SECURENN: 3-PARTY SECURE COMPUTATION FOR NEURAL NETWORK TRAINING

WHITEPAPER

Sameer Wagh*
Princeton University
snwagh@gmail.com

Divya Gupta
Microsoft Research, India
divya.gupta@microsoft.com

Nishanth Chandran
Microsoft Research, India
nichandr@microsoft.com

February 20, 2019

ABSTRACT

Neural Networks (NN) provide a powerful method for machine learning training and inference. For effective training, it is often desirable for multiple parties to combine their data – however, doing so conflicts with data privacy. In this work, we provide novel three-party secure computation protocols for exact computation of popular non-linear activation functions such as Rectified Linear Units, Maxpool, normalization and so on. We piece our protocol with existing work from literature to enable efficient training and inference of a broad class of NN architectures (CNN's). Experimentally, we implement and deploy our system over Amazon EC2 servers in both the LAN and WAN setting and 4 different neural networks (CNN's and DNN's) of varying complexity to showcase the promise of our approach. Our contributions can be summarized as follows:

- **Theoretical Novelty:** We develop new protocol for the exact computation of non-linear functions that are more communication efficient ($> 8\times$) than currently known techniques. Our theoretical improvements when implemented in practice show significant performance improvement over state-of-the-art work in privacy preserving machine learning.
- **Experimental Evaluation:** For secure inference, our system outperforms prior 2 and 3-server works (SecureML, MiniONN, Chameleon, Gazelle) by $6\times - 113\times$ (with larger gains obtained in more complex networks). For secure training, compared to the only prior work (SecureML) that considered a much smaller fully connected network, our protocols are $79\times$ and $7\times$ faster than their 2 and 3-server protocols. In the WAN setting, these improvements are more dramatic and we obtain an improvement of $553\times$!

This enables us to be the first work to widely explore the practicality of privacy preserving neural network training on complex networks such as Convolutional Neural Networks (CNNs) with $>$ 99% accuracy on the MNIST dataset. Our efficiency gains come from a significant improvement in communication through the complete elimination of expensive garbled circuits and oblivious transfer protocols.

1 Introduction

Neural networks (NN) have proven to be a very effective tool in producing predictive models that are widely used in applications such as healthcare, image classification, finance, and so on. The accuracy of these models improves with the amount and diversity of training data [38]. Large amounts of training data can be obtained by pooling in data from multiple contributors, but this data is sensitive and cannot be revealed in the clear due to proprietary reasons or compliance requirements [15, 6]. To enable training of NN models with good accuracy, it is desirable to securely train over data from multiple contributors such that plaintext data is hidden from the training entities.

In this work, we consider a number of clients $C_1, C_2, ..., C_M$, who wish to execute training over their joint data using 3 servers. These 3 servers run an interactive protocol to train a neural network over the joint data to produce a trained

*Work done primarily while at Microsoft Research, India

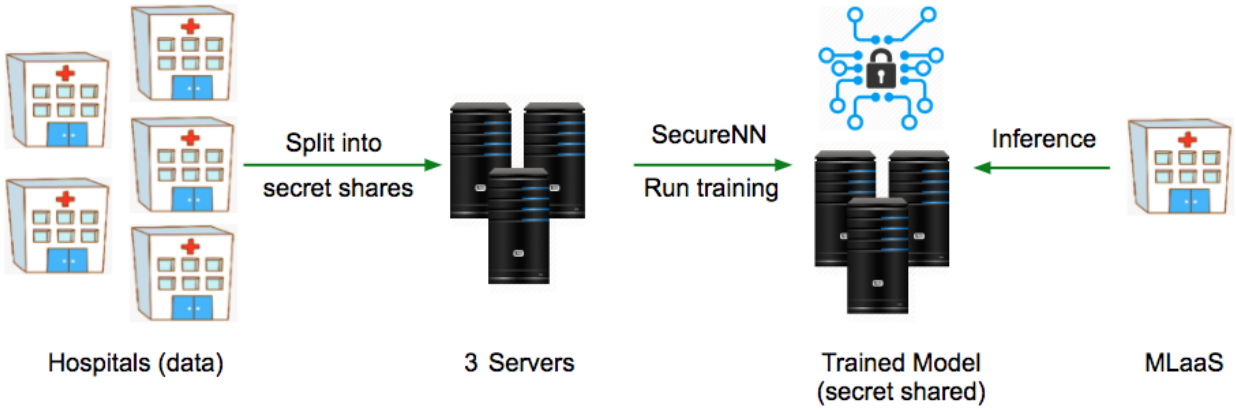

Figure 1: **Architecture for SecureNN usage scenario**

model which can later be used for inference. The security requirement is that no client or server learns any information about any other party's training data. At the same time, the trained model is secret shared between the servers or can be reconstructed to obtain the trained model in the clear. Even if the model is retained as secret shares between the 3 servers, the inference/prediction can still be executed using the trained model on any new input – keeping the model, the new input, and the predicted output private from the other parties as well as the servers. For example, a group of M hospitals, each having sensitive patient data (such as heart rate readings, blood group, sugar levels etc.) can use the above architecture to train a model to run Machine Learning as a Service (MLaaS) and help predict some disease or irregular health behavior. The system can be set up such that the patient's sensitive input and predicted output are only revealed to the patient, and remains hidden from everyone else. This architecture is graphically represented in Figure 1.

Over the past few years, there are been significant research progress in pushing the frontiers of secure multi-party computation addressed to the above problem [9, 10, 14, 30, 7, 22, 27, 17]. Our theoretical contributions form our biggest differentiator from prior work – we develop new protocols for the exact computation of popular non-linear activation functions required by popular neural network architectures. We showcase the promise of our alternative approach in enabling efficient non-linear function computation to conventional techniques relying on interconversion protocols (such as ABY [20]) between standard secret sharing schemes.

