Truncation Untangled: Scaling Fixed-Point Arithmetic for Privacy-Preserving Machine Learning to Large Models and Datasets

Christopher Harth-Kitzerow
Technical University of Munich, BMW Group
christopher.harth-kitzerow@tum.de

Georg Carle
Technical University of Munich
carle@net.in.tum.de

Abstract

Fixed point arithmetic (FPA) is essential to enable practical Privacy-Preserving Machine Learning. When multiplying two fixed-point numbers, truncation is required to ensure that the product maintains correct precision. While multiple truncation schemes based on Secure Multiparty Computation (MPC) have been proposed, which of the different schemes offers the best trade-off between accuracy and efficiency on common PPML datasets and models has remained underexplored.

In this work, we study several different stochastic and exact truncation approaches found in the MPC literature that require different slack sizes, i.e. additional bits required by each secret share to ensure correctness. We provide novel, improved construction for each truncation approach in the semi-honest 3-PC and malicious 4-PC settings which reduce communication and round complexity up to three times. Moreover, we propose a truncation scheme that does not introduce any communication overhead in the online phase and matches the accuracy of plaintext PyTorch inference of VGG-16 on the ImageNet dataset with over 80% accuracy using shares with a bitlength of only 32. This is the first time that high PPML accuracy is demonstrated on ImageNet.

1 Introduction

Privacy-Preserving Machine Learning (PPML) [23] aims to enable the training and inference of Machine Learning models while keeping model parameters and data private using cryptographic techniques. Secure Multiparty Computation (MPC) [19] is a key technology to enable PPML. MPC allows multiple parties to jointly compute a function on their private inputs without revealing any information about the inputs to the other parties. While private training of state-of-the-art neural network models is still out of reach given the performance overhead of MPC, private inference is already practical for many models and datasets: The PIGEON PPML framework [10] recently demonstrated that private inference of various sate-of-the-art convolutional neural networks (CNNs) on the popular CIFAR-10 image classification dataset [17] achieves throughputs of more than 100 images per second. These performance improvements enable for the first time to systematically study a wide range of different MPC-specific configurations and their impact on the performance and accuracy of large CNN models such as VGG-16 [28] and various ResNet architectures [11].

One aspect that helped PPML achieve practical performance is the use of fixed point arithmetic (FPA). FPA allows parties to perform computation using integer-only arithmetic, which is significantly more efficient in MPC than floating point arithmetic [14]. A downside of fixed point arithmetic is that the resulting accuracy may not be equivalent to plaintext floating point inference. Therefore, it is crucial to carefully choose the fixed point precision and bit width of numbers to minimize accuracy loss.

In FPA, real numbers are represented by ℓ-bit integers where k bits are used for the fractional part and $\ell - k$ bits are used for the integer part. The product of two fixed point numbers results in $2k$ fractional bits and $\ell - 2k$ of integer bits. In order to prevent overflow and maintain the correct precision, the number of fractional bits must be reduced back to k bits by performing an arithmetic right shift. This process is called truncation. While truncating a number is trivial in plaintext, it is a non-trivial cryptographic primitive in MPC.

Over the last years, several different truncation approaches have been proposed [6, 8, 21, 22]. Exact truncation approaches [8] are independent of the value or randomness of a secret share and are equivalent to an arithmetic right shift in the plaintext domain. Stochastic truncation approaches [6, 21, 22], on the other hand, are biased toward the nearest truncated value and may differ by one bit from the plaintext truncation. While there are highly efficient stochastic truncation schemes [21, 22], they can cause truncation failure with a high probability, i.e. the result of the truncation can be significantly different from the expected value. The probability of truncation failure can be reduced by either increasing the ring size of the MPC protocols or by using more expensive stochastic truncation schemes that only require a slack of one bit [6].

Similarly, exact truncation schemes may require a slack of one bit or no slack at all, depending on the specific protocol used [7, 8]. Primitives that reduce slack size are typically less efficient in terms of communication complexity. However, the reduced slack may allow using smaller ring sizes for the entire forward pass, which in turn can reduce the communication and computational complexity of the entire computation.

Contributions The tradeoffs of utilizing one truncation scheme over another such as different ring sizes and probabilities of truncation failures, have not been systematically studied in the context of PPML inference. Additionally, existing truncation schemes often introduce a high overhead in both total communication complexity and round complexity, which throttles the performance of PPML inference.

Our contributions are two-fold: First, we propose the following novel techniques to reduce the communication overhead and required slack-size of various truncation schemes in the context of PPML in general.

- We provide efficient constructions for several truncation approaches in the semi-honest 3-PC and malicious 4-PC settings based on the Trio and Quad MPC protocols [9]. Our primitives reduce the communication overhead of these truncation schemes compared to previous work. Table 1 provides an overview of the communication complexity of our truncation primitives compared to the state-of-the-art, with improvements of up to 3 times. Our constructions are described in §4.
- The focus on PPML allows us to exploit the typical structure of neural networks and fuse truncation primitives with the evaluation of other layers. This further reduces or even eliminates the communication overhead of truncation. Our PPML-specific optimizations are described in §5, while an overview is again provided in Table 1.
- We also propose several tweaks that reduce the required slack size when using truncation in the context of PPML. For instance, public denominators in average pooling can often be expressed with fewer fractional bits while guaranteeing equivalent precision which reduces the probability of truncation failure.
- Based on these tweaks we propose a novel truncation approach that utilizes different truncation strategies for different layers of a neural network. Our approach does not introduce any communication overhead in the online phase and matches the accuracy of plaintext VGG-16 inference on the ImageNet dataset with over 80% accuracy using shares with a bitlength of only 32.

Second, we conduct a large-scale systematic evaluation of several truncation schemes, ring sizes, neural network architectures, and datasets to provide clear guidelines on how to choose the right truncation scheme while focusing solely on PPML inference.

- We implement all studied truncation primitives into the open-source HPMPC [9] MPC framework and evaluate the runtime and accuracy of these truncation approaches for different ring sizes and neural network architectures based on various benchmark datasets. For the first, time we also evaluate PPML inference accuracy on the ImageNet dataset [26] and thus answer the long-standing question of whether fixed-point MPC can scale to large-scale models and datasets [23]. Our accuracy-related results are presented in §7.
- We also study which choices in plaintext training lead to a reduced probability of truncation failure in PPML inference. We find that training models with the ADAMW [20] optimizer benefits stochastic truncation due to its weight decay mechanism.

Given our extensive evaluation, we provide end-to-end guidelines on regularization techniques to consider for plaintext training, which ring size and fractional bitlength to use for inference, and which truncation approach offers the best tradeoff between communication complexity and accuracy.

Table 1: Communication complexity of truncating a share by t bits

Primitive	Scheme	PRE	ON	Rounds
Stochastic Truncation, Large Slack	ABY3 [21]	0	ℓ	1
	Ours (3PC)	ℓ	**0**	**0**
	Ours (3PC)[p]	0	*0*	*0*
	Tetrad [16]	ℓ	ℓ	0[c]
	Ours (4PC)	ℓ	ℓ	0[c]
	Tetrad [16][p]	0	0	0
	Ours (4PC)[p]	0	0	0
Stochastic Truncation, 1-bit Slack	Dalskov [6]	0	8ℓ	3
	Ours (3PC)	3ℓ	$\mathbf{2}\ell$	**1**
	Ours (3PC)[p]	3ℓ	*0*	*0*
	Fantastic [7]	0	12ℓ	3
	Ours (4PC)	4ℓ	$\mathbf{4}\ell$	**1**
	Ours (4PC)[p]	6ℓ	*0*	*0*
Ecact Truncation, 1-bit Slack	Escudero [8]	$2A_{\ell,t}$	$2A_{\ell,t}$	$2A_{\ell}$
	Ours (3PC)	$A_{\ell,t}$	$\mathbf{A}_{\ell,\mathbf{t}}$	$\mathbf{A}_{\ell}$
	Ours (3PC)[p]	A_t	A_t	*0*
	Fantastic [7]	$3A_{\ell,t}$	$3A_{\ell,t}$	$2A_{\ell}$
	Ours (4PC)	$A_{\ell,t}$	$\mathbf{A}_{\ell,\mathbf{t}}$	$\mathbf{A}_{\ell}$
	Ours (4PC)[p]	A_t	A_t	*0*
Trunc. prior to Mult.[d]	Bicoptor 2.0 [31]	2T	2T	T
	Ours	**T**	**T**	T

Costs are measured in ring elements (ℓ), ℓ-bit and t bit extraction circuits ($A_{\ell,t}$), or truncation primitives (T).
[c] Constant-round online communication [9].
[p] Optimized construction when fusing truncation with certain PPML layers such as ReLU or BatchNorm.
[d] Tweak to truncate two shares prior to multiplication to reduce the probability of truncation failure.

2 Related Work

We focus on truncation primitives for MPC protocols computing over the ring $\mathbb{Z}_{2^\ell}$ with ring sizes $\ell \in \{16, 32, 64\}$ bits as computation over these rings can be expressed with native integer operations on modern hardware. While truncation is straightforward for protocols computing over fields due to the availability of division, field-based protocols introduce significant real-world overhead due to the lack of native hardware support for field operations.

While several works proposed efficient ring-based truncation techniques, the comparison of these techniques is lacking with a few exceptions. Piranha [30] analyzed the impact of different numbers of fractional bits on the PPML inference accuracy and recommends a ring size of 64 bits and 26 bits representing the fractional part when using stochastic truncation based on ABY3 [21]. However, Bicoptor 2.0 [31] found that Piranha did not sample random values for their experiments which hides the impact of truncation failure. Bicoptor 2.0 [31] thus found in their experiments that 15 bits for the fractional part are appropriate for the ring size of 64 bits to avoid truncation failure which matches the results of Koti et al. [15] who use 13 bits for the fractional part for a ring size of 64 bits. Fantastic Four [7] is the only work that empirically compared two different truncation schemes. When comparing their stochastic truncation scheme requiring only one bit of slack to the stochastic truncation scheme based on ABY3 [21] requiring a large slack, they found that their scheme is two times more efficient on a ring size of 64 bit than the stochastic truncation scheme on 80 bit. The overhead, however, was mostly attributed to the computational inefficiency of performing 80-bit computations on 64-bit hardware. Thus it is left to examine whether such a large ring size is indeed necessary for the stochastic truncation when considering that prior works achieved high accuracy with a ring size of 64 bits for various neural network architectures using the truncation schemes based on ABY3 [21] and SecureML [22].

Bicoptor 2.0 also showed that truncating two factors before multiplication rather than truncating the product after multiplication can reduce the probability of truncation failure. As a downside, their construction requires the parties to perform two truncations for each multiplication instead of one.

Stochastic Truncation SecureML [22] proposed the first stochastic truncation protocol in the 2PC setting. ABY3 [21] picked up on their construction and proposed a stochastic truncation protocol in the more efficient semi-honest 3PC setting. Since then, several stochastic truncation primitives have been proposed in the 3PC [9, 25] and the malicious 4PC settings [5, 7, 9, 15, 16] based on these two prior works. ABY3 [21] and Trident [5] first proposed fusing the multiplication with the truncation primitive in the 3PC and 4PC settings respectively to reduce the communication overhead of truncation. This idea was picked up by Tetrad [16], Trio, and Quad [9] to integrate stochastic truncation into the multiplication protocol at no additional communication cost in the 3PC and 4PC settings. Truncating a share after multiplication with a public fixed-point value, however, still requires parties to communicate in these protocols.

The probability of truncation failure when using stochastic truncation schemes based on ABY3 [21] and SecureML [22] increases proportionally to the closeness of the absolute plaintext value to the ring size 2^ℓ [31]. For this reason, frameworks that utilize these truncation schemes typically increase the ring size to reduce the probability of truncation failure. This overhead in ring size is referred to as a "slack".

Security of Stochastic Truncation Li et al. [18] raised concerns about stochastic truncation schemes being inherently insecure using standard security requirements of MPC protocols [3] as the output of the truncation function depends on the same randomness that already masks the input share. This finding motivated Orca [12] to propose a stochastic truncation scheme that does not rely on the same randomness as the input share. However, Santos et al. [27] showed that by using an alternative ideal functionality for stochastic truncation, all earlier covered truncation schemes can be in fact proven secure.

Stochastic Truncation with Reduced Slack Dalskov et al. [6] proposed a stochastic truncation scheme that requires no slack but their scheme only guarantees correct results if the plaintext value is not negative. They provide an efficient construction for their truncation scheme in the semi-honest 3PC setting. Fantastic Four [7] extended their protocol to the malicious 4PC setting. To enable Dalskov et al.'s truncation scheme for both negative and positive values, Escudero et al. [8] introduced a simple trick that requires a slack of 1 bit.

Exact Truncation Exact truncation schemes typically require share conversion from the arithmetic to the boolean domain, and vice versa. While they can be implemented without requiring any slack, the communication overhead of share conversion is significant due to the necessity of computing a boolean circuit for sign bit extraction or the addition of decomposed shares. Boolean adders can be implemented using Ripple Carry Adders (RCAs), or Parallel Prefix Adders (PPAs) [21] using either regular AND gates, multi-input AND gates [24], or multi-input scalar products [2]. These approaches have different tradeoffs in terms of communications rounds and number of messages exchanged but each circuit requires at least $O(log(\ell))$ communication rounds and $O(\ell)$ messages exchanged.

Exact Truncation with Slack Escudero et al. [8] proposed a generic exact truncation primitive that requires computing two bit extraction circuits sequentially and requires a slack of

1 bit. Fantastic Four [7] proposed an exact truncation primitive with 1 bit of slack in the 4PC setting that utilizes local share splitting and requires computing only one most-significant bit extraction circuit and one t-least-significant bit extraction circuit in parallel, thus achieving fewer communication rounds. However, the sharing semantics of Fantastic Four require the parties to compute each adder on four inputs which in turn increases both the number of communication rounds and the number of messages exchanged.

3 Preliminaries

In this section, we provide the preliminaries, including truncation notations and sharing semantics. We build our truncation primitives on top of the semi-honest Trio 3PC protocol and the maliciously secure Quad protocol in the honest majority setting [9]. As these protocols use similar sharing semantics as ABY2 [24], Astra [4], and Tetrad [16], our truncation primitives can be adapted to these protocols as well.

Notations We use $\mathcal{P}$ to denote the set of parties and P_i to denote the ith party. $\mathcal{P}_\Phi$ denotes a subset of $\mathcal{P}$ consisting of parties in the set Φ. For instance, $\mathcal{P}_\Phi$ or simply $\mathcal{P}_{i,j}$ indicates the set of parties $\Phi = \{P_i, P_j\}$. Similarly, x_Φ or simply $x_{i,j}$ denote a value possessed by all parties in $\Phi = \{P_i, P_j\}$.

Truncation of a value x by t-bits is denoted by $x^t = \lfloor \frac{x}{2^t} \rfloor$. We denote the exact truncation of x as $(x)^t$ and stochastic truncation as $(x)^{st}$. We refer to a stochastic and exact truncation scheme requiring a slack of s as $TS_{\{s\}}$ and $TE_{\{s\}}$, respectively. Further, we denote by $TS_{\{L\}}$ a truncation scheme that requires a large slack that is not further specified. Finally, we denote by $TS_{\{Mix\}}$ our truncation strategy that utilizes different stochastic truncation approaches throughout the network.

Sharing Schemes We use different sharing schemes throughout this work, and their overview is provided below.

1. $[\cdot]$-sharing: A value $x \in \mathbb{Z}_{2^\ell}$ is $[\cdot]$-shared among $\mathcal{P}_\Phi$, if each $P_i \in \mathcal{P}_\Phi$ holds x^i such that $\sum_i x^i = x$.
2. $[\![\cdot]\!]$-sharing: A value $x \in \mathbb{Z}_{2^\ell}$ is $[\![\cdot]\!]$-shared among $\mathcal{P}_\Phi$, if parties in $\mathcal{P}_\Phi$ hold m_x and $[\lambda_x]$ such that $\mathsf{m}_x = x + \lambda_x$.
3. $\langle\cdot\rangle$-sharing: To denote a generic secret share of $x \in \mathbb{Z}_{2^\ell}$ without specifying its sharing semantics, we use $\langle x \rangle$.

