Momentum, ours is 1.5 to 4.2 times faster to reach the same accuracy of the trained network. We note that FALCON has the lowest total communication overhead; the lower overall running time of our protocols is due to more efficient computation and interaction.

Results for AlexNet and VGG16. In FALCON [64], the total running time for training on AlexNet and VGG16 was estimated through extrapolation since these networks require a significant amount of computation for training, even in the clear. We follow this method

	Security	Setting	AlexNet [s]	VGG16 [s]
FALCON [64]	Passive	LAN	10,892	523,127
Ours	Passive	LAN	3,139	43,150
FALCON [64]	Active	LAN	41,537	2,051,751
Ours	Active	LAN	15,021	161,481
FALCON [64]	Passive	WAN	23,489	575,699
Ours	Passive	WAN	49,833	347,928
FALCON [64]	Active	WAN	75,838	2,240,515
Ours	Active	WAN	159,781	1,293,226

Table 10. Measured running time per epoch for training AlexNet and VGG16 on CIFAR-10.

to estimate the running time of ours and FALCON (re-evaluated in our environment) in the same way.

In Table 10, we show the measured running time to complete a single epoch for AlexNet and VGG16 using the CIFAR-10 dataset, both for passive and active security, as well as in the LAN and WAN settings. Table 12 includes the corresponding communication overhead, where the overhead of our protocol is measured, but the overhead for FALCON is based on the figures from [64]. Note, however, that the time to complete a single epoch is not indicative of the overall performance difference between FALCON and our framework, as the underlying optimization methods are different, and require a different number of epochs to train a network achieving a certain prediction accuracy.

To determine the number of epochs needed for Adam (implemented in our framework) and SGD (implemented in FALCON), we ran Adam and SGD for AlexNet and VGG16 on CIFAR-10 in the clear, and measured the achieved accuracy. The results are shown in Fig. 1. For AlexNet, we see that the accuracy converges towards 60% $\sim$ 73%, with Adam achieving a maximum of 72.10% and SGD a maximum of 61.22% in our test. For both AlexNet and VGG16, we see that Adam significantly outperforms SGD, and after relatively few epochs, achieves an accuracy not obtained by SGD, even after 60 epochs. We note that achieving an accuracy exceeding 60% on AlexNet requires 2 and 19 epochs for Adam and SGD, respectively, whereas exceeding an accuracy of 70% on VGG16 requires 5 and 20 epochs, respectively.

Based on the observations above, we estimate the running time of achieving an accuracy of 60% for AlexNet and 70% for VGG16, for active/passive security in the LAN/WAN setting. The result is shown in Table 11. We see that in the LAN setting, the total running time of our framework outperforms the online phase of FALCON with a factor of 26 $\sim$ 33 for AlexNet and 48 $\sim$ 51 for VGG16, depending on the security setting, whereas in the WAN setting, the factors are about

7 FALCON does not return a fully trained model, but 15 epochs is suggested in the paper [64], and plaintext training confirms that this yields an accuracy of 92% $\sim$ 94%. The accuracy for one epoch of 3FF and SPDZ was measured to 93.68% and 93.46%, respectively.

	Security	Setting	AlexNet [h]	VGG16 [h]
FALCON [64]	Passive	LAN	57	2,906
Ours	Passive	LAN	2	60
FALCON [64]	Active	LAN	219	11,398
Ours	Active	LAN	8	224
FALCON [64]	Passive	WAN	124	3,198
Ours	Passive	WAN	28	483
FALCON [64]	Active	WAN	400	12,447
Ours	Active	WAN	88	1,796

Table 11. Estimated running time for training of AlexNet (60% accuracy) and VGG16 (70% accuracy) on CIFAR-10.

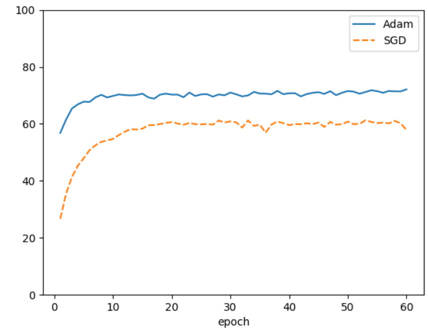
	Security	AlexNet			VGG16		
		Per epoch [GB]	Epochs	Total [GB]	Per epoch [GB]	Epochs	Total [GB]
FALCON	Passive	247	19	4,693	5,640	20	112,800
Ours	Passive	2,787	2	5,574	26,382	5	131,910
FALCON	Active	1,481	19	28,139	22,769	20	455,380
Ours	Active	12,092	2	24,184	112,220	5	561,100

Table 12. Estimated total communication cost for training of AlexNet (60% accuracy), VGG16 (70% accuracy) on CIFAR-10

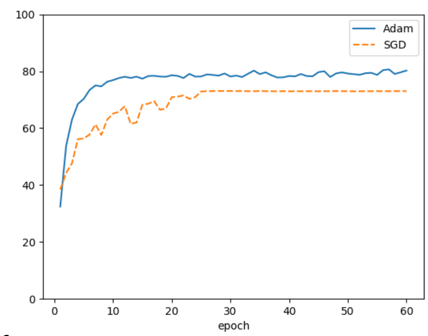
4 and 7, respectively. In Table 12, the total communication cost to train both networks is shown, estimated based on the per epoch cost.