1.1 Our Contributions

In this work, we propose novel, specialized protocols (for the 3-server setting) tailored to popular functions in neural network training and inference and push the frontiers on privacy preserving machine learning. Specifically, our contributions can be summarized as follows:

- **Theoretical:** Our main technical contribution is the construction of efficient 3-party protocols for various non-linear functions commonly used in machine learning algorithms – Rectified Linear Unit (ReLU), Maxpool, normalization and their derivatives. There protocols can be combined to provide an efficient and secure MPC protocols for NN training and inference.
- **Experimental:** We implement and evaluate our end-to-end NN training and inference protocols over Amazon EC2 over 4 different CNN's and DNN's (of varying complexity) and in different settings (LAN and WAN). We obtain much larger gains on more complex network architectures as well as in the WAN setting. These mainly stem from the fact that our protocols are much more communication efficient ($> 8\times$) by developing alternatives to expensive interconversion protocols used by prior works (such as SecureML [31], MiniONN [29], Gazelle [25]).

We also analyze the security of our system in two models. First, we prove full security (privacy and correctness) against *semi-honest corruption* of a single server. At the same time, we are able to provide privacy (but not correctness) against a *single malicious server*, a notion formalized by Araki *et al.* [7]. Privacy against malicious server informally guarantees that a malicious server cannot learn anything about the inputs or outputs of the honest clients even if it deviates arbitrarily from the protocol specification (as long as the outputs of the computation are not revealed to the adversary). Given our 3-server setting, single corruption is the best corruption threshold that one can achieve [18].

1.2 Experimental Results

We evaluate our end-to-end implementation over 3 popular neural networks for the MNIST dataset: (A) a 3-layer DNN from SecureML [31], (B) a 4-layer CNN from MiniONN [29] and (C) the 4-layer CNN popularly known as LeNet [26]. We evaluate our protocols both on the task of secure training as well as secure prediction in the LAN and WAN settings and provide details below. These are briefly described in Fig. 4. For our comparisons, we run our protocols on similar hardware and network conditions as prior works. Refer to Section 8 for more details.

Secure Training. We train all the above networks on the MNIST dataset [3]. The *overall execution time* of our MPC protocol for Network A model over a LAN network is roughly an hour. For our largest CNN (Network C) our secure protocols execute in under 10 hours to achieve $> 98\%$ accuracy and in under 30 hours to achieve $> 99\%$ accuracy.

Comparison with prior work: We remark that this is the first work that implements training for networks B and C and these networks are much larger and give much higher accuracy ($> 98\%$) than network A (93%) considered by prior work. SecureML [31] provides secure NN training protocols in both the 2-server and 3-server models secure against a single semi-honest adversary for Network A. Their 3-server protocol is practical in the LAN setting only, and our protocols outperform theirs by $7\times$. Compared to their 2-server protocols, we give an improvement of $79\times$ and $553\times$ in the LAN and WAN settings, respectively.

Secure Inference. We also evaluate our protocols for secure inference one the same networks (and additionally on Network D) assuming a trained model secret shared between the servers. For the smallest network A, a single prediction takes roughly $0.04s$ and $2.43s$ in the LAN and WAN settings, respectively. For the largest network C, a single prediction takes $0.23s$ in LAN and $4.08s$ in the WAN setting. Also, as is observed by previous works as well, doing batch predictions is much faster in the amortized sense than doing multiple predictions serially. For instance, for network C, a batch of 128 predictions take only $10.82s$ in the LAN and $30.45s$ in the WAN setting and scales well with larger batch sizes.

Comparison with prior work: We compare our protocols with state-of-the-art works in secure inference in the 2-server [31, 29, 25] and 3-server [35] settings. As our experiments show, the total execution time of our protocols are $113\times$ faster than [31], $71.7\times$ faster than MiniONN [29], $35.5\times$ faster than Chameleon [35] and $6.23\times$ faster than Gazelle [25]. These gains come from a $74.2\times, 3.2\times$ and $7.9\times$ reduction in communication over MiniONN, Chameleon, and Gazelle, respectively[2].

1.3 Organization of the paper

We begin by giving a technical roadmap in Section 2. We describe the security model, the neural network algorithms and develop the notation in Section 3. Section 4 contains our building block protocols. Section 5 describe the main protocols. We discuss theoretical efficiency of our protocols in Section 7 and present our experimental evaluation in Section 8. Finally, acknowledge our limitations in Section 9, present related work in Section 10 and conclude in Section 11.

2 Technical Roadmap

Secure protocols for neural network algorithms generally follow the paradigm of executing arithmetic computation, such as matrix multiplication or convolutions, using Beaver triplets and executing non-linear computation such as ReLU, Maxpool and its derivatives, using Boolean circuits or Yao's garbled circuits. In order to make these protocols compatible with each other, interconversion protocols are also used to move from an arithmetic encoding (either arithmetic sharing or homomorphic encryption ciphertext) to a Boolean encoding (garbled encoding) and vice-versa. The use of Yao's garbled circuits for non-linear function computation incurs a multiplicative factor overhead of 128 (the security parameter, κ). Similarly, interconversion protocols required to switch between different encodings incurs overheads. Our protocols propose a novel way to compute these non-linear functions exactly, while reducing the communication overhead. Given that multi-party protocols, especially in the WAN setting, are communication bound, our protocols demonstrate significant speed-up in total execution time.

In the 3-server setting, we denote the servers by P_0, P_1 and P_2. At the start of any protocol, parties P_0 and P_1 hold 2-out-of-2 additive secret shares of the inputs to the protocol. All our protocols maintain the invariant that at the end of the protocol P_0 and P_1 hold 2-out-of-2 shares of the output.

[2]SecureML does not mention their communication complexity.

2.1 Overview of Protocols

SecureNN protocols are not symmetric in the 3 participating servers – the data is initially shared in a 2-out-of-2 arithmetic sharing between P_0 and P_1. Party P_2 takes the role of "assistant" in all protocols and has no input to protocols. We split the different functionalities required for various NN architectures into various *main protocols*. We also develop a number of *building block protocols*, which are combined together to form the main protocols.

Main protocols. In all our main protocols (Section 5), we maintain the invariant that parties P_0 and P_1 begin with "fresh" shares of input value (over $\mathbb{Z}_L$) and output a "fresh" share of the output value (again over $\mathbb{Z}_L$) at the end of the protocol – this will enable us (as shown in Section 5.7) to arbitrarily compose our main protocols to obtain protocols for a variety of neural networks.

Building blocks. In this section, we describe various building blocks to our main protocols. Some of these protocols deviate from the invariant described above – i.e., P_0 and P_1 do not necessarily begin/end protocols with shares of input/output over $\mathbb{Z}_L$. Furthermore, P_2 may receive outputs in these protocols. We provide some of the security proofs in Appendix B and defer the remaining proofs to the full version.