Primitives in this work that are based on $[\![\cdot]\!]$-sharing are constructed specifically for the Trio and Quad protocols, while primitives in this work that are based on $\langle\cdot\rangle$-sharing can be implemented with any linear secret sharing scheme and are not tied to the Trio and Quad protocols. Additionally, $\langle\cdot\rangle^B$ represents Boolean sharing, where addition and multiplication are replaced by XOR and AND gates, respectively. Similarly, $\langle\cdot\rangle^A$ denotes arithmetic sharing. The superscript is omitted when the type of sharing is clear from the context.

Table 2: Sharing semantic for 3PC and 4PC protocols

	Party	Trio (3PC)	Quad (4PC)
Sharing Semantics $[\![x]\!]$	P_0	λ_x^1, λ_x^2	$\mathsf{m}_x^*, \lambda_x^1, \lambda_x^2$
	P_1	$\mathsf{m}_{x,2}, \lambda_x^1$	$\mathsf{m}_x, \lambda_x^*, \lambda_x^1$
	P_2	$\mathsf{m}_{x,1}, \lambda_x^2$	$\mathsf{m}_x, \lambda_x^*, \lambda_x^2$
	P_3	-	$\lambda_x^*, \lambda_x^1, \lambda_x^2$
	Correlation	$\mathsf{m}_{x,1} = x + \lambda_x^1$	$\lambda_x = \lambda_x^1 + \lambda_x^2$
		$\mathsf{m}_{x,2} = x + \lambda_x^2$	$\mathsf{m}_x = x + \lambda_x$
			$\mathsf{m}_x^* = x + \lambda_x^*$

Table 2 summarizes the sharing semantics for Trio and Quad protocols. We refer to shares of λ_x as input-independent shares, and m_x as input-dependent shares. All input-independent shares are generated non-interactively in the preprocessing phase, while computing input-dependent shares may require interaction between parties. Some shares in Quad only serve the purpose of verification and are only required by the end of the protocol. For instance, all communication tied to m_x^* is considered constant-round communication that does not affect the round complexity of the online phase. For further details, we refer readers to [9].

Functionalities Our primitives utilize cryptographically secure implementations of the $\mathcal{F}_{\mathsf{SRNG}}$ functionality, which allows a subset of parties $\mathcal{P}_\Phi$ to generate fresh random values without interaction. Random values are generated with the help of a pseudorandom function (PRF). The protocol assumes a shared-key setup ($\mathcal{F}_{\mathsf{setup}}$) is established at the beginning, which is the case with most existing protocols [1,5,7]. To achieve malicious security in Quad, each party needs to verify the correctness of the messages it receives. To enable this, the parties have access to a Compare-View functionality, similar to the joint-message passing in SWIFT [15] and the jsnd primitive in Tetrad [16]. We refer readers to [9] for the formal descriptions of Compare-View (Π_{CV}) and sampling shared random values (Π_{SRNG}).

4 Truncation Approaches

In this section, we introduce the different truncation approaches that we investigate in this work along with efficient constructions for each truncation approach in Trio and Quad.

4.1 Truncation-Related Primitives

To construct efficient truncation protocols, we first observe that existing truncation schemes often benefit from parties holding a $\binom{2}{2}$ additive sharing ($[\cdot]$-sharings), i.e. a subset of parties holds x^1 and a disjoint subset of parties holds x^2 with $x = x^1 + x^2$. Parties can use this sharing to first locally compute modulus or truncation operations on x^1 and x^2 respectively, second re-share the modified values, and third aggregate the modified values.

The Trio and Quad MPC protocols [9] are especially suitable for this routine as a subset of parties can locally obtain $-\lambda_x$ and a disjoint subset of parties can locally obtain m_x with $x = (-\lambda_x) + \mathsf{m}_x$. To also enable efficient re-sharing we introduce the subset-sharing primitives that allow $\mathcal{P}_{\lambda_x}$, i.e. the subset of parties that can locally obtain λ_x, and $\mathcal{P}_{\mathsf{m}_x}$, i.e. the subset of parties that can locally obtain m_x to share local values with the parties of the other subset.

Figures 1,2 and 3, 4 show the subset-sharing primitives for the 3-PC and 4-PC settings respectively. The primitives minimize the communication overhead of secret sharing by exploiting that some shares can be locally obtained using Π_{SRNG} while other shares can be set to 0. The 4PC primitives utilize the verify-send technique introduced by SWIFT [15] where out of two parties holding a message, one party sends the message to the recipient while the other parties verify the message with the recipient using Π_{CV}. The honest majority assumption ensures that the recipient receives the correct message or aborts the protocol. Further, the 4PC protocols utilize that m_x^* is only required by P_0 at the end of the protocol and therefore related online communication does not add to the round complexity of the protocol. Table 3 shows the communication complexity of our subset-sharing primitives.

Protocol $\Pi_{\mathsf{SH}_{3PC}}(x, P_0) \rightarrow [\![x]\!]$

Preprocessing:

1. P_0 samples λ_x^1 with P_1 using Π_{SRNG}.
2. P_0 computes $\lambda_x^2 = -\lambda_x^1 - x$ and sends λ_x^2 to P_2.

Online:

$\mathcal{P}_{1,2}$ set their input-dependent shares to 0.

Figure 1: 3PC Subset-Sharing by $\mathcal{P}_{\lambda_x}$

Protocol $\Pi_{\mathsf{SH}_{3PC}}(x, \mathcal{P}_{1,2}) \rightarrow [\![x]\!]$

Preprocessing:

All parties set their input-independent shares to 0.

Online:

$\mathcal{P}_{1,2}$ set their input-dependent shares to x.

Figure 2: 3PC Subset-Sharing by $\mathcal{P}_{\mathsf{m}_x}$

Protocol $\Pi_{\mathsf{SH}_{4PC}}(x, \mathcal{P}_{0,3}) \rightarrow [\![x]\!]$

Preprocessing:

1. $\mathcal{P}_{0,3}$ samples λ_x^1 with P_1 using Π_{SRNG}.
2. $\mathcal{P}_{0,3}$ computes $\lambda_x^2 = -\lambda_x^1 - x$ and verify-sends λ_x^2 to P_2.
3. $\mathcal{P}_{1,2,3}$ set λ_x^* to 0.

Online:

P_0 sets m_x^* to x while $P_{1,2}$ set m_x to 0.

Figure 3: 4PC Subset-Sharing by $\mathcal{P}_{\lambda_x}$

Protocol $\Pi_{\mathsf{SH}_{4PC}}(x, \mathcal{P}_{1,2}) \rightarrow [\![x]\!]$

Preprocessing:

1. $P_{1,2,3}$ sample λ_x^* using Π_{SRNG}.
2. The parties set all remaining input-independent shares to 0.

Online:

1. $\mathcal{P}_{1,2}$ set their input-dependent share $\mathsf{m}_x = x$.
2. $\mathcal{P}_{1,2}$ verify-send $\mathsf{m}_x^* = x + \lambda_x^*$ to P_0 as part of constant-round communication.

Figure 4: 4PC Subset-Sharing by $\mathcal{P}_{\mathsf{m}_x}$

Table 3: Communication complexity of Subset-Sharing primitives

Setting	Subset holding x	PRE	ON	Rounds
3PC	$\mathcal{P}_{\lambda_x} = P_0$	ℓ	0	0
	$\mathcal{P}_{\mathsf{m}_x} = \mathcal{P}_{1,2}$	0	0	0
4PC	$\mathcal{P}_{\lambda_x} = P_{0,3}$	ℓ	0	0
	$\mathcal{P}_{\mathsf{m}_x} = \mathcal{P}_{1,2}$	0	ℓ	0

4.2 Stochastic Truncation

Stochastic truncation is the most widely used truncation approach in the MPC literature. It was first introduced by SecureML [22], later picked up by ABY3 [21], and has since been used in many state-of-the-art MPC frameworks [9, 13, 29, 30]. Stochastic truncation primitives based on these works assume that $\mathcal{P}$ can create an additive sharing $[x] = x^1 + x^2$ from their established sharing $\langle x \rangle$ such that a subset of parties holds x^1 and a subset of parties holds x^2. The sets of parties then locally compute $\left(x^1\right)^t$ and $\left(x^2\right)^t$ respectively. $\mathcal{P}$ can then re-share and add the truncated values to obtain $\langle (x)^{st} \rangle = \langle \left(x^1\right)^t \rangle + \langle \left(x^2\right)^t \rangle$. Figure 5 shows a general procedure for probabilistic truncation. This approach of stochastic truncation is referred to as $TS_{\{L\}}$ in this work.

Protocol $\Pi_{TS_{\{L\}}}(\langle x \rangle) \rightarrow \langle (x)^{st} \rangle$

1. Create a $\binom{2}{2}$ $[\cdot]$-sharing of $\langle x \rangle$ denoted by $[x] = x^1 + x^2$.
2. Compute $\left(x^1\right)^t$ and $\left(x^2\right)^t$ using local truncation and create a $\langle \cdot \rangle$-sharing of the two values.
3. Output $\langle \left(x^1\right)^t \rangle + \langle \left(x^2\right)^t \rangle$.

Figure 5: $TS_{\{L\}}$: Stochastic Truncation requiring a large slack [22]

$TS_{\{L\}}$ suffers from a small one-off error (e_0) and with a certain probability a large error (e_1) that causes truncation failure. The one-off error e_0 causes the truncated value to be one bit larger or smaller than the corresponding truncated plaintext value. The large error e_1 causes the truncated value to differ significantly from the expected value. To illustrate the impact of e_1, we borrow an example from [31]:

$$\begin{aligned}
&[\![x]\!] = 0100\ 1011, \ell = 8, t = 4\\
&\lambda_x = 1110\ 0000\\
&\mathsf{m}_x = x + \lambda_x \bmod 2^8 = 0010\ 1011,\\
&([\![x]\!])^{st}\\
&= (\mathsf{m}_x)^t \bmod 2^8 - (\lambda_x)^t \bmod 2^8) \bmod 2^8\\
&= (0000\ 0010 - 0000\ 1110) \bmod 2^8 = 1111\ 0100
\end{aligned}$$

The actual result of probabilistic truncation in this example is 1111 0100 while truncating plaintext x results in 0000 0100.

This error can be quite devastating to the accuracy of ML applications. The e_1 error is sometimes referred to as a wrap-around error as a carry bit is falsely propagated through the truncated values hence causing the large error. The closer the actual value x is to the ring modulus ℓ, the higher the probability that $TS_{\{L\}}$ truncation produces truncation failure due to a falsely propagated carry bit. More precisely assuming two's complement representation and $x \in [0, 2^{\ell_x}) \bigcup (2^\ell - 2^{\ell_x}, 2^\ell)$ in $\mathbb{Z}_{2^\ell}$, the probability of truncation failure as analyzed by [31] is given by:

$$P = \frac{1}{2^{\ell - \ell_x - 1}}$$

For this purpose, state-of-the-art frameworks utilize a "slack". The slack increases the utilized ring size ℓ in order to reduce the probability of this type of error. As a result, an application might have to use a ring size of $\mathbb{Z}_{2^{64}}$ in settings where all inputs fit within the ring $\mathbb{Z}_{2^{32}}$ without overflow.

Stochastic Truncation in Trio and Quad Figure 6 shows our construction of $TS_{\{L\}}$ in Trio and Quad. Observe that the parties in Trio can locally create an additive secret sharing $[x]$ using their established sharing as $\mathcal{P}_{1,2}$ can locally compute $\mathsf{m}_x = \mathsf{m}_{x,1} + \lambda_x^2 = \mathsf{m}_{x,2} + \lambda_x^1$ and P_0 can compute $-\lambda_x = -(\lambda_x^1 + \lambda_x^2)$. Similarly, the parties in Quad can locally create an additive sharing as $\mathcal{P}_{1,2}$ hold $\mathsf{m}_x = x + \lambda_x$ and $\mathcal{P}_{0,3}$ hold λ_x. The parties can use this insight to locally truncate their shares of x and re-share the truncated values using the subset-sharing primitives.

Protocol $\Pi_{\mathsf{TS}_{\{L\}}}$ $([\![x]\!] \to [\![(x)^{st}]\!]$

1. $\mathcal{P}_{\lambda_x}$ locally computes $(x^1)^t = (-\lambda_x)^t$ followed by $\Pi_{\mathsf{SH}}(\mathcal{P}_{\lambda_x}, (x^1)^t)$.
2. $\mathcal{P}_{\mathsf{m}_x}$ locally computes $(x^2)^t = (\mathsf{m}_x)^t$ followed by $\Pi_{\mathsf{SH}}(\mathcal{P}_{\mathsf{m}_x}, (x^2)^t)$.
3. Output $[\![(x^1)^t]\!] + [\![(x^2)^t]\!]$.

Figure 6: $TS_{\{L\}}$ in Trio and Quad

4.3 Stochastic Truncation with Reduced Slack

Stochastic truncation with reduced slack (c.f. Figure 7) was first proposed by Dalskov et al. [6]. This type of stochastic truncation also introduces an e_0 error but no e_1 error. However, this truncation scheme only guarantees correct results if the most significant bit of x is 0. Escudero et al. [8] overcame this limitation by adding $2^{\ell-1}$ prior to truncation and subtracting $2^{\ell-t-1}$ after truncation. This trick ensures that the most significant bit of x is 0 but introduces a slack of 1 bit to ensure correctness. We refer to this type of truncation as $TS_{\{1\}}$.

Protocol $\Pi_{\mathsf{TS}_{\{1\}}}$ $(\langle x\rangle) \to \langle (x)^{st}\rangle$

1. Add $2^{\ell-1}$ to $\langle x\rangle$ to ensure $\mathsf{MSB}(x) = 0$.
2. Generate ℓ random shared bits $\langle r_i\rangle^B$ and compute $\langle r\rangle^A \leftarrow \sum_i \langle r_i\rangle \cdot 2^i$.
3. Open $c \leftarrow \langle x\rangle + \langle r\rangle$ and compute $c' \leftarrow ((c)^t) \bmod 2^{\ell-t-1}$.
4. Compute $\langle b\rangle \leftarrow \langle r_{\ell-1}\rangle \oplus \mathsf{MSB(c)}$.
5. Compute $\langle y\rangle = c' - \sum_{i=t}^{\ell-2} \langle r_i\rangle \cdot 2^{i-t} + \langle b\rangle \cdot 2^{\ell-t-1}$
6. Output $\langle y\rangle - 2^{\ell-t-1}$.

Figure 7: $TS_{\{1\}}$: Stochastic Truncation requiring a slack of 1 bit [6]

The intuition of $TS_{\{1\}}$ is that parties generate a new mask for x using $\langle r\rangle$ and replace the old mask of x with the new mask r by opening $\langle x\rangle + \langle r\rangle$ in step 2. By calculating $c' \leftarrow ((c)^t) \bmod 2^{\ell-t-1}$, the parties truncate the value c to the desired number of fractional bits t but discard the t most significant bits of c. Note that these most significant bits could be affected by truncation failure when relying on $TS_{\{L\}}$. Steps 3-4 exploit that parties also hold a bit decomposition of mask $\langle r\rangle$ in order to recover the t most significant bits of c' in a deterministic way. While Dalskov et al. [6] and Fantastic Four [7] provide tailor-made constructions for the 3PC and 4PC settings respectively, we show how to construct the slack-free truncation protocol in Trio and Quad with three times lower round complexity and up to four times lower online complexity by utilizing the subset-sharing primitives.