The above comparison illustrates the advantage of the approach taken in our framework; by constructing efficient (and highly accurate) protocols that allow advanced ML algorithms such as Adam to be evaluated, despite these containing "MPC-unfriendly functions", we gain a significant advantage in terms of overall performance compared to previous works like FALCON, that attempt to achieve efficiency by simplifying the underlying ML algorithms, and optimizing the evaluation of these. As shown, the advantage when considering larger more realistic networks can in some cases be significantly more pronounced than suggested by the evaluation results on benchmark networks such as 3DNN, which is illustrated by the obtained 48 times faster evaluation of VGG16 in the LAN setting. We again highlight that this is obtained despite the advantages offered to FALCON in the comparison.

Comparison to CRYPTGPU. The CRYPTGPU [62] framework is optimized for and runs entirely on GPUs, whereas our framework is implemented on standard CPUs. Due to this difference in underlying architecture, a direct one-to-one comparison on the same hardware is not possible. However, [62] reports on the running time for a single iteration of training for AlexNet and VGG16 on CIFAR-10 using NVIDIA V100 Tensor Core GPUs [1], which we can use as a basis for a comparison of absolute running times. Specifically, from [62] we can derive that the running time of CRYPTGPU per epoch for AlexNet and VGG16 is 1364s and 22763s, respectively, whereas the running time of our implementation is 3139s and 43150s (see Table 10). While this corresponds to a factor of $1.8 \sim 2.3$ slower running time per epoch for our framework, we implement ADAM whereas CRYPTGPU implements the simpler SGD optimization. Hence, estimating the total running time to train AlexNet and VGG16 to 60% and 70% accuracy, respectively, as done above in the comparison to FALCON, we obtain that CRYPTGPU requires 7 and 126 hours, whereas our framework requires 2 and 60 hours, respectively, corresponding to a factor of $2.1 \sim 4.1$ faster running time. This is a somewhat surprising result, as GPUs are capable of providing significantly better performance for highly paralellizable task such as CNN training, and will in plaintext significantly outperform CPUs. The NVIDIA V100 GPU in particular is a powerful chip optimized to accelerate AI and high performance computing, providing performance equivalent to $9 \sim 93$ dual Xeon CPUs depending on the chosen benchmark [2]. It remains an interesting open question what performance our framework can achieve if ported to and optimized for GPUs.



(a) AlexNet



(b) VGG16

Fig. 1. Accuracy of AlexNet and VGG16 trained with Adam and SGD on CIFAR-10.

Acknowledgements

We would like to thank the reviewers of PETS 2022 for their thoughtful comments and efforts to improve our manuscript. We also thank Daichi Takenouchi for his insightful comments and experimental support. This work was partially supported by JST CREST Grant Number JPMJCR19F6, JST AIP Acceleration Research JPMJCR22U5, and JSPS KAKENHI Grant Number 19H01109.

References

[1] NVIDIA V100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/v100/.

[2] NVIDIA V100 GPU Benchmarks. https://developer.nvidia.com/hpc-application-performance.

[3] keras. https://keras.io/.

[4] Mixed precision. https://www.tensorflow.org/guide/mixed_precision.

[5] Pytorch. https://pytorch.org/.

[6] Tensorflow. https://www.tensorflow.org/.

[7] Training with mixed precision. https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html.

[8] N. Agrawal, A. S. Shamsabadi, M. J. Kusner, and A. Gascón. QUOTIENT: Two-party secure neural network training and prediction. In *ACM CCS 2019*, pp. 1231–1247.

[9] A. Aly and N. P. Smart. Benchmarking privacy preserving scientific operations. In *ACNS 19*, pp. 509–529.

[10] T. Araki, A. Barak, J. Furukawa, M. Keller, Y. Lindell, K. Ohara, and H. Tsuchida. Generalizing the SPDZ compiler for other protocols. In *ACM CCS 2018*, pp. 880–895.

[11] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy*, pp. 843–862, 2017.

[12] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *ACM CCS 2016*, pp. 805–817, 2016.

[13] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pp. 1–10. ACM Press, May 1988.

[14] F. Bourse, M. Minelli, M. Minihold, and P. Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *CRYPTO 2018, Part III*, pp. 483–512, 2018.

[15] M. Byali, H. Chaudhari, A. Patra, and A. Suresh. Flash: Fast and robust framework for privacy-preserving machine learning. In *PoPETs 2020*, pp. 459 – 480.

[16] O. Catrina and S. de Hoogh, Improved primitives for secure multiparty integer computation. *SCN, 2010*, pp. 182–199.

[17] O. Catrina and A. Saxena. Secure computation with fixed-point numbers. In *FC 2010*, pp. 35–50.

[18] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi. EzPC: Programmable, efficient, and scalable secure two-party computation for machine learning. EuroS&P 2019: 496-511

[19] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh. ASTRA: high throughput 3pc over rings with application to secure prediction. In *CCSW@CCS 2019*, pp. 81–92.

[20] H. Chaudhari, R. Rachuri, and A. Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *NDSS 2020*.

[21] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *CRYPTO 2018*, pp. 34–64.

[22] K. Chida, K. Hamada, D. Ikarashi, R. Kikuchi, and B. Pinkas. High-throughput secure AES computation. In *WAHC@CCS 2018*, pp. 13–24, 2018.

[23] R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *TCC 2005*, pp. 342–362, 2005.

[24] A. Dalskov, D. Escudero, and M. Keller. Secure Evaluation of Quantized Neural Networks. Proc. Priv. Enhancing Technol.,pp. 355–375, 2020.

[25] A. Dalskov, D. Escudero, and M. Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX Security 2021*, pp. 2183–2200.

