
Secure Quantized Training for Deep Learning

Marcel Keller¹ Ke Sun^{1,2}

Abstract

We implement training of neural networks in secure multi-party computation (MPC) using quantization commonly used in said setting. We are the first to present an MNIST classifier purely trained in MPC that comes within 0.2 percent of the accuracy of the same convolutional neural network trained via plaintext computation. More concretely, we have trained a network with two convolutional and two dense layers to 99.2% accuracy in 3.5 hours (under one hour for 99% accuracy). We have also implemented AlexNet for CIFAR-10, which converges in a few hours. We develop novel protocols for exponentiation and inverse square root. Finally, we present experiments in a range of MPC security models for up to ten parties, both with honest and dishonest majority as well as semi-honest and malicious security.

1. Introduction

Secure multi-party computation (MPC) is a cryptographic technique that allows a set of parties to compute a public output on private inputs without revealing the inputs or any intermediate results. This makes it a potential solution to federated learning where the sample data stays private and only the model or even only inference results are revealed.

Imagine a set of healthcare providers holding sensitive patient data. MPC allows them to collaboratively train a model. This model could then either be released or even kept private for inference using MPC again. See Figure 1 for an illustration. Note that MPC is oblivious to how the input data is split among participants, that is, it can be used for the horizontal as well as the vertical case.

A more conceptual example is the well-known *millionaires' problem* where two people want to find out who is richer without revealing their wealth. There is clearly a difference

between the one bit of information desired and the full figures.

There has been a sustained interest in applying secure computation to machine learning and neural networks going back to at least Barni et al. (2006). More recent advances in practical MPC have led to an increased effort in implementing both inference and training.

A number of works such as Mohassel & Zhang (2017), Mohassel & Rindal (2018), Wagh et al. (2019), Wagh et al. (2021) implement neural network training with MPC at least in parts. However, for the task of classifying MNIST digits, they either give accuracy figures below 95% or figures obtained using plaintext training. For the latter case, the works do not clarify how close the computation for plaintext training matches the lower precision and other differences in the MPC setting. Agrawal et al. (2019) achieved an accuracy of 99.38% in a comparable setting for a convolutional neural network with more channels than we use. However, their actual implementation only uses dense layers, and we achieve comparable accuracy in this model. All works use quantization in the sense that a fractional number x is represented as $\lfloor x \cdot 2^f \rfloor$ where $\lfloor \cdot \rfloor$ denotes rounding to the nearest integer, and $f \in \mathbb{Z}$ (\mathbb{Z} : the set of integers) is a precision parameter. This makes addition considerably faster in the secure computation setting because it reduces to integer addition (at the cost of compounding rounding errors). Furthermore, some of the works suggest replacing the softmax function that uses exponentiation with a ReLU-based replacement. Keller & Sun (2020) discovered that this softmax replace-

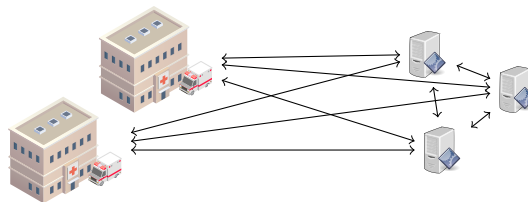


Figure 1: Outsourced computation: Data holders (on the left) secret-share their data to a number of computing parties (on the right), who then return the desired result (e.g., a model or inference results on further queries). All communication except the outputs is secret-shared and thus secure if no two computing parties collude.

¹CSIRO's Data61, Sydney, Australia ²The Australian National University. Correspondence to: Marcel Keller <marcel.keller@data61.csiro.au>, Ke Sun <ke.sun@data61.csiro.au>.

ment deteriorates accuracy in multilayer perceptrons to the extent that it does not justify the efficiency gains.

The concurrent work of [Tan et al. \(2021\)](#) gives some figures on the learning curves when trained with secure computation. However, they stop at five epochs for MNIST training and achieve 94% accuracy, whereas we present the figures up to 50 epochs and 99.2% accuracy using AMSGrad. Our implementation using stochastic gradient descent (SGD) is 40% faster than theirs. Note that we use the CPU of one AWS c5.9xlarge instance per party whereas [Tan et al.](#) use one NVIDIA Tesla V100 GPU per party. We believe this somewhat counter-intuitive result comes from MPC heavily relying on communication, which is an aspect where GPUs do not have an advantage over CPUs. While GPUs are very efficient in computing on data in their cache, it is no more efficient to transfer data in and out of these caches.

In this paper, we present an extensible framework for secure deep learning based on MP-SPDZ by [Keller \(2020\)](#), a software library for multi-party computation.¹ Similar to TensorFlow and PyTorch, our approach allows representing deep learning models as a succession of layers. We then use this implementation to obtain accuracy figures for MNIST and CIFAR-10 training by utilizing the MP-SPDZ emulator, which allows to run the plaintext equivalent of secure computation, that is, the same algorithms with the same precision. Finally, we run one of the most promising instantiations in real secure computation in order to benchmark it, confirming the results from the plaintext emulator. We use less sophisticated datasets because the relatively high cost of MPC makes it less feasible to use datasets with millions of examples rather than tens of thousands.

For our implementation, we have designed novel protocols for exponentiation and inverse square root computation. Both play an important role in deep learning, the first in softmax, and the second in the Adam and AMSGrad optimizers. Our protocols are based on mixed-circuit computation, that is, we combine a protocol that works over \mathbb{Z}_M for some large modulus M and thus integer-like computation with a protocol in the same security model that works over \mathbb{Z}_2 , which allows computing binary circuits.²

There are several projects that integrate secure computation directly into popular machine learning frameworks such as CrypTen by [Knott et al. \(2021\)](#), PySyft by [Ryffel et al. \(2018\)](#), and TF Encrypted by [Dahl et al. \(2018\)](#). Our approach differs from all of them by running the protocol as native CPU code (implemented using C++). This allows for much faster execution. For example, CrypTen provides an MNIST training example (`mpc_autograd_cnn`), which

¹Code available at <https://github.com/data61/MP-SPDZ>.

² $\mathbb{Z}_M = \mathbb{Z}/M\mathbb{Z}$ denotes integer computation modulo M .

we adapted to full LeNet ([LeCun et al., 1998](#)) training. It took over three hours to run one epoch on one machine. In comparison, our implementation takes 11 minutes to run one epoch with the full dataset of 60,000 samples with the same hardware across instances.

An alternative to MPC would be using purely homomorphic encryption. [Lou et al. \(2020\)](#) trained an MNIST network to 98.6% accuracy in 8 days whereas we achieved the same accuracy in less than 5 days when using a two-party protocol based on homomorphic encryption.

A number of works ([Juvekar et al., 2018](#); [Mishra et al., 2020](#); [Rathee et al., 2020](#)) only implemented deep learning inference. None of them implements all building blocks necessary for training. It is therefore not possible to make any statement on the training performance of these works.

Another line of work (e.g., [Quoc et al. 2021](#)) uses trusted execution environments that provide computation outside the reach of the operating system. This is a different security model to multi-party computation that works with distributing the information among several entities.

The rest of this paper is structured as follows: We highlight some secure computation aspects in Section 2 and do the same for machine learning in Section 3. Section 4 analyzes the error introduced by the quantization. Finally, we present our implementation and our experimental results for MNIST and CIFAR-10 classification in Section 5.

2. Secure Computation Building Blocks

Our secure implementation has two layers of abstraction. The lower level is concerned with a few basic operations that depend on the actual protocol and security model. The basic operations include integer addition and multiplication, input, output, and domain conversion, that is converting between integer-focused and bit-focused computation. Appendices A and B give the details of our honest-majority three-party protocol and our dishonest-majority protocol, respectively. Honest and dishonest majority refer to how many computing parties are being trusted. For an honest majority, strictly more than half parties must follow the protocol and not reveal information to anyone else. This setting allows for more efficient protocols while the dishonest-majority setting requires more expensive cryptographic schemes such as homomorphic encryption to be used. See [Keller \(2020\)](#) for a cost comparison between a range of protocols. The higher level of abstraction is common to all protocol as it builds in a generic way on the primitives in the lower level. Notable operations on this level are division, exponentiation, and inverse square root. Appendix C provides more details.

In this section, we focus on two aspects: How to implement fractional-number computation with integers, and how to im-

plement exponentiation. The former plays a crucial role in efficient secure computation, and the latter presents a novel protocol. We also contribute a novel protocol for inverse square root computation, which is deferred to Appendix C due to space constraints.

2.1. Quantization

While Aliasgari et al. (2013) showed that it is possible to implement floating-point computation, the cost is far higher than integer computation. It is therefore common to represent fractional numbers using quantization (also called fixed-point representation) as suggested by Catrina & Saxena (2010). A real number x is represented as $Q^f(x) := \lfloor x \cdot 2^f \rfloor$ where f is a positive integer specifying the precision. The linearity of the representation allows to compute addition by simply adding the representing integers. Multiplication however requires adjusting the result because it will have twice the precision: $(x \cdot 2^f) \cdot (y \cdot 2^f) = xy \cdot 2^{2f}$. There are two ways to rectify this:

- An obvious correction would be to shift the result by f bits after adding 2^{f-1} to the integer representation, as $\lfloor xy \cdot 2^f \rfloor = \lfloor 2^{-f} \cdot (xy \cdot 2^{2f} + 2^{f-1}) \rfloor$, where $\lfloor \cdot \rfloor$ is the floor function. This ensures rounding to the nearest number possible in the representation, with the tie being broken by rounding up.
- However, Catrina & Saxena found that in the context of secure computation it is more efficient to use probabilistic truncation. This method rounds up or down probabilistically depending on the input. For example, probabilistically rounding 0.75 to an integer would see it rounded up with probability 0.75 and down with probability 0.25.

Our quantization scheme is related to quantized neural networks (see e.g. Hubara et al. 2016). However, our consideration is not to compress the model, but to improve the computational speed and to reduce communication.

2.2. Exponentiation

Exponentiation is a core computational module in deep learning and is useful to evaluate non-linear activation functions such as the softmax. Mohassel & Zhang (2017) introduced a simplification of softmax to avoid computing exponentiation. In this section, we present an optimized exponentiation protocol instead.

As noted by Aly & Smart (2019), exponentiation can be reduced to exponentiation with base two using $a^x = 2^{x \log_2 a}$. Furthermore, if the base is public, this reduction only costs one public-private multiplication, and it introduces an ac-

Algorithm 1 Exponentiation with base two (Aly & Smart, 2019)

Input: Secret share $\langle x; f \rangle^A$ with precision f where $-(k - f - 1) < x < k - f - 1$

Output: $\langle 2^x; f \rangle^A$ with precision f and total bit length k

- 1: $\langle s \rangle^A \leftarrow \langle x; f \rangle^A < 0$ {sign}
- 2: $\langle x; f \rangle^A \leftarrow \langle x; f \rangle^A - 2 \cdot \langle s \rangle^A \cdot \langle x; f \rangle^A$ {absolute value}
- 3: $\langle i \rangle^A \leftarrow \text{Trunc}(\langle x; f \rangle^A)$ {integer component}
- 4: $\langle r; f \rangle^A \leftarrow \langle x; f \rangle^A - \langle i \rangle^A$ {fractional component}
- 5: $\ell \leftarrow \lceil \log_2(k - f) \rceil$
- 6: $\langle i_0 \rangle^A, \dots, \langle i_{\ell-1} \rangle^A \leftarrow \text{BitDec}(\langle r \rangle^A, \ell)$ {bit decomposition}
- 7: $\langle d \rangle^A \leftarrow \prod_{j=0}^{\ell-1} (\langle i_j \rangle^A \cdot 2^{2^j} + 1 - \langle i_j \rangle^A)$ {integer exponentiation}
- 8: $\langle u; f \rangle^A \leftarrow \text{Approx}_{2^*}(\langle r; f \rangle^A)$ {polynomial approximation}
- 9: $\langle g; f \rangle^A \leftarrow \langle u; f \rangle^A \cdot \langle d \rangle^A$
- 10: **return** $\langle s \rangle^A \cdot \left(\frac{1}{\langle g; f \rangle^A} - \langle g; f \rangle^A \right) + \langle g; f \rangle^A$ {correct for sign}

Algorithm 2 Exponentiation with base two (ours)

Input: Secret share $\langle x; f \rangle^A$ with precision f where $x < k - f - 1$

Output: $\langle 2^x; f \rangle^A$ with precision f and total bit length k

- 1: $\langle x_0 \rangle^B, \dots, \langle x_{k-1} \rangle^B \leftarrow \text{A2B}(\langle x; f \rangle^A)$ {bit decomposition}
- 2: $\langle z \rangle^B \leftarrow \sum_{i=0}^{k-1} 2^{i-f} \langle x_i \rangle^B < -(k - f - 1)$ {binary comparison with fixed-representation}
- 3: $\ell \leftarrow \lceil \log_2(k - f) \rceil$
- 4: **for** $j = f, \dots, f + \ell - 1$ **do**
- 5: $\langle x_j \rangle^A \leftarrow \text{Bit2A}(\langle x_j \rangle^B)$
- 6: **end for**
- 7: $\langle d \rangle^A \leftarrow \prod_{j=0}^{\ell-1} (\langle x_{f+j} \rangle^A \cdot 2^{2^j} + 1 - \langle x_{f+j} \rangle^A)$ {integer exponentiation}
- 8: $\langle r \rangle^A \leftarrow \text{B2A}(\langle x_0 \rangle^B, \dots, \langle x_{f-1} \rangle^B)$ {fractional component}
- 9: $\langle u; f \rangle^A \leftarrow \text{Approx}_{2^*}(\langle r; f \rangle^A)$ {polynomial approximation}
- 10: $\langle g; f \rangle^A \leftarrow \langle u; f \rangle^A \cdot \langle d \rangle^A$
- 11: $\langle g'; f \rangle^A \leftarrow \text{Round}(\langle g; f \rangle^A, f + 2^\ell, 2^\ell)$ {result for negative input}
- 12: $\langle h; f \rangle^A \leftarrow \text{Bit2A}(\langle x_{k-1} \rangle^B) \cdot (\langle g'; f \rangle^A - \langle g; f \rangle^A) + \langle g; f \rangle^A$ {correct for sign}
- 13: **return** $(1 - \text{Bit2A}(\langle z \rangle^B)) \cdot \langle h; f \rangle^A$ {output 0 if input is too small}

ceptable error as

$$2^{x \log_2 a \pm 2^{-f}} = 2^{x \log_2 a} \cdot 2^{\pm 2^{-f}}.$$

Algorithm 3 Procedures in Algorithms 1 and 2

$\langle x_0 \rangle^B, \dots, \langle x_{k-1} \rangle^B \leftarrow \text{A2B}(\langle x; f \rangle^A)$ Arithmetic to binary conversion of the k least significant bits, which can be implemented using edaBits (Escudero et al., 2020) or local share conversion (Demmler et al., 2015; Araki et al., 2018) if available in the underlying protocol.