Stochastic Truncation with Reduced Slack in Trio and Quad Figure 8 shows our construction of $TS_{\{1\}}$ in Trio and Quad. To implement $TS_{\{1\}}$, we exploit that $\mathcal{P}_{\mathsf{m}_x}$ can locally define $c = \mathsf{m}_x$ while $\mathcal{P}_{\lambda_x}$ can locally define $r = \lambda_x$. This observation allows parties to skip all communication-related operations in steps 1 and 2 of Figure 7 including generating doubly authenticated bits $[\![r]\!]$ and even opening the value c. Additionally, parties can locally precompute some operations such as $r' = \sum_{i=t}^{\ell-2} \lambda_i \cdot 2^{i-t}$ and $c' = (\mathsf{m}_x)^t \bmod 2^{\ell-t-1}$ to avoid computing these expressions jointly. Finally, parties use our efficient subset-sharing primitives to re-share the locally modified shares and compute the final result analogous to the original protocol. The XOR operation in step 4 can be evaluated in the arithmetic domain by computing $a \oplus b = a + b - 2ab$.

Protocol $\Pi_{\mathsf{TS-1}}(\llbracket x \rrbracket) \rightarrow (\llbracket x \rrbracket)^{st}$

1. Add $2^{\ell-1}$ to $\llbracket x \rrbracket$ to ensure $\mathsf{MSB}(x) = 0$.
2. $\mathcal{P}_{\lambda_x}$ locally compute $r' = \sum_{i=t}^{\ell-2} \lambda_{x,i} \cdot 2^{i-t}$ and $r_{\ell-1} = \mathsf{MSB}(\lambda_x)$.
3. $\mathcal{P}_{\mathsf{m}_x}$ locally compute $c' = (\mathsf{m}_x)^t \bmod 2^{\ell-t-1}$ and $\mathsf{MSB}(c) = \mathsf{MSB}(\mathsf{m}_x)$.
4. Create $\llbracket \cdot \rrbracket^A$-sharings $\llbracket r' \rrbracket, \llbracket r_{\ell-1} \rrbracket, \llbracket c' \rrbracket, \llbracket MSB(c) \rrbracket$ using Π_{SH}.
5. Compute $\llbracket b \rrbracket^A = \llbracket r_{\ell-1} \rrbracket^A \oplus \llbracket MSB(c) \rrbracket^A$.
6. Compute $\llbracket y \rrbracket = \llbracket c' \rrbracket - \llbracket r' \rrbracket + \llbracket b \rrbracket \cdot 2^{\ell-t-1}$.
7. Output $\llbracket y \rrbracket - 2^{\ell-t-1}$.

Figure 8: $TS_{\{1\}}$ in Trio and Quad

4.4 Exact Truncation

Exact Truncation computes $(x)^t$ deterministically without causing an e_0 or e_1 error. Figure 9 shows a general procedure for exact truncation without requiring any slack which we refer to as $TE_{\{0\}}$. The protocol first converts the arithmetic sharing $\langle x \rangle^A$ to a boolean sharing $\langle x \rangle^B$ using arithmetic-to-binary conversion (Π_{A2B}). In the boolean domain, parties can locally perform an arithmetic right shift by shifting all bits of their shares t positions to the right and setting all t vacated bits to the value of the original sign bit. The parties then convert the boolean sharing of $(x)^t$ back to an arithmetic sharing using binary-to-arithmetic conversion (Π_{B2A}). Protocols Π_{A2B} and Π_{B2A} require evaluating a boolean addition circuit which can be implemented using one of the variants described in §2.

Protocol $\Pi_{\mathsf{TE}_{\{0\}}}$ $(\langle x \rangle) \rightarrow \langle (x)^t \rangle$

1. Use Π_{A2B} to convert $\langle x \rangle^A$ to $\langle x \rangle^B$.
2. Compute $\langle x' \rangle = \langle x \rangle^t$ using an arithmetic right shift of $\langle x \rangle^B$ by t bits using only local bit assignments.
3. Output $\Pi_{\mathsf{B2A}}(\langle x' \rangle^B)$.

Figure 9: $TE_{\{0\}}$: Exact Truncation without requiring any slack

Fantastic Four [7] proposed a more efficient exact truncation scheme based on additive sharing that replaced the need for full-bit adders with bit extraction circuits. These can be implemented with the same number of rounds but often fewer gates. The resulting approach requires a slack of 1 bit and is referred to as $TE_{\{1\}}$ in this work. For $f(x) = x - (x \bmod 2^\ell)$, they compute the truncation of $[x]$ with $\mathsf{MSB}(x) = 0$ as follows:

$$(x)^t = \sum_{i=0}^{n-1} x^i/2^t + \left(\sum_i (x^i \bmod 2^t)\right)/2^t - f\left(\sum_i x^i\right)/2^t$$

The intuition behind this formula is that the first term calculates the truncation of each share individually similar to $TS_{\{L\}}$. The second term corrects the one-off error (e_0) while the third term corrects the wrap-around error (e_1) introduced by the first term. The first term in the sum can be computed by each party locally dividing each share, while the remaining terms can be computed interactively in the boolean domain using binary adders. The authors also noticed that the second term, $\sum_i (x^i \bmod 2^m)/2^m$ is always smaller than n, where n is the number of unique shares held by the parties. Hence, a bit extraction circuit computing the carry bits at positions $[t, t+\log_n -1]$ suffices to obtain the result. Similarly, the third term can only contain non-zero bits at the $log(n)$ most significant bits which requires computing another bit extraction circuit on $log(n)+\ell$ bits. Note that extracting the carry bits of t-bit terms is less expensive than extracting the carry bits of ℓ-bit terms, thus computing the last term to correct e_1 accounts for most of the amortized communication complexity.

The downside of Fantastic Four's exact truncation primitive is that Fantastic Four protocol requires four shares per party. Thus, the bit extraction circuits need to be evaluated in a tree-based fashion with three adders and two levels of adders in total. Evaluating the expression based on a $\binom{2}{2}$ additive sharing on the other hand would enable the parties to not only reduce the number of bits to extract from four to two but also to only require a single adder per term.

Exact Truncation in Trio and Quad Fortunately, creating a $\binom{2}{2}$ additive sharing in Trio and Quad is straightforward as parties can create sharings of m_x and $-\lambda_x$ with little communication overhead using subset sharing as exploited by the previous protocols as well. Figure 10 shows the deterministic truncation protocol based on the sharing semantics of Trio and Quad. Steps 2-4 create the necessary sharings to compute all sums in the truncation formula. Steps 5-6 calculate the potential non-zero bits in the boolean domain. Finally, step 7 converts the carry bits to the arithmetic domain, and step 8 aggregates the terms to obtain the final result. Note that combining the correction of the one-off error from Figure 10 with the more efficient correction of the wrap-around error by $TS_{\{1\}}$ (c.f. Figure 8) could potentially introduce further reductions in communication complexity. We leave the investigation of joining these two approaches for future work.

Protocol $\Pi_{\mathsf{TE}_{\{1\}}}$ $(\llbracket x \rrbracket) \rightarrow \llbracket (x)^t \rrbracket$

1. Add $2^{\ell-1}$ to $\llbracket x \rrbracket$ to ensure $\mathsf{MSB}(x) = 0$.
2. $\mathcal{P}_{\mathsf{m}_x}$ create $\llbracket \mathsf{m}_x/2^t \rrbracket$, while $\mathcal{P}_{\lambda_x}$ create $\llbracket -\lambda_x/2^t \rrbracket$ using Π_{SH}. Parties compute $\llbracket x/2^t \rrbracket = \llbracket \mathsf{m}_x/2^t \rrbracket + \llbracket -\lambda_x/2^t \rrbracket$.
3. $\mathcal{P}_{\mathsf{m}_x}$ create $\llbracket \mathsf{m}_x \bmod 2^t \rrbracket^B$, while $\mathcal{P}_{\lambda_x}$ create $\llbracket -\lambda_x \bmod 2^t \rrbracket^B$ using Π_{SH}.
4. $\mathcal{P}_{\mathsf{m}_x}$ create $\llbracket \mathsf{m}_x \rrbracket^B$, while $\mathcal{P}_{\lambda_x}$ create $\llbracket -\lambda_x \rrbracket^B$ using Π_{SH}.
5. Calculate the carry bit $\llbracket b_1 \rrbracket^B$ for bit position $t+1$ of $\llbracket \mathsf{m}_x \bmod 2^t \rrbracket^B + \llbracket -\lambda_x \bmod 2^t \rrbracket^B$ using a t-bit carry adder.
6. Calculate the carry bit for bit position $\ell+1$ of $\llbracket b_2 \rrbracket^B$ of $\llbracket \mathsf{m}_x \rrbracket^B + \llbracket -\lambda_x \rrbracket^B$ using an ℓ-bit carry adder.
7. Convert $\llbracket b_1 \rrbracket^B$ and $\llbracket b_2 \rrbracket^B$ to $\llbracket b_1 \rrbracket^A$ and $\llbracket b_2 \rrbracket^A$ using Π_{Bit2A}.
8. Compute $\llbracket y \rrbracket = \llbracket x/2^t \rrbracket + \llbracket b_1 \rrbracket^A - \llbracket b_2 \rrbracket^A \cdot 2^{\ell-t}$.
9. Output $\llbracket y \rrbracket - 2^{\ell-t-1}$.

Figure 10: $TE_{\{1\}}$: Exact Truncation requiring a 1-bit slack in Trio and Quad

5 Applying Truncation in PPML

In this section, we propose how to efficiently integrate the different truncation approaches into the PPML inference of neural network layers. We observe that the properties of several layers allow us to reduce the slack size required by a truncation scheme as well as reduce its communication complexity.

Linear Layers and Batch Normalization Linear layers such as fully connected layers and convolutional layers require matrix multiplication of fixed point shares. Thus, each output share needs to be truncated. Batch Normalization computes $y(x) = \frac{x-\mu}{\sqrt{\sigma^2+\varepsilon}} \cdot \gamma + \beta$ where the parameters $\mu, \sigma, \gamma, \beta$ are model parameters obtained during training, and ε is a small public constant to avoid division by zero. Thus, during inference, the party holding the model parameters locally computes $\hat{\sigma} = \gamma \cdot \frac{1}{\sqrt{\sigma^2+\varepsilon}}$ and shares it along with μ and β among the parties. Using these shares, the parties can compute the layer with a single fixed point multiplication. When using $TS_{\{L\}}$, we exploit that truncation can be integrated into the multiplication protocols of Trio and Quad at no additional communication costs [9]. The formal protocol is described in the authors' work.

As Batch Normalization typically appears directly after a linear layer, $TS_{\{1\}}$ of the linear layer can be fused with the multiplication in Batch Normalization using multi-input multiplication gates [24] to reduce the $TS_{\{1\}}$ overhead in round complexity to 0. These can be further optimized to multi-input scalar products [2] to also reduce the overhead in online communication to 0. To do so, observe that step 3 in $\Pi_{\mathsf{TS}-1}$ requires an XOR operation of the two shares $[\![m]\!]^A = [\![r_{l-1}]\!]$ and $[\![n]\!]^A = [\![\mathsf{MSB}(c)]\!]$ followed by a multiplication with public value $k = 2^{\ell-t-1}$. The results need to be added to $[\![o]\!] = [\![c']\!] - [\![r']\!] - 2^{\ell-t-1} - [\![\mu]\!]$ to obtain the first factor to compute the batch normalization. Hence, the parties wish to compute the following expression to obtain the layer output y of batch normalization which can be expressed as a single scalar product consisting of one two-input multiplication and one three-input multiplication in a single round of communication as follows:

$$\begin{aligned}
[\![y]\!] &= [\![\hat{\sigma}]\!] \cdot ([\![o]\!] + ([\![m]\!] \oplus [\![n]\!])k + \beta \\
&= [\![\hat{\sigma}]\!] \cdot [\![o]\!] + [\![\hat{\sigma}]\!] \cdot ([\![km]\!] + [\![kn]\!] - 2[\![km]\!] \cdot [\![kn]\!]) + \beta \\
&= [\![\hat{\sigma}]\!]([\![o]\!] + [\![km]\!] + [\![kn]\!]) + [\![-2\hat{\sigma}]\!] \cdot [\![km]\!] \cdot [\![kn]\!] + \beta
\end{aligned}$$

Activation Functions Normalization layers or linear layers are typically followed by an activation function. The most frequently used activation function in convolutional neural networks is ReLU. The ReLU operation is defined as $\mathsf{ReLU}(x) = \max(x, 0)$. To perform a ReLU operation, parties convert $\langle x \rangle^A$ to $\langle x \rangle^B$, evaluate a sign bit extraction circuit, and negate the result to obtain $\mathsf{DReLU}(x) = \langle \neg\mathsf{MSB}(x) \rangle^B$. $\mathsf{ReLU}(x)$ can then be computed as $\langle \mathsf{DReLU}(x) \rangle^B \cdot \langle x \rangle^A$ using Bit Injection [21].

Protocol $\Pi_{\mathsf{ReLU+TE}_{\{0\}}}(\langle x \rangle) \to \langle \mathsf{ReLU}((x)^t) \rangle$

1. Execute steps 2-5 of $\Pi_{\mathsf{TE}_{\{1\}}}$.
2. Execute step 6 of $\Pi_{\mathsf{TE}_{\{1\}}}$ but also extract $[\![\mathsf{MSB}(x)]\!]^B$ as part of the same circuit.
3. Execute step 7-8 of $\Pi_{\mathsf{TE}_{\{1\}}}$ and multiply the result with $[\![\mathsf{MSB}(x)]\!]^B$ using Π_{BitInj}.

Figure 11: ReLU with exact truncation ($TE_{\{0\}}$)

All truncation schemes can benefit from delaying the truncation of the layer prior to the ReLU operation, both in terms of reduced slack and communication overhead. The slack-related benefit from delaying truncation until the next ReLU layer is that after an activation all negative values are guaranteed to be 0. As truncating 0 has a negligible probability of truncation failure, this optimization significantly reduces the number of truncation failures in PPML where negative values are as common as positive values. Additionally, the $TE_{\{1\}}$ and $TS_{\{1\}}$ schemes can benefit from the ReLU operation as they do not need to respect their non-negativity constraint: In case of truncation failure, the ReLU operations set the output share to 0 provided that ReLU is calculated based on the untruncated share. Consequently, the parties can omit the addition and subtraction operations required by the $TE_{\{1\}}$ and $TS_{\{1\}}$ schemes which transform them into $TE_{\{0\}}$ and $TS_{\{0\}}$ schemes, respectively.

The performance-related benefit of delaying truncation applies to the $TS_{\{1\}}$, $TE_{\{1\}}$, and $TE_{\{0\}}$ schemes. Trivially, ReLU can be fused with little communication overhead with $TE_{\{0\}}$ as proposed by [12] by performing a full bit decomposition, applying truncation and ReLU in the boolean domain, and performing a full bit composition to obtain the result.

However, we observe that ReLU can also be merged with the more efficient $TE_{\{1\}}$ and $TS_{\{1\}}$ schemes. The conversion and bit extraction of ReLU can be fused without additional overhead into $\Pi_{\mathsf{TE}_{\{1\}}}$ by letting the carry adder in step 6 of the protocol (c.f. Figure 10) also compute the sign bit of $[\![x]\!]^B$. Hence, the only communication overhead of adding a ReLU operation to the truncation primitive is performing a bit injection which is typically similarly complex to performing a single multiplication in $\mathbb{Z}_{2^\ell}$. Figure 11 describes the protocol for fusing ReLU with $\Pi_{\mathsf{TE}_{\{1\}}}$. Note that $\Pi_{\mathsf{TE}_{\{1\}}}$ can also implement $TS_{\{1\}}$ by skipping all computations related to computing $[\![b_1]\!]$ which includes steps 3,5, and one Π_{Bit2A} operation (c.f. Figure 10). When optimizing for communication rounds, a generic way of fusing ReLU with $\Pi_{\mathsf{TS}_{\{1\}}}$ is to compute DReLU on the untruncated share $\langle x \rangle$ while computing $\langle (x)^t \rangle$ in parallel to the ReLU computation. The parties then bit inject the result of $\mathsf{DReLU}(\langle x \rangle)$ into the truncated share $\langle (x)^t \rangle$ to obtain $\mathsf{ReLU}(\langle (x)^t \rangle)$ without any overhead in round complexity.