[26] I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO 2007*, pp. 572–590, 2007.

[27] R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and fact-track multiparty computations with applications to threshold cryptography. In *PODC*, pp. 101–111, 1998.

[28] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML 2016*, pp. 201–210.

[29] O. Goldreich. *The Foundations of Cryptography - Volume 1, Basic Techniques*. Cambridge University Press, 2001.

[30] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *19th ACM STOC*, pp. 218–229, 1987.

[31] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR 2016*, pp. 770–778.

[32] M. Ito, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. In *Proceedings IEEE Globecom '87*, pp. 99–102. IEEE, 1987.

[33] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security 2018*, pp. 1651–1669.

[34] M. Keller and K. Sun. Effectiveness of mpc-friendly softmax replacement, 2020.

[35] R. Kikuchi, N. Attrapadung, K. Hamada, D. Ikarashi, A. Ishida, T. Matsuda, Y. Sakai, and J. C. N. Schuldt. Field extension in secret-shared form and its applications to efficient secure computation. In *ACISP 19*, pp. 343–361, 2019.

[36] R. Kikuchi, D. Ikarashi, T. Matsuda, K. Hamada, and K. Chida. Efficient bit-decomposition and modulus-conversion protocols with an honest majority. In *ACISP 18*,

volume 10946 of *LNCS*, pp. 64–82, 2018.
[37] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015*.
[38] H. Kitai, J. P. Cruz, N. Yanai, N. Nishida, T. Oba, Y. Unagami, T. Teruya, N. Attrapadung, T. Matsuda, and G. Hanaoka. MOBIUS: model-oblivious binarized neural networks. *IEEE Access*, 7:139021–139034, 2019.
[39] N. Koti, M. Pancholi, A. Patra, and A. Suresh. SWIFT: super-fast and robust privacy-preserving machine learning. USENIX Security Symposium 2021: 2651-2668
[40] A. Krizhevsky, V. Nair, and G. Hinton. The cifar-10 dataset, 2014.
[41] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, 2017.
[42] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. CrypTFlow: Secure TensorFlow inference. In *2020 IEEE Symposium on Security and Privacy*, pp. 336–353. IEEE Computer Society Press, May 2020.
[43] S. Laur, J. Willemson, and B. Zhang. Round-efficient oblivious database manipulation. In *ISC*, pp. 262–277, 2011.
[44] Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.
[45] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In *ACM CCS 2017*, pp. 619–631, 2017.
[46] Q. Lou, B. Feng, G. C. Fox, and L. Jiang. Glyph: Fast and accurately training deep neural networks on encrypted data. In *NeurIPS 2020*.
[47] W. Lu, Y. Fang, Z. Huang, C. Honge, C. Chen, H. Qu, Y. Zhou, and K. Ren. Faster Secure Multiparty Computation of Adaptive Gradient Descent. In *PPMLP 2020*.
[48] L. Luo, Y. Xiong, Y. Liu, and X. Sun. Adaptive gradient methods with dynamic bound of learning rate. In *ICLR 2019*.
[49] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security 2020*, pp. 2505–2522, 2020.
[50] P. Mohassel and P. Rindal. ABY^3: A mixed protocol framework for machine learning. In *CCS 2018*, pp. 35–52, 2018.
[51] P. Mohassel and Y. Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pp. 19–38, 2017.
[52] K. Nandakumar, N. K. Ratha, S. Pankanti, and S. Halevi. Towards deep neural network training on encrypted data. In *CVPR Workshops 2019*, pp. 40–48.
[53] A. Patra and A. Suresh. BLAZE: blazing fast privacy-preserving machine learning. In *NDSS 2020*.
[54] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
[55] R. Rachuri and A. Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *NDSS 2020*.
[56] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. E. Lauter, and F. Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In *USENIX Security 2019*, pp. 1501–1518.
[57] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *ASIACCS 18*, pp. 707–721, 2018.
[58] B. D. Rouhani, M. S. Riazi, and F. Koushanfar. Deepsecure: scalable provably-secure deep learning. In *Annual Design Automation Conference, DAC 2018*, pp. 2:1–2:6, 2018.
[59] S. Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.
[60] A. S. Shamsabadi, A. Gascón, H. Haddadi, and A. Cavallaro. Privedge: From local to distributed private training and prediction. *IEEE Transactions on Information Forensics and Security*, 15:3819–3831, 2020.
[61] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR 2015*.
[62] S. Tan, B. Knott, Y. Tian, and D. J. Wu. Cryptgpu: Fast privacy-preserving machine learning on the gpu. *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 1021–1038.
[63] S. Wagh, D. Gupta, and N. Chandran. SecureNN: 3-party secure computation for neural network training. *PoPETs*, 2019(3):26–49, July 2019.
[64] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin. FALCON: Honest-majority maliciously secure framework for private deep learning. *Proceedings on Privacy Enhancing Technologies*, 2021(1):188 – 208, 2021.
[65] H. W. H. Wong, J. P. K. Ma, D. P. H. Wong, L. K. L. Ng, S. S. M. Chow. Learning Model with Error — Exposing the Hidden Model of BAYHENN. In *IJCAI 2020*, pp. 3529–3535.
[66] P. Xie, B. Wu, and G. Sun. Bayhenn: Combining bayesian deep learning and homomorphic encryption for secure dnn inference. *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, Aug 2019.
[67] A. C.-C. Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pp. 160–164, 1982.
[68] Q. Zhang, C. Xin, and H. Wu. Privacy-preserving deep learning based on multiparty secure computation: A survey. *IEEE Internet of Things Journal*, 8(13):10412–10429, 2021.