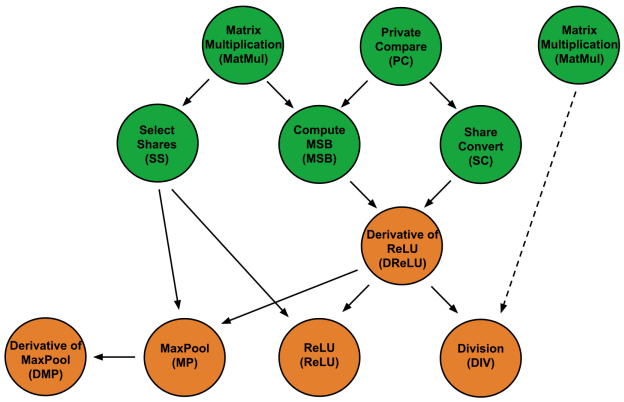

Figure 2: **Dependency graph of main protocols (orange) and building blocks (green) in SecureNN.**

Fig. 2 shows a dependency graph between the various protocols (main and building blocks) used in SecureNN. Next, we briefly describe the central protocol of our work - the derivative of the ReLU function (DReLU in Fig. 2).

2.2 Derivative of ReLU

The derivative of ReLU functionality is central to the construction of other non-linear protocols in this work. We give a detailed intuition behind this protocol and defer the formal description of all the protocols to Sections 4 and 5.

(1) ReLU' and MSB connection: We know that $\mathsf{ReLU}'(x)$ is 1 if $x \geq 0$ and 0 otherwise. Using our encoding system (refer to Appendix A), we see that $\mathsf{ReLU}'(x)$ is closely related to most-significant bit (MSB[3]) of x. That is, $\mathsf{ReLU}'(x)$ is 1 iff $\mathsf{MSB}(x) = 0$. Hence, it suffices to compute the $\mathsf{MSB}(x)$.

(2) Computing MSB: Next, since computing LSB of a number is much easier than computing the MSB (as it does not require bit extraction), we flip the problem of computing MSB to a problem of computing LSB as follows: $\mathsf{MSB}(a) = \mathsf{LSB}(2a)$ if we are working over an odd ring[4]. For now, let us assume that we are working over an odd ring and we later describe how we go from even ring $\mathbb{Z}_{2^{64}}$ to a odd ring $\mathbb{Z}_{2^{64}-1}$.

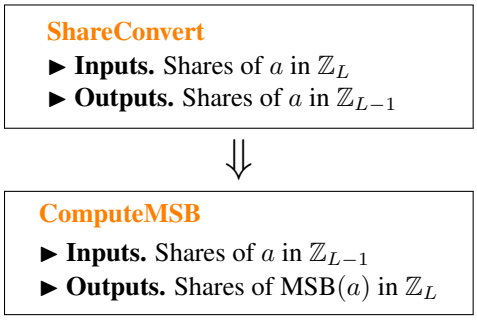

Figure 3: **Intuition behind the derivative of ReLU protocol**

At the start of the protocol, P_0 and P_1 hold shares of a (over $\mathbb{Z}_{2^{64}-1}$), using which they locally compute shares of $y = 2a$. Now, the helper party P_2 would assist in computing the LSB of y as follows: From now on, we denote $\mathsf{LSB}(y) = y[0]$. The first observation is that for three numbers u, v, w such that $u = v + w$, $u[0] = v[0] \oplus w[0]$ if the addition does not "wrap around" the ring and $u[0] = u[0] \oplus v[0] \oplus 1$ if addition wraps around. The second observation is that if x is a random number chosen by P_2 and is unknown to P_0 and P_1, then it is okay for P_0, P_1 to learn $r = y + x$. This is because the secret y is masked by random x. Hence,

[3] MSB refers to leftmost bit in the 64-bit representation of the argument

[4] In a group of order n, we have $\mathsf{MSB}(x) = 1$ iff $x > n/2$ iff $n > 2x - n > 0$; if n is odd, then so is $2x - n$ and it follows that $\mathsf{MSB}(x) = 1$ iff $\mathsf{LSB}(2x) = 1$.

P_2 gives secret shares of x as well as shares of $x[0]$ to P_0, P_1 and they reconstruct r. Now, all that is left is to figure out whether the addition $y + x$ wraps around the ring or not. For this, the third observation is that this addition wraps around if and only if the final sum is less than one of the individual values – that is, $x + y$ is greater than the ring iff $x > r$. Thus, if we can compute shares of $x > r$ between P_0 and P_1, then we are done.

(3) Specialized comparison protocol: Next, we construct a protocol called private compare (denoted by $\mathcal{F}_{\text{PC}}$) for comparison of a public integer with a private integer shared in a specific manner. This protocol assumes that P_0 and P_1 each have a share of the bits of $\ell-$bit value x (over some field $\mathbb{Z}_p$) as well as a common number r and a random bit β as input. This functionality computes the bit $(x > r)$ (which is 1 if $x > r$ and 0 otherwise) and XOR masks it with the bit β. This output $\beta \oplus (x > r)$ is given to the third party P_2. We implement this functionality by building on the techniques of [19, 34] and providing a much more efficient variant for the 3-server model. Note that this protocol requires parties to have shares of bits of x over field $\mathbb{Z}_p$. These would be provided to P_0, P_1 by P_2. With these protocols, we are in a position to compute the $\mathsf{ReLU}'(\cdot)$ function if P_0 and P_1 began with shares of the input over an odd ring.

(4) Converting shares: Now, we revisit the requirement of an odd ring. As we explained above, all of this works, if we had shares of a over an odd ring. Now, we could execute our protocol over the ring $\mathbb{Z}_N$ with N being odd. However doing so is fairly inefficient as matrix multiplication over the ring $\mathbb{Z}_{2^{64}}$ (or $\mathbb{Z}_{2^{32}}$) is much faster. This is because (as observed in [31]), native implementation of matrix multiplication over `long` (or `int`) automatically implements the modulo operation over $\mathbb{Z}_{2^{64}}$ (or $\mathbb{Z}_{2^{32}}$) and many libraries heavily optimize matrix multiplication over these rings, which give significant efficiency improvements compared to operations over any other ring. We then provide a protocol that converts values ($\neq L - 1$) that are secret shared over $\mathbb{Z}_L$ into shares over $\mathbb{Z}_{L-1}$. This protocol also uses the private compare protocol and may be of independent interest.