We can design a more efficient protocol for Trio and Quad by fusing the bit injection performed during ReLU with the online communication required by $\Pi_{\mathsf{TS}_{\{1\}}}$ similar to our approach during batch normalization. An additional challenge here is that the output of ReLU prior to Bit Injection is given in the boolean domain and thus the process is more involved than simply utilizing multi-input scalar products. We observe that the online phases of bit injection and $\Pi_{\mathsf{TS}_{\{1\}}}$ can be merged at no additional communication cost or round complexity.

Fusing Truncation and Bit Injection Similar to our fused Batch Normalization approach we define $[\![m]\!]^A = [\![r_{l-1}]\!]$ and $[\![n]\!]^A = [\![\mathsf{MSB}(c)]\!]$, The shares need to be first XOR-ed, then multiplied with $k = 2^\ell - t - 1$, and finally added to $[\![o]\!] = [\![c']\!] - [\![r']\!] - 2^{\ell-t-1}$ to obtain the first factor to compute the fused bit injection. The second factor $[\![x]\!]^B$ is the negated most-significant bit of the untruncated $[\![x]\!]$ obtained during the ReLU operation. The output $[\![y]\!]$ of fusing truncation and bit injection is thus given by.

$$\begin{aligned}
[\![y]\!] &= [\![x]\!]^B \cdot ([\![o]\!]^A + ([\![m]\!]^A \oplus [\![n]\!]^A)k) \\
&= [\![x]\!]^B \cdot ([\![o]\!] + [\![km]\!] + [\![kn]\!] - 2[\![km]\!] \cdot [\![kn]\!]) \\
&= (\mathsf{m}_x + \lambda_x - 2\mathsf{m}_x\lambda_x) \cdot ([\![o]\!] + [\![km]\!] + [\![kn]\!] - 2[\![km]\!][\![kn]\!]) \\
&= (\mathsf{m}_x + \lambda_x - 2\mathsf{m}_x\lambda_x)(\mathsf{m}_o - \lambda_o + \mathsf{m}_{km} - \lambda_{km} + \mathsf{m}_{kn} - \lambda_{kn} \\
&\quad - 2((\mathsf{m}_{km} - \lambda_{km})(\mathsf{m}_{kn} - \lambda_{kn})) \\
&= (\mathsf{m}_x + \lambda_x - 2\mathsf{m}_x\lambda_x)(\mathsf{m}_o - \lambda_o + \mathsf{m}_{km} - \lambda_{km} + \mathsf{m}_{kn} - \lambda_{kn} \\
&\quad - 2(\mathsf{m}_{km}\mathsf{m}_{kn} - \mathsf{m}_{km}\lambda_{kn} - \mathsf{m}_{kn}\lambda_{km} + \lambda_{km}\lambda_{kn}))
\end{aligned}$$

When fully expanding the last equation the parties obtain different combinations of input-dependent and input-independent subterms, e.g., $\mathsf{m}_x \cdot \mathsf{m}_o$, $\lambda_x \cdot \lambda_o$, $\mathsf{m}_x \cdot \lambda_o$. Note that all input-dependent terms can be computed locally by $\mathcal{P}_{\mathsf{m}_x}$. To also evaluate the input-independent terms, they need to obtain additive shares of all relevant λ-terms from $\mathcal{P}_{\lambda_x}$ in the preprocessing phase, use these to calculate an additive sharing $[y]$ and reshare the result to all parties to obtain $[\![y]\!]$. The complete procedure is described in §A.

Pooling Layers Out of the pooling layers, only average pooling requires truncation. Computing an average in MPC requires a division operation, which is not natively supported in ring-based MPC. However, since the divisor d is public, we can approximate the division by multiplying the input $\langle x \rangle$ with the FPA representation of the reciprocal $r = 1/d$. We can exploit several slack-related optimizations to reduce the probability of truncation failure of that multiplication in average pooling, mainly by exploiting that d is a public value.

Following our proposed approach of computing an average naively would require to approximate r using t bits of precision followed by computing $\langle y \rangle = r \cdot \langle x \rangle$ with $2t$ fractional bits which leads to the same probability of truncation failure as the multiplication of two secret shares. However, we observe that common denominators in average pooling are powers of two with the most common denominator being 4 resulting from a kernel size of 2x2. For $d = 2^k$, the reciprocals can be expressed with k fractional bits without any loss of precision. For denominators that are not powers, we can exploit that the denominator in FPA is approximated using t fractional bits but not all of these bits are significant. For instance, $r = 1/9$ resulting from a kernel size of 3x3 is approximated as 00111000 for $t = 8$ but can be expressed as 111000 for $t = 6$ without any loss of precision. Finally, parties can also exploit that when a reciprocal is between two FPA approximations, choosing the one with lower precision reduces precision by less than 2^{-t} but still reduces the probability of truncation failure by a factor of 2. Hence, parties may additionally choose a threshold to decide when to use an approximation with fewer fractional bits that introduce loss of precision. Note that several networks such as VGG-16 or ResNet architectures use adaptive average pooling which dynamically determines the kernel size of the average pooling layer. These layers frequently create kernels of size 1x1 which results in no required truncation. Figure 12 describes the protocol for computing a division with a reduced probability of truncation failure that includes all described considerations.

Protocol $\Pi_{\mathsf{Division}}(\langle x \rangle, d, t, \mathsf{threshold}) \rightarrow \langle y \rangle \approx x/d$

1. Approximate $r_t \approx 1/d$ and $r_{t-1} \approx 1/d$ using t and $t-1$ fractional bits in FPA respectively.
2. Calculate (using floating points) $e_t = |1/d - r_t|$ and $e_{t-1} = |1/d - r_{t-1}|$.
3. While $e^t - e^{t-1} \leq \mathsf{threshold}$, decrement t. If $t = 0$, output $\langle x \rangle$.
4. Compute $\langle y \rangle = r_t \cdot \langle x \rangle$.
5. Output $(\langle y \rangle)^t$.

Figure 12: Division with reduced probability of truncation failure

Finally, note that average pooling typically follows after a ReLU layer. Hence, $TE_{\{1\}}$ and $TS_{\{1\}}$ can be implemented without the 1-bit slack requirement thus leading to $TE_{\{0\}}$ and $TS_{\{0\}}$ schemes, respectively. Given that we also achieved 0 bits of slack by delaying the truncation of the layer prior to ReLU, all common neural network architectures that do not use Batch Normalization such as AlexNet, LeNet5, and VGG-16 are completely evaluated without any slack using $TE_{\{1\}}$ and $TS_{\{1\}}$ schemes while layers that use Batch Normalization such as ResNet architectures achieve 0 bits of slack in more than half of all layers.

Mixed Truncation Given the introduced optimizations, we propose a mixed truncation strategy that combines the benefits of all truncation schemes depending on the PPML layer type. We observed that average pooling deals with small public reciprocals $r \leq 1/4$ that can additionally often be expressed with a smaller fixed point multiplicator without causing any loss of precision. Hence, this multiplication is an optimal candidate for $TS_{\{L\}}$. In neural network architectures, pooling is typically followed by a linear layer, in some cases followed by a normalization layer, and finally by a ReLU layer. If the linear layer is followed by a Batch Normalization layer, we can apply our proposed $TS_{\{1\}}$ optimization to merge the truncation of the linear layer with the multiplication in Batch Normalization without any overhead in round complexity or online complexity. The untruncated output of the layer prior to ReLU is then truncated within the Bit injection operation required by ReLU as described previously. Observe that the total overhead of the mixed truncation strategy to the forward pass is 0 in terms of round complexity and online communication complexity. Table 4 shows which truncation approach to apply in each layer to the input $[\![x]\!]$ and output $[\![y]\!]$ of the layer. Note that the described sequence of layers is used by all common CNN architectures such as VGG-16, ResNets, AlexNet, and LeNet.

Table 4: Mixed Truncation Strategy

Layer	trunc($[\![x]\!]$)	trunc($[\![y]\!]$)	Optimization
MaxPool	-	-	-
AvgPool	-	$TS_{\{L\}}$	Reduced t
Linear	-	Delay	-
BN	$TS_{\{1\}}$	Delay	Fuse $TS_{\{1\}}$ & BN
ReLU	$TS_{\{1\}}$	-	Fuse $TS_{\{1\}}$ & ReLU[a]

[a] Including slack-based optimization.

6 Evaluation

In this section, we evaluate our PPML-specific optimizations for different truncation schemes from both an accuracy and communication complexity perspective. Our evaluation is based on our implementation of all truncation schemes in the HPMPC framework [9] while we utilize PyTorch for plaintext training and inference.

Slack-related optimizations Our slack-related optimization introduced in §5 consists of exploiting non-negativity and reducing the number of fractional bits for the denominator during average pooling. Our concept of delaying truncation does not only reduce communication complexity by fusing the ReLU and truncation primitives but also reduces the number of truncation failures by exploiting the non-negativity guarantee by ReLU. Also layers following the activation function such as average pooling benefit that their output cannot be negative thus eliminating the one bit of slack required by $TS_{\{1\}}$ and $TE_{\{1\}}$. Our tweaks of reducing the number of fractional bits during average pooling similarly decrease the probability of overflows and truncation failures.

Table 5 shows that the accuracy of $TS_{\{1\}}$ and $TS_{\{Mix\}}$ improve significantly from these tweaks, allowing them to closely match the accuracy of their deterministic counterparts and achieve only little accuracy decay compared to plaintext inference.

Table 5: Truncation accuracy in % for VGG-16 on CIFAR-10 with bitlengh $\ell = 32$ and $t = 5$ fractional bits. Plaintext Accuracy: 81.74%.

	$\neg$OPT$^{\text{MSB}}$		OPT$^{\text{MSB}}$	
Scheme	$\neg$OPT$^{\text{AVG}}$	OPT$^{\text{AVG}}$	$\neg$OPT$^{\text{AVG}}$	OPT$^{\text{AVG}}$
$TS_{\{L\}}$	11.62	18.36	10.75	18.36
$TS_{\{1\}}$	51.47	81.05	66.31	81.05
$TE_{\{0\}}$	80.76	80.86	80.76	80.86
$TE_{\{1\}}$	80.76	80.86	80.76	80.86
$TS_{\{Mix\}}$	61.04	66.90	71.88	79.49

OPT^{MSB}: Exploiting non-negativity during pooling and ReLU layers. Includes delayed truncation (c.f. §5).
OPT^{AVG}: Slack-related optimizations to the denominator during average pooling (c.f. §5).

Plaintext-training-related optimizations Several techniques such as dropout, weight decay, and weight clipping can be used to reduce the magnitude of weights and therefore reduce the probability of overflows and truncation failures. While weight clipping and dropout should lead to universally better results in PPML as long as the plaintext accuracy remains high, weight decay might reduce accuracy by requiring more fractional bits to accurately represent the smaller weights. To study this tradeoff, we train multiple models on CIFAR-10 with the ADAMW classifier and set the weight decay hyperparameter to 0.03. Figure 15 shows the accuracy of the different truncation schemes on CIFAR-10 using ResNet50 with a bitlength of 32. The plots for several other architectures and bitlengths can be found in §D (c.f. Figure 19). We observe that it is indeed more challenging for the models to match plaintext accuracy with a low number of fractional bits as setting the fractional bits to 5 suffices for most truncation approaches to closely match plaintext accuracy without weight deacy but shows a noticeable decay in accuracy with weight decay. However, this decay is mitigated as the number of fractional bits increases. More importantly, the results show that the use of weight decay indeed reduces the number of truncation failures as $TS_{\{L\}}$ comes close to achieving plaintext accuracy with weight decay and 6 fractional bits while it does not come close to plaintext accuracy without weight decay for any number of fractional bits.

Performance-related optimizations Our various tweaks for different truncation schemes are based on delaying the truncation of the output of one layer to the next layer where we can fuse the truncation operation into another primitive. Table 6 shows the reduction in communication complexity of an entire forward pass for different truncation schemes based on

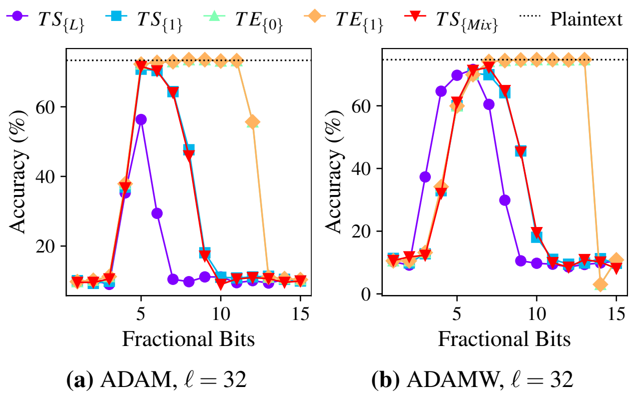


Figure 13: Accuracy of different truncation schemes with ResNet50 on CIFAR-10. Weight decay for ADAMW is set to 0.03.

VGG16 on ImageNet (preprocessing + online phase). Results for other models, datasets, and bitlengths are attached in §E.

The table shows that the reduction achieved in total communication is significant especially for the exact truncation schemes since fusing them with the ReLU operation eliminates most or all of the truncation-related which apparently can contribute to more than 25% of the total communication complexity of a forward pass. However stochastic truncation schemes also benefit from delaying truncation. An exception is $TS_{\{L\}}$ as the operation can already be fused into each multiplication operation at no additional cost in Trio and Quad. However, reducing the overhead of $TS_{\{L\}}$ when delaying truncation to 0 could be achieved by designing a similar fused bit injection scheme as for $TS_{\{1\}}$.

While we do not verify the round complexity empirically, note that the impact of our tweaks is likely even more significant as the exact truncation primitives are often multi-round protocols whereas linear layers, average pooling, and Batch Normalization can be evaluated in 0 to 1 communication rounds. In this context, even the reduction in communication complexity of $TS_{\{1\}}$ and $TS_{\{Mix\}}$ from 1 to 0 communication rounds when fusing truncation with other layers should be considered a significant improvement as it halves the round complexity of these layers when truncation is fused with other operations. The overall impact on the round complexity also depends on which boolean circuit is used during sign bit extraction which ranges from $\log_4(\ell)$ using multi-input scalar products and a parallel prefix adder to $\ell - 1$ round when using a ripple carry adder.

7 Truncation Schemes Compared

In this section, we show the results of our large-scale comparison of the different truncation approaches on various datasets. Analogously to §6 our implementation is based on HPMPC for secure inference and PyTorch for plaintext inference.

Table 6: Trio and Quad: Reduction in communication complexity of different truncation schemes for VGG16 on ImageNet when delaying truncation.