A Comparison Table

We provide a comprehensive comparison among 3-party PPML systems supporting DNN in Table 13. Note that this list is by no means exhaustive; we refer to more related works in a survey paper of [68].

B Active Security for Division

Protocol 9 shows our actively secure division protocol.

Table 13. Comparison among various 3-party privacy-preserving ML systems.

	System	Secure Capability		Supported Algorithms			Threat Model		Based Techniques			LAN/ WAN		Evaluation Dataset		Network Architectures	
		Theoretical metric										Evaluation metric					
		Prediction	*Training*	*Basic*	*Batch-Norm*	*Advance*	*Semi-honest*	*Malicious*	*HE*	*GC*	*SS*	*LAN*	*WAN*	*Small*	*Large*	*Simple*	*Complex*
	ABY3 [50]	●	●	●	○	○	●	●	○	●	●	●	◐	●	○	●	○
	SecureNN [63]	●	●	●	◐	○	●	◐	○	○	●	●	●	●	○	●	○
	CryptFlow [42]	●	○	●	○	○	●	●	○	○	●	●	○	●	●	●	◐
	QuantizedNN [24]	●	○	●	◐	○	●	●	●	○	●	●	●	○	●	○	◐
	ASTRA [19]	●	○	●	○	○	●	●	○	●	●	●	●	●	○	●	○
3PC	BLAZE [53]	●	◐	●	○	○	●	●	○	●	●	●	●	●	○	●	○
	FALCON [64]	●	●	●	●	○	●	●	○	○	●	●	●	●	●	●	●
	CRYPTGPU [62]	●	●	●	●	○	●	○	○	○	●	●	○	●	●	●	●
	FantasticFour [25]	●	◐	●	○	○	●	●	○	○	●	●	○	●	●	●	◐
	Lu *et al.* [47]	●	●	◐	○	◐	●	○	○	○	●	●	●	●	○	●	○
	This work	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●

This table is inspired by Table 1 in [64]. "Basic" for Supported ML Algorithms refers to more basic ones such as linear operations, convolution, ReLU, Maxpool, and/or SGD optimizer. "Advance" refers to advance optimizers, namely, ADAM (considered in this work) and AMSGrad (in Quotient). HE, GC, SS refer to homomorphic encryption, garbled circuit, and secret sharing, respectively. "Small" for Evaluation Dataset refers to MNIST, except for BLAZE, which uses Parkinson disease dataset (its dimension is similar to MNIST) "Large" refers to larger datasets such as the well-known CIFAR-10 in particular (in all the systems that tick except QuantizedNN), or TinyImageNet (in CryptFlow and QuantizedNN, and partially in Falcon). "Simple" for Network Architectures refers to simple neural networks such as the basic 3-layer DNN (3DNN) from SecureML in particular, or other slightly different small networks from [45, 63]. "Complex" refers to more complex networks such as the well-known AlexNet and VGG-16 in particular (both are considered in Falcon and this work). ● indicates that such a system support a feature, ○ indicates that such a system does not so support so, ◐ refers to fair comparison being difficult due to various reasons; *e.g.*, , [47] provided essentially only an invert square root protocol (but no details on other ML algorithms). We refer to the full version for explanations on other ◐'s.

Protocol 9 Actively Secure Division by Public Value

Functionality: $[\![c]\!] \leftarrow \mathsf{Div}^{\mathsf{Mal}}_{(2,3)}([\![a]\!], d)$
Input: $[\![a]\!]$ and d, where a and d are multiples of 4
Output: $[\![c]\!]$, where $c \approx \frac{a}{d}$

1: Let α_p and r_p be $p = \alpha_p d + r_p$, where $0 \le r_p < d$.
2: $[\![q]\!] \leftarrow \mathcal{F}_{\mathrm{QT}}([\![a]\!])$
3: $z := 1$ if $r_p \ge d/2$ or $z := 0$ otherwise.
4: Let a_i be a sub-share of $[\![a]\!]$, *i.e.,* $a_1 + a_2 + a_3 = a \mod p$
5: **for** $1 \le j \le 3$ **do**
6: $\quad P_j$ and P_{j+1} compute
$$b_j := \begin{cases} a_j + (d - r_p) + (d - r_p)/2 \text{ in } \mathbb{N} & \text{if } j = 0 \\ a_j & \text{otherwise} \end{cases}$$
7: $\quad P_j$ and P_{j+1} set $b'_j := \begin{cases} b_j/d + 1 & \text{if } \frac{b_j}{d} - b_j/d \ge \frac{d}{2} \\ b_j/d & \text{otherwise} \end{cases}$
8: $[\![b']\!]_i := (b'_i, b'_{i+1})$ for $i = 1, 2, 3$
9: Output $[\![b']\!] - (\alpha_p + z)[\![q]\!] - 1$

C Efficiency of Division Protocol

We obtain the concrete efficiency of our protocols by considering efficient instantiations of the required building blocks. The quotient transfer protocol in [36] requires 2 bits communication and 1 round, besides a single call of $\mathcal{F}_{\mathrm{mod}}$. The modulus conversion protocol in [36] and $\mathsf{ConvertToRep}$ require $3|p| + 3$ bits and $2|p|$ bits of communication, respectively, and both 1 communication round. Furthermore, we can reduce a round required in Protocol 2 by parallel execution of $\mathcal{F}_{\mathrm{QT}}$ and $\mathsf{ConvertToRep}$. Consequently, instantiating $\mathcal{F}_{\mathrm{QT}}$ (and $\mathcal{F}_{\mathrm{mod}}$ used in QT internally) by the protocols in [36], Protocol 1 and 2 require $3|p| + 5$ and $5|p| + 5$ bits of communication and 2 communications rounds in total.