$\langle x \rangle^A \leftarrow \text{Bit2A}(\langle x \rangle^B)$ Bit to arithmetic conversion, which can be implemented using daBits (Rotaru & Wood, 2019).

$\langle x \rangle^A \leftarrow \text{B2A}(\langle x_0 \rangle^B, \dots, \langle x_k \rangle^B)$ Binary to arithmetic conversion of k -bit values, which can be implemented using edaBits.

$\langle y; f \rangle^A \leftarrow \text{Round}(\langle x; f \rangle^A, k, m)$ Truncate a k -bit value by m bits with either nearest or probabilistic rounding, corresponding to a division by 2^m . Both also benefit from mixed-circuit computation as shown by Dalskov et al. (2021).

$\langle y; f \rangle^A \leftarrow \text{Approx}_{2^*}(\langle x; f \rangle^A)$ Approximate 2^x for $x \in [0, 1]$ using a Taylor series.

The second multiplicand on the right-hand side is easily seen to be within $[1 - 2^{-f}, 1 + 2^{-f}]$.

Algorithm 1 shows the exponentiation method by Aly & Smart. It uses $\langle \cdot \rangle^A$ to indicate integer sharing and $\langle \cdot; f \rangle^A$ to indicate fixed-point sharing. We use the superscript A to indicate that both secret sharing are in the arithmetic domain, that is, in Z_M for some large M . This distinguishes them from the bit-wise secret sharing indicated by B used below. The algorithm uses a number of standard components such as addition, multiplication, comparison, and bit decomposition. Furthermore, it uses a polynomial approximation of 2^x for $x \in [0, 1]$. They propose to use the polynomial P_{1045} by Hart (1978). However, we found that Taylor approximation at 0 leads to better results than P_{1045} .

Algorithm 1 suffers from two shortcomings. First, it is unnecessarily restrictive on negative inputs because it uses the absolute value of the input, the exponentiation of which falls outside the representation range for values as small as $k - f - 1$ where k is a parameter defining the range of inputs $([-2^{k-f-1}, 2^{k-f-1}])$. Second, the truncation used to find the integer component resembles bit decomposition in many protocols. It is therefore wasteful to separate the truncation and the bit decomposition. Algorithm 2 fixes these two issues. It also explicitly uses mixed circuits, with binary secrets denoted using $\langle \cdot \rangle^B$.

The general structure of Algorithm 2 is the same as Algorithm 1. It splits the input into an integer and a fractional

Table 1: Total communication in kbit for Algorithms 1 and 2 across a range of security models with one corrupted party. ‘‘SH’’ stands for semi-honest security and ‘‘Mal.’’ for malicious security.

| | 3 Parties | | 2 Parties | |
|--------------------|-----------|------|-----------|---------|
| | SH | Mal. | SH | Mal. |
| Aly & Smart (2019) | 27 | 498 | 1,338 | 214,476 |
| Algorithm 2 (ours) | 16 | 323 | 813 | 121,747 |

part and then computes the exponential separately, using an exact algorithm for the integer part and an approximation for the fractional part. It differs in that it uses mixed circuits and it allows any negative input. The biggest difference however is in step 11, where the exponentiation of a negative number is not computed by inverting the exponentiation of the absolute value. Instead, we use the following observation. If $-2^\ell \leq x < 0$, $x = -2^\ell + y + r$, where $y = x + 2^\ell - r$ and r is the fractional component of x . Then,

$$2^x = 2^{-2^\ell} \cdot 2^y \cdot 2^r.$$

2^y equals d in Algorithm 2 because y is the composition of the bits used in integer exponentiation, namely $y = \sum_{i=0}^{\ell-1} x_{f+i}$. Therefore, $g \approx 2^y \cdot 2^r$, and



$$g' \approx \frac{g}{2^{2^\ell}} \approx 2^x$$








for g and g' in steps 10 and 11 because rounding by 2^ℓ bits implies division by 2^{2^ℓ} . This saves a relatively expensive division compared to Algorithm 1.

Table 1 shows how the two algorithms compare in terms of communication across a number of security models. All figures are obtained using MP-SPDZ (Keller, 2020) with $f = 16$, $k = 31$, computation modulo 2^{64} for three-party computation, and computation modulo a 128-bit prime for two-party computation (as in Section 5.1.1). We use edaBits for mixed-circuit computation in Algorithm 2. Across all security models, our algorithm saves about 30 percent in communication despite the improved input range. In the context of the figures in Table 5 the improvement is less than one percent because exponentiation is only used once for every example per epoch, namely for the computation of softmax.

3. Deep Learning Building Blocks

In this section, we give a high-level overview of our secure deep learning framework, and how it builds upon our low-level computational modules (implemented from scratch) to tackle challenges in secure computation. Table 2 shows a non-exhaustive list of basic operations that is our computational alphabet, with a rough estimation of their cost in

Table 2: Basic operations and their cost. The column “MPC Cost” shows the roughly approximated computational cost in MPC, as compared to integer multiplication. The last column displays the computational and communication resource consumption. “” means the cheapest computation; “” means the most expensive.

| Operation | MPC Cost (\approx) | Efficiency |
|-------------------------------|------------------------|---|
| $\llbracket x > y \rrbracket$ | 10^1 |  |
| $x \pm y$ | 10^{-1} |  |
| $x \times y$ (integer) | 10^0 |  |
| $x \times y$ (fractional) | 10^1 |  |
| x/y | 10^2 |  |
| $1/\sqrt{x}$ | 10^2 |  |
| $\exp(x), \log(x)$ | 10^2 |  |

MPC. The “MPC Cost” column shows how an operation relates to integer multiplication in terms of magnitude order because integer multiplication is the most basic operation that involves communication. Most notably, comparison is not the basic operation as in silicon computation.

Linear Operations Matrix multiplication is the workhorse of deep learning frameworks. The forward and backward computation for both dense layers and convolutional layers are implemented as matrix multiplication, which in turn is based on dot products. A particular challenge in secure computation is to compute a number of outputs in parallel to save communication rounds. We overcome this challenge by having a dedicated infrastructure that computes all dot products for a matrix multiplication in *a single batch of communication*, thus reducing the number of communication rounds. Another challenge is the accumulated error when evaluating matrix multiplication using quantized numbers. The following Section 4 presents a careful analysis on our solution to this potential issue. Our secure computation implementation uses efficient dot products when available (Appendix A) or efficient use of SIMD-style homomorphic encryption (Appendix B).

Non-linear Operations A major part of the non-linear computation is the non-linear neural network layers. Both rectified linear unit (ReLU; Nair & Hintón 2010) and max pooling are based on comparison followed by oblivious selection. For example, $\text{ReLU}(x) := \max(x, 0) = (x > 0 ? x : 0)$, where the C expression (condition ? a : b) outputs a if condition is true and outputs b if otherwise. In secure computation, it saves communication rounds if the process uses a balanced tree rather than iterating over all input values of one maximum computation. For example, two-dimensional max pooling over a 2x2 window requires computing the maximum of four values. We compute the

maximum of two pairs of values followed by the maximum of the two results. Our softmax and sigmoid activation functions are implemented using the exponentiation algorithm in section 2.2. Another source of non-linear computation is the *inverse square root* $1/\sqrt{x}$, which appears in batch normalization (Ioffe & Szegedy, 2015), our secure version of Adam (Kingma & Ba, 2015) and AMSGrad (Reddi et al., 2018) optimizers, and Glorot initialization (Glorot & Bengio, 2010) of the neural network weights. Our novel implementation of $1/\sqrt{x}$ (explained in Appendix C) is more efficient than state-of-the-art alternatives (Aly & Smart, 2019; Lu et al., 2020). It helps to avoid numerical division, which is much more expensive than multiplication. Taking Adam as an example, the increment of the parameter θ_i in each learning step is altered to $\gamma g_i / \sqrt{v_i + \varepsilon}$ (γ : learning rate; g_i and v_i : first and second moments of the gradient; $\varepsilon = 10^{-8}$) as compared to $\gamma g_i / (\sqrt{v_i} + \varepsilon)$ in Kingma & Ba’s original algorithm, so that we only need to take the inverse square root of $v_i + \varepsilon$, then multiply by γ and g_i .

Random Operations Our secure framework supports random initialization of the network weights (Glorot & Bengio, 2010), random rounding, random shuffling of the training samples, and Dropout (Srivastava et al., 2014). Our implementation relies on uniform random samplers (Dalakov et al., 2021; Escudero et al., 2020) based on mixed-circuit computation. Random rounding and Dropout require to draw samples from Bernoulli random variables. By a reparameterization trick, a sample x from the distribution $\text{Prob}(x = a) = p$, $\text{Prob}(x = b) = 1 - p$ ($\text{Prob}(\cdot)$: probability of statement being true) can be expressed as $x = b + (a - b)\lfloor p + r \rfloor$, where $r \sim \mathcal{U}(0, 1)$ (uniform distribution on $[0, 1]$).

4. An Analysis of Probabilistic Rounding

In this section, we analyze the rounding error of our secure quantized representation for matrix multiplication. We show that our rounding algorithm ① is unbiased; ② has acceptable error; ③ is potentially helpful to training. Fixed point numbers (Lin et al., 2016) and stochastic rounding (Wang et al., 2018) have been applied to deep learning. The rounding error analysis presented here is potentially useful to similar quantized representations.

Recall that any $x \in \mathbb{R}$ is stored as an integer (equivalently, a fixed point number) $Q^f(x) := \lfloor x \cdot 2^f \rfloor$. As the maximum error of the nearest rounding $\lfloor \cdot \rfloor$ is $1/2$, we have $|x - Q^f(x)2^{-f}| = 2^{-f}|x \cdot 2^f - Q^f(x)| \leq 2^{-f-1} = \epsilon/2$, where $\epsilon := 2^{-f}$ is the smallest positive number we can represent. Our multiplication algorithm outputs $R^f(xy)$, the integer representation of the product xy . By definition,

$$R^f(xy) := \lfloor \mu \rfloor + b, \quad (1)$$

$$\mu := Q^f(x)Q^f(y)2^{-f}, \quad b \sim \mathcal{B}(\{\mu\}),$$

where $\{\mu\}$ is the fractional part of μ ($0 \leq \{\mu\} < 1$), the random variable $b \in \{0, 1\}$ means rounding down or up, and $\mathcal{B}(\cdot)$ denotes a Bernoulli distribution with given mean. Notice that the input $Q^f(x)$ and $Q^f(y)$ and the output $R^f(xy)$ are integers. They all have to be multiplied by 2^{-f} to get the underlying fixed point number. Eq. (1) can be computed purely based on integer operations (see Appendix C). The Bernoulli random variable b in Eq. (1) is sampled independently for each numerical product we compute.

Given two real matrices $A_{m \times n}$ and $B_{n \times p}$ represented by $Q^f(A)$ and $Q^f(B)$, where $Q^f(\cdot)$ is elementwisely applied to the matrix entries, the integer representation of their product is $R^f(AB) := (\sum_i R^f(A_{il}B_{lj}))_{m \times p}$. Its correctness is guaranteed in the following proposition.

Proposition 4.1. $\forall A \in \mathbb{R}^{m \times n}, \forall B \in \mathbb{R}^{n \times p}, E(R^f(AB)) = 2^{-f}Q^f(A)Q^f(B)$, where $E(\cdot)$ denotes the expectation with respect to the random rounding in Eq. (1).

The expectation of the random matrix $R^f(AB)$ is exactly the matrix product of $Q^f(A)$ and $Q^f(B)$ up constant scaling. Hence, our matrix multiplication is *unbiased*. Product of fixed point numbers can also be based on nearest rounding by replacing Eq. (1) with $\tilde{R}^f(xy) := \lfloor Q^f(x)Q^f(y)2^{-f} \rfloor$. As a deterministic estimation, it is unbiased only if the rounding error is absolutely zero, which is not true in general. Therefore, matrix product by nearest rounding is *biased*.

We bound the worst case error during matrix multiplication as follows.

Proposition 4.2. Assume $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times p}$, and the absolute value of all entries are bounded by 2^k ($k \geq 0$). Then, $\|R^f(AB) - 2^f AB\| < \sqrt{mp} \cdot n \cdot (2^k + 1 + \epsilon/4)$, where $\epsilon := 2^{-f}$, and $\|\cdot\|$ is the Frobenius norm.