	$\ell = 32$			$\ell = 64$		
Scheme	¬ D	D	Δ	¬ D	D	Δ
			3PC			
$TS_{\{L\}}$	773.3	827.5	-6.55%	1554	1662	-6.52%
$TS_{\{1\}}$	1044	995.3	4.93%	2095	1992	5.17%
$TE_{\{0\}}$	1305	1039	25.62%	2627	2091	25.68%
$TE_{\{1\}}$	1404	1192	17.79%	2842	2412	17.80%
$TS_{\{Mix\}}$	1044	995.3	4.93%	2095	1992	5.17%
			4PC			
$TS_{\{L\}}$	1331	1440	-7.53%	2674	2891	-7.50%
$TS_{\{1\}}$	1819	1719	5.83%	3649	3441	6.05%
$TE_{\{0\}}$	2182	1702	28.19%	4392	3424	28.25%
$TE_{\{1\}}$	2421	2050	18.12%	4900	4149	18.11%
$TS_{\{Mix\}}$	1819	1719	5.83%	3649	3441	6.05%

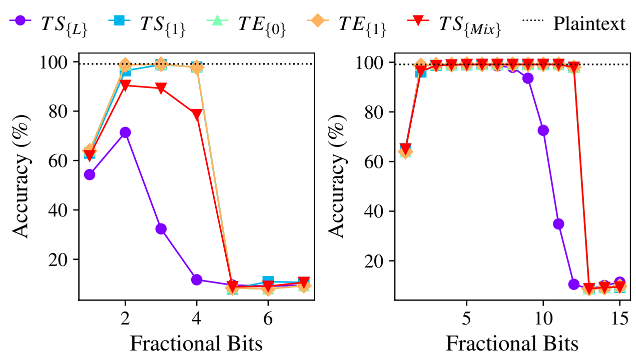
¬D: Total communication in MB if not delaying truncation.
D: Total communication in MB if delaying truncation.
Δ: Percentage reduction in communication complexity.

7.1 Accuracy Comparison

To compare the accuracy across truncation schemes we utilize all presented optimizations. We set the threshold for losing precision during average pooling (c.f. §5) to 0 to ensure that we only reduce the number of fractional bits if it does not affect approximation accuracy.

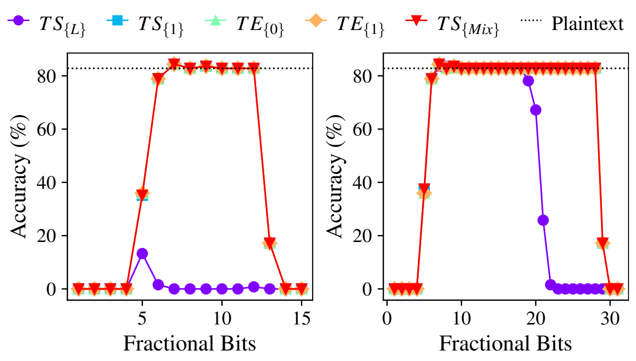
MNIST To evaluate the accuracy of MNIST, we train a plaintext PyTorch model based on the ADAM optimizer on the LeNet5 architecture but using only average pooling for pooling and only ReLU for activations to achieve an MPC-friendly architecture. The results in Figure 14 show that most truncation schemes already come close to the plaintext accuracy of more than 99% with a bitlength of 16 and 3 factional bits. Only $TS_{\{Mix\}}$ and $TS_{\{L\}}$ requires a bitlength of 32 to match plaintext accuracy.

CIFAR-10 Similarly, we evaluate the accuracy of CIFAR-10 by training a PyTorch model with the ADAM optimizer based on different architectures such as ResNet50, VGG16, and AlexNet while again replacing max pooling with average pooling. Figure 15 shows the accuracy of the different truncation schemes on ResNet50, while the plots for other architectures can be found in §D (c.f. Figure 18). We observe that all truncation schemes except $TS_{\{L\}}$ nearly match the plaintext accuracy with a bitlength of 32 and 5 fractional bits. $TS_{\{L\}}$ requires a bitlength of 64 to match the plaintext accuracy. With a bitlength of 16 none of the truncation schemes achieves more than 40% accuracy.



(a) $\ell = 16$ **(b)** $\ell = 32$

Figure 14: Accuracy of different truncation schemes with LeNet5 on MNIST.

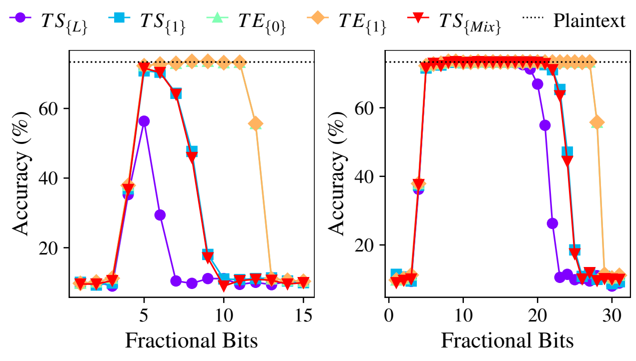


(a) $\ell = 32$ **(b)** $\ell = 64$

Figure 16: Accuracy of different truncation schemes with VGG16 on ImageNet

ImageNet ImageNet contains over one million images with 224x224x3 pixels per image so even plaintext training on the whole training dataset would take weeks. Hence, we take the pre-trained PyTorch models of VGG-16 and AlexNet that achieve over 80% and 60% plaintext accuracy respectively. Both models use maxpooling. We measure the accuracy based on 128 images from the validation dataset. Figure 16 shows the accuracy of the different truncation schemes with VGG16 on ImageNet. The plots for AlexNet can be found in §D (c.f. Figure 18). The results show that all truncation schemes except $TS_{\{L\}}$ can match the plaintext accuracy with a bitlength of 32 and 7 fractional bits. $TS_{\{L\}}$ requires a bitlength of 64 to match the plaintext accuracy. With a bitlength of 16 none of the truncation schemes achieves more than 20% accuracy.

Recommended fixed point ranges We conduct a large-scale evaluation of different truncation schemes, datasets, and model architectures. We obtain the fractional ranges that provide 0%, 1%, and 5% accuracy loss compared to plaintext training. The results are shown in §D (c.f. Table 10). When aggregating these results to give general recommendations based on dataset dimensions we arrive at the fractional ranges shown in Table 7. Each fractional range shown in the table provides high accuracy for most models on the given dataset, while values in bold indicate our recommended bitlength. The recommended bitlength and fractional range closely match the plaintext accuracy for most models and increasing it only provides little to no accuracy improvement for most models.



(a) $\ell = 32$ **(b)** $\ell = 64$

Figure 15: Accuracy of different truncation schemes with ResNet50 on CIFAR-10.

Table 7: Recommended fixed-point range for different truncation schemes.

Dataset	Scheme	Bitlength		
		$\ell = 16$	$\ell = 32$	$\ell = 64$
MNIST	$TS_{\{L\}}$	-	**5**	2-23
	$TS_{\{1\}}$	3	**6-11**	2
	$TE_{\{0\}}$	3-4	**5-11**	5
	$TE_{\{1\}}$	3-4	**5-11**	5
	$TS_{\{Mix\}}$	3-4	**5-11**	5
CIFAR-10	$TS_{\{L\}}$	-	6	**6-7**
	$TS_{\{1\}}$	-	**5-9**	8-12
	$TE_{\{0\}}$	-	**8-9**	8-12
	$TE_{\{1\}}$	-	**8-9**	8-12
	$TS_{\{Mix\}}$	-	**5-6**	8-12
ImageNet	$TS_{\{L\}}$	-	-	**8-18**
	$TS_{\{1\}}$	-	**10**	8-28
	$TE_{\{0\}}$	-	**8-12**	8-28
	$TE_{\{1\}}$	-	**8-12**	8-28
	$TS_{\{Mix\}}$	-	**10**	8-28

Additional results We also study the impact of truncating two factors before the multiplication compared to truncating their product after multiplication as proposed by Bicoptor 2.0 [31]. We find that truncating prior to multiplication can be expressed using the regular truncation approach without

Table 8: Runtime (s) for different truncation schemes in MAN: 1 Gbit/s bandwidth, 2 ms latency.

Setting	Scheme	CIFAR-10				ImageNet	
		ResNet50		VGG-16		VGG-16	
		32	64	32	64	32	64
3PC	$TS_{\{L\}}$	5.07 ± 0.17	8.12 ± 0.00	2.18 ± 0.00	4.20 ± 0.00	8.16 ± 0.02	19.28 ± 0.31
	$TS_{\{1\}}$	5.68 ± 0.05	9.40 ± 0.30	2.23 ± 0.13	5.26 ± 0.30	8.75 ± 0.02	20.81 ± 0.02
	$TE_{\{0\}}$	21.95 ± 0.79	54.01 ± 0.14	5.23 ± 0.13	13.96 ± 0.82	9.47 ± 0.02	22.40 ± 0.29
	$TE_{\{1\}}$	11.92 ± 0.15	31.89 ± 0.80	3.03 ± 0.09	12.42 ± 1.62	9.34 ± 0.09	21.48 ± 0.08
	$TS_{\{Mix\}}$	5.23 ± 0.38	9.12 ± 0.01	2.22 ± 0.07	4.95 ± 0.00	8.63 ± 0.13	20.14 ± 0.02
4PC	$TS_{\{L\}}$	5.50 ± 0.10	12.72 ± 0.33	2.77 ± 0.14	8.25 ± 0.01	17.21 ± 0.04	41.73 ± 0.30
	$TS_{\{1\}}$	6.36 ± 0.04	12.20 ± 0.92	3.01 ± 0.41	8.63 ± 0.00	17.69 ± 0.06	42.30 ± 0.01
	$TE_{\{0\}}$	26.38 ± 1.34	39.09 ± 0.87	5.30 ± 0.04	13.78 ± 0.54	18.62 ± 0.28	44.33 ± 0.04
	$TE_{\{1\}}$	15.65 ± 0.00	21.39 ± 0.03	3.27 ± 0.14	7.23 ± 2.18	18.24 ± 0.07	43.44 ± 0.01
	$TS_{\{Mix\}}$	7.69 ± 0.02	11.53 ± 0.30	2.85 ± 0.00	8.42 ± 0.00	17.52 ± 0.04	42.21 ± 0.02

any communication overhead. We refer the reader to §B for the full results. Additionally, we evaluate the impact of the common optimization [29] to replace MaxPooling with AveragePooling on the plaintext accuracy and find that given the significant reduction in communication complexity, it is a worthwhile tradeoff (c.f. §C). While we have already shown that PPML based on fixed point arithmetic can achieve identical accuracy to plaintext inference for large models and datasets, we aim to find an indicator of the limits of fixed point approximation. Based on the small accumulated fixed point error in the final layer of neural networks we conclude that fixed point arithmetic is sufficient for secure inference even for larger models than evaluated in this work (c.f. §C). Finally, we refer readers to §D for our extensive evaluation of utilizing different truncation schemes in practice.

7.2 Performance Comparison

Table 6 showed that as expected that $TS_{\{L\}}$ achieves the lowest communication complexity compared to the other approaches with the same bitlength. However, if we set the bitlength according to the accuracy achieved by different truncation schemes, we arrive at a different conclusion. As shown by table 10 in §D, $TS_{\{Mix\}}$ and $TS_{\{1\}}$ exactly match plaintext accuracy of all ImageNet models with $\ell = 32$ and 10 fractional bits while using $TS_{\{L\}}$ causes an accuracy loss of more than 5% compared to plaintext accuracy with a bitlength of 32 for all numbers of fractional bits. This accuracy loss is mitigated when increasing the bitlength to 64. However, at a bitlength of 64, $TS_{\{L\}}$ requires more communication than all other truncation schemes. Hence, we conclude that $TS_{\{Mix\}}$ and $TS_{\{1\}}$ are the most efficient truncation schemes when accounting for both accuracy and total communication complexity.

We compare the end-to-end inference runtime (preprocessing + online phase) of the different truncation schemes in three network settings: A LAN setting with 25 Gbit/s bandwidth and 0.3 ms latency, a MAN setting with 1 Gbit/s bandwidth and 2 ms latency, and a WAN network with 200 Mbit/s bandwidth and 40 ms latency. Table 8 shows the runtime in seconds for different models in the MAN setting, while the other settings can be found in §F (c.f. Table 13 and 14).

In line with the results on communication complexity, $TS_{\{L\}}$ achieves the lowest runtime compared to the other truncation schemes with the same bitlength but a lower runtime compared to the other stochastic truncation schemes when applying the recommended bitlength. The runtime results also demonstrate the advantage of $TE_{\{1\}}$ over $TE_{\{0\}}$ to parallelize the computation of boolean adders. Despite its high communication complexity, $TE_{\{1\}}$ achieves a lower runtime than $TE_{\{0\}}$ in MAN and WAN for all models and bitlengths. Note also that for the VGG-16 model on ImageNet, the relative difference in runtime between truncation schemes is lower as the MaxPooling layers present in the default architecture are responsible for the majority of the runtime.

7.3 Takeaways

We conclude that against common intuition, $TS_{\{L\}}$ is not the most efficient stochastic truncation scheme when accounting for both accuracy and runtime. Deterministic truncation schemes achieve similar accuracy as stochastic ones but can be more reliable for a larger number of fractional bits. $TE_{\{1\}}$ matches the accuracy of $TE_{\{0\}}$ while requiring significantly fewer communication rounds. Hence, practitioners should consider $TS_{\{Mix\}}$, $TS_{\{1\}}$, and $TE_{\{1\}}$ for efficient inference. For all these schemes using a bitlength of 32 and 10 fractional bits is a good starting point that closely matches plaintext accuracy for most models. For plaintext training, practitioners should consider replacing MaxPooling with AveragePooling and use weight clipping and weight decay. Finally, our result suggests that 64-bit fixed point arithmetic is future-proof for secure inference of large models and datasets.

References

[1] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 805–817, 2016.

[2] Andreas Brüggemann, Robin Hundt, Thomas Schneider, Ajith Suresh, and Hossein Yalame. Flute: fast and secure lookup table evaluations. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 515–533. IEEE, 2023.

[3] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY*, 13:143–202, 2000.

[4] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. Astra: high throughput 3pc over rings with application to secure prediction. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 81–92, 2019.

[5] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. In *NDSS*, 2020.

[6] Anders Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. *Proceedings on Privacy Enhancing Technologies*, 2020.

[7] Anders PK Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX Security Symposium*, pages 2183–2200, 2021.

[8] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for mpc over mixed arithmetic-binary circuits. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II 40*, pages 823–852. Springer, 2020.

[9] Christopher Harth-Kitzerow, Ajith Suresh, Yongqin Wang, Hossein Yalame, Georg Carle, and Murali Annavaram. High-throughput secure multiparty computation with an honest majority in various network settings. Cryptology ePrint Archive, Paper 2024/386, 2024.

[10] Christopher Harth-Kitzerow, Yongqin Wang, Rachit Rajat, Georg Carle, and Murali Annavaram. PIGEON: A framework for private inference of neural networks. Cryptology ePrint Archive, Paper 2024/1371, 2024.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[12] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. Orca: Fss-based secure training and inference with gpus. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 63–63. IEEE Computer Society, 2023.

[13] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 1575–1590, 2020.

[14] Marcel Keller, Peter Scholl, and Nigel P Smart. An architecture for practical actively secure mpc with dishonest majority. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 549–560, 2013.

[15] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. In *USENIX Security Symposium*, pages 2651–2668, 2021.

[16] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. Tetrad: Actively secure 4pc for secure training and inference. *arXiv preprint arXiv:2106.02850*, 2021.

[17] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, Toronto, Ontario, 2009.

[18] Yun Li, Yufei Duan, Zhicong Huang, Cheng Hong, Chao Zhang, and Yifan Song. Efficient {3PC} for binary circuits with application to {Maliciously-Secure}{DNN} inference. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5377–5394, 2023.

[19] Yehuda Lindell. Secure multiparty computation (mpc). *IACR Cryptol. ePrint Arch.*, 2020:300, 2020.

[20] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[21] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 35–52, 2018.

[22] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *IEEE Symposium on Security and Privacy*, pages 19–38, 2017.

[23] Lucien KL Ng and Sherman SM Chow. Sok: Cryptographic neural-network computation. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 497–514. IEEE, 2023.

[24] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. {ABY2. 0}: Improved {Mixed-Protocol} secure {Two-Party} computation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2165–2182, 2021.

[25] Arpita Patra and Ajith Suresh. Blaze: blazing fast privacy-preserving machine learning. *arXiv preprint arXiv:2005.09042*, 2020.

[26] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115:211–252, 2015.