Steps (1)-(4) together give a protocol to compute the derivative of ReLU starting with 2-out-of-2 arithmetic sharing of the argument of ReLU between Parties P_0 and P_1. Steps (2), (3), (4) are building block protocols Compute MSB (Algorithm 5), Private Compare (Algorithm 3), and Share Convert (Algorithm 4) respectively. Our design enables us to run our comparison protocol (the protocol that realizes $\mathcal{F}_{\text{PC}}$ above) over a *small* field $\mathbb{Z}_p$ (we choose $p = 67$ concretely) and this reduces the communication complexity significantly. Other non-linear protocols are simple extensions of the derivative of ReLU protocol, with the exception of derivative of Maxpool, which is constructed exploiting specific number-theoretic properties.

3 PRELIMINARIES

3.1 Threat Model and Security

In this work, we consider full semi-honest security as well as privacy against malicious adversaries.

Semi-honest Security. A *semi-honest* (also known as honest-but-curious) adversary follows the protocol specifications honestly. We consider a semi-honest adversary that corrupts a single server (and any number of clients) and provide full security (i.e., privacy and correctness) of our protocols in the simulation paradigm [24, 13, 12]. We prove the security of our protocols using a simulation based argument.

Malicious Security. A *malicious* adversary can arbitrarily deviate from the protocol specification. Araki *et al.* [7] formalized the notion of privacy against malicious adversaries in the client-server model. Our protocol provides privacy (but not correctness) against a malicious adversary that corrupts any one of the three servers. Intuitively, privacy against malicious server guarantees that even a malicious adversary cannot break the privacy of inputs or outputs of the honest parties. Even though this notion is strictly weaker than simulation based malicious security (in particular, this does not guarantee correctness), it does guarantee that privacy is not compromised by malicious behavior.

Theoretically, our protocols are information theoretically secure. However, in practice, we use pseudorandom functions to generate shared randomness as well as point-to-point secure channels between all pairs of parties thereby relying on computational assumptions for implementational purposes.

3.2 Notation

In our protocols, we use additive secret sharing over the four rings $\mathbb{Z}_L$, $\mathbb{Z}_{L-1}$, $\mathbb{Z}_p$ and $\mathbb{Z}_2$, where $L = 2^\ell$ and p is a prime. Note that $\mathbb{Z}_{L-1}$ is a ring of odd size and $\mathbb{Z}_p$ is a field. We use 2-out-of-2 secret sharing and use $\langle x \rangle_0^t$ and $\langle x \rangle_1^t$ to denote the two shares of x over $\mathbb{Z}_t$ – specifically, the scheme generates $r \xleftarrow{\$} \mathbb{Z}_t$, sets $\langle x \rangle_0^t = r$ and $\langle x \rangle_1^t = x - r \pmod t$. We also use $\langle x \rangle^t$ to denote sharing of x over $\mathbb{Z}_t$ (we abuse notation and write $\langle x \rangle^B$ to

denote sharing of x over $\mathbb{Z}_2$). The algorithm $\mathsf{Share}^t(x)$ generates the two shares of x over the ring $\mathbb{Z}_t$ and algorithm $\mathsf{Reconst}^t(x_0, x_1)$ reconstructs a value x using x_0 and x_1 as the two shares over $\mathbb{Z}_t$ (reconstruction is simply $x_0 + x_1$ over $\mathbb{Z}_t$). Also, for any $\ell-$bit integer x, we use $x[i]$ to denote the i^{th} bit of x. Then, $\{\langle x[i]\rangle^t\}_{i\in[\ell]}$ denotes the shares of bits of x over $\mathbb{Z}_t$. For an $m \times n$ matrix X, we denote by $\langle X\rangle_0^t$ and $\langle X\rangle_1^t$ the matrices that are created by secret sharing the elements of X component-wise (other notation on X, such as $\mathsf{Reconst}^t(X_0, X_1)$ is similarly defined component-wise).

We assume that any pair of parties pre-share fresh shares of 0. In our implementation, the parties exchange a PRF key to generate common randomness efficiently. These shares of 0 can be used to "refresh" the secret shares (local addition by each party locally). When we use the term "fresh share" of some value x, we mean that the randomness used to generate the share of x has not been used anywhere else in that or any other protocol. In the following, we say "party P_i generates shares $\langle x\rangle_j^t$ for $j \in \{0, 1\}$ and sends to P_j" to mean "party P_i generates $(\langle x\rangle_0^t, \langle x\rangle_1^t) \leftarrow \mathsf{Share}^t(x)$ and sends $\langle x\rangle_j^t$ to P_j for $j \in \{0, 1\}$".

4 Building Block Protocols

In this section, we present our building block protocols. These can be combined together to enable the main protocols (described in Section 5) that correspond to various neural network operations such as linear/convolutional layers, ReLU, normalization etc. All of our protocols satisfy perfect correctness and standard simulation-based semi-honest security against a single server corruption. We defer the formal proofs of security to the full version. Finally, in Section 6, we argue that all our protocols satisfy the privacy against a malicious corruption of a single server in the client-server model as defined by [7].

4.1 Matrix Multiplication

Algorithm 1 describes our 3-party protocol for secure multiplication (functionality $\mathcal{F}_{\text{MATMUL}}$) where parties P_0 and P_1 hold shares of $X \in \mathbb{Z}_L^{m\times n}$ and $Y \in \mathbb{Z}_L^{n\times v}$ and the functionality outputs fresh shares of $Z = X \cdot Y$ to P_0, P_1.

Algorithm 1 Matrix Multiplication $\Pi_{\mathsf{MatMul}}(P_0, P_1, P_2)$:

Input: P_0 and P_1 hold shares of matrices X and Y.
Output: P_0 and P_1 hold shares of the matrix $X \cdot Y$.