D Communication Complexities

We list the communication and round complexities of building blocks and proposed protocols in Table 14.

E Quotient Transfer Protocol

We describe the quotient transfer protocols for $[\![\cdot]\!]$ and $\langle\!\langle\cdot\rangle\!\rangle$ proposed in [36]. The key observation is that if we use an odd prime and the secret's LSB is zero, the addition of the truncated shares' LSBs corresponds to q.

In the presence of passive adversaries, we use $\langle\!\langle a\rangle\!\rangle$ as an input of the quotient transfer protocol. Because $\langle\!\langle a\rangle\!\rangle$ consists of two sub-shares, the quotient q is 0 or 1 and the (single) LSB must be 0.

In the presence of active adversaries, we use $[\![a]\!]$ as an input of the quotient transfer protocol. Because $[\![a]\!]$ consists of three sub-shares, the quotient q is 0, 1, or 2, and the second LSBs must be 0s to contain 2. The step

Table 14. Communication costs and round complexities of our protocols (and previous building blocks we use).

	Passive		Active	
	Communication cost (bits)	**#Round**	**Communication cost (bits)**	**#Round**
Previous building blocks				
Mult in $[\![\cdot]\!]$ [22]	$3\lvert p\rvert$	1	$6\lvert p\rvert$	1
Mult in $[\cdot]$ [22]	3	1	$3(\kappa+1)$	1
Modulus conversion [36] (*cf.* §F)	$3\lvert p\rvert+3$	3	$15\lvert p\rvert+6$	4
Bit decomposition [36] (*cf.* §F)	$5\ell+2$	$\ell+1$	$9\ell\kappa+6\ell+9\kappa+3$	$\ell+2$
Bit composition [10] (*cf.* §F)	$6\lvert p\rvert+6\ell+9$	$\ell+3$	$30\lvert p\rvert+15\ell\kappa+6\ell-15\kappa+9$	$2\ell+2$
Quotient transfer [36] (*cf.* §E)	$3\lvert p\rvert+5$	4	$30\lvert p\rvert+21\kappa+15$	5
Our protocols				
Division (public div)	$5\lvert p\rvert+5$	4	$30\lvert p\rvert+21\kappa+15$	5
Real-number mult.	$8\lvert p\rvert+5$	5	$36\lvert p\rvert+21\kappa+15$	6
MSNZBFit	$9\lvert p\rvert+14\ell+8$	$3\ell+4$	$36\lvert p\rvert+27\ell\kappa+15\ell-9\kappa+9$	$4\ell+4$
MSNZBFitExt	$18\lvert p\rvert+14\ell+6\left\lfloor\frac{\ell}{2}\right\rfloor+20$	$3\ell+4$	$81\lvert p\rvert+27\ell\kappa+15\ell-24\kappa$ $+3(5\kappa+2)\left\lfloor\frac{\ell}{2}\right\rfloor+24$	$4\ell+4$
Inversion	$17\lvert p\rvert+14\ell+13$ $+2(I-1)(8\lvert p\rvert+5)$ $(=: C_{\mathsf{inv}})$	$3\ell+5I+4$	$72\lvert p\rvert+27\ell\kappa+15\ell+12\kappa+24$ $+6(I-1)(12\lvert p\rvert+7\kappa+5)$ $(=: C'_{\mathsf{inv}})$	$4\ell+6I+4$
Inv square root	$34\lvert p\rvert+14\ell+6\left\lfloor\frac{\ell}{2}\right\rfloor+30$ $+3(I-1)(8\lvert p\rvert+5)$ $(=: C_{\mathsf{invsqrt}})$	$3\ell+10I-1$	$153\lvert p\rvert+27\ell\kappa+15\ell+18\kappa+3(5\kappa+2)\left\lfloor\frac{\ell}{2}\right\rfloor$ $+3(I-1)(36\lvert p\rvert+21\kappa+15)+54$ $(=: C'_{\mathsf{invsqrt}})$	$4\ell+12I-2$
Exponentiation	$19\lvert p\rvert+5\ell+t(14\lvert p\rvert+8)$ $+(I-1)(8\lvert p\rvert+5)+13$ $(=: C_{\mathsf{exp}})$	$\ell+5\lceil\log(t+1)\rceil)+14$	$87\lvert p\rvert+9\ell\kappa+6\ell+51\kappa+t(51\lvert p\rvert+21\kappa+21)$ $+(I-1)(36\lvert p\rvert+21\kappa+15)+36$ $(=: C'_{\mathsf{exp}})$	$\ell+18$ $+6\lceil\log(t+1)\rceil$
ReLU	$5\ell+6\lvert p\rvert+5$	$\ell+5$	$9\ell\kappa+6\ell+9\kappa+21\lvert p\rvert+9$	$\ell+7$
Derivative of ReLU	$5\ell+3\lvert p\rvert+5$	$\ell+4$	$9\ell\kappa+6\ell+9\kappa+15\lvert p\rvert+9$	$\ell+6$
Max pooling	$(n-1)(9\lvert p\rvert+5\ell+10)$	$(\ell+7)\log n$	$(n-1)(27\lvert p\rvert+9\ell\kappa+18\kappa+6\ell+15)$	$(\ell+9)\log n$
Softmax	$KC_{\mathsf{exp}}+C_{\mathsf{inv}}$	$4\ell+5I+5\lceil\log(t+1)\rceil+18$	$KC'_{\mathsf{exp}}+C'_{\mathsf{inv}}$	$5\ell+6I+6\lceil\log(t+1)\rceil+22$
Batch-normalizaiton	$m(8\lvert p\rvert+5)+5\lvert p\rvert+5+C_{\mathsf{invsqrt}}$	$3\ell+10I+8$	$m(36\lvert p\rvert+21\kappa+15)+30\lvert p\rvert+21\kappa+15+C'_{\mathsf{invsqrt}}$	$4\ell+12I+9$