[27] Manuel B Santos, Dimitris Mouris, Mehmet Ugurbil, Stanislaw Jarecki, José Reis, Shubho Sengupta, and Miguel de Vega. Curl: Private llms through wavelet-encoded look-up tables. *Cryptology ePrint Archive*, 2024.

[28] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[29] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. Cryptgpu: Fast privacy-preserving machine learning on the gpu. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038. IEEE, 2021.

[30] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A {GPU} platform for secure computation. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 827–844, 2022.

[31] Lijing Zhou, Qingrui Song, Su Zhang, Ziyu Wang, Xianggui Wang, and Yong Li. Bicoptor 2.0: Addressing challenges in probabilistic truncation for enhanced privacy-preserving machine learning. *arXiv preprint arXiv:2309.04909*, 2023.

A Merging Truncation and Bit Injection

When expanding the equation from §5, we obtain the equation below. Each $P_i \in \mathcal{P}_{\mathsf{m}_x}$ can compute an additive share of y by holding the input-dependent shares in the clear and obtaining $[\cdot]$-sharings of the relevant input-independent shares from $\mathcal{P}_{\lambda_x}$.

$$\begin{aligned}
&y = (\mathsf{m}_x + \lambda_x - 2\mathsf{m}_x\lambda_x)(\mathsf{m}_o - \lambda_o + \mathsf{m}_{km} - \lambda_{km} + \mathsf{m}_{kn} - \lambda_{kn} - 2(\mathsf{m}_{km}\mathsf{m}_{kn} - \mathsf{m}_{km}\lambda_{kn} - \mathsf{m}_{kn}\lambda_{km} + \lambda_{km}\lambda_{kn})) && (1)\\
&[y] = \mathsf{m}_x\mathsf{m}_o - \mathsf{m}_x[\lambda_o] + \mathsf{m}_x\mathsf{m}_{km} - \mathsf{m}_x[\lambda_{km}] + \mathsf{m}_x\mathsf{m}_{kn} - \mathsf{m}_x[\lambda_{kn}] && (2)\\
&- 2(\mathsf{m}_x\mathsf{m}_{km}\mathsf{m}_{kn} - \mathsf{m}_x\mathsf{m}_{km}[\lambda_{kn}] - \mathsf{m}_x\mathsf{m}_{kn}[\lambda_{km}] + \mathsf{m}_x[\lambda_{km}\lambda_{kn}]) && (3)\\
&+ \mathsf{m}_o[\lambda_x] - [\lambda_x\lambda_o] + \mathsf{m}_{km}[\lambda_x] - [\lambda_x\lambda_{km}] + \mathsf{m}_{kn}[\lambda_x] - [\lambda_x\lambda_{kn}] && (4)\\
&- 2(\mathsf{m}_{km}\mathsf{m}_{kn}[\lambda_x] - \mathsf{m}_{km}[\lambda_x\lambda_{kn}] - \mathsf{m}_{kn}[\lambda_x\lambda_{km}] + [\lambda_x\lambda_{km}\lambda_{kn}]) && (5)\\
&- 2(\mathsf{m}_x\mathsf{m}_o[\lambda_x] - \mathsf{m}_x[\lambda_x\lambda_o] + \mathsf{m}_x\mathsf{m}_{km}[\lambda_x] - \mathsf{m}_x[\lambda_x\lambda_{km}] + \mathsf{m}_x\mathsf{m}_{kn}[\lambda_x] - \mathsf{m}_x[\lambda_x\lambda_{kn}] && (6)\\
&+ 4(\mathsf{m}_x\mathsf{m}_{km}\mathsf{m}_{kn}[\lambda_x] - \mathsf{m}_x\mathsf{m}_{km}[\lambda_x\lambda_{kn}] - \mathsf{m}_x\mathsf{m}_{kn}[\lambda_x\lambda_{km}] + \mathsf{m}_x[\lambda_x\lambda_{km}\lambda_{kn}]) && (7)\\
&= [\lambda_x](\mathsf{m}_o + \mathsf{m}_{km} + \mathsf{m}_{kn} - 2\mathsf{m}_{km}\mathsf{m}_{kn} - 2\mathsf{m}_x\mathsf{m}_o - 2\mathsf{m}_x\mathsf{m}_{km} - 2\mathsf{m}_x\mathsf{m}_{kn} + 4\mathsf{m}_x\mathsf{m}_{km}\mathsf{m}_{kn}) && (8)\\
&+ [\lambda_o](-\mathsf{m}_x) + [\lambda_{km}](-\mathsf{m}_x + 2\mathsf{m}_x\mathsf{m}_{kn}) + [\lambda_{kn}](-\mathsf{m}_x + 2\mathsf{m}_x\mathsf{m}_{km}) + [\lambda_{km}\lambda_{kn}](-2\mathsf{m}_x) + [\lambda_x\lambda_o](-1 + 2\mathsf{m}_x) && (9)\\
&+ [\lambda_x\lambda_{km}](-1 + 2\mathsf{m}_{kn} + 2\mathsf{m}_x - 4\mathsf{m}_x\mathsf{m}_{kn}) + [\lambda_x\lambda_{kn}](-1 + 2\mathsf{m}_{km} + 2\mathsf{m}_x - 4\mathsf{m}_x\mathsf{m}_{km}) + [\lambda_x\lambda_{km}\lambda_{kn}](-2 + 4\mathsf{m}_x) && (10)\\
&+ \mathsf{m}_x(\mathsf{m}_o + \mathsf{m}_{km} + \mathsf{m}_{kn} - 2\mathsf{m}_{km}\mathsf{m}_{kn}) && (11)
\end{aligned}$$

Observe that $\mathcal{P}_{\mathsf{m}_x}$ require the following additive shares to completely evaluate the equation:

$$\begin{aligned}
&[\lambda_x], [\lambda_o], [\lambda_{km}], [\lambda_{kn}], [\lambda_{km}\cdot\lambda_{kn}],\\
&[\lambda_x\cdot\lambda_o], [\lambda_x\cdot\lambda_{km}], [\lambda_x\cdot\lambda_{kn}], [\lambda_x\cdot\lambda_{km}\cdot\lambda_{kn}]
\end{aligned}$$

Out of these terms $\mathcal{P}_{\mathsf{m}_x}$ already hold $[\lambda_o]$, $[\lambda_{km}]$ and $[\lambda_{kn}]$ but not any of the cross-terms and not $[\lambda_x]^A$ since they only initially hold $[\lambda_x]^B$ in $\mathbb{Z}_2$. Thus, there are six remaining input-independent cross-terms that $\mathcal{P}_{\lambda_x}$ need to compute locally and share with $\mathcal{P}_{\mathsf{m}_x}$. $\mathcal{P}_{\mathsf{m}_x}$ can then proceed to compute $[y]$. Each $P_i \in \mathcal{P}_{\mathsf{m}_x}$ then samples λ_y^i, computes $\mathsf{M}_i = y^i + \lambda_y^i$ and sends it to the other party $P_j \in \mathcal{P}_{\mathsf{m}_x}$. Finally the parties set $\mathsf{m}_y = \mathsf{M}_1 + \mathsf{M}_2$ and all parties hold consistent sharings of $[\![y]\!]$ according to the Trio sharing semantics.

While this approach accounts for all steps to construct a semi-honest 3PC protocol in Trio, the malicious 4PC protocol in Quad requires additional steps to verify correctness. Note that the sharing of input-independent terms can be trivially verified as both P_0 and P_3 can compute and verify-share the input-independent terms with P_1 and P_2. However, the online reconstruction of $[\![y]\!]$ needs to be secured against a malicious P_1 or P_2 who might send incorrect messages. Thus, $\mathcal{P}_{0,1,2}$ engage in a similar computation to compute $[\bar{y}]$ which is only equal to $[y]$ if both P_1 and P_2 honestly communicated their messages. To do so, $\mathcal{P}_{0,1,2}$ sets $\overline{\mathsf{m}}_x = \mathsf{m}_x^* + \lambda_x$ for each input share used in the long equation, while $\mathcal{P}_{1,2}$ set $\overline{\mathsf{m}}_x = \mathsf{m}_x + \lambda_x^*$. Note that the parties inherently hold $\left[\overline{\lambda}_x\right] = \lambda_x^* + \lambda_x$ where λ_x is held by P_0 and λ_x^* is held by $\mathcal{P}_{1,2}$. The parties proceed to evaluate the identical equation that computes $[y]$ but with these new shares to compute $[\overline{y}]$ while P_3 supplies the six input-dependent terms denoted by $\overline{\lambda}$.

After obtaining $[\overline{y}]$, $\mathcal{P}_{1,2}$ verify-send their masked share to P_0 such that it can obtain $\mathsf{m}_{\overline{y}}^* = \overline{y} + \overline{\lambda}_y$. Finally, $\mathcal{P}_{0,1,2}$ compare their views of $\mathsf{m}_y + \overline{\lambda}_y$ and $\overline{\mathsf{m}}_y + \lambda_y$ using Π_{CV} to verify the correctness of the computation. Note that only if both P_1 and P_2 honestly communicated M_1 and M_2 the verification will succeed. As $\mathsf{M}_{\{1,2\}}$ is computed non-interactively by both P_1 and P_2, the verification is secure against a malicious corruption by one of the two parties.

Note that the compare-views used are identical to the ones used by Quad's multiplication protocol [9] and the authors provide simulation-based security proofs for the protocol. Also, note that $\mathsf{M}_{\{1,2\}}$ is only used for verification and thus part of the constant-round communication [9]. Figure 17 summarizes the protocol for semi-honest Trio and malicious Quad. Finally, observe that we can combine some input-independent terms to further reduce the number of input-independent shares that need to be sent from $\mathcal{P}_{\lambda_x}$ to $\mathcal{P}_{\mathsf{m}_x}$ to five.

$$\begin{aligned}
[y] = {} & [\lambda_x]\,(\mathsf{m}_o + \mathsf{m}_{km} + \mathsf{m}_{kn} - 2\mathsf{m}_{km}\mathsf{m}_{kn} - 2\mathsf{m}_x\mathsf{m}_o \\
& - 2\mathsf{m}_x\mathsf{m}_{km} - 2\mathsf{m}_x\mathsf{m}_{kn} + 4\mathsf{m}_x\mathsf{m}_{km}\mathsf{m}_{kn}) \\
& + [\lambda_{km}]\,(-\mathsf{m}_x + 2\mathsf{m}_x\mathsf{m}_{kn}) + [\lambda_{kn}]\,(-\mathsf{m}_x + 2\mathsf{m}_x\mathsf{m}_{km}) \\
& + [\lambda_x\lambda_{km}]\,(-1 + 2\mathsf{m}_{kn} + 2\mathsf{m}_x - 4\mathsf{m}_x\mathsf{m}_{kn}) \\
& + [\lambda_x\lambda_{kn}]\,(-1 + 2\mathsf{m}_{km} + 2\mathsf{m}_x - 4\mathsf{m}_x\mathsf{m}_{km}) \\
& + [\boldsymbol{\lambda_o + 2\lambda_{km}\lambda_{kn}}](-\mathsf{m}_x) \\
& + [\boldsymbol{\lambda_x\lambda_o + 2\lambda_x\lambda_{km}\lambda_{kn}}](-1 + 2\mathsf{m}_x) \\
& + \mathsf{m}_x(\mathsf{m}_o + \mathsf{m}_{km} + \mathsf{m}_{kn} - 2\mathsf{m}_{km}\mathsf{m}_{kn})
\end{aligned}$$

The total communication complexity of the protocol is thus five elements in the preprocessing phase and two elements in the online phase for Trio and ten elements in the preprocessing phase and three elements in the online phase for Quad. Since Bit Injection requires the same online complexity and two resp. four elements of communication in the preprocessing phase, the total overhead of fusing truncation and Bit Injection is three preprocessing elements for Trio and six for Quad. This overhead is identical to the preprocessing costs of one three-input multiplication and thus we achieve exactly the same overhead as fusing Batch Normalization and stochastic truncation as shown in §5.

Protocol $\Pi_{\mathsf{BitInj+TS}_{\{1\}}}$ $(\llbracket x \rrbracket^B, \llbracket o \rrbracket, \llbracket m \rrbracket, \llbracket m \rrbracket, k) \rightarrow \llbracket y \rrbracket$

Preprocessing:

1. $\mathcal{P}_{\lambda_x}$ subset share $\lambda_x, \lambda_x \cdot \lambda_o, \lambda_{km} \cdot \lambda_{kn} \lambda_x \cdot \lambda_{km} \lambda_x \cdot \lambda_{kn} \lambda_x \cdot \lambda_{km} \cdot \lambda_{kn}$ with $\mathcal{P}_{\mathsf{m}_x}$.
2. P_3 shares $\overline{\lambda}_x, \overline{\lambda}_x \cdot \overline{\lambda}_o, \overline{\lambda}_{km} \cdot \overline{\lambda}_{kn}, \overline{\lambda}_x \cdot \overline{\lambda}_{km}, \overline{\lambda}_x \cdot \overline{\lambda}_{kn}, \overline{\lambda}_x \cdot \overline{\lambda}_{km} \cdot \overline{\lambda}_{kn}$ with $\mathcal{P}_{0,1,2}$.

Online:

1. $\mathcal{P}_{\mathsf{m}_x}$ locally compute $[y]$ using their input shares and preprocessing material provided by $\mathcal{P}_{\lambda_x}$.
2. $\mathcal{P}_{0,1,2}$ locally compute $[\overline{y}]$ using their input shares and preprocessing material provided by $\mathcal{P}_3$.
3. Each party $P_i \in \mathcal{P}_{\mathsf{m}_x}$ samples λ_y^i with $\mathcal{P}_{\lambda_x}$, computes $\mathsf{M}_i = y^i + \lambda_y^i$ and sends it to the other party $P_j \in \mathcal{P}_{\mathsf{m}_x}$. $\mathcal{P}_{\mathsf{m}_x}$ set $\mathsf{m}_y = \mathsf{M}_1 + \mathsf{M}_2$.
4. $\mathcal{P}_{1,2}$ sample $\overline{\lambda}_y$ with $\mathcal{P}_3$, compute $\mathsf{M}_{\{1,2\}} = [\overline{y}] + \overline{\lambda}_y$, and verify-send it to P_0. P_0 sets $\mathsf{m}_{\overline{y}}^* = [\overline{y}] + \mathsf{M}_{\{1,2\}}$.
5. $\mathcal{P}_{0,1,2}$ compare their views of $\mathsf{m}_y + \overline{\lambda}_y$ and $\overline{\mathsf{m}}_y + \lambda_y$ using Π_{CV}.
6. P_0 sets $\mathsf{m}_{\overline{y}}^* = \overline{\mathsf{m}}_y - \overline{\lambda}_y$ while all other parties set $\lambda_y^* = \overline{\lambda}_y$. All parties now hold a consistent sharing of $\llbracket y \rrbracket$.

Figure 17: Merged Bit Injection and Truncation in Quad. Steps 2 of the preprocessing phase and steps 2,4,5,6 of the online phase are omitted for semi-honest Trio.

B Truncation before Multiplication

Bicoptor 2.0 [31] proposed to utilize truncation before multiplication to reduce the probability of truncation failure. When multiplying $c = a \cdot b$, truncation can either be applied to a and b individually before the multiplication or to the result c after the multiplication. When truncating c after multiplication, t is set to k where k is the number of fractional bits used to represent a value. When parties instead truncate a and b prior to multiplication, t is set to $\frac{k}{2}$. For large absolute values of a and b, truncation before multiplication can significantly reduce the probability of truncation failure by producing intermediary products with k instead of $2k$ fractional bits. As truncation prior to multiplication requires two individual truncations of a and b instead of a single one of c, the communication overhead is increased by a factor of two. However, we find that truncating prior to multiplication can be implemented without any additional communication overhead compared to truncating c after the multiplication.