1: P_2 generates random matrices U, V and $Z = U \cdot V$
2: **for** $i \in \{0, 1\}$ **do**
3: P_2 sends respective shares of U, V, and Z to P_i.
4: P_0, P_1 reconstruct A, B such that $\langle A\rangle_i^L = \langle X\rangle_i^L - \langle U\rangle_i^L$ and $\langle B\rangle_i^L = \langle Y\rangle_i^L - \langle V\rangle_i^L$.
5: **return** P_i outputs $\langle W\rangle_i^L = -A \cdot B + A \cdot \langle V\rangle_i^L + \langle U\rangle_i^L \cdot B + \langle Z\rangle_i^L$.
6: **end for**

Intuition: Our protocol relies on standard cryptographic technique for multiplication of using Beaver triplets [8] generalized to the matrix setting. P_2 generates these triplet shares and sends the shares to parties P_0, P_1.

4.2 Select Share

Algorithm 2 describes our 3-party protocol realizing the select share functionality $\mathcal{F}_{\text{SS}}$, which is as follows: Parties P_0, P_1 hold shares of x, y over $\mathbb{Z}_L$ and shares of a selection bit $\alpha \in \{0, 1\}$ over $\mathbb{Z}_L$. In the end, Parties P_0, P_1 get fresh shares of x if $\alpha = 0$ and fresh shares of y if $\alpha = 1$.

Algorithm 2 Select Share $\Pi_{\mathsf{SS}}(P_0, P_1, P_2)$:

Input: P_0, P_1 hold shares of $\alpha, x, y \in \mathbb{Z}_L$.
Output: P_0, P_1 get shares of $(1 - \alpha)x + \alpha y$.

1: P_0, P_1 compute $w = y - x$.
2: P_0, P_1, P_2 run Π_{MatMul} on α, w to get c.
3: **return** P_0, P_1 output $x + c$.

Intuition: We note that selecting between x and y can be arithmetically expressed as $(1-\alpha)\cdot x+\alpha\cdot y = x+\alpha\cdot(y-x)$. The last term is computed using a call to Π_{MatMul}.

4.3 Private Compare

Algorithm 3 describes our 3-party protocol realizing the functionality $\mathcal{F}_{\mathsf{PC}}$ for comparison that is as follows: The parties P_0 and P_1 holds shares of bits of ℓ-bit integer x in $\mathbb{Z}_p$, i.e., $\{\langle x[i]\rangle_0^p\}_{i\in[\ell]}$ and $\{\langle x[i]\rangle_1^p\}_{i\in[\ell]}$, respectively. P_0, P_1 also hold an ℓ-bit integer r and a bit β. At the end of the protocol, P_2 learns a bit $\beta' = \beta \oplus (x > r)$, where $(x > r)$ denotes the bit which is 1 when $x > r$ over the integers and 0 otherwise.

Algorithm 3 PrivateCompare $\Pi_{\mathsf{PC}}(P_0, P_1, P_2)$:

Input: P_0, P_1 hold shares of bits of x in $\mathbb{Z}_p$, a public ℓ bit integer r and a common random bit β.
Output: P_2 gets the bit $\beta' = \beta \oplus (x > r)$.
Common Randomness: P_0, P_1 have a common random permutation π over ℓ numbers and ℓ common random values $\{s_i\}_{i=1}^{\ell}$ from $\mathbb{Z}_p^*$.

1: P_0, P_1 perform local computations for Steps 2-7. All shares are over $\mathbb{Z}_p$.
2: **for** $i = \{\ell, \ell-1, \ldots, 1\}$ **do**
3: Compute shares of $w[i] = x[i] \oplus r[i]$
4: Compute shares of $c[i] = (-1)^{\beta}(x[i] - r[i]) + 1 + \sum_{k=i+1}^{\ell} w[k]$.
5: **end for**
6: Compute share of $d[i] = s_i \cdot c[i]$.
7: Permute $\{d[i]\}_{i=1}^{\ell}$ according to π and send the shares to P_2.
8: P_2 reconstructs $\{d[i]\}_{i=1}^{\ell}$ and sets $\beta' = 0$.
9: **if** $\exists i \in \{1, 2, \ldots, \ell\}$ s.t $d_i = 0$ **then**
10: $\beta' = 1$
11: **end if**
12: **return** P_2 outputs β'.

Intuition: Our starting point is the idea for 2-party comparison present in [19, 34]. We build on this to give a much more efficient information theoretic 3-party protocol. We want to compute $\beta' = \beta \oplus (x > r)$. That is, for $\beta = 0$, we compute $x > r$ and for $\beta = 1$, we compute $1 \oplus (x > r) \equiv (x \leq r) \equiv (x < (r + 1))$ over integers. In the corner case of $r = 2^{\ell} - 1$, $x \leq r$ is always true.

Consider the case of $\beta = 0$. In this case, $\beta' = 1$ iff $(x > r)$ or at the leftmost bit where $x[i] \neq r[i]$, $x[i] = 1$. We compute $w_i = x[i] \oplus r[i] = x[i] + r[i] - 2x[i]r[i]$ and $c[i] = r[i] - x[i] + 1 + \sum_{k=i+1}^{\ell} w_k$. Since r is known to both P_0, P_1, shares of both w_i and c_i can be computed locally. Now, we can prove that $\exists\ i.c_i = 0$ iff $x > r$. Hence, both P_0, P_1 send shares of c_i to P_2 who reconstructs c_i and looks for a 0. To ensure security against a corrupt P_2, we hide exact values of non-zero c_i's and position of (a possible) 0 by multiplying with random s_i and permuting these values by a common permutation π. These s_i and π are common to both P_0 and P_1.

In the case when $\beta = 1$ and $r \neq 2^{\ell} - 1$, we compute $(r + 1) > x$ using similar logic as above. In the corner case of $r = 2^{\ell} - 1$, both parties P_0, P_1 know that result of $x \leq r \equiv (r + 1) > x$ over integers should be true. Hence, they together pick shares of c_i such that there is exactly one 0. This is done by P_0, P_1 having common values u_i that they use to create a valid share of a 0 and $\ell - 1$ shares of 1 (see Step 11 in the published version).

4.4 Share Convert

Algorithm 4 describes our three-party protocol for converting shares over $\mathbb{Z}_L$ to $\mathbb{Z}_{L-1}$ realizing the functionality $\mathcal{F}_{\text{SC}}$. Here, parties P_0, P_1 hold shares of $\langle a\rangle^L$ such that $a \neq L - 1$. At the end of the protocol, P_0, P_1 hold fresh shares of same value over $L - 1$, i.e., $\langle a\rangle^{L-1}$.