In Proposition 4.2, k corresponds to the number of bits to present the integer part of any $x \in \mathbb{R}$. To get some intuition, we have the trivial bound $\|AB\| \leq \sqrt{mp} \cdot n \cdot 2^{k+1}$. By Proposition 4.2, the computational error $\|2^{-f}R^f(AB) - AB\|$ has a smaller order of magnitude, as it scales with $\epsilon = 2^{-f}$. The precision f should be high enough to avoid a large error. Let us remark that nearest rounding has similar deterministic error bounds (omitted here).

The following proposition shows our rounding in eq. (1) satisfies another *probabilistic bound*, which improves over the above worst case bound.

Proposition 4.3. Let $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times p}$. With probability at least $1 - \frac{1}{4t^2}$, the following is true

$$\|R^f(AB) - 2^{-f}Q^f(A)Q^f(B)\| \leq \iota\sqrt{mnp}.$$

By Proposition 4.1, $2^{-f}Q^f(A)Q^f(B)$ is the expectation of $R^f(AB)$. Therefore the first term is the ‘‘distance’’ between the random matrix $R^f(AB)$ and its mean. One can

roughly approximate $2^{-f}Q^f(A)Q^f(B) \approx 2^f AB$ to compare Proposition 4.3 with Proposition 4.2. The probabilistic bound scales the deterministic bound by a factor of $2^{-k}/\sqrt{n}$. That means our matrix multiplication is potentially more accurate than alternative implementations based on nearest rounding, especially for large matrices (n is large).

One can write Dropout (Srivastava et al., 2014) in a similar way to eq. (1): $\text{Dropout}(A) = A \odot B, b_{ij} \sim \mathcal{B}(\tau)$, where \odot is Hadamard product, and $\tau > 0$ is a dropout rate. Our probabilistic rounding is naturally equipped with noise injection, which is helpful to avoid bad local optima and over-fitting. Based on our observations, one can achieve better classification accuracy than nearest rounding in general.

5. Implementation and Benchmarks

We build our implementation on MP-SPDZ by Keller (2020). MP-SPDZ not only implements a range of MPC protocols, but also comes with a high-level library containing some of the building blocks in Appendix C. We have added the exponentiation in Section 2.2, the inverse square root computation in Appendix C, and all machine learning building blocks.

MP-SPDZ allows implementing the computation in Python code, which is then compiled into a specific bytecode. This code can be executed by a virtual machine performing the actual secure computation. The process allows optimizing the computation in the context of MPC.

The framework also features an emulator that executes the exact computation in the clear that could be done securely. This allowed us collecting the accuracy figures in the next section at a lower cost. Our implementation is licensed under a BSD-style license.

5.1. MNIST Classification

For a concrete measurement of accuracy and running times, we train a multi-class classifier³ for the widely-used MNIST dataset (LeCun et al., 2010). We work mainly with the models used by Wagh et al. (2019) with secure computation, and we reuse their numbering (A–D). The models contain up to four layers. Network C is a convolutional neural network that appeared in the seminal work of LeCun et al. (1998) whereas the others are simpler networks used by related literature on secure computation such as Mohassel & Zhang (2017), Liu et al. (2017), and Riazi et al. (2018). See Figures 5–8 in the Appendix for the exact network structures.

Figure 2 shows the learning curves for various quantization precisions and two rounding options, namely, nearest and

³Scripts are available at <https://github.com/csiro-mlai/deep-mpc>.

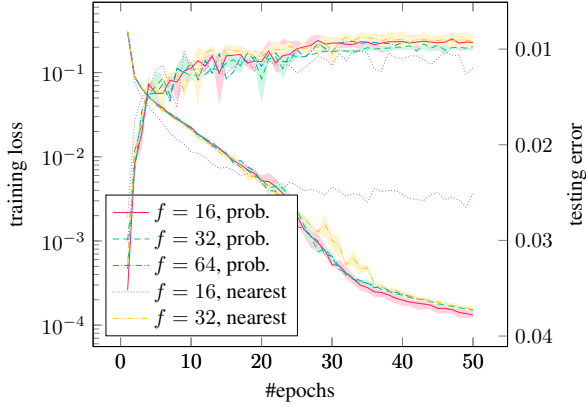


Figure 2: Loss and accuracy for LeNet and various precision options when running SGD with a learning rate of 0.01. Interval only available for $f = 32$ and $f = 16$ with probabilistic rounding.

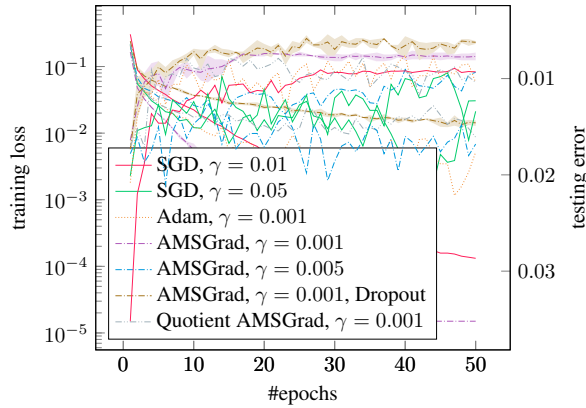


Figure 3: Loss and accuracy for LeNet with various optimizer options, $f = 16$, and probabilistic truncation. γ is the learning rate. Interval only available for AMSGrad with $\gamma = 0.0001$.

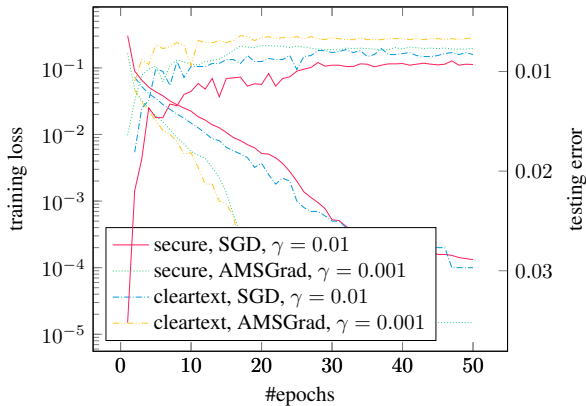


Figure 4: Comparison of cleartext training and secure training for LeNet with $f = 16$ and probabilistic truncation. γ is the learning rate.

probabilistic rounding. We use SGD with learning rate 0.01, batch size 128, and the usual MNIST training/test split. Most configurations perform similarly except for 16-bit precision with nearest rounding which gives worse results. We run the best-performing configurations several times. The range of the performance scores is indicated by the shaded area. For the rest of the paper, we focus on $f = 16$ with probabilistic rounding because it offers the best overall performance in terms of accuracy and efficiency.

The choice of $f = 16$ (and $k = 31$) is informed on one hand by the fact that it is close to the lower limit for reasonable results. The works listed in Table 3 use values in the range 13–20, and we found that $f = 8$ leads to divergence. On the other hand, Figure 2 shows that increasing the precision does not lead to an increase in accuracy. Finally, the division algorithm by [Catrina & Saxena \(2010\)](#) requires that k is roughly twice f , and that k is half the domain size (strictly less than half the domain size for efficiency). A domain size of 64 is natural given the 64-bit word size of common hardware. The choice of f and k follows from these constraints. Learning rate and minibatch size are set empirically based on satisfactory performance in plaintext training.

Figure 3 reports the results with a variety of optimizers. AMSGrad ([Reddi et al., 2018](#)) stands out in terms of convergence and final accuracy. [Agrawal et al. \(2019\)](#) suggest using normalized gradients in AMSGrad for training binary neural networks to improve performance. In our experiments using quantized weights with higher precision, similar improvements are not observed.

Furthermore, Figure 3 also shows that using a Dropout layer as described in Figure 7 in Appendix E only slightly improves the performance. This is possibly due to the fact that the reduced precision and probabilistic rounding already prevent overfitting to some degree.

Resources We run the emulator on AWS `c5.9xlarge` instances. One epoch takes a few seconds to several minutes depending on the model that is being trained. For all our experiments, we used a few weeks of computational time including experiments not presented here.

5.1.1. SECURE COMPUTATION

In order to verify our emulation results, we run LeNet with precision $f = 16$ and probabilistic rounding in our actual multi-party computation protocol. We could verify that it converges on 99.2% accuracy at 25 epochs, taking 3.6 hours (or one hour for 99% accuracy). Tables 3 and 4 compare our results to previous works in a LAN setting.

Note that [Wagh et al. \(2019\)](#) and [Wagh et al. \(2021\)](#) give accuracy figures. The authors have told us in personal com-

Secure Quantized Training for Deep Learning

Table 3: Benchmarks in the three-party LAN setting with one corruption. Accuracy N/A means that the accuracy figures were not given or computed in a way that does not reflect the secure computation. (*) Wagh et al. (2021) only implemented their online phase.

| Network | | Epoch time (s) | Comm. per epoch (GB) | Acc. (# epochs) | Precision (f) |
|---------|--------------------------|----------------|----------------------|-----------------|-------------------|
| A | Mohassel & Rindal (2018) | 180 | N/A | 94.0% (15) | N/A |
| | Wagh et al. (2019) | 247 | N/A | N/A | 13 |
| | Wagh et al. (2021)* | 41 | 3 | N/A | 13 |
| | Ours (SGD) | 31 | 26 | 97.8% (15) | 16 |
| | Ours (AMSGrad) | 88 | 139 | 98.1% (15) | 16 |
| B | Wagh et al. (2019) | 4,176 | N/A | N/A | 13 |
| | Wagh et al. (2021)* | 891 | 108 | N/A | 13 |
| | Ours (SGD) | 144 | 201 | 98.0% (15) | 16 |
| | Ours (AMSGrad) | 187 | 234 | 98.6% (15) | 16 |
| C | Wagh et al. (2019) | 7,188 | N/A | N/A | 13 |
| | Wagh et al. (2021)* | 1,412 | 162 | N/A | 13 |
| | Tan et al. (2021) | 1,036 | 534 | 94.0% (5) | 20 |
| | Knott et al. (2021) | 10,940 | N/A | 96.7% (4) | 16 |
| | Ours (SGD) | 343 | 352 | 98.5% (5) | 16 |
| | Ours (AMSGrad) | 513 | 765 | 99.0% (5) | 16 |
| D | Mohassel & Rindal (2018) | 234 | N/A | N/A | N/A |
| | Wagh et al. (2021)* | 101 | 11 | N/A | 13 |
| | Ours (SGD) | 41 | 41 | 98.1% (15) | 16 |
| | Ours (AMSGrad) | 95 | 137 | 98.5% (15) | 16 |

Table 4: Benchmarks in the two-party LAN setting. In the column “Epoch time (s)”, two numbers refer to online and offline time. Accuracy N/A means that the accuracy figures were not given or computed in a way that does not reflect the secure computation. Figures for Network C are estimates based on ten batch iterations.

| Network | | Epoch time (s) | Comm. per epoch (GB) | Acc. (# epochs) | Precision (f) |
|---------|-------------------------|----------------|----------------------|-----------------|-------------------|
| A | Mohassel & Zhang (2017) | 283/19,333 | N/A | 93.4% (15) | 13 |
| | Agrawal et al. (2019) | 31,392 | N/A | 95.0% (10) | N/A |
| | Ours (SGD) | 3,741 | 1,128 | 97.8% (15) | 16 |
| | Ours (AMSGrad) | 5,688 | 4,984 | 98.1% (15) | 16 |
| D | Ours (SGD) | 141,541 | 25,604 | 98.1% (15) | 16 |
| | Ours (AMSGrad) | 196,745 | 51,770 | 98.5% (15) | 16 |

Table 5: Benchmarks for Network C (LeNet) with AMSGrad across a range of security models in the LAN setting. Figures are mostly estimates based on ten batch iterations.

| Protocol | # parties | # corruptions | Malicious | Time per epoch (s) | Comm. per epoch (GB) |
|-----------------------|-----------|---------------|-----------|--------------------|----------------------|
| Appendix B | 2 | 1 | ✗ | 196,745 | 51,770 |
| Appendix A | 3 | 1 | ✗ | 513 | 765 |
| Dalskov et al. (2021) | 3 | 1 | ✓ | 4,961 | 9,101 |
| Appendix B | 3 | 2 | ✗ | 357,214 | 271,595 |
| Dalskov et al. (2021) | 4 | 1 | ✓ | 1,175 | 2,945 |
| Appendix B.2 | 10 | 1/8 | ✗ | 29,078 | 99,775 |
| Goyal et al. (2021) | 10 | 4 | ✗ | 129,667 | 434,138 |
| Appendix B | 10 | 9 | ✗ | 2,833,641 | 13,875,834 |

munication that their figures do not reflect the secure computation. This is related to the fact that they do not implement exponentiation and thus softmax but use the ReLU-based alternative, which has been found to perform poorly by Keller & Sun (2020). The same holds for Mohassel & Rindal (2018) and Mohassel & Zhang (2017). Agrawal et al. (2019) instead use ternary weights, which explains their reduced accuracy.

Tan et al. (2021) rely entirely on Taylor series approximation for exponentiation whereas we use approximation only for the fractional part of the input (Algorithm 2). This might explain their lower accuracy.