Our key observation to perform the optimization is that each plaintext value can be pre-truncated by $t = \frac{k}{2}$ without any communication overhead before entering the MPC protocol. As a result, all secret shares are already pre-truncated, and multiplying them produces shares with k fractional bits. From that point on, truncating prior-to-multiplication can be implemented as truncating after multiplication by half the number of fractional bits as truncating $\langle c \rangle$ after multiplication by $t = \frac{k}{2}$ produces a pre-truncated share $\langle (c)^t \rangle$ that can be used either as the next $\langle (a)^t \rangle$ or $\langle (b)^t \rangle$ for further multiplications. However, observe also that pre-truncating all plaintext values by $k/2$ bits is equivalent to representing all fixed point values with $k/2$ fractional bits to begin with. Hence, truncating prior to multiplication is actually no different from simply using half the number of fractional bits and utilizing the standard truncation after multiplication approach.

C Replacing MaxPooling with AveragePooling

A common optimization in MPC to reduce communication complexity is to replace max pooling with average pooling [29]. Computing the maximum of n values requires computing $n-1$ pairwise comparisons along a tree of height $\log_2(n)$. Each pairwise comparison requires on DReLU operation. Hence, maxpooling with common kernel sizes such as 3x3 can become the most expensive layer in PPML while average pooling only requires a single fixed-point truncation and is typically the cheapest layer in PPML. To evaluate whether this optimization affects the accuracy of ML models we train different ResNet models on CIFAR-10 with maxpooling and average pooling using various optimizers and compare the accuracy. The results are shown in table 9. On average, the models with maxpooling achieve 72.59% accuracy while the models with average pooling achieve 71.00% accuracy. Given the significant reduction in communication complexity, we replace average pooling with max pooling for all further evaluations except ImageNet-based models where we rely on the official pretrained PyTorch models that use max pooling.

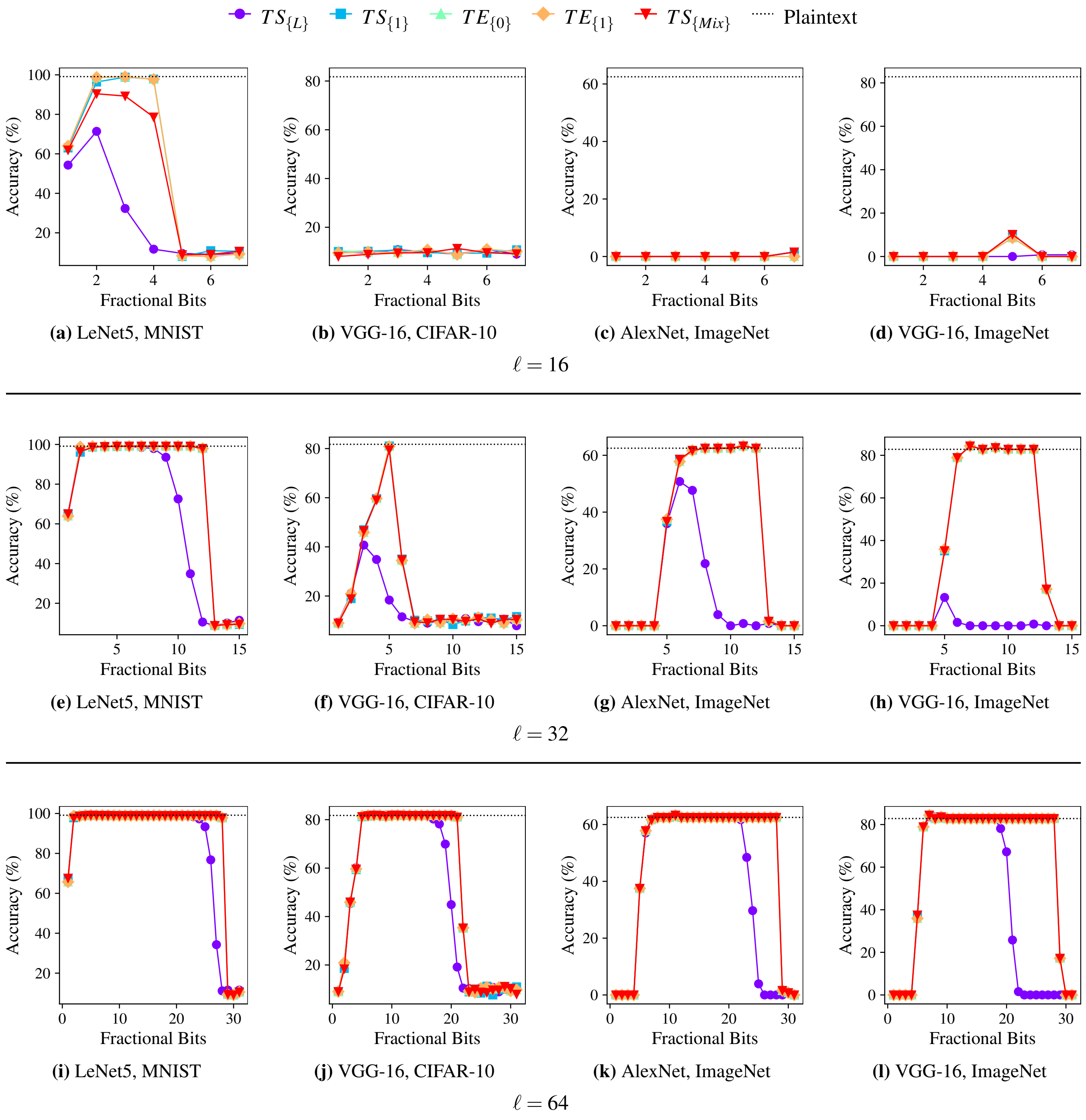
Table 9: Accuracy in % for ResNet models on CIFAR-10 with max pooling and average pooling.

Model	Optimizer	MaxPool	AvgPool
ResNet50	ADAM	77.14	72.04
	ADAMW	77.39	74.86
	SGD	66.41	63.30
	SGDW	73.38	72.95
ResNet101	ADAM	74.55	76.82
	ADAMW	76.00	75.22
	SGD	65.30	64.04
	SGDW	73.50	71.23
ResNet152	ADAM	76.33	75.31
	ADAMW	74.69	73.22
	SGD	62.77	61.45
	SGDW	73.62	71.58

[a] SGDW refers to SGD with 0.03 weight decay.

How far can FPA scale? FPA enables parties to use 16-bit, 32-bit, or 64-bit integer arithmetic that has low overhead on modern hardware. While ring sizes of more than 64-bit are possible, they introduce significant computational overhead. To obtain an indicator for how much precision is lost when utilizing 64-bit fixed point arithmetic in PPML, we investigate the outputs computed by the last layer of the VGG-16 model on ImageNet and calculate δ^f as the fixed point deviation of each fixed point value with respect to its plaintext floating point value. Note that by the last layer, all errors from previous layers' fixed-point calculations accumulate. We then calculate δ^c as the minimum difference of two final class predictions in floating point. Intuitively, the closer δ^f and δ^c are the higher the probability that the fixed point errors cause swapping the likelihood of two classes compared to floating point arithmetic. As only swapping the likelihood order of top predictions is relevant in practice we only consider the top 5 most likely predictions. Over multiple batches we observe that the minimum δ^c found for VGG-16 on ImageNet is 0.22 meaning that two classes in the top 5 predictions are at least separated by 0.22. The maximum δ^f for $TS_{\{1\}}$ when using a bitlength of 64 and 13 fractional bits is 0.012 meaning that the fixed point representation of a class value is at most 0.012 off from the final layer's floating point value in plaintext inference. Given this discrepancy, we can conclude that using 64-bit fixed point arithmetic in PPML can most likely be used for even larger models and datasets without risking misclassification due to the accumulation of fixed point errors.

D Additional Accuracy Evaluation



(a) LeNet5, MNIST **(b)** VGG-16, CIFAR-10 **(c)** AlexNet, ImageNet **(d)** VGG-16, ImageNet

$\ell = 16$

(e) LeNet5, MNIST **(f)** VGG-16, CIFAR-10 **(g)** AlexNet, ImageNet **(h)** VGG-16, ImageNet

$\ell = 32$

(i) LeNet5, MNIST **(j)** VGG-16, CIFAR-10 **(k)** AlexNet, ImageNet **(l)** VGG-16, ImageNet

$\ell = 64$

Figure 18: Accuracy of different truncation schemes on various models and datasets. Each row corresponds to a different Bitlength ℓ as indicated.

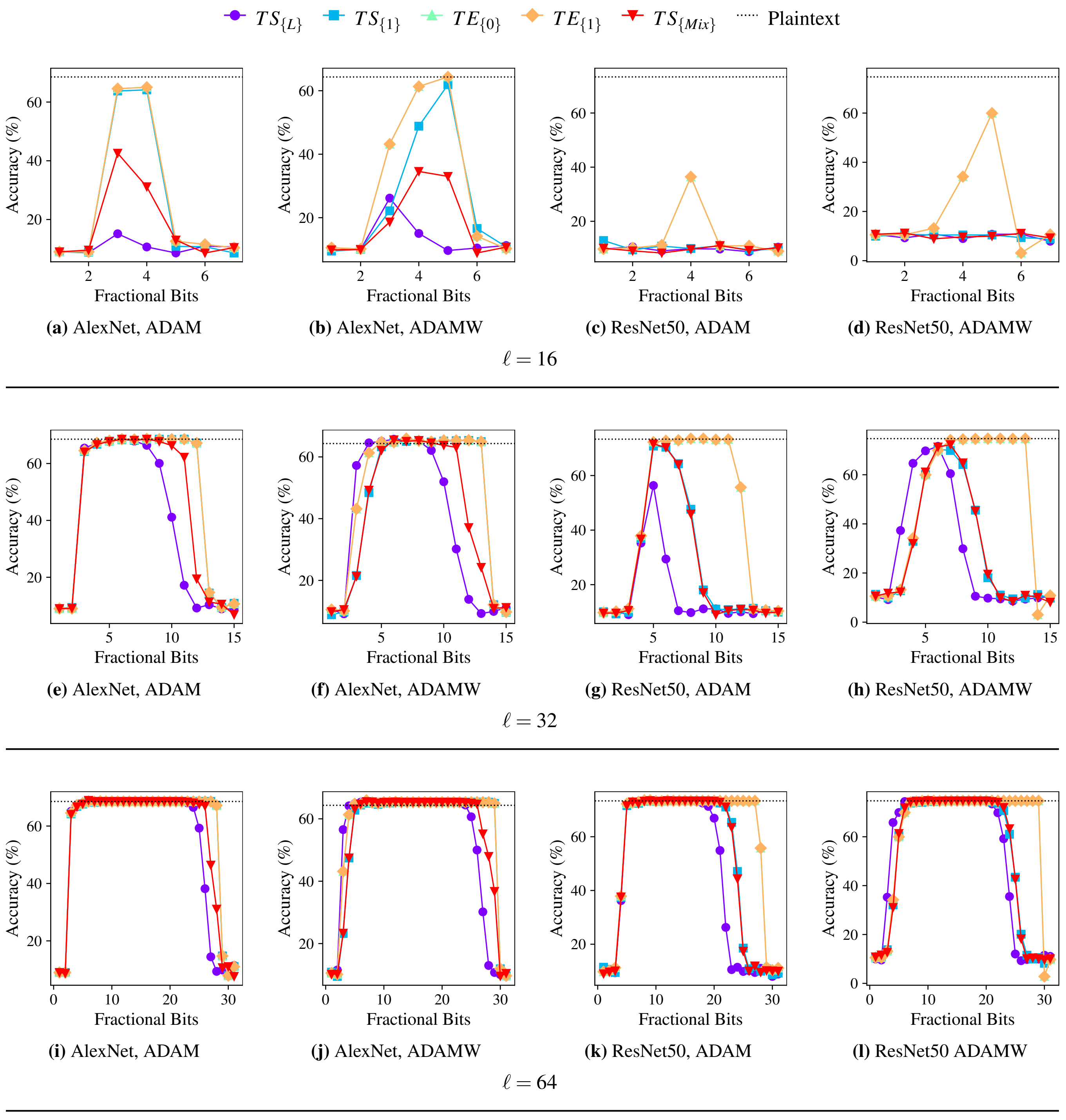


Figure 19: Accuracy of different truncation schemes for various models trained on CIFAR=10 with ADAMW and 0.03 weight decay or regular ADAM without weight decay. Each row corresponds to a different Bitlength ℓ as indicated.

Table 10: Ranges of fractional bits that introduce at most x% of accuracy loss compared to plaintext inference

Model	Plaintext Accuracy	Scheme	x=0%			x=1%			x=5%		
			$\ell = 16$	$\ell = 32$	$\ell = 64$	$\ell = 16$	$\ell = 32$	$\ell = 64$	$\ell = 16$	$\ell = 32$	$\ell = 64$
		MNIST (28x28x1)									
		$TS_{\{L\}}$	-	5	-	-	2-7	2-23	-	2-8	2-24
		$TS_{\{1\}}$	-	6-11	2	3	2-11	2-27	2-4	2-12	2-28
LeNet	99.12%	$TE_{\{0\}}$	-	5-11	5	2-3	2-11	2-27	2-4	2-12	2-28
		$TE_{\{1\}}$	-	5-11	5	2-3	2-11	2-27	2-4	2-12	2-28
		$TS_{\{Mix\}}$	-	5-11	5	-	2-11	2-27	-	2-12	2-28
		CIFAR-10 (32x32x3)									
		$TS_{\{L\}}$	-	6	11-21	-	6-7	5-23	-	3-8	3-24
		$TS_{\{1\}}$	-	8-9	8-27	-	5-11	6-27	-	3-12	4-28
AlexNet	68.55%	$TE_{\{0\}}$	-	8-11	8-27	-	5-11	5-27	-	4-12	4-28
		$TE_{\{1\}}$	-	8-11	8-27	-	5-11	5-27	-	4-12	4-28
		$TS_{\{Mix\}}$	-	6	6-23	-	5-8	6-25	-	4-10	3-26
		$TS_{\{L\}}$	-	4-8	5-24	-	4-8	4-24	-	4-9	4-24
		$TS_{\{1\}}$	-	5-13	5-29	-	5-13	5-29	4-5	4-13	4-29
AlexNet[w]	64.33%	$TE_{\{0\}}$	5	5-13	5-29	5	5-13	5-29	4-5	4-13	4-29
		$TE_{\{1\}}$	5	5-13	5-29	5	5-13	5-29	4-5	4-13	4-29
		$TS_{\{Mix\}}$	-	4-9	4-26	-	4-10	4-26	-	4-11	4-26
		$TS_{\{L\}}$	-	-	8-9	-	-	6-18	-	-	5-19
		$TS_{\{1\}}$	-	-	8-9	-	-	6-21	-	5-7	5-23
ResNet50	73.34%	$TE_{\{0\}}$	-	8-9	8-9	-	6-11	6-27	-	5-11	5-27
		$TE_{\{1\}}$	-	8-9	8-9	-	6-11	6-27	-	5-11	5-27
		$TS_{\{Mix\}}$	-	-	8-9	-	-	6-22	-	5-7	5-23
		$TS_{\{L\}}$	-	-	9-10	-	-	6-20	-	6	6-21
		$TS_{\{1\}}$	-	-	6-7	-	6-7	6-22	-	6-7	6-23
ResNet50 [w]	74.71%	$TE_{\{0\}}$	-	-	-	-	7-13	7-29	-	7-13	7-29
		$TE_{\{1\}}$	-	-	-	-	7-13	7-29	-	7-13	7-29
		$TS_{\{Mix\}}$	-	-	6-7	-	-	6-22	-	5-7	6-23
		$TS_{\{L\}}$	-	-	6-7	-	-	5-16	-	-	5-18
		$TS_{\{1\}}$	-	-	10-12	-	5	5-21	-	5	5-21
VGG-16	81.74%	$TE_{\{0\}}$	-	-	11-12	-	-	5-21	-	5	5-21
		$TE_{\{1\}}$	-	-	11-12	-	-	5-21	-	5	5-21
		$TS_{\{Mix\}}$	-	-	10-12	-	-	5-21	-	5	5-21
		ImageNet (224x224x3)									
		$TS_{\{L\}}$	-	-	11	-	-	8-21	-	-	7-22
		$TS_{\{1\}}$	-	10-11	25-26	-	8-12	8-28	-	7-12	7-28
AlexNet[p]	62.50%	$TE_{\{0\}}$	-	11	11	-	8-12	8-28	-	7-12	7-28
		$TE_{\{1\}}$	-	11	11	-	8-12	8-28	-	7-12	7-28
		$TS_{\{Mix\}}$	-	10-11	25-26	-	8-12	8-28	-	7-12	7-28
		$TS_{\{L\}}$	-	-	7-9	-	-	7-18	-	-	6-18
		$TS_{\{1\}}$	-	7-10	7-9	-	7-12	7-28	-	6-12	6-28
VGG-16[p]	82.81%	$TE_{\{0\}}$	-	7	7	-	7-12	7-28	-	6-12	6-28
		$TE_{\{1\}}$	-	7	7	-	7-12	7-28	-	6-12	6-28
		$TS_{\{Mix\}}$	-	7-10	7-9	-	7-12	7-28	-	6-12	6-28

[w] Weight decay of 0.03 [p] Pretrained weights provided by PyTorch. Unmodified model architecture.