In this algorithm, $\alpha := \mathsf{wrap}(x, y, L)$ is 1 if $x + y \geq L$ over integers and 0 otherwise. That is, α denotes the wrap-around bit for the computation $x + y \mod L$.

Intuition: Let $\theta = \mathsf{wrap}(\langle a\rangle_0^L, \langle a\rangle_1^L, L)$. Now, we note that if $\theta = 1$, i.e., if the original shares wrapped around L, then we need to subtract 1, else original shares are also valid shares of same value of $L - 1$. Hence, in the protocol we compute shares of bit θ over $L - 1$ and subtract from original shares locally. This protocol makes use of novel modular arithmetic to securely compute these shares of θ, an idea which is potentially of independent interest. We explain these relations in the correctness proof below.

Lemma 1. *Protocol* $\Pi_{\mathsf{SC}}(\{P_0, P_1\}, P_2)$ *in Algorithm 4 realizes* $\mathcal{F}_{\text{SC}}$.

Algorithm 4 ShareConvert $\Pi_{\mathsf{SC}}(P_0, P_1, P_2)$:

Input: P_0, P_1 hold shares of $a \in \mathbb{Z}_L$, $a \neq L-1$
Output: P_0, P_1 get shares of $a \in \mathbb{Z}_{L-1}$
Common Randomness: P_0, P_1 hold a random bit η''.

1: P_0, P_1 generate shares $\langle r\rangle_0^L, \langle r\rangle_1^L$ of a random number r and $\alpha = \mathsf{wrap}(\langle r\rangle_0^L, \langle r\rangle_1^L, L)$.
2: P_0, P_1 locally compute $\langle \tilde{a}\rangle_j^L = \langle a\rangle_j^L + \langle r\rangle_j^L$ and $\beta_j = \mathsf{wrap}(\langle a\rangle_j^L, \langle r\rangle_j^L, L)$.
3: P_0, P_1 send their shares of $\tilde{a}$ to P_2.

4: P_2 reconstructs $x \equiv \langle \tilde{a}\rangle_0^L + \langle \tilde{a}\rangle_1^L \pmod L$ and $\delta = \mathsf{wrap}(\langle \tilde{a}\rangle_0^L, \langle \tilde{a}\rangle_1^L, L)$.
5: P_2 generates shares of bits of $x \in \mathbb{Z}_p$ and shares of $\delta \in \mathbb{Z}_{L-1}$.
6: P_2 sends the above shares to P_0 and P_1.

7: P_0, P_1 and P_2 run Π_{PC} on shares of bits of $x \in \mathbb{Z}_p, r$ and η''.
8: P_2 learns η', the output of Π_{PC}.
9: P_2 generates shares of $\eta' \in \mathbb{Z}_{L-1}$ and sends them to P_0 and P_1.

10: P_0 and P_1 compute shares of $\eta = 1 \oplus \eta' \oplus \eta''$.
11: P_0 and P_1 compute shares of $\theta = \beta_0 + \beta_1 + \eta - \delta - \alpha$ over $\mathbb{Z}_{L-1}$.
12: **return** P_0 and P_1 output $\langle y\rangle_j^{L-1} = \langle a\rangle_j^L - \langle \theta\rangle_j^{L-1}$ (over $L-1$) for $j \in \{0, 1\}$.

Proof. For correctness we need to prove that $\mathsf{Reconst}^{L-1}(\langle y\rangle_0^{L-1}, \langle y\rangle_1^{L-1}) = \mathsf{Reconst}^{L}(\langle a\rangle_0^{L}, \langle a\rangle_1^{L}) = a$. Looking at Step 11 of the protocol and the intuition above, it suffices to prove that $\mathsf{Reconst}^{L-1}(\langle \theta\rangle_0^{L-1}, \langle \theta\rangle_1^{L-1}) = \theta = \mathsf{wrap}(\langle a\rangle_0^L, \langle a\rangle_1^L, L)$. First, by correctness of protocol Π_{PC}, $\eta' = \eta'' \oplus (x > r-1)$. Next, let $\eta = \mathsf{Reconst}^{L-1}(\langle \eta\rangle_0^{L-1}, \langle \eta\rangle_1^{L-1}) = \eta' \oplus \eta'' = (x > r-1)$. Next, note that $x \equiv a + r \mod L$. Hence, $\mathsf{wrap}(a, r, L) = 0$ iff $x > r-1$. By the correctness of wrap, following relations hold over the integers:

(1) $r = \langle r\rangle_0^L + \langle r\rangle_1^L - \alpha L$.

(2) $\langle \tilde{a}\rangle_j^L = \langle a\rangle_j^L + \langle r\rangle_j^L - \beta_j L$.

(3) $x = \langle \tilde{a}\rangle_0^L + \langle \tilde{a}\rangle_1^L - \delta L$.

(4) $x = a + r - (1-\eta)L$.

(5) Let θ be such that $a = \langle a\rangle_0^L + \langle a\rangle_1^L - \theta L$.

Computing, $(1) - (2) - (3) + (4) + (5)$ gives us $\theta = \beta_0 + \beta_1 - \alpha + \delta + \eta - 1$. This is exactly, what the parties P_0 and P_1 calculate in Step 10. □

4.5 Compute MSB

Algorithm 5 describes our 3-party protocol realizing the functionality $\mathcal{F}_{\text{MSB}}$ that computes the most significant bit[5] (MSB) of a value $a \in \mathbb{Z}_{L-1}$. P_0, P_1 hold shares of a over odd ring $\mathbb{Z}_{L-1}$ and end with shares of $\mathsf{MSB}(a)$ over $\mathbb{Z}_L$.