ℓ is the input bit length, p is a prime for share domain $\mathbb{Z}_p$, I is # of iterations, κ is a statistical security parameter for active security (assume $|p|>\kappa$), K is the number of classes, m is a batch size, t is the lookup table vs Taylor series threshold, and n is the vector size.

Protocol 10 Quotient Transfer for $\langle\!\langle\cdot\rangle\!\rangle$

Functionality: $\langle\!\langle q\rangle\!\rangle \leftarrow \mathsf{QT}(\langle\!\langle a\rangle\!\rangle)$
Input: $\langle\!\langle a\rangle\!\rangle$ where a is a multiple of 2.
Output: $\langle\!\langle q\rangle\!\rangle$ where $\langle\!\langle a\rangle\!\rangle_1 + \langle\!\langle a\rangle\!\rangle_2 = a + qp$

1: P_0 and P_1 secret-share LSBs of $\langle\!\langle a\rangle\!\rangle_1$ and $\langle\!\langle a\rangle\!\rangle_2$ in modulo 2, respectively. Let them be $[\langle\!\langle a\rangle\!\rangle_1^{(1)}]$ and $[\langle\!\langle a\rangle\!\rangle_2^{(1)}]$.
2: $[q] := [\langle\!\langle a\rangle\!\rangle_1^{(1)}] \oplus [\langle\!\langle a\rangle\!\rangle_2^{(1)}]$.
3: $[\![q]\!] \leftarrow \mathcal{F}_{\mathrm{mod}}([q])$
4: $\langle\!\langle q\rangle\!\rangle \leftarrow \mathsf{ConvertToAdd}([\![q]\!])$
5: Output $\langle\!\langle q\rangle\!\rangle$

Protocol 11 Quotient Transfer for $[\![\cdot]\!]$

Functionality: $[\![q]\!] \leftarrow \mathsf{QT}([\![a]\!])$
Input: $[\![a]\!]$ where a is a multiple of 4.
Output: $[\![q]\!]$ where $a_1 + a_2 + a_3 = a + qp$

1: The parties *locally* generate shares of the second LSBs of a_1, a_2, and a_3 in modulo 2, respectively. Let them be $[a_1^{(1)}]$, $[a_1^{(2)}]$, $[a_2^{(1)}]$, $[a_2^{(2)}]$, $[a_3^{(1)}]$, and $[a_3^{(2)}]$.
2: $[q_1] := [a_1^{(1)}] \oplus [a_2^{(1)}] \oplus [a_3^{(1)}]$.
3: $[c] \leftarrow ([a_1^{(1)}] \oplus [a_3^{(1)}]) \cdot ([a_2^{(1)}] \oplus [a_3^{(1)}]) \oplus [a_3^{(1)}]$.
4: $[q_2] := [a_1^{(2)}] \oplus [a_2^{(2)}] \oplus [a_3^{(2)}] \oplus [c]$.
5: $[\![q_1]\!] \leftarrow \mathcal{F}_{\mathrm{mod}}([q_1])$
6: $[\![q_2]\!] \leftarrow \mathcal{F}_{\mathrm{mod}}([q_2])$
7: Output $[\![q]\!] := [\![q_1]\!] + 2[\![q_2]\!]$

3 and the last term of step 4 come from the fact that the carry of a_1, a_2, a_3 is $(a_1 \oplus a_3)(a_2 \oplus a_3) \oplus a_3$. This protocol is secure against an active adversary using a general compiler, such as [21], to compute multiplication. Note, in the step 1, the "share of sub-shares" can be generated locally. For details, see Section 4.4 in [36].

F Conversion Protocols

In the following, we describe the used bit-composition, bit-decomposition, and modulus conversion protocols.

The bit-composition protocol is obtained by modifying the protocol [10] designed to work for values in $\mathbb{Z}_{2^\ell}$ to be applicable to values in $\mathbb{Z}_p$. Here, the main difference lie in how the ℓ-th bit carry is handled. The resulting protocol is shown in Protocol 12. In the protocol, Carry denotes an algorithm that on input $[x]$ outputs $[y]$, where y is the carry obtained by addition of the sub-shares of $[x]$: x_1, x_2, and x_3. This is the same as the step 1-3 of Protocol 11.