The figures for CrypTen (Knott et al., 2021) were obtained by running an adaption of their `mpc_autograd_cnn` example to the full LeNet with SGD and learning rate 0.01. We found that it would eventually diverge even with a relatively low learning rate of 0.001. The given accuracy is the highest we achieved before divergence. We estimate that this is due to the approximations used in CrypTen. It uses the following approximation for the exponential function:

$$(1 + x/2^n)^{2^n}.$$

According to the CrypTen code⁴, n is set to 9. Furthermore, they use a fixed-point precision of 16. This leads to a relatively low precision. For $x = -4$, their approximation is 0.018030 when rounding to the nearest multiple of 2^{-16} at every step. In contrast, the correct value is 0.018315, a relative error of more than one percent. In comparison, our solution produces values in the range $[0.018310, 0.018325]$, a relative error of less than 10^{-3} and an absolute error of less than the representation step of 2^{-16} .

There are a number of factors that make it hard to compare the performance. Wagh et al. (2021) have only implemented their online phase, which makes their figures incomplete. The implementation of Knott et al. (2021) is purely in Python whereas all others use C/C++. Finally, we rely on figures obtained using different platforms except for Knott et al. (2021). In general, it is likely that MP-SPDZ benefits from being an established framework that has undergone more work than other code bases.

Further security models MP-SPDZ supports a wide range of security models, that is, choices for the number of (corrupted) parties and the nature of their corruption, i.e., whether they are assumed to follow the protocol or not. Table 5 shows our results across a range of security models.

5.1.2. COMPARISON TO CLEARTEXT TRAINING

Figure 4 compares the performance of our secure training with cleartext training in TensorFlow. It shows that secure

⁴<https://github.com/facebookresearch/CrypTen>

training performs only slightly worse with the same optimizer.

5.2. CIFAR-10 Classification

In order to highlight the generality of our approach, we have implemented training for CIFAR-10 with a network by Wagh et al. (2021), which is based on AlexNet (Krizhevsky et al., 2017). See Figure 9 in Appendix E for details. Unlike the networks for MNIST, the network for CIFAR-10 uses batch normalization (Ioffe & Szegedy, 2015). Table 6 shows our results. Our results are not comparable with Wagh et al. (2021) because they do not implement the full training. Furthermore, Tan et al. (2021) do not provide accuracy figures for training from scratch, which makes it unclear how their less accurate approximations perform in this setting. Figure 12 in Appendix G shows that our secret training comes within a few percentage points of cleartext training. Similar figures for MPC training on CIFAR-10 are not widely seen in the literature.

Table 6: Time (seconds) and communication (GB) per epoch, accuracy after ten epochs, and fixed-point precision for CIFAR-10 training in the three-party LAN setting with one corruption. Tan et al. (2021) do not train from scratch, which is why we do not include their accuracy figure. (*)Wagh et al. (2021) only implemented their online phase.

| Network | s/ep. | GB/ep. | Acc. | f |
|---------------------|-------|--------|-------|-----|
| Wagh et al. (2021)* | 3,156 | 80 | N/A | 13 |
| Tan et al. (2021) | 1,137 | 535 | N/A | 20 |
| Ours (SGD) | 1,603 | 771 | 64.9% | 16 |
| Ours (Adam) | 2,431 | 3,317 | 64.7% | 16 |
| Ours (AMSGrad) | 2,473 | 3,285 | 63.6% | 16 |

6. Conclusions

We implement deep neural network training purely in multi-party computation, and we present extensive results of convolutional neural networks on benchmark datasets. We find that the low precision of MPC computation increases the error slightly. We only consider one particular implementation of division and exponentiation, which are crucial to the learning process as part of softmax. Future work might consider different approximations of these building blocks.

References

- Agrawal, N., Shamsabadi, A. S., Kusner, M. J., and Gascón, A. QUOTIENT: Two-party secure neural network training and prediction. In Cavallaro, L., Kinder, J., Wang, X., and Katz, J. (eds.), *ACM CCS 2019*, pp. 1231–1247. ACM Press, November 2019. doi: 10.1145/3319535.3339819.

- Aliasgari, M., Blanton, M., Zhang, Y., and Steele, A. Secure computation on floating point numbers. In *NDSS 2013*. The Internet Society, February 2013.
- Aly, A. and Smart, N. P. Benchmarking privacy preserving scientific operations. In Deng, R. H., Gauthier-Umaña, V., Ochoa, M., and Yung, M. (eds.), *ACNS 19*, volume 11464 of *LNCS*, pp. 509–529. Springer, Heidelberg, June 2019. doi: 10.1007/978-3-030-21568-2_25.
- Araki, T., Furukawa, J., Lindell, Y., Nof, A., and Ohara, K. High-throughput semi-honest secure three-party computation with an honest majority. In Weippl, E. R., Katzenbeisser, S., Kruegel, C., Myers, A. C., and Halevi, S. (eds.), *ACM CCS 2016*, pp. 805–817. ACM Press, October 2016. doi: 10.1145/2976749.2978331.
- Araki, T., Barak, A., Furukawa, J., Keller, M., Lindell, Y., Ohara, K., and Tsuchida, H. Generalizing the SPDZ compiler for other protocols. In Lie, D., Mannan, M., Backes, M., and Wang, X. (eds.), *ACM CCS 2018*, pp. 880–895. ACM Press, October 2018. doi: 10.1145/3243734.3243854.
- Barni, M., Orlandi, C., and Piva, A. A privacy-preserving protocol for neural-network-based computation. In *Proceedings of the 8th workshop on Multimedia and security*, pp. 146–151, 2006.
- Beaver, D. Efficient multiparty protocols using circuit randomization. In Feigenbaum, J. (ed.), *CRYPTO’91*, volume 576 of *LNCS*, pp. 420–432. Springer, Heidelberg, August 1992. doi: 10.1007/3-540-46766-1_34.
- Benaloh, J. C. and Leichter, J. Generalized secret sharing and monotone functions. In Goldwasser, S. (ed.), *CRYPTO’88*, volume 403 of *LNCS*, pp. 27–35. Springer, Heidelberg, August 1990. doi: 10.1007/0-387-34799-2_3.
- Catrina, O. and Saxena, A. Secure computation with fixed-point numbers. In Sion, R. (ed.), *FC 2010*, volume 6052 of *LNCS*, pp. 35–50. Springer, Heidelberg, January 2010.
- Cock, M. d., Dowsley, R., Nascimento, A. C., and Newman, S. C. Fast, privacy preserving linear regression over distributed datasets based on pre-distributed data. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, AISEC ’15, pp. 3–14, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338264. doi: 10.1145/2808769.2808774.
- Cramer, R., Damgård, I., and Nielsen, J. B. Multiparty computation from threshold homomorphic encryption. In Pfitzmann, B. (ed.), *EUROCRYPT 2001*, volume 2045 of *LNCS*, pp. 280–299. Springer, Heidelberg, May 2001. doi: 10.1007/3-540-44987-6_18.
- Dahl, M., Mancuso, J., Dupis, Y., Decoste, B., Giraud, M., Livingstone, I., Patriquin, J., and Uhma, G. Private machine learning in TensorFlow using secure computation. *CoRR*, abs/1810.08130, 2018.
- Dalskov, A., Escudero, D., and Keller, M. Fantastic four: Honest-majority four-party secure computation with malicious security. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- Dalskov, A. P. K., Escudero, D., and Keller, M. Secure evaluation of quantized neural networks. *PoPETs*, 2020(4): 355–375, October 2020. doi: 10.2478/popets-2020-0077.
- Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J. B., and Toft, T. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Halevi, S. and Rabin, T. (eds.), *TCC 2006*, volume 3876 of *LNCS*, pp. 285–304. Springer, Heidelberg, March 2006. doi: 10.1007/11681878_15.
- Damgård, I., Pastro, V., Smart, N. P., and Zakarias, S. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini, R. and Canetti, R. (eds.), *CRYPTO 2012*, volume 7417 of *LNCS*, pp. 643–662. Springer, Heidelberg, August 2012. doi: 10.1007/978-3-642-32009-5_38.
- Damgård, I., Escudero, D., Frederiksen, T. K., Keller, M., Scholl, P., and Volgushev, N. New primitives for actively-secure MPC over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy*, pp. 1102–1120. IEEE Computer Society Press, May 2019. doi: 10.1109/SP.2019.00078.
- Demmler, D., Schneider, T., and Zohner, M. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015.
- Eerikson, H., Keller, M., Orlandi, C., Pullonen, P., Puura, J., and Simkin, M. Use your brain! Arithmetic 3PC for any modulus with active security. In Kalai, Y. T., Smith, A. D., and Wichs, D. (eds.), *ITC 2020*, pp. 5:1–5:24. Schloss Dagstuhl, June 2020. doi: 10.4230/LIPIcs.ITC.2020.5.
- Escudero, D., Ghosh, S., Keller, M., Rachuri, R., and Scholl, P. Improved primitives for MPC over mixed arithmetic-binary circuits. In Micciancio, D. and Ristenpart, T. (eds.), *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pp. 823–852. Springer, Heidelberg, August 2020. doi: 10.1007/978-3-030-56880-1_29.
- Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics (AISTATS)*, pp. 249–256, 2010.

- Goldschmidt, R. E. Applications of division by convergence. Master's thesis, MIT, 1964.
- Goyal, V., Li, H., Ostrovsky, R., Polychroniadou, A., and Song, Y. ATLAS: Efficient and scalable MPC in the honest majority setting. In Malkin, T. and Peikert, C. (eds.), *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pp. 244–274, Virtual Event, August 2021. Springer, Heidelberg. doi: 10.1007/978-3-030-84245-1_9.
- Halevi, S. and Shoup, V. Algorithms in HElib. In Garay, J. A. and Gennaro, R. (eds.), *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pp. 554–571. Springer, Heidelberg, August 2014. doi: 10.1007/978-3-662-44371-2_31.
- Hart, J. F. *Computer approximations*. Krieger Publishing Co., Inc., 1978.
- Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 29. Curran Associates, Inc., 2016.
- Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Bach, F. and Blei, D. (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 448–456, Lille, France, 2015. PMLR.
- Juvekar, C., Vaikuntanathan, V., and Chandrakasan, A. GAZELLE: A low latency framework for secure neural network inference. In Enck, W. and Felt, A. P. (eds.), *USENIX Security 2018*, pp. 1651–1669. USENIX Association, August 2018.
- Keller, M. MP-SPDZ: A versatile framework for multi-party computation. In Ligatti, J., Ou, X., Katz, J., and Vigna, G. (eds.), *ACM CCS 2020*, pp. 1575–1590. ACM Press, November 2020. doi: 10.1145/3372297.3417872.
- Keller, M. and Sun, K. Effectiveness of MPC-friendly softmax replacement. In *Privacy Preserving Machine Learning - PriML and PPML Joint Edition (NeurIPS 2020 workshop)*, 2020.
- Keller, M., Pastro, V., and Rotaru, D. Overdrive: Making SPDZ great again. In Nielsen, J. B. and Rijmen, V. (eds.), *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pp. 158–189. Springer, Heidelberg, April / May 2018. doi: 10.1007/978-3-319-78372-7_6.
- Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.
- Knott, B., Venkataraman, S., Hannun, A., Sengupta, S., Ibrahim, M., and van der Maaten, L. CrypTen: Secure multi-party computation meets machine learning. In Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, 2021.
- Kolesnikov, V., Sadeghi, A.-R., and Schneider, T. A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design. *Journal of Computer Security*, 21(2):283–315, 2013.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, may 2017.
- LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- LeCun, Y., Cortes, C., and Burges, C. MNIST handwritten digit database. *ATT Labs [Online]*. Available: <http://yann.lecun.com/exdb/mnist>, 2, 2010. Creative Commons Attribution-Share Alike 3.0 license, <https://creativecommons.org/licenses/by-sa/3.0/>.
- Li, H., De, S., Xu, Z., Studer, C., Samet, H., and Goldstein, T. Training quantized nets: A deeper understanding. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30. Curran Associates, Inc., 2017.
- Lin, D., Talathi, S., and Annapureddy, S. Fixed point quantization of deep convolutional networks. In Balcan, M. F. and Weinberger, K. Q. (eds.), *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pp. 2849–2858, New York, New York, USA, 20–22 Jun 2016. PMLR.
- Liu, J., Juuti, M., Lu, Y., and Asokan, N. Oblivious neural network predictions via MiniONN transformations. In Thuraisingham, B. M., Evans, D., Malkin, T., and Xu, D. (eds.), *ACM CCS 2017*, pp. 619–631. ACM Press, October / November 2017. doi: 10.1145/3133956.3134056.
- Lou, Q., Feng, B., Fox, G. C., and Jiang, L. Glyph: Fast and accurately training deep neural networks on encrypted data. In *Proceedings of the 34th International Conference on Neural Information Processing Systems, NIPS'20*, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.
- Lu, W.-j., Fang, Y., Huang, Z., Hong, C., Chen, C., Qu, H., Zhou, Y., and Ren, K. Faster secure multiparty computation of adaptive gradient descent. In *Proceedings of the*