Intuition: Note that when the shares of the private input (say a) are over an odd ring (such as after using Π_{SC}), the MSB computation can be converted into an LSB computation. More precisely, over an odd ring, $\mathsf{MSB}(a) = \mathsf{LSB}(y)$, where $y = 2a$. Now, P_2 assists in computation of shares of $\mathsf{LSB}(y)$ as follows: P_2 picks a random integer $x \in \mathbb{Z}_{L-1}$ and sends shares of x over $\mathbb{Z}_{L-1}$ and shares of $x[0]$ over $\mathbb{Z}_L$ to P_0, P_1. Next, P_0, P_1 compute shares of $r = y + x$ and reconstruct r by exchanging shares. We note that $\mathsf{LSB}(y) = y[0] = r[0] \oplus x[0] \oplus \mathsf{wrap}(y, x, L-1)$ over an odd ring. Also, $\mathsf{wrap}(y, x, L-1) = (x > r)$, which can be computed using comparison protocol Π_{PC}. To enable this, P_2 also secret shares $\{x[i]\}_{i\in[\ell]}$ over $\mathbb{Z}_p$. Steps 6-10 compute the equation $\mathsf{LSB}(y) = y[0] = r[0] \oplus x[0] \oplus (x > r)$ by using the arithmetic equation for XOR computation (note that $x \oplus r = x + r - 2xr$.

[4] In the corner case when $r = 0$, both P_0 and P_1 set the output of Π_{PC} to be 1 and execute it. This is similar to the other corner case discussed in Section 4.3.

[5] Most significant bit of a number is defined as the value of the leftmost bit in the bit representation.

Algorithm 5 ComputeMSB $\Pi_{\mathsf{MSB}}(P_0, P_1, P_2)$:

Input: P_0, P_1 hold shares of $a \in \mathbb{Z}_{L-1}$.
Output: P_0, P_1 get shares of $\mathsf{MSB}(a) \in \mathbb{Z}_2$.
Common Randomness: P_0, P_1 hold a random bit β.

1: P_2 generates a random $r \in \mathbb{Z}_{L-1}$ and generates shares of $r \in \mathbb{Z}_{L-1}$, shares of bits of $r \in \mathbb{Z}_p$ and shares of LSB of $r \in \mathbb{Z}_L$ and sends appropriate shares to P_0, P_1.
2: P_0, P_1 reconstruct $c \equiv 2a + r \pmod{L-1}$.
3: P_0, P_1 and P_2 run Π_{PC} on shares of bits of $r \in \mathbb{Z}_p$, c and β.
4: P_2 learns β', the output of Π_{PC}.
5: P_2 generates shares of $\beta' \in \mathbb{Z}_2$ and sends them to P_0 and P_1.
6: P_0 and P_1 compute shares of $\gamma = \beta \oplus \beta'$.
7: P_0 and P_1 compute shares of $\delta = \mathrm{LSB}[c] \oplus \mathrm{LSB}[r]$.
8: **return** P_0 and P_1 output shares of $\alpha = \gamma \oplus \delta$.

5 Main Protocols

In this section, we describe all our main protocols for functionalities such as linear layer, derivative of ReLU, ReLU, division needed for normalization during training, Maxpool and its derivative. We maintain the invariant that parties P_0 and P_1 begin with "fresh" shares of input value (over $\mathbb{Z}_L$) and output a "fresh" share of the output value (again over $\mathbb{Z}_L$) at the end of the protocol. Party P_2 takes the role of "assistant" in all protocols and has no input or output.

5.1 Linear and Convolutional Layer

We note that a linear (or fully connected) layer in a neural network is exactly a matrix multiplication. Similarly, a convolutional layer can also be expressed as a (larger) matrix multiplication. As an example the 2-dimensional convolution of a 3×3 input matrix X with a kernel K of size 2×2 can be represented by the matrix multiplication shown below.

$$\mathsf{Conv2d}\left(\begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix}, \begin{bmatrix} k_1 & k_2 \\ k_3 & k_4 \end{bmatrix}\right) = \begin{bmatrix} x_1 & x_2 & x_4 & x_5 \\ x_2 & x_3 & x_5 & x_6 \\ x_4 & x_5 & x_7 & x_8 \\ x_5 & x_6 & x_8 & x_9 \end{bmatrix} \times \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{bmatrix}$$

For a generalization, see e.g. [5] for an exposition on convolutional layers. Hence both these layers can be directly implemented using Algorithm 1 from Section 4.

5.2 Derivative of ReLU

Algorithm 6 describes our 3-party protocol for realizing the functionality $\mathcal{F}_{\text{DReLU}}$ that computes the derivative of ReLU, denoted by ReLU'. Parties P_0, P_1 hold secret shares of a over ring $\mathbb{Z}_L$ and end up with secret shares of $\mathsf{ReLU}'(a)$ over $\mathbb{Z}_L$. Note that $\mathsf{ReLU}'(a) = 1$ if $\mathsf{MSB}(a) = 0$, else $\mathsf{ReLU}'(a) = 0$.

Algorithm 6 ReLU', $\Pi_{\mathsf{DReLU}}(P_0, P_1, P_2)$:

Input: P_0, P_1 hold shares of $a \in \mathbb{Z}_L$.
Output: P_0, P_1 get shares of $\mathsf{ReLU}'(a) \in \mathbb{Z}_L$.

1: P_0, P_1 compute shares of $c = 2a$ (hence $c \neq L - 1$).
2: P_0, P_1, P_2 run $\Pi_{\mathsf{SC}}(P_0, P_1, P_2)$ on shares of c to get $y \in \mathbb{Z}_{L-1}$
3: P_0, P_1, P_2 run $\Pi_{\mathsf{MSB}}(P_0, P_1, P_2)$ with on y to get shares of $\alpha = \mathrm{MSB}(y)$.
4: P_0, P_1 output $1 - \alpha$

Intuition: As is clear from the function ReLU' itself, the protocol computes the shares of $\mathsf{MSB}(a)$ and flips it to compute $\mathsf{ReLU}'(a)$. Recall that the protocol Π_{MSB} expects shares of a over $\mathbb{Z}_{L-1}$. Hence, we need to convert shares over $\mathbb{Z}_L$ to fresh shares over $\mathbb{Z}_{L-1}$ of the same value. Recall that for correctness of the share convert protocol, we

require that value is not equal to $L-1$. This is ensured by first computing shares of $c = 2a$ and then calling Π_{SC}. We ensure[6] that $\mathsf{ReLU}'(a) = \mathsf{ReLU}'(c)$ by requiring that $a \in [0, 2^k) \cup (2^\ell - 2^k, 2^\ell - 1]$, where $k < \ell - 1$.