The bit-decomposition and modulus conversion protocols are obtained by simplifying the corresponding protocols from [36] by assuming the use of a Mersenne prime. The resulting protocols are shown in Protocol 13 and Protocol 14. $\mathcal{F}_{\mathrm{rand}}$ denotes the functionality of generating a share of a random number, which can be implemented via local computation [23]

Note that in Protocol 14, the required communication complexity for computing $[\![r_1 \oplus r_2 \oplus r_3]\!]$ in Step 3 and 4 is only $3|p|$ although two multiplications are used. We use the IKHC multiplication protocol [22] in which P_i on input $[\![a]\!]_i = (a_i, a_{i+1})$ and $[\![b]\!]_i = (b_i, b_{i+1})$ sends $d_i := a_i b_{i+1} + a_{i+1} b_i - s_{i+2,i}$ to P_{i+1}, where $s_{i+2,i}$ is a random value shared between P_i and P_{i+2}. At the start

Protocol 12 Bit-composition

Functionality: $[\![a]\!] \leftarrow \mathsf{BC}([a_1], \dots, [a_\ell])$
Input: $[a_1], \dots, [a_\ell]$
Output: $[\![a]\!]$, where $a = \sum_{i=1}^{\ell} 2^{i-1} a_i$
Parameter: The bit-length of secret, ℓ

1: $[a'_0] := 1 \oplus [a_0]$
2: $[q_0] \leftarrow \mathsf{Carry}([a'_0])$
3: $[c_0] := [a_0]$
4: **for** $2 \leq i \leq \ell$ **do**
5: $\quad [a'_i] := [a_i] \oplus [c_{i-1}] \oplus (1 \oplus [q_{i-1}])$
6: $\quad [q_i] \leftarrow \mathsf{Carry}([a'_i])$
7: $\quad [c_i] := ([a_i] \oplus [c_{i-1}])((1 \oplus [q_{i-1}]) \oplus [c_{i-1}]) \oplus [c_{i-1}]$
8: $[\![c_\ell]\!] \leftarrow \mathcal{F}_{\mathrm{mod}}([c_\ell])$
9: $[\![q_\ell]\!] \leftarrow \mathcal{F}_{\mathrm{mod}}([q_\ell])$
10: $[\![b]\!] \leftarrow \sum_{i=1}^{\ell} 2^{i-1} [a'_i] \mod p$
11: Output $[\![b]\!] + 2^{\ell+1}([\![c_\ell]\!] + (1 \oplus [\![q_\ell]\!])) + \sum_{i=\ell}^{|p|} 2^i$

Protocol 13 Bit-decomposition

Functionality: $([a^{(1)}], [a^{(2)}], \dots, [a^{(\ell)}]) \leftarrow \mathsf{BDC}([\![a]\!])$
Input: $[\![a]\!]$ where a is an ℓ-bit value and $p > 2a$.
Output: $([a^{(1)}], [a^{(2)}], \dots, [a^{(\ell)}])$ where $\sum_{i=1}^{\ell} 2^{i-1} a^{(i)} = a$

1: $\langle\!\langle a \rangle\!\rangle \leftarrow \mathsf{ConvertToAdd}([\![a]\!])$
2: $\langle\!\langle a' \rangle\!\rangle := 2\langle\!\langle a \rangle\!\rangle$, and $\langle\!\langle a \rangle\!\rangle_i^{(j)}$ be the j-th bit of $\langle\!\langle a \rangle\!\rangle_i$.
3: P_1 and P_2 secret-share the least $\ell + 1$ bits of their shares bit-by-bit in $\mathbb{Z}_2$, respectively. (The parties obtain $[\langle\!\langle a' \rangle\!\rangle_i^{(j)}]$ for $i = 1, 2$ and $1 \leq j \leq \ell + 1$)
4: The parties obtain $([b^{(1)}], \dots, [b^{(\ell+1)}])$ by computing an adder circuit: an input of j-th bit is $[\langle\!\langle a' \rangle\!\rangle_1^{(j)}]$, $[\langle\!\langle a' \rangle\!\rangle_2^{(j)}]$, and the carry from the previous bit, except the 1st bit is $[\langle\!\langle a' \rangle\!\rangle_1^{(1)}]$, $[\langle\!\langle a' \rangle\!\rangle_2^{(1)}]$, and $[\langle\!\langle a' \rangle\!\rangle_1^{(1)}] + [\langle\!\langle a' \rangle\!\rangle_2^{(1)}]$.
5: $[a^{(j)}] = [b^{(j+1)}]$ for $1 \leq j \leq \ell$ ▷ Discard LSB
6: Output $([a^{(1)}], [a^{(2)}], \dots, [a^{(\ell)}])$

of Step 3, P_1 has $[\![r_1]\!]_1 = (r_1, 0)$ and $[\![r_2]\!]_1 = (0, r_2)$; similarly, P_2 has $(0,0)$ and $(r_2, 0)$, and P_3 has $(0, r_1)$ and $(0,0)$. Multiplying $[\![r_1]\!]$ with $[\![r_2]\!]$, $d_i = -s_{i+2,i}$ for $i = 2, 3$, means that d_2 and d_3 are publicly known to *all* the parties. Therefore, these values can be constants, e.g., 0, and do not need to be sent. Similarly, when multiplying $[\![r_1 \oplus r_2]\!]$ with $[\![r_3]\!]$, P_1 has $[\![r_3]\!]_1 = (0,0)$ and $d_1 = -s_{3,1}$. By using the same technique, P_1 sends nothing. The elimination of these communications results in a total communication of $3\,|p|$, since the communication costs of the first and second XORs are $|p|$ and $2\,|p|$.