- 2020 Workshop on Privacy-Preserving Machine Learning in Practice, PPMLP'20, pp. 47–49, 2020.
- Mishra, P., Lehmkuhl, R., Srinivasan, A., Zheng, W., and Popa, R. A. Delphi: A cryptographic inference service for neural networks. In Capkun, S. and Roesner, F. (eds.), *USENIX Security 2020*, pp. 2505–2522. USENIX Association, August 2020.
- Mohassel, P. and Rindal, P. ABY³: A mixed protocol framework for machine learning. In Lie, D., Mannan, M., Backes, M., and Wang, X. (eds.), *ACM CCS 2018*, pp. 35–52. ACM Press, October 2018. doi: 10.1145/3243734.3243760.
- Mohassel, P. and Zhang, Y. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pp. 19–38. IEEE Computer Society Press, May 2017. doi: 10.1109/SP.2017.12.
- Nair, V. and Hinton, G. E. Rectified linear units improve Restricted Boltzmann machines. In *International Conference on Machine Learning (ICML)*, pp. 807–814, 2010.
- Quoc, D. L., Gregor, F., Arnautov, S., Kunkel, R., Bhatotia, P., and Fetzer, C. secureTF: A secure TensorFlow framework. *CoRR*, abs/2101.08204, 2021.
- Rathee, D., Rathee, M., Kumar, N., Chandran, N., Gupta, D., Rastogi, A., and Sharma, R. CrypTFlow2: Practical 2-party secure inference. In Ligatti, J., Ou, X., Katz, J., and Vigna, G. (eds.), *ACM CCS 2020*, pp. 325–342. ACM Press, November 2020. doi: 10.1145/3372297.3417274.
- Reddi, S. J., Kale, S., and Kumar, S. On the convergence of Adam and beyond. In *International Conference on Learning Representations (ICLR)*, 2018.
- Riazi, M. S., Weinert, C., Tkachenko, O., Songhori, E. M., Schneider, T., and Koushanfar, F. Chameleon: A hybrid secure computation framework for machine learning applications. In Kim, J., Ahn, G.-J., Kim, S., Kim, Y., López, J., and Kim, T. (eds.), *ASIACCS 18*, pp. 707–721. ACM Press, April 2018.
- Rotaru, D. and Wood, T. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In Hao, F., Ruj, S., and Sen Gupta, S. (eds.), *INDOCRYPT 2019*, volume 11898 of *LNCS*, pp. 227–249. Springer, Heidelberg, December 2019. doi: 10.1007/978-3-030-35423-7_12.
- Ryffel, T., Trask, A., Dahl, M., Wagner, B., Mancuso, J., Rueckert, D., and Passerat-Palmbach, J. A generic framework for privacy preserving deep learning. In *Privacy Preserving Machine Learning (NeurIPS 2018 Workshop)*, 2018.
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research (JMLR)*, 15:1929–1958, 2014.
- Tan, S., Knott, B., Tian, Y., and Wu, D. J. CryptGPU: Fast privacy-preserving machine learning on the GPU. In *IEEE Symposium on Security and Privacy*, 2021.
- Wagh, S., Gupta, D., and Chandran, N. SecureNN: 3-party secure computation for neural network training. *PoPETs*, 2019(3):26–49, July 2019. doi: 10.2478/popets-2019-0035.
- Wagh, S., Tople, S., Benhamouda, F., Kushilevitz, E., Mittal, P., and Rabin, T. Falcon: Honest-majority maliciously secure framework for private deep learning. *PoPETs*, 2021(1):188–208, January 2021. doi: 10.2478/popets-2021-0011.
- Wang, N., Choi, J., Brand, D., Chen, C.-Y., and Gopalakrishnan, K. Training deep neural networks with 8-bit floating point numbers. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.
- Xiao, H., Rasul, K., and Vollgraf, R. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017.

A. An Efficient Secure Three-Party Computation Protocol

There is a wide range of MPC protocols with a variety of security properties (see Keller (2020) for an overview). In this section, we focus on the setting of *three-party computation with one semi-honest corruption*. This means that, out of the three parties, two are expected to behave honestly, i.e., they follow the protocol and keep their view of the protocol secret, and one party is expected to follow the protocol but might try to extract information from their view. The reason for choosing this setting is that it allows for an efficient MPC protocol while still providing secure outsourced computation. The protocol does not easily generalize to any other setting. However, protocols exist for any number of parties, see Keller for an overview. The concrete protocol we use goes back to Benaloh & Leichter (1990) with further aspects by Araki et al. (2016), Mohassel & Rindal (2018), and Eerikson et al. (2020). We summarize the core protocol below. The mathematical building blocks in the next section mostly use the aspects below.

Secret sharing The simplest variant of secure computation is only data-oblivious, that is, the participants are aware of the nature of the computation (addition, multiplication, etc.) but not the values being operated on. In the context of machine learning, this means that they know the hyper-parameters such as the layers but not the sample data or the neural network weights. All of these values in our protocol are stored using replicated secret sharing. A secret value x is represented as a random sum $x = x_0 + x_1 + x_2$, and party P_i holds (x_{i-1}, x_{i+1}) where the indices are computed modulo three. Clearly, each party is missing one value to compute the sum. On the other hand, each pair of parties hold all necessary to reconstruct the secret. For a uniformly random generation of shares, the computation domain has to be finite. Most commonly, this domain is defined by integer computation modulo a number. We use 2^k for k being a multiple of 64 as well as 2 as the moduli. The first case corresponds to an extension of 64-bit arithmetic found on most processors. We will refer to the two settings as arithmetic and binary secret sharing throughout the paper.

Input sharing The secret sharing scheme implies a protocol to share inputs where the inputting party samples the shares and distributes them accordingly. Eerikson et al. (2020) propose a more efficient protocol where the inputting party only needs to send one value instead of two pairs of values. If P_i would like to input x , x_i is set to zero, and x_{i-1} is generated with a pseudo-random generator using a key previously shared between P_i and P_{i+1} . P_i can compute $x_{i+1} = x - x_{i-1}$ and send it to P_{i-1} . While the resulting secret sharing is not entirely random, the fact that P_i already knows x makes randomizing x_i obsolete.

Addition, subtraction, and scalar multiplication The commutative nature of addition allows to add secret sharings without communication. More concretely, secret sharings $x = x_0 + x_1 + x_2$ and $y = y_0 + y_1 + y_2$ imply the secret sharing $x + y = (x_0 + y_0) + (x_1 + y_1) + (x_2 + y_2)$. The same works for subtraction. Furthermore, the secret sharing $x = x_0 + x_1 + x_2$ allows to compute $\lambda x = \lambda x_0 + \lambda x_1 + \lambda x_2$ locally.

Multiplication The product of $x = x_0 + x_1 + x_2$ and $y = y_0 + y_1 + y_2$ is

$$\begin{aligned} x \cdot y &= (x_0 + x_1 + x_2) \cdot (y_0 + y_1 + y_2) \\ &= (x_0y_0 + x_0y_1 + x_1y_0) + (x_1y_1 + x_1y_2 + x_2y_1) + (x_2y_2 + x_2y_0 + x_0y_2). \end{aligned}$$

On the right-hand side (RHS), each term in parentheses only contains shares known by one of the parties. They can thus compute an additive secret sharing (one summand per party) of the product. However, every party only holding one share does not satisfy the replication requirement for further multiplications. It is not secure for every party to pass their value on to another party because the summands are not distributed randomly. This can be fixed by rerandomization: Let $xy = z_0 + z_1 + z_2$ where z_i is known to P_i . Every party P_i computes $z'_i = z_i + r_{i,i+1} - r_{i-1,i}$ where $r_{i,i+1}$ is generated with a pseudo-random generator using a key pre-shared between P_i and P_{i+1} . The resulting sum $xy = z'_0 + z'_1 + z'_2$ is pseudo-random, and it is thus secure for P_i to send z'_i to P_{i+1} in order to create a replicated secret sharing $((xy)_{i-1}, (xy)_{i+1}) = (z'_i, z'_{i-1})$.

Domain conversion Recall that we use computation modulo 2^k for k being a multiple of 64 as well as 1. Given that the main operations are just addition and multiplication in the respective domain, it is desirable to compute integer arithmetic in the large domain but operations with a straight-forward binary circuit modulo two. There has been a long-running interest in this going back to least Kolesnikov et al. (2013). We mainly rely on the approach proposed by Mohassel & Rindal (2018) and Araki et al. (2018). Recall that $x \in 2^k$ is shared as $x = x_0 + x_1 + x_2$. Now let $\{x_0^{(i)}\}_{i=0}^{k-1}$ the bit decomposition of x_0 ,

that is, $x_0^{(i)} \in \{0, 1\}$ and $x_0 = \sum_{i=0}^{k-1} x_0^{(i)} 2^i$. It is self-evident that $x_0^{(i)} = x_0^{(i)} + 0 + 0$ is a valid secret sharing modulo two (albeit not a random one). Furthermore, every party holding x_0 can generate $x_0^{(i)}$. It is therefore possible for the parties to generate a secret sharing modulo two of a *single share* modulo 2^k . Repeating this for all shares and computing the addition as a binary circuit allow the parties to generate a secret sharing modulo two from a secret sharing modulo 2^k . Conversion in the other direction can be achieved using a similar technique or using “daBits” described by [Rotaru & Wood \(2019\)](#). In the following, we will use the term mixed-circuit computation for any technique that works over both computation domains.

Dot products Dot products are an essential building block of linear computation such as matrix multiplication. In light of quantization, it is possible to reduce the usage of truncation by deferring it to after the summation. In other words, the dot product in the integer representations is computed before truncating. This not only reduces the truncation error, but also is more efficient because the truncation is the most expensive part in quantized secure multiplication. Similarly, our protocol allows to defer the communication needed for multiplication. Let \vec{x} and \vec{y} be two vectors where the elements are secret-shared, that is, $\{x^{(i)}\} = x_0^{(i)} + x_1^{(i)} + x_2^{(i)}$ and similarly for $y^{(i)}$. The inner product then is

$$\begin{aligned} \sum_i x^{(i)} \cdot y^{(i)} &= \sum_i (x_0^{(i)} + x_1^{(i)} + x_2^{(i)}) \cdot (y_0^{(i)} + y_1^{(i)} + y_2^{(i)}) \\ &= \sum_i (x_0^{(i)} y_0^{(i)} + x_0^{(i)} y_1^{(i)} + x_1^{(i)} y_0^{(i)}) + \sum_i (x_1^{(i)} y_1^{(i)} + x_1^{(i)} y_2^{(i)} + x_2^{(i)} y_1^{(i)}) \\ &\quad + \sum_i (x_2^{(i)} y_2^{(i)} + x_2^{(i)} y_0^{(i)} + x_0^{(i)} y_2^{(i)}). \end{aligned}$$

The three sums in the last term can be computed locally by one party each before applying the same protocol as for a single multiplication.

Comparisons Arithmetic secret sharing does not allow to access the individual bits directly. It is therefore not straightforward to compute comparisons such as “less than”. There is a long line of literature on how to achieve this going back to at least [Damgård et al. \(2006\)](#). More recently, most attention has been given to combining the power of arithmetic and binary secret sharing in order to combine the best of worlds. One possibility to do so is to plainly convert to the binary domain and compute the comparison circuit there. In our concrete implementation, we use the more efficient approach by [Mohassel & Rindal \(2018\)](#). It starts by taking the difference between the two inputs. Computing the comparison then reduces to comparing to zero, which in turn is equivalent to extracting the most significant bit as it indicates the sign. The latter is achieved by converting the shares locally to bit-wise sharing of the arithmetic shares, which sum up to the secret value. It remains to compute the sum of the binary shares in order to come up with the most significant bit.

Shifting and truncation [Dalskov et al. \(2021\)](#) present an efficient implementation of the deterministic truncation using mixed-circuit computation, and [Dalskov et al. \(2020\)](#) present an efficient protocol for the probabilistic truncation.

The probabilistic truncation involves the truncation of a randomized value, that is the computation of $\lfloor (x + r)/2^f \rfloor$ for a random f -bit value r . It is easy to see that

$$\lfloor (x + r)/2^f \rfloor = \begin{cases} \lfloor x/2^f \rfloor & \text{if } (x \bmod 2^f + r) < 2^f \\ \lfloor x/2^f \rfloor + 1 & \text{otherwise.} \end{cases}$$

Therefore, the larger $(x \bmod 2^f)$ is, the more likely the latter condition is true.

B. An Efficient Multi-Party Computation Protocol Based on Homomorphic Encryption

While the protocol in the previous is very efficient, the multiplication only works with an honest majority, that is, the number of corrupted parties is strictly less than half of the total number. Multiplications with more corruptions than that requires more involved cryptographic schemes. In this section, we present a semi-honest protocol based on homomorphic encryption that tolerates $n - 1$ corrupted parties out of n .

Secret sharing The simplest secret sharing is additive secret sharing, which can be used in this setting. The share of party i is a random element x_i of the relevant domain such that the sum of all elements is the secret.

Input sharing The canonical input sharing would be for the inputting party to generate random shares such that they add up to the input and then distribute the shares. Using a one-time setup, it is possible to share without communication as shown in Algorithm 4. It is easy to see that the resulting sharing is correct and that the unknown shares are indistinguishable to random for an adversary corrupting any number of parties.

Algorithm 4 Additive input sharing without communication.

Setup: PRG key K_{ij} known only to parties P_i and P_j

Input: Party P_i has input x .

Output: $\langle x \rangle^A$

- 1: Party P_i sets $x_i \leftarrow x - \sum_{j \neq i} \text{PRG}_{K_{ij}}()$.
 - 2: For all $j \neq i$, party P_j sets $x_j \leftarrow \text{PRG}_{K_{ij}}()$.
-

Addition, subtraction, and scalar multiplication As with any linear secret sharing scheme, these operation can trivially be computed locally.

Multiplication We use Beaver’s technique (Beaver, 1992). Assume that a and b are random numbers in the domain, and $c = ab$. Then, for any x, y , it holds that

$$\begin{aligned} xy &= (x + a - a) \cdot (y + b - b) \\ &= (x + a) \cdot (y + b) - (y + b) \cdot a - (x + a) \cdot b + c. \end{aligned}$$

As a and b are random, they can be used to mask x and y , respectively. Once $x + a$ and $y + b$ have been revealed, the operation becomes linear in a and b :

$$[xy] = (x + a) \cdot (y + b) - (y + b) \cdot [a] - (x + a) \cdot [b] + [c]$$

It remains to produce a fresh triple (a, b, c) for every multiplication. Keller et al. (2018) show how to do this using semi-homomorphic encryption based on Learning With Errors (LWE), a widely used cryptographic assumption. They make optimal use of the SIMD (single instruction, multiple data) nature of LWE-based encryption. Their protocol involves pair-wise communication between parties, which scale quadratically in the number of parties. In the following section, we propose a protocol that scales better, which is inspired by Cramer et al. (2001).