E Additional Evaluation of Communication Complexity

Table 11: Trio (3PC): Reduction in communication complexity of different truncation schemes for various models and datasets when delaying truncation.

Model	Scheme	$\ell = 16$			$\ell = 32$			$\ell = 64$		
		$\neg$ D	D	Δ	$\neg$ D	D	Δ	$\neg$ D	D	Δ
MNIST (28x28x1)										
LeNet	$TS_{\{L\}}$	0.144	0.157	-8.29%	0.291	0.317	-8.26%	0.589	0.641	-8.22%
	$TS_{\{1\}}$	0.222	0.212	4.97%	0.447	0.423	5.58%	0.900	0.853	5.52%
	$TE_{\{0\}}$	0.298	0.235	26.94%	0.605	0.476	27.22%	1.235	0.969	27.46%
	$TE_{\{1\}}$	0.332	0.282	17.76%	0.664	0.561	18.44%	1.366	1.152	18.62%
	$TS_{\{Mix\}}$	0.209	0.199	5.28%	0.422	0.398	5.93%	0.849	0.802	5.88%
CIFAR-10 (32x32x3)										
ResNet18	$TS_{\{L\}}$	3.672	4.276	-14.13%	7.390	8.599	-14.06%	14.82	17.24	-14.01%
	$TS_{\{1\}}$	6.706	6.533	2.65%	13.46	13.06	3.00%	26.95	26.13	3.17%
	$TE_{\{0\}}$	9.519	8.410	13.19%	19.31	17.05	13.27%	38.90	34.34	13.28%
	$TE_{\{1\}}$	10.81	9.951	8.64%	21.52	19.75	8.96%	43.70	40.11	8.95%
	$TS_{\{Mix\}}$	6.672	6.499	2.66%	13.39	13.00	3.02%	26.83	26.00	3.18%
ResNet50	$TS_{\{L\}}$	5.794	6.705	-13.59%	11.66	13.49	-13.52%	23.40	27.04	-13.49%
	$TS_{\{1\}}$	10.36	10.07	2.97%	20.80	20.13	3.33%	41.67	40.25	3.53%
	$TE_{\{0\}}$	14.60	12.75	14.50%	29.62	25.84	14.60%	59.63	52.04	14.58%
	$TE_{\{1\}}$	16.54	15.10	9.58%	32.94	29.96	9.94%	66.85	60.85	9.86%
	$TS_{\{Mix\}}$	10.33	10.03	2.97%	20.74	20.06	3.35%	41.54	40.12	3.54%
VGG-16	$TS_{\{L\}}$	6.052	6.606	-8.39%	12.21	13.32	-8.32%	24.53	26.74	-8.27%
	$TS_{\{1\}}$	9.073	8.621	5.24%	18.25	17.25	5.82%	36.60	34.49	6.12%
	$TE_{\{0\}}$	11.92	9.257	28.81%	24.18	18.75	28.99%	48.71	37.72	29.14%
	$TE_{\{1\}}$	13.23	11.12	18.99%	26.42	22.09	19.62%	53.56	44.80	19.55%
	$TS_{\{Mix\}}$	8.822	8.372	5.38%	17.75	16.75	5.99%	35.60	33.49	6.30%
AlexNet	$TS_{\{L\}}$	0.267	0.291	-8.35%	0.538	0.586	-8.30%	1.080	1.178	-8.26%
	$TS_{\{1\}}$	0.403	0.383	5.19%	0.810	0.766	5.77%	1.624	1.531	6.09%
	$TE_{\{0\}}$	0.532	0.415	28.31%	1.080	0.840	28.57%	2.177	1.690	28.81%
	$TE_{\{1\}}$	0.591	0.498	18.69%	1.181	0.990	19.34%	2.398	2.007	19.48%
	$TS_{\{Mix\}}$	0.388	0.369	5.37%	0.781	0.737	6.01%	1.567	1.474	6.32%
ImageNet (224x224x3)										
AlexNet	$TS_{\{L\}}$	28.15	29.14	-3.40%	56.86	58.84	-3.37%	114.3	118.2	-3.33%
	$TS_{\{1\}}$	33.10	32.28	2.52%	66.73	64.93	2.77%	134.0	130.2	2.91%
	$TE_{\{0\}}$	37.66	32.90	14.48%	76.25	66.53	14.61%	153.4	133.8	14.67%
	$TE_{\{1\}}$	39.77	35.99	10.50%	79.84	72.08	10.77%	161.2	145.5	10.79%
	$TS_{\{Mix\}}$	33.10	32.28	2.52%	66.73	64.93	2.77%	134.0	130.2	2.91%
VGG-16	$TS_{\{L\}}$	383.2	410.3	-6.60%	773.3	827.5	-6.55%	1554	1662	-6.52%
	$TS_{\{1\}}$	518.8	496.7	4.45%	1044	995.3	4.93%	2095	1992	5.17%
	$TE_{\{0\}}$	644.3	513.6	25.45%	1305	1039	25.62%	2627	2091	25.68%
	$TE_{\{1\}}$	701.8	598.4	17.28%	1404	1192	17.79%	2842	2412	17.80%
	$TS_{\{Mix\}}$	518.8	496.7	4.45%	1044	995.3	4.93%	2095	1992	5.17%

$\neg$D: Total communication in MB if not delaying truncation.
D: Total communication in MB if delaying truncation.
Δ: Percentage reduction in communication complexity.

Table 12: Quad (4PC): Reduction in communication complexity of different truncation schemes for various models and datasets when delaying truncation.

Model	Scheme	$\ell = 16$			$\ell = 32$			$\ell = 64$		
		$\neg$ D	D	Δ	$\neg$ D	D	Δ	$\neg$ D	D	Δ
MNIST (28x28x1)										
LeNet	$TS_{\{L\}}$	0.251	0.277	-9.42%	0.506	0.558	-9.36%	1.021	1.127	-9.37%
	$TS_{\{1\}}$	0.391	0.369	5.92%	0.784	0.736	6.50%	1.579	1.483	6.47%
	$TE_{\{0\}}$	0.495	0.381	29.92%	1.005	0.772	30.18%	2.051	1.572	30.44%
	$TE_{\{1\}}$	0.575	0.487	18.08%	1.149	0.968	18.74%	2.363	1.987	18.89%
	$TS_{\{Mix\}}$	0.369	0.347	6.31%	0.740	0.692	6.89%	1.491	1.394	6.90%
CIFAR-10 (32x32x3)										
ResNet18	$TS_{\{L\}}$	6.283	7.491	-16.13%	12.64	15.06	-16.06%	25.35	30.19	-16.01%
	$TS_{\{1\}}$	11.74	11.38	3.22%	23.56	22.75	3.55%	47.19	45.50	3.70%
	$TE_{\{0\}}$	15.62	13.61	14.77%	31.69	27.60	14.83%	63.85	55.60	14.84%
	$TE_{\{1\}}$	18.62	17.11	8.84%	37.05	33.95	9.14%	75.26	68.97	9.12%
	$TS_{\{Mix\}}$	11.68	11.32	3.22%	23.44	22.64	3.57%	46.96	45.27	3.73%
ResNet50	$TS_{\{L\}}$	9.922	11.74	-15.51%	19.97	23.61	-15.44%	40.05	47.35	-15.40%
	$TS_{\{1\}}$	18.15	17.52	3.58%	36.41	35.04	3.91%	72.94	70.06	4.11%
	$TE_{\{0\}}$	23.98	20.64	16.20%	48.66	41.85	16.28%	98.00	84.28	16.28%
	$TE_{\{1\}}$	28.50	25.96	9.78%	56.73	51.52	10.12%	115.2	104.7	10.04%
	$TS_{\{Mix\}}$	18.09	17.46	3.59%	36.30	34.92	3.93%	72.71	69.84	4.11%
VGG-16	$TS_{\{L\}}$	10.48	11.59	-9.56%	21.13	23.34	-9.49%	42.43	46.87	-9.45%
	$TS_{\{1\}}$	15.90	14.97	6.25%	31.97	29.93	6.82%	64.13	59.87	7.12%
	$TE_{\{0\}}$	19.83	15.02	32.05%	40.22	30.42	32.20%	81.01	61.23	32.31%
	$TE_{\{1\}}$	22.87	19.17	19.33%	45.66	38.07	19.94%	92.57	77.22	19.88%
	$TS_{\{Mix\}}$	15.46	14.53	6.44%	31.10	29.06	7.02%	62.38	58.12	7.33%
AlexNet	$TS_{\{L\}}$	0.463	0.511	-9.53%	0.932	1.029	-9.45%	1.871	2.066	-9.42%
	$TS_{\{1\}}$	0.707	0.666	6.17%	1.420	1.331	6.76%	2.848	2.660	7.06%
	$TE_{\{0\}}$	0.885	0.673	31.44%	1.795	1.363	31.68%	3.619	2.743	31.94%
	$TE_{\{1\}}$	1.023	0.860	19.00%	2.042	1.706	19.65%	4.145	3.461	19.77%
	$TS_{\{Mix\}}$	0.682	0.641	6.45%	1.370	1.280	7.03%	2.747	2.560	7.34%
ImageNet (224x224x3)										
AlexNet	$TS_{\{L\}}$	48.23	50.20	-3.92%	97.37	101.3	-3.89%	195.7	203.5	-3.88%
	$TS_{\{1\}}$	57.12	55.44	3.03%	115.2	111.5	3.29%	231.2	223.6	3.42%
	$TE_{\{0\}}$	63.43	54.83	15.69%	128.4	110.9	15.79%	258.3	223.0	15.84%
	$TE_{\{1\}}$	68.32	61.67	10.78%	137.1	123.5	11.00%	276.8	249.3	11.02%
	$TS_{\{Mix\}}$	57.12	55.44	3.03%	115.2	111.5	3.29%	231.2	223.6	3.42%
VGG-16	$TS_{\{L\}}$	659.8	714.0	-7.59%	1331	1440	-7.53%	2674	2891	-7.50%
	$TS_{\{1\}}$	903.8	858.0	5.34%	1819	1719	5.83%	3649	3441	6.05%
	$TE_{\{0\}}$	1077	841.0	28.05%	2182	1702	28.19%	4392	3424	28.25%
	$TE_{\{1\}}$	1211	1029	17.63%	2421	2050	18.12%	4900	4149	18.11%
	$TS_{\{Mix\}}$	903.8	858.0	5.34%	1819	1719	5.83%	3649	3441	6.05%

$\neg$D: Total communication in MB if not delaying truncation.
D: Total communication in MB if delaying truncation.
Δ: Percentage reduction in communication complexity.

F Additional Runtime Evaluation

Table 13: Runtime (s) for different truncation schemes in LAN: 25 Gbit/s bandwidth, 0.3 ms latency.

Setting	Scheme	CIAR-10				ImageNet	
		ResNet50		VGG-16		VGG-16	
		32	64	32	64	32	64
3PC	$TS_{\{L\}}$	3.35 ± 0.06	6.25 ± 0.05	2.72 ± 0.15	6.04 ± 0.03	6.51 ± 0.00	15.29 ± 0.05
	$TS_{\{1\}}$	4.17 ± 0.02	6.96 ± 0.38	3.23 ± 0.00	6.82 ± 0.00	6.90 ± 0.03	15.80 ± 0.17
	$TE_{\{0\}}$	14.59 ± 0.29	29.74 ± 0.32	6.74 ± 0.01	14.16 ± 0.02	6.75 ± 0.04	15.90 ± 0.10
	$TE_{\{1\}}$	8.72 ± 0.14	29.92 ± 0.01	3.56 ± 0.00	11.87 ± 0.01	7.23 ± 0.08	16.22 ± 0.27
	$TS_{\{Mix\}}$	3.87 ± 0.19	6.70 ± 0.08	3.11 ± 0.00	6.67 ± 0.12	6.86 ± 0.05	15.64 ± 0.04
4PC	$TS_{\{L\}}$	3.45 ± 0.03	6.41 ± 0.03	3.35 ± 0.02	6.67 ± 0.01	15.93 ± 0.08	38.34 ± 0.03
	$TS_{\{1\}}$	4.34 ± 0.00	7.01 ± 0.09	3.72 ± 0.01	7.04 ± 0.00	16.60 ± 0.02	39.09 ± 0.16
	$TE_{\{0\}}$	14.77 ± 0.53	15.48 ± 2.98	7.13 ± 0.04	14.75 ± 0.01	16.85 ± 0.24	39.34 ± 0.19
	$TE_{\{1\}}$	8.71 ± 0.08	5.24 ± 0.41	4.07 ± 0.01	13.34 ± 0.09	17.27 ± 0.03	40.30 ± 0.80
	$TS_{\{Mix\}}$	4.31 ± 0.00	5.13 ± 1.86	3.75 ± 0.00	6.99 ± 0.02	15.95 ± 0.33	39.15 ± 0.03

Table 14: Runtime (s) for different truncation schemes in WAN: 0.2 Gbit/s bandwidth, 40 ms latency.

Setting	Scheme	CIFAR-10			
		ResNet50		VGG-16	
		32	64	32	64
3PC	$TS_{\{L\}}$	69.05 ± 0.01	132.38 ± 0.00	20.95 ± 0.00	41.27 ± 0.04
	$TS_{\{1\}}$	75.43 ± 0.03	138.98 ± 0.11	22.46 ± 0.09	42.27 ± 0.00
	$TE_{\{0\}}$	282.40 ± 0.09	559.10 ± 0.92	54.03 ± 0.08	106.50 ± 0.27
	$TE_{\{1\}}$	153.10 ± 0.06	292.61 ± 0.15	29.33 ± 0.06	58.57 ± 0.00
	$TS_{\{Mix\}}$	72.44 ± 0.05	136.98 ± 0.12	22.31 ± 0.06	42.15 ± 0.12
4PC	$TS_{\{L\}}$	69.08 ± 0.00	132.53 ± 0.04	21.43 ± 0.02	41.87 ± 0.02
	$TS_{\{1\}}$	75.56 ± 0.07	138.77 ± 0.01	22.85 ± 0.02	43.02 ± 0.13
	$TE_{\{0\}}$	282.32 ± 0.06	559.33 ± 0.01	54.13 ± 0.04	106.65 ± 0.05
	$TE_{\{1\}}$	152.96 ± 0.33	292.49 ± 0.76	29.99 ± 0.04	56.51 ± 0.02
	$TS_{\{Mix\}}$	75.44 ± 0.01	138.74 ± 0.02	22.60 ± 0.03	42.69 ± 0.01