G Parameters

The parameters used for the elementary functions are listed in Table 15. The parameters for Adam are as follows: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\eta = 0.001$, and $\epsilon = 0$. Note that ϵ in the original paper [37] is a value added to prevent division by 0, but as our InvSqrt protocol has the property that the output is 0 when the input is 0, we

Protocol 14 Modulus-conversion ($[\cdot]$ to $[\![\cdot]\!]$)

Functionality: $[\![a]\!] \leftarrow \mathsf{ModConv}([a])$
Input: $[a]$
Output: $[\![a]\!]$

1: $[r] \leftarrow \mathcal{F}_{\mathrm{rand}}$
2: The parties locally generate shares of sub-shares of $[r]$ in modulus p. ▷ obtain $[\![r_1]\!]$, $[\![r_2]\!]$, and $[\![r_3]\!]$, where $r_1 + r_2 + r_3 = r \mod 2$.
3: $[\![r_1 \oplus r_2]\!] = [\![r_1]\!] + [\![r_2]\!] - 2[\![r_1]\!][\![r_2]\!]$
4: $[\![r]\!] = [\![r_1 \oplus r_2 \oplus r_3]\!] = [\![r_1 \oplus r_2]\!] + [\![r_3]\!] - 2[\![r_1 \oplus r_2]\!][\![r_3]\!]$
5: Reconstruct $a + r$ from $[a + r] = [a] + [r]$.
6: Output $[\![r]\!]$ if $a + r = 0$, $1 - [\![r]\!]$ otherwise.

Parameter	α	δ	ℓ	I	t	μ	β
Exponent	20	14	25	4	4	−15	27
Inv	14	14	29	4	-	-	-
InvSqrt	26	10	28	6	-	-	-

Table 15. Parameters used for elementary functions.

can simply set $\epsilon = 0$ in our implementation. This yields the same result as in plaintext evaluation as $\hat{V}_{t+1} = 0$ implies $\hat{M}_{t+1} = 0$, and hence $W_{t+1} = W_t$ in Adam.

H Comparison to Related Work for Elementary Functions

Comparing Inversion/Division to [17]. The division from Catrina-Saxena [17] is based on a similar approximation approach to our Taylor-series-based inversion which trivially extends to division ([17] uses Goldschmith's method). The performance of this approximation is highly dependent on how truncation is done for the iterative fixed-point multiplication and how the initial value is obtained. While [17] uses an accurate efficient truncation and scaling approach, these both have a communication overhead of $\mathcal{O}(|p|^2)$, whereas our truncation and MSNZBFit protocols has overhead $\mathcal{O}(|p|)$, leading to an overall more communication efficient inversion achieving the same accuracy and round complexity.

Comparing Inverse Square-root to [47]. The inverse square root protocol of Lu *et al.* [47] is based on polynomial approximation, whereas our protocol make use of Newton's method that allows fine-tuning of output precision. We can infer from [47][Table 1] that the output accuracy is $8 \sim 12$ bits, which is much lower than our achieved $26 \sim 29$ bits accuracy, despite [47] using a 128-bit ring that is about twice the size of the field used in our protocol (see Table 16). Assuming bit-

Table 16. Output Accuracy for Elementary Protocols

	Protocol	Field Size [bits]	Accuracy [bits]	Accuracy / Field Size [%]
Inverse Sqrt	Lu *et al.* [47]	128	$8 \sim 12$	$6.2 \sim 9.3\%$
	Ours	61	$26 \sim 29$	$42.6 \sim 47.5\%$
Exponentiation	Aly *et al.* [9]	245	40	16%
	Ours	61	$23 \sim 25$	$37.7 \sim 40.9\%$

composition is implemented in $\mathcal{O}(\ell)$ rounds (*e.g.,* [10]), the round complexity of both protocols is $\mathcal{O}(\ell)$.[8]

Comparing Exponentiation to [9]. The exponentiation from Aly-Smart[9] is based on splitting the input into an integral part and fractional remainder, and then computing the exponentiation of these separately. The later is obtained via a Padé polynomial approximation with coefficients requiring more than 80 bits of precision. To ensure the computation is numerically stable, the authors recommend a large internal precision for this computation, leading to a 245-bit field size for their concrete implementation. In contrast, our protocol is fine-tune to our setting and ensures $23 \sim 25$-bit output accuracy with a field size of only 61 bits (see Table 16). Using bit-decomposition of $\mathcal{O}(\ell)$ rounds (*e.g.,* [36]), the round complexity of both [9] and ours is $\mathcal{O}(\ell)$.

I Convergence: 3DNN

For completeness, we compare convergence of our secure training with that obtained using TensorFlow. Table 17 shows the obtained accuracy for the trained 3DNN network for 1 to 10 epochs in the passive security setting. As shown, training converges after ~ 5 rounds, obtaining an accuracy of ~ 97.7.

J SoftMax Clipping

As typically done in plaintext evaluation, we stabilize the softmax function via input clipping. Specifically, before evaluating softmax, we limit any input value a to the range $-15 \leq a \leq 15$, by setting any input outside this range to -15 or 15 (for $a < -15$ and $a > 15$, respectively), via simple comparison and conditional assignment.

#epochs	1	2	3	4	5	6	7	8	9	10
TensorFlow	95.50	96.66	97.43	97.28	97.48	97.68	97.79	97.84	97.51	97.75
Ours	95.60	96.85	97.23	97.67	97.68	97.50	97.58	97.75	98.06	97.75

Table 17. Measured accuracy of 3DNN: TensorFlow (plaintext) vs. Our secure training.

8 The round complexity stated in [47] suggests bit-composition is implemented in $\mathcal{O}(\log(\ell))$ rounds, but it is not explained how this is achieved. Improved bit-composition would benefit both [47] and our protocol, but we note that to achieve our stated accuracy, we require 19 rounds of iterations in Newton's method.