Matrix multiplication Cock et al. (2015) have extended Beaver’s technique to matrix multiplication by replacing x, a with $m \times n$ and y, b with $n \times p$ matrices. Using the diagonal packing by Halevi & Shoup (2014) allows to implement the matrix triple generation with $O(m(n + l))$ communication instead of the naïve $O(mnl)$. Unlike Halevi and Shoup, we only use the semi-homomorphic encryption, that is, only public-private multiplications. This avoids the need for cycling or shifting because we can simply generate the necessary cleartexts as needed.

Domain conversion daBits (Rotaru & Wood, 2019) and edaBits (Escudero et al., 2020) are a generic method to convert between integer and binary computation. For the special case of semi-honest two-party computation, Demmler et al. (2015) have presented a more efficient protocol similar to the one in the previous section.

Comparison, shifting, and truncation Escudero et al. (2020) have shown how to use edaBits to implement these computations. Essentially, they make minimal use of binary computation before switching back to integer computation so that the result is available for further computation in the integer domain.

B.1. Linear-cost Triple Generation with Semi-homomorphic Encryption

Assume an encryption system that allows to multiply a ciphertext and a plaintext, that is, $\text{Dec}(\text{Enc}(\mathbf{a}) \cdot \mathbf{b}) = \mathbf{a} * \mathbf{b}$ where \mathbf{a} and \mathbf{b} are vectors and $*$ denotes the Schur product. LWE allows the efficient creation of such a system. Furthermore, assume a distributed key setup as used by Damgård et al. (2012). Since we only want to achieve semi-honest security, it is straight-forward to generate without using any protocol to check the correctness. Algorithm 5 show the details of our protocol.

Algorithm 5 Triple Generation with distributed semi-homomorphic encryption

Input: Distributed key setup

Output: Random multiplication triples

- 1: Each party P_i samples \mathbf{a}_i and send $\text{Enc}(\mathbf{a}_i)$ to P_1 .
 - 2: P_1 computes $C_a = \sum_i \text{Enc}(\mathbf{a}_i)$ and broadcasts it.
 - 3: Each P_i samples \mathbf{b}_i and sends $C_i = C_a * \mathbf{b}_i + \text{Enc}(0)$ to P_1 .
 - 4: P_1 computes $C_c = \sum_i C_i$ and broadcasts it.
 - 5: The parties run distributed the decryption to obtain the share \mathbf{c}_i .
-

By definition,

$$\begin{aligned}
 \sum \mathbf{c}_i &= \text{Dec}(C_c) \\
 &= \text{Dec}\left(\sum_i C_a * \mathbf{b}_i + \text{Enc}(0)\right) \\
 &= \text{Dec}\left(C_a * \sum_i \mathbf{b}_i\right) \\
 &= \text{Dec}\left(\sum_i \text{Enc}(\mathbf{a}_i) * \sum_i \mathbf{b}_i\right) \\
 &= \left(\sum_i \mathbf{a}_i\right) * \left(\sum_i \mathbf{b}_i\right).
 \end{aligned}$$

This proves correctness. Furthermore, the distributed key setup hides encrypted values from all parties, and the addition of $\text{Enc}(0)$ in step 3 ensures that P_1 only receives a fresh encryption of $\mathbf{a} * \mathbf{b}_i$.

B.2. Replacing Homomorphic Encryption by a Dealer

It is possible to generate all correlated randomness (triples, random bits, (e)daBits etc.) by one party. The security then relies on the fact that either the dealer is trusted or all of the remaining parties. The offline phase simply consists of the dealer generating the necessary randomness (for example, a random triple (a, b, ab)) and then sending an additive secret sharing to the remaining parties.

C. High-Level Secure Computation Building Blocks

In this section, we will discuss how to implement computation with MPC with a focus on how it differs from computation on CPUs or GPUs. Most of the techniques below are already known individually. To the best of our knowledge, however, we are the first to put them together in an efficient and extensible framework for secure training of deep neural networks.

Oblivious Selection Plain secure computation does not allow branching because the parties would need to be aware of which branch is followed. Conditional assignment can be implemented as follows. If $b \in \{0, 1\}$ denotes the condition, $x + b \cdot (y - x)$ is either x or y depending on b . If the condition is available in binary secret sharing but x and y in arithmetic secret sharing, b has to be converted to the latter. This can be done using a daBit (Rotaru & Wood, 2019), which is a secret random bit shared both in arithmetic and binary. It allows to mask a bit in one world by XORing it. The result is then revealed and the masking is undone in the other world.

Division Catrina & Saxena (2010) show how to implement quantized division using the algorithm by Goldschmidt (1964). It mainly uses arithmetic and the probabilistic truncation already explained. In addition, the initial approximation requires a full bit decomposition as described above. The error of the output depends on the error in the multiplications used for Goldschmidt’s iteration, which compounds in particular when using probabilistic truncation. Due to the nature of secure computation, the result of division by zero is undefined. One could obtain a secret failure bit by testing the divisor to zero. However, that is unnecessary in our algorithm, because so far we only use division by secret value for the softmax function, where the divisor is guaranteed to strictly positive.

Algorithm 6 Separation (Sep) (Lu et al., 2020)

Input: Secret share $\langle \tilde{x}; f \rangle^A$ where $2^{-f+1} \leq \tilde{x} \leq 2^{f-1}$.

Output: $\langle \tilde{u}; f \rangle^A$ and $\langle z_0 \rangle^B, \dots, \langle z_{k-1} \rangle^B$ such that $z_{e+f} = 1$, $z_i = 0$ for $i \neq e + f$, and $\tilde{u} = \tilde{x}^{-1} \cdot 2^{e+1} \in [0.25, 0.5)$.

- 1: $\langle z_0 \rangle^B, \dots, \langle z_{k-1} \rangle^B \leftarrow \text{NP2}(\langle \tilde{x}; f \rangle^A)$
 - 2: $\langle 2^{e-1}; f \rangle^A \leftarrow \text{B2A}(\langle z_{2f-1} \rangle^B, \dots, \langle z_0 \rangle^B)$
 - 3: $\langle \tilde{u}; f \rangle^A \leftarrow \langle \tilde{x}; f \rangle^A \cdot \langle 2^{e-1}; f \rangle^A$
 - 4: **return** $\langle \tilde{u}; f \rangle^A, (\langle z_0 \rangle^B, \dots, \langle z_{k-1} \rangle^B)$
-

Algorithm 7 Square-root compensation (SqrtComp)

Input: Secret share $\langle z_0 \rangle^B, \dots, \langle z_{k-1} \rangle^B$ such that $z_{e+f} = 1$, $z_i = 0$ for $i \neq e + f$

Output: $\langle 2^{-(e-1)/2}; f \rangle^A$

- 1: $k' \leftarrow k/2$, $f' \leftarrow f/2$, $c_0 \leftarrow 2^{f/2+1}$, $c_1 = 2^{(f+1)/2+1}$
 - 2: **for** $i = 0, \dots, k'$ **do**
 - 3: $\langle a_i \rangle^B \leftarrow \langle z_{2i} \rangle^B \vee \langle z_{2i+1} \rangle^B$ {only $a_{e'} = 1$ for $e' = \lfloor (e+f)/2 \rfloor$ }
 - 4: **end for**
 - 5: $\langle 2^{-e'-2}; f \rangle^A \leftarrow \text{B2A}(\langle a_{2f'-1} \rangle^B, \dots, \langle a_0 \rangle^B)$
 - 6: $\langle b \rangle^B = \bigoplus_{i=0}^{k'-1} \langle z_{2i} \rangle^B$ { $b = \text{LSB}(e+f)$ }
 - 7: **return** $\langle 2^{-(e-1)/2}; f \rangle^A \leftarrow \text{MUX}(c_0, c_1, \langle b \rangle^B) \cdot \langle 2^{-e'-2}; f \rangle^A$
-

Algorithm 8 InvertSqrt (Lu et al., 2020)

Input: Share $\langle \tilde{x}; f \rangle^A$ where $2^{-f+1} \leq \tilde{x} \leq 2^{f-1}$ where $2^{e-1} \leq \tilde{x} \leq 2^e$ for some $e \in \mathbb{Z}$

Output: Share $\langle \tilde{y}; f \rangle^A$ such $\tilde{y} \approx 1/\sqrt{\tilde{x}}$

- 1: $\langle \tilde{u}; f \rangle^A, \langle z_0 \rangle^B, \dots, \langle z_{k-1} \rangle^B \leftarrow \text{Sep}(\langle \tilde{x}; f \rangle^A) \{z_{e+f} = 1\}$
 - 2: $\langle \tilde{c}; f \rangle^A \leftarrow 3.14736 + \langle \tilde{u}; f \rangle^A \cdot (4.63887 \cdot \langle \tilde{u}; f \rangle^A - 5.77789)$
 - 3: $\langle 2^{-(e-1)/2}; f \rangle^A \leftarrow \text{SqrtComp}(\langle z_0 \rangle^B, \dots, \langle z_{k-1} \rangle^B)$
 - 4: **return** $\langle \tilde{c}; f \rangle^A \cdot \langle 2^{-(e-1)/2}; f \rangle^A$
-

Algorithm 9 Procedures in Algorithms 6–8

$\langle x_0 \rangle^B, \dots, \langle x_{k-1} \rangle^B \leftarrow \text{NP2}(\langle x; f \rangle^A)$ Next power of two. This returns a one-hot vector of bits such indicating the closes larger power of two. We adapt Protocol 3.5 by [Catrina & Saxena \(2010\)](#) for this.

$\langle x \rangle^A \leftarrow \text{B2A}(\langle x_0 \rangle^B, \dots, \langle x_{k-1} \rangle^B)$ Domain conversion from binary to arithmetic as above.

$\langle c_b; f \rangle^A \leftarrow \text{MUX}(c_0, c_1, \langle b \rangle^B)$ Oblivious selection as in Appendix C.

Logarithm Computation logarithm with any public base can be reduced to logarithm to base two using $\log_x y = \log_2 y \cdot \log_x 2$. [Aly & Smart \(2019\)](#) propose to represent y as $y = a \cdot 2^b$, where $a \in [0.5, 1)$ and $b \in \mathbb{Z}$. This then allows to compute $\log_2 y = \log_2 a + b$. Given the restricted range of a , $\log_2 a$ can be approximated using a division of polynomials. Numerical stability and input range control are less of an issue here, because we only use logarithm for the loss computation, which does not influence the training.

Inverse square root [Aly & Smart \(2019\)](#) propose to compute square root using Goldschmidt and Raphson-Newton iterations. We could combine this with the division operator introduced above. However, [Lu et al. \(2020\)](#) propose a more direct computation that avoids running two successive iterations. We have optimized their algorithm SqrtComp computing the square root of a power of two as shown in Algorithm 7. As an essential optimization, we compute the least significant bit of $e + f$ by simply XORing every other z_i in step 6 of Algorithm 7. Given that XOR does not require communication with binary secret sharing, this is much more efficient than computing the least significant bit of a sum of oblivious selections done by [Lu et al.](#) Remarkably, our optimization cuts the cost by roughly half. This is due to the reduction in conversions

Table 7: Total communication in kbit for inverse square root across a range of security models with one corrupted party. “SH” stands for semi-honest security and “Mal.” for malicious security.

| | 3 Parties | | 2 Parties | |
|------------------|-----------|------|-----------|--------|
| | SH | Mal. | SH | Mal. |
| Lu et al. (2020) | 19 | 160 | 481 | 25,456 |
| Ours | 9 | 114 | 342 | 21,522 |

from binary to arithmetic. Furthermore, we correct some issues which result in outputs that are off by a multiplication with a power of two. We present the rest of their approach in Algorithms 6 and 8 (unchanged) for completeness. Unlike Lu et al. (2020), we do not explicitly state the truncation after fixed-point multiplication.

Table 7 shows how our approach compares to Lu et al. (2020). In the context of the Figures in Table 5, this implies an improvement of up one quarter. The importance of inverse square root stems from the fact that it is computed for every iteration and trainable parameter in the parameter update.

Uniformly random fractional number Limiting ourselves to intervals of the form $[x, x + 2^e)$ for a potentially negative integer e , we can reduce the problem to generate a random $(f + e)$ -bit number where f is the fixed-point precision. Recall that we represent a fractional number x as $\lfloor x \cdot 2^{-f} \rfloor$. Generating a random n -bit number is straightforward using random bits, which in our protocol can be generated as presented by Damgård et al. (2019). However, Dalskov et al. (2021) and Escudero et al. (2020) present more efficient approaches that involve mixed-circuit computation.

Communication cost Table 8 shows the total communication cost of some of the building blocks in our three-party protocol for $f = 16$. This setting mandates the modulus 2^{64} because the division protocol requires a bit length of $4f$.

Table 8: Communication cost of select computation for $f = 16$ and integer modulus 2^{64} .

| | Bits |
|---|--------|
| Integer multiplication | 192 |
| Probabilistic truncation | 960 |
| Nearest truncation | 2,225 |
| Comparison | 668 |
| Division (prob. truncation) | 10,416 |
| Division (nearest truncation) | 24,081 |
| Exponentiation (prob. truncation) | 16,303 |
| Exponentiation (nearest truncation) | 43,634 |
| Invert square root (prob. truncation) | 9,455 |
| Invert square root (nearest truncation) | 15,539 |

D. Deep Learning Building Blocks

In this section, we will use the building blocks in Appendix C to construct high-level computational modules for deep learning.

Fully connected layers Both forward and back-propagation of fully connected layers can be seen as matrix multiplications and thus can be implemented using dot products. A particular challenge in secure computation is to compute a number of outputs in parallel in order to save communication rounds. We overcome this challenge by having a dedicated infrastructure in our implementation that computes all dot products for a matrix multiplication in a single batch of communication, thus reducing the number of communication rounds.

2D convolution layers Similar to fully connected layers, 2D convolution and its corresponding gradient can be implemented using only dot products, and we again compute several output values in parallel.

ReLU Given the input $x \in \mathbb{R}$, a rectified linear unit (ReLU [Nair & Hinton 2010](#)) outputs

$$\text{ReLU}(x) := \max(x, 0) = \llbracket x > 0 \rrbracket \cdot x.$$

It can thus be implemented as a comparison followed by an oblivious selection. For back-propagation, it is advantageous to reuse the comparison results from forward propagation due to the relatively high cost of secure computation. Note that the comparison results are stored in secret-shared form and thus there is no reduction in security.

Max pooling Similar to ReLU, max pooling can be reduced to comparison and oblivious selection. In secure computation, it saves communication rounds if the process uses a balanced tree rather than iterating over all input values of one maximum computation. For example, two-dimensional max pooling with a 2x2 window requires computing the maximum of four values. We compute the maximum of two pairs of values followed by the maximum of the two results. For back-propagation, it again pays off to store the intermediate results from forward propagation in secret-shared form.

Dropout Dropout layers ([Srivastava et al., 2014](#)) require generating a random bit according to some probability and oblivious selection. For simplicity, we only support probabilities that are a power of two.

Batch normalization As another widely used deep learning module, batch normalization (BN, [Ioffe & Szegedy 2015](#)). Based on a mini-batch $\mathcal{B} = \{x_i\}$ of any input statistic x , the BN layer outputs

$$y_i = \gamma \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \beta, \quad (2)$$

where $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$ are the first and second central moments of x , and γ, β and ϵ are hyper-parameters. In addition to basic arithmetic, it requires inverse square root, which is not trivial in MPC.

Softmax output and cross-entropy loss For classification tasks, consider softmax output units $\frac{\exp(x_i)}{\sum_j \exp(x_j)}$, where $\mathbf{x} = (x_1, x_2, \dots)$ is the linear output (logits) of the last layer, and the cross-entropy loss $\ell = -\sum_i y_i \log \frac{\exp(x_i)}{\sum_j \exp(x_j)} = -\sum_i x_i y_i + \log \sum_j \exp(x_j)$, where $\mathbf{y} = (y_1, y_2, \dots)$ is the ground truth one-hot vector. This usual combination requires computing the following gradient for back-propagation:

$$\begin{aligned} \nabla_i &:= \frac{\partial \ell}{\partial x_i} = \frac{\partial}{\partial x_i} \left(-\sum_k y_k \cdot x_k + \log \sum_j \exp(x_j) \right) \\ &= -y_i + \frac{\exp(x_i)}{\sum_j \exp(x_j)}. \end{aligned} \quad (3)$$

On the RHS of the above eq. (3), the values in the denominator are potentially large due to the exponentiation. This is prone to numerical overflow in our quantized representation, because the latter puts strict limits on the values. We therefore optimize the computation by first taking the maximum of the input values: $x_{\max} = \max_i x_i$. Then we evaluate ∇_i based on

$$\nabla_i = \frac{\exp(x_i - x_{\max})}{\sum_j \exp(x_j - x_{\max})} - y_i.$$

As $\forall j, x_j \leq x_{\max}$, we have $1 \leq \sum_j \exp(x_j - x_{\max}) \leq L$, where L is the number of class labels ($L = 10$ for MNIST). Hence numerical overflow is avoided. The same technique can be used to compute the sigmoid activation function, as $\text{sigmoid}(x) := \frac{1}{1 + \exp(-x)} = \frac{\exp(0)}{\exp(0) + \exp(-x)}$ is a special case of softmax.

Stochastic gradient descent The model parameter update in SGD only involves basic arithmetic:

$$\theta_j \leftarrow \theta_j - \frac{\gamma}{B} \sum_{i=1}^B \frac{\partial \ell_i}{\partial \theta_j}, \quad (4)$$

where θ_j is the parameter indexed by j , $\gamma > 0$ is the learning rate, B is the mini-batch size, ℓ_i is the cross-entropy loss with respect to the i 'th sample in the mini-batch. To tackle the limited precision with quantization, we *defer dividing by the*

batch size B to the model update in eq. (4). In other words, the back-propagation computes the gradient $\frac{\partial \ell_i}{\partial \theta_j}$, where the back-propagated error terms are not divided by B . Division by the batch size B only happens when the parameter update is performed. Since the batch size is a power of two (128), it is sufficient to use probabilistic truncation instead of full-blown division. This both saves time and decreases the error.

Adam (Kingma & Ba, 2015) and AMSGrad (Reddi et al., 2018) Optimizers in this category perform a more sophisticated parameter update rule⁵:

$$\theta_j \leftarrow \theta_j - \frac{\gamma}{\sqrt{v_j + \epsilon}} g_j,$$

where $\epsilon > 0$ is a small constant to prevent division by zero, and g_j and v_j are the first and second moments of the gradient $\frac{\partial \ell}{\partial \theta_j} = \frac{1}{B} \sum_{i=1}^B \frac{\partial \ell_i}{\partial \theta_j}$, respectively. Division by the batch size B can be skipped in computing $\frac{\partial \ell}{\partial \theta_j}$, because scaling of the gradient leads to the same scaling factor of g_j and $\sqrt{v_j}$. Both g_j and v_j are computed from the back-propagation result using simple arithmetic and comparison (in the case of AMSGrad). We compute the inverse square root as described in Section C above.

Parameter initialization We use the widely adopted initialization method proposed by Glorot & Bengio (2010). Each weight w between two layers with size d_{in} and d_{out} is initialized by

$$w \sim \mathcal{U} \left[-\sqrt{\frac{6}{d_{\text{in}} + d_{\text{out}}}}, \sqrt{\frac{6}{d_{\text{in}} + d_{\text{out}}}} \right],$$

where $\mathcal{U}[a, b]$ means the uniform distribution in the given range $[a, b]$. Besides basic operations, it involves generating a uniformly distributed random fractional value in a given interval, which is introduced in the previous section C. All bias values are initialized to 0.

E. Models

The neural network structures investigated in this paper are given by the following Figures 5, 6, 7, 8 and 9. The structure of each network is formatted in Keras⁶ code.

```
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(128, activation='relu'),
tf.keras.layers.Dense(128, activation='relu'),
tf.keras.layers.Dense(10, activation='softmax')
```

Figure 5: Network A used by Mohassel & Zhang (2017)

```
tf.keras.layers.Conv2D(16, 5, 1, 'same', activation='relu'),
tf.keras.layers.MaxPooling2D(2),
tf.keras.layers.Conv2D(16, 5, 1, 'same', activation='relu'),
tf.keras.layers.MaxPooling2D(2),
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(100, activation='relu'),
tf.keras.layers.Dense(10, activation='softmax')
```

Figure 6: Network B used by Liu et al. (2017)

F. Fashion MNIST

We run our implementation on Fashion MNIST (Xiao et al., 2017) for a more complete picture. Figure 10 shows our results. See Xiao et al. (2017) for an overview on how other models perform in cleartext.

⁵Our implementation is slightly different from the original Adam and AMSGrad, as we put ϵ inside the square root. This is because the inverse square root is implemented as a basic operation that can be efficiently computed in MPC.

⁶<https://keras.io>

```
tf.keras.layers.Conv2D(20, 5, 1, 'valid', activation='relu'),
tf.keras.layers.MaxPooling2D(2),
tf.keras.layers.Conv2D(50, 5, 1, 'valid', activation='relu'),
tf.keras.layers.MaxPooling2D(2),
tf.keras.layers.Flatten(),
tf.keras.layers.Dropout(0.5),
tf.keras.layers.Dense(100, activation='relu'),
tf.keras.layers.Dense(10, activation='softmax')
```

Figure 7: Network C used by [LeCun et al. \(1998\)](#) (with optional Dropout layer)

```
tf.keras.layers.Conv2D(5, 5, 2, 'same', activation='relu'),
tf.keras.layers.Flatten(),
tf.keras.layers.Dense(100, activation='relu'),
tf.keras.layers.Dense(10, activation='softmax')
```

Figure 8: Network D used by [Riazi et al. \(2018\)](#)

```
#1st Convolutional Layer
Conv2D(filters=96, input_shape=(32,32,3), kernel_size=(11,11), strides=(4,4),
padding=9),
Activation('relu'),
MaxPooling2D(pool_size=3, strides=(2,2)),
BatchNormalization(),

#2nd Convolutional Layer
Conv2D(filters=256, kernel_size=(5, 5), strides=(1,1), padding=1),
Activation('relu'),
BatchNormalization(),
MaxPooling2D(pool_size=(2,2), strides=1),

#3rd Convolutional Layer
Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding=1),
Activation('relu'),

#4th Convolutional Layer
Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding=1),
Activation('relu'),

#5th Convolutional Layer
Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding=1),
Activation('relu'),

#Passing it to a Fully Connected layer
# 1st Fully Connected Layer
Dense(256),
Activation('relu'),

#2nd Fully Connected Layer
Dense(256),
Activation('relu'),
```

```
#Output Layer
Dense (10)
```

Figure 9: AlexNet for CIFAR-10 by [Wagh et al. \(2021\)](#). Note that our implementation allows any padding unlike Keras.

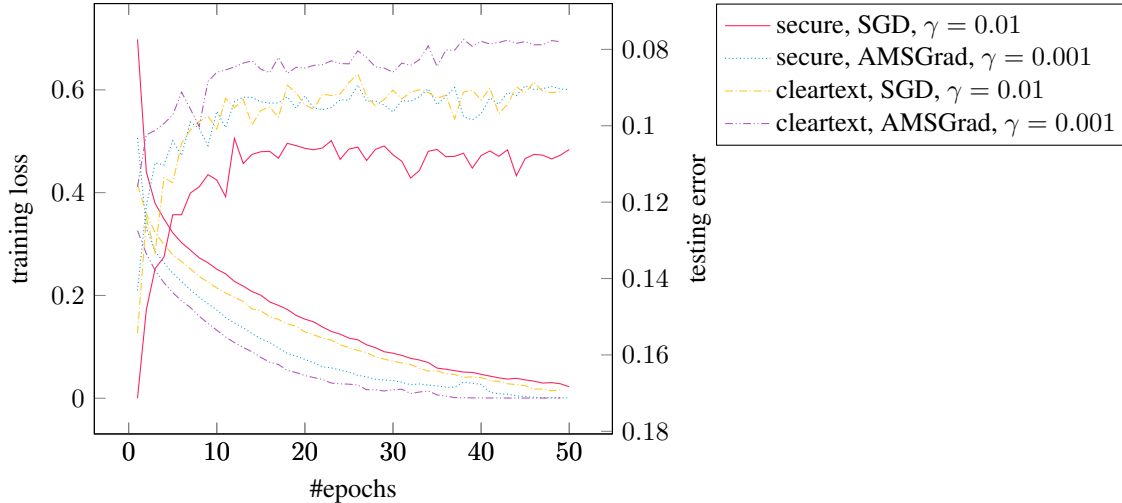


Figure 10: Learning curves for image classification using network C on the Fashion MNIST dataset, with $f = 16$ and probabilistic truncation. γ is the learning rate.

G. More Experimental Results

For the task of classifying MNIST digits, Figure 11 shows the learning curves for all networks used by [Wagh et al. \(2019\)](#). Among all the investigated structures, LeNet (network C) performs the best. Furthermore, AMSGrad consistently improves the classification performance as compared to SGD.

Figure 12 shows the comparison of cleartext training and secure training for CIFAR-10 with $f = 16$ and probabilistic truncation. Our MPC implementation has gained similar performance with the cleartext counterpart.

H. Hyperparameter Settings

In the following we discuss our choice of hyperparameters.

Number of epochs As we found convergence after 100 epochs, we have run most of our benchmarks for 150 epochs, except for the comparison of optimizers where we stopped at 100.

Early stop We have not used early stop.

Mini-batch size We have used 128 throughout as it is a standard size. We briefly trialed 1024 as suggested by [Li et al. \(2017\)](#), but did not find any improvement.

Reshuffling training samples At the beginning of each epoch, we randomly re-shuffle the training samples using the Fisher-Yates shuffle with MP-SPDZ’s internal pseudo-random number generator as randomness source.

Learning rate We have tried a number of learning rates as documented in Figure 3. As a result, we settled for 0.01 for SGD and 0.001 for AMSGrad in further benchmarks.

Learning rate decay/schedule We have not used either.

Random initialization The platform uses independent random initialization by design.

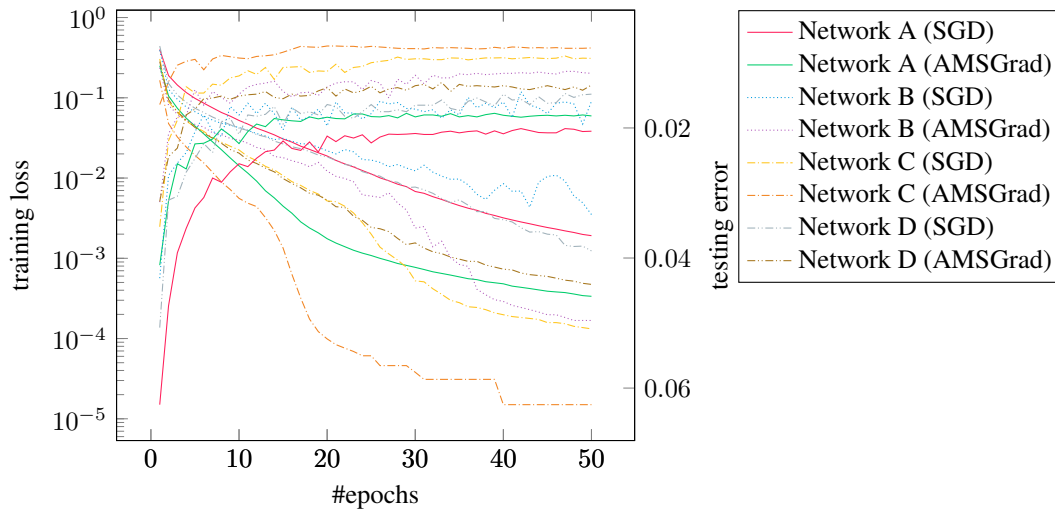


Figure 11: Loss and accuracy for various networks, $f = 16$, and probabilistic truncation.

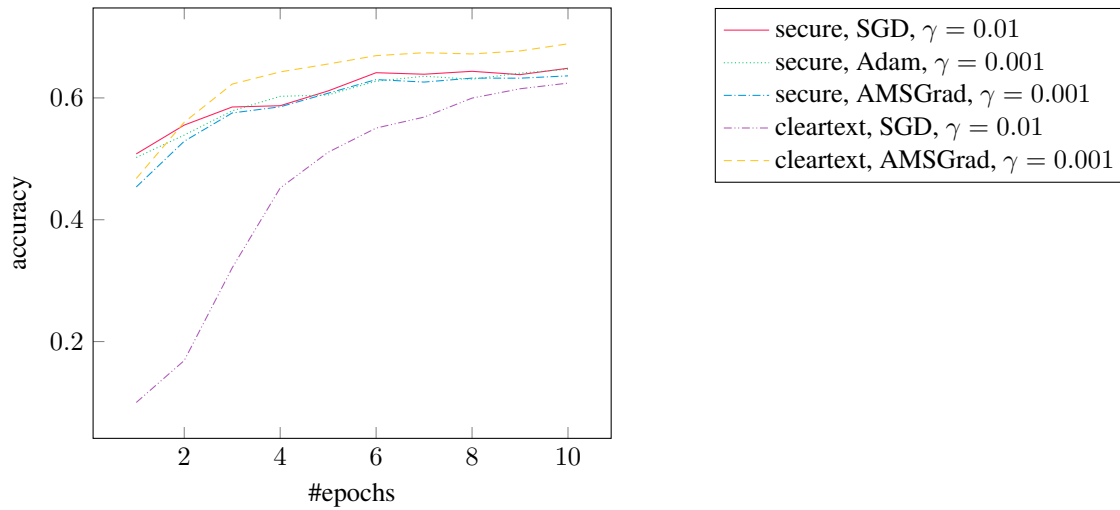


Figure 12: Comparison of cleartext training and secure training for CIFAR-10 with $f = 16$ and probabilistic truncation. γ is the learning rate.

Dropout We have experimented with Dropout but not found any improvement as shown in Figure 3.

Input preprocessing We have normalized the inputs to $[0, 1]$.

Test/training split We have used the usual MNIST split.

Hyperparameters for Adam/AMSGrad We use the common choice $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$.

I. List of Symbols

Table 9: List of Symbols

| | |
|---------------------------|---|
| \mathbb{R} | the set of real numbers |
| \mathbb{Z} | the set of integers |
| x, y, λ | real numbers in \mathbb{R} |
| \mathbf{x}, \mathbf{y} | real vectors |
| $\lceil x \rceil$ | nearest rounding of x : $\lceil x \rceil := \arg \min_{z \in \mathbb{Z}} x - z $ |
| $\lfloor x \rfloor$ | floor function: $\lfloor x \rfloor := \max\{z \in \mathbb{Z} \mid z \leq x\}$ |
| $\{x\}$ | the fractional part of x : $\{x\} := x - \lfloor x \rfloor$ |
| $Q^f(x)$ | the integer representation of x : $Q^f(x) = \lfloor x \cdot 2^f \rfloor$ |
| $\mathcal{U}(a, b)$ | uniform distribution on $[a, b]$ |
| $\mathcal{B}(p)$ | Bernoulli distribution (with probability p the random variable equals 1) |
| $\llbracket c \rrbracket$ | Iverson bracket ($\llbracket c \rrbracket = 1$ if c is true otherwise 0) |
| $ \cdot $ | absolute value |
| $\ \cdot\ $ | Frobenius norm |
| $\text{Prob}(\cdot)$ | probability of the parameter statement being true |
| (condition ? a : b) | oblivious selection |

J. Proofs

J.1. Proof of Proposition 4.1

Proof. By Eq. (1),

$$E(b) = 0 \times (1 - \{\mu\}) + 1 \times \{\mu\} = \{\mu\}.$$

Therefore

$$E(R^f(xy)) = E(\lfloor \mu \rfloor + b) = \lfloor \mu \rfloor + E(b) = \lfloor \mu \rfloor + \{\mu\} = \lfloor \mu \rfloor + (\mu - \lfloor \mu \rfloor) = \mu = Q^f(x)Q^f(y)2^{-f}.$$

Notice that $Q^f(A)$ is simply obtained through entry-wise quantization $Q^f(a_{ij})$. By our matrix multiplication algorithm,

$$\begin{aligned} E(R_{ij}^f(AB)) &= E\left(\sum_k R^f(A_{ik}B_{kj})\right) \\ &= \sum_k E(R^f(A_{ik}B_{kj})) \\ &= \sum_k Q^f(A_{ik})Q^f(B_{kj})2^{-f} \\ &= 2^{-f} \sum_k Q_{ik}^f(A)Q_{kj}^f(B) \\ &= 2^{-f}(Q^f(A)Q^f(B))_{ij}. \end{aligned}$$

In matrix form, it gives

$$E(R^f(AB)) = 2^{-f}Q^f(A)Q^f(B).$$

□

J.2. A Lemma of Probabilistic Rounding

Lemma J.1. Given $x, y \in (-2^k, 2^k) \subset \mathbb{R}$,

$$|xy - R^f(xy)2^{-f}| < (\max\{|x|, |y|\} + 1)\epsilon + \epsilon^2/4 < (2^k + 1)\epsilon + \epsilon^2/4,$$

where $\epsilon := 2^{-f}$.

Proof. We have

$$|xy - R^f(xy)2^{-f}| \leq |xy - \mu 2^{-f}| + |R^f(xy)2^{-f} - \mu 2^{-f}| = |xy - \mu 2^{-f}| + 2^{-f} |R^f(xy) - \mu|.$$

By the definition of $R^f(xy)$,

$$|R^f(xy) - \mu| = |\lfloor \mu \rfloor + b - \mu| = |b - (\mu - \lfloor \mu \rfloor)| = |b - \{\mu\}| < 1.$$

Therefore, we have

$$|xy - R^f(xy)2^{-f}| < |xy - \mu 2^{-f}| + 2^{-f} = |xy - \mu 2^{-f}| + \epsilon.$$

Now we only need to bound the deterministic term $|xy - \mu 2^{-f}|$. We have

$$\begin{aligned} |xy - \mu 2^{-f}| &= |xy - Q^f(x)Q^f(y)2^{-2f}| \\ &= |Q^f(x)Q^f(y)2^{-2f} - xy| \\ &= |(Q^f(x)2^{-f} - x)(Q^f(y)2^{-f} - y) + 2^{-f}Q^f(x)y + 2^{-f}Q^f(y)x - 2xy| \\ &= |(Q^f(x)2^{-f} - x)(Q^f(y)2^{-f} - y) + (2^{-f}Q^f(x) - x)y + (2^{-f}Q^f(y) - y)x| \\ &\leq |Q^f(x)2^{-f} - x| \cdot |Q^f(y)2^{-f} - y| + |2^{-f}Q^f(x) - x| \cdot |y| + |2^{-f}Q^f(y) - y| \cdot |x|. \end{aligned}$$

We notice that $\forall x \in \mathbb{R}$,

$$\begin{aligned} |2^{-f}Q^f(x) - x| &= |2^{-f}\lfloor x \cdot 2^f \rfloor - x| = 2^{-f}|\lfloor x \cdot 2^f \rfloor - x \cdot 2^f| \\ &\leq 2^{-f} \times \frac{1}{2} = 2^{-f-1} = \frac{\epsilon}{2}. \end{aligned}$$

Then, we get

$$\begin{aligned} |xy - \mu 2^{-f}| &\leq \frac{\epsilon}{2} \cdot \frac{\epsilon}{2} + \frac{\epsilon}{2} \cdot (|x| + |y|) = \frac{\epsilon^2}{4} + \frac{\epsilon}{2} \cdot (|x| + |y|) \\ &\leq \frac{\epsilon^2}{4} + \frac{\epsilon}{2} \cdot (2 \max\{|x|, |y|\}) = \frac{\epsilon^2}{4} + \max\{|x|, |y|\}\epsilon. \end{aligned}$$

In summary,

$$|xy - R^f(xy)2^{-f}| < |xy - \mu 2^{-f}| + \epsilon \leq \frac{\epsilon^2}{4} + \max\{|x|, |y|\}\epsilon + \epsilon = (\max\{|x|, |y|\} + 1)\epsilon + \frac{\epsilon^2}{4}.$$

The second “<” is trivial by noting our assumption

$$\max\{|x|, |y|\} < 2^k.$$

□

J.3. Proof of Proposition 4.2

Proof. The proof is based on Lemma J.1.

$$\begin{aligned}
 \|AB - R^f(AB)2^{-f}\| &= \sqrt{\sum_{i,j} \left((AB)_{ij} - R_{ij}^f(AB)2^{-f} \right)^2} \\
 &= \sqrt{\sum_{i,j} \left(\sum_l A_{il}B_{lj} - \sum_l R^f(A_{il}B_{lj})2^{-f} \right)^2} \\
 &= \sqrt{\sum_{i,j} \left| \sum_l (A_{il}B_{lj} - R^f(A_{il}B_{lj})2^{-f}) \right|^2} \\
 &\leq \sqrt{\sum_{i,j} \left(\sum_l |A_{il}B_{lj} - R^f(A_{il}B_{lj})2^{-f}| \right)^2} \\
 &< \sqrt{\sum_{i,j} \left[\sum_l \left((2^k + 1)\epsilon + \frac{\epsilon^2}{4} \right) \right]^2} \\
 &= \sqrt{mp \cdot n^2 \cdot \left((2^k + 1)\epsilon + \frac{\epsilon^2}{4} \right)^2} \\
 &= \sqrt{mpn} \left((2^k + 1)\epsilon + \frac{\epsilon^2}{4} \right).
 \end{aligned}$$

Multiplying both sides by 2^f , we get

$$\|R^f(AB) - 2^f AB\| < \sqrt{mpn} \left((2^k + 1) + \frac{\epsilon}{4} \right).$$

□

J.4. Proof of Proposition 4.3

Proof. Note that the variance, denoted as $\sigma^2(\cdot)$, of $R^f(xy)$ is

$$\begin{aligned}
 \sigma^2(R^f(xy)) &= \sigma^2(b) = E(b^2) - (E(b))^2 = E(b) - (E(b))^2 = \{\mu\} - (\{\mu\})^2 = \{\mu\}(1 - \{\mu\}) \\
 &\leq \left(\frac{\{\mu\} + (1 - \{\mu\})}{2} \right)^2 = \frac{1}{4}.
 \end{aligned}$$

Therefore,

$$\begin{aligned}
 \sigma^2(R_{ij}^f(AB)) &= \sigma^2 \left(\sum_k R^f(A_{ik}B_{kj}) \right) = \sum_k \sigma^2(R^f(A_{ik}B_{kj})) \\
 &\leq \sum_k \frac{1}{4} = \frac{n}{4}.
 \end{aligned}$$

In matrix form, the element-wise variance of the random matrix $R^f(AB)$ is

$$\sigma^2(R^f(AB)) \leq \frac{n}{4} \mathbf{1}\mathbf{1}^\top,$$

where $\mathbf{1}$ is the column vector of 1's. Note all mp entries in $R^f(AB)$ are independent. By the Chebyshev inequality, with probability at least $1 - \frac{mp}{\rho}$, the following is true

$$\frac{\|R^f(AB) - E(R^f(AB))\|^2}{n/4} \leq \rho,$$

which can be re-written as

$$\|R^f(AB) - E(R^f(AB))\| \leq \frac{\sqrt{\rho n}}{2}.$$

Let $\iota := \frac{1}{2} \sqrt{\frac{\rho}{mp}}$. Then $\rho = 4\iota^2 mp$. In conclusion, with probability at least

$$1 - \frac{mp}{\rho} = 1 - \frac{mp}{4\iota^2 mp} = 1 - \frac{1}{4\iota^2},$$

the following is true:

$$\|R^f(AB) - E(R^f(AB))\| \leq \frac{\sqrt{\rho n}}{2} = \frac{\sqrt{4\iota^2 mp \cdot n}}{2} = \iota \sqrt{mnp}.$$

Based on Proposition 4.1, the statement can be equivalently written as

$$\|R^f(AB) - 2^{-f} Q^f(A) Q^f(B)\| \leq \iota \sqrt{mnp}.$$

□