5.3 ReLU

Algorithm 7 describes our 3-party protocol for realizing the functionality $\mathcal{F}_{\text{RELU}}$ that computes $\mathsf{ReLU}(a)$. Parties P_0, P_1 hold secret shares of a over ring $\mathbb{Z}_L$ and end up with secret shares of $\mathsf{ReLU}(a)$ over $\mathbb{Z}_L$. Note that $\mathsf{ReLU}(a) = a$ if $\mathsf{MSB}(a) = 0$, else 0. That is, $\mathsf{ReLU}(a) = \mathsf{ReLU}'(a) \cdot a$.

Intuition: Our protocol implements the above relation by using one call each to Π_{DReLU} and Π_{MatMul}. Note that Π_{MatMul} is invoked for multiplying two matrices of dimension 1×1 (or just one integer multiplication).

Algorithm 7 ReLU, $\Pi_{\mathsf{ReLU}}(P_0, P_1, P_2)$:

Input: P_0, P_1 hold shares of $a \in \mathbb{Z}_L$.
Output: P_0, P_1 get shares of $\mathsf{ReLU}(a) \in \mathbb{Z}_L$.

1: P_0, P_1, P_2 run $\Pi_{\mathsf{DReLU}}(P_0, P_1, P_2)$ on shares of a to get shares of α.
2: P_0, P_1, P_2 run $\Pi_{\mathsf{MatMul}}(P_0, P_1, P_2)$ on shares of α and a to get c.
3: P_0, P_1 output shares of c.

5.4 Division

Algorithm 8 describes a protocol to securely realize the functionality $\mathcal{F}_{\text{DIV}}$. Parties P_0, P_1 hold shares of x and y over $\mathbb{Z}_L$. At the end of the protocol, parties P_0, P_1 hold shares of $\lfloor x/y \rfloor$ over $\mathbb{Z}_L$ when $y \neq 0$.

Algorithm 8 Division $\Pi_{\mathsf{Div}}(P_0, P_1, P_2)$:

Input: P_0, P_1 hold shares of $x, y \in \mathbb{Z}_L$, $x \leq y$.
Output: P_0, P_1 get shares of x/y (up to a fixed precision f).

1: P_0, P_1 initialize $p = 0$ and $w = 0$.
2: P_0, P_1, P_2 perform Steps 3-10.
3: **for** $i = \{1, 2, \ldots, f\}$ **do**
4: Run Π_{ST} on y to get shares of $z = y/2^i$.
5: Run Π_{DRELU} on input $(x - p - z)$ to get output bit b.
6: Run Π_{SS} on input z and selection bit b to get shares d.
7: Run Π_{SS} on input $2^{(f-i)}$ and selection bit b to get shares q.
8: $p \leftarrow p + d$.
9: $w \leftarrow w + q$.
10: **end for**
11: **return** P_0, P_1 output their share of w.

Intuition: Our protocol implements long division where the quotient is computed bit-by-bit sequentially starting from the most significant bit. In each iteration, we compute the current dividend by subtracting the correct multiple of the divisor. Then we compare the current dividend with a multiple of the divisor ($2^i y$ in round i). Depending on the output of the comparison, i^{th} bit of the quotient is 0 or 1. This comparison can be written as a comparison with 0 and hence can be computed using a single call to Π_{DReLU}. We use this selection bit to select between 0 and 2^i for quotient and 0 and $2^i y$ for what to subtract from dividend. This selection can be implemented using Π_{MatMul} (similar to ReLU computation). Hence, division protocol proceeds in iterations and each iteration makes one call to Π_{DReLU} and one call[7] to Π_{MatMul}.

5.5 Maxpool

Algorithm 9 describes our 3-party protocol realizing the functionality $\mathcal{F}_{\text{MAXPOOL}}$ to compute the maximum of n values. Parties P_0, P_1 hold shares of $\{x_i\}_{i \in [n]}$ over $\mathbb{Z}_L$ and end up with fresh shares of $\mathsf{max}(\{x_i\}_{i \in [n]})$.

[6]This essentially means that the absolute value of a is not very large, and in particular not larger than 2^k. This is not a limitation in any of the ML applications that we work with.

[7]Note that multiplication with 2^i can be done locally.

Algorithm 9 Maxpool $\Pi_{\mathsf{MP}}(P_0, P_1, P_2)$:

Input: P_0, P_1 hold shares of a vector $\{x_i\}_{i\in[n]}$.
Output: P_0, P_1 get shares of z where $z = \max_{i\in[n]}\{x_i\}$ and shares of index where $\text{index} = \text{argmax}_{i\in[n]}\{x_i\}$.

1: P_0, P_1 set max equal to shares of x_0, index $= 0$, and $\{\text{indexShares}_i\}_{i=0}^{n-1}$ equal to shares of i.
2: **for** $i = \{1, \ldots, n-1\}$ **do**
3: Set diff $=$ max $- x_i$ and diffIndex $=$ maxIndex $- y_i$.
4: P_0, P_1, P_2 call $\Pi_{\mathsf{DReLU}}(P_0, P_1, P_2)$ to get shares $\beta = \mathsf{ReLU}'(\text{diff})$.
5: P_0, P_1, P_2 call $\Pi_{\mathsf{SS}}(P_0, P_1, P_2)$ on bit β and diff and get u.
6: P_0, P_1, P_2 call $\Pi_{\mathsf{SS}}(P_0, P_1, P_2)$ on bit β and diffIndex and get v.
7: max $\leftarrow$ max $+ u$
8: index $\leftarrow$ index $+ v$
9: **end for**
10: P_0, P_1 output max and index.

Intuition: The protocol implements the max algorithm that runs in $(n-1)$ sequential steps. We start with $\mathsf{max}_1 = x_1$. In step i, we compute the shares of $\mathsf{max}_i = \mathsf{max}(x_1, \ldots, x_i)$ as follows: We compute shares of $w_i = x_i - \mathsf{max}_{i-1}$. Then, we compute shares of $\beta_i = \mathsf{ReLU}'(w_i)$ that is 1 if $x_i \geq \mathsf{max}_{i-1}$ and 0 otherwise. Next, we use Π_{SS} to select between max_{i-1} and x_i using β_i to compute max_i. Note, that in a similar manner, we can also calculate the index of maximum value, i.e. k such that $x_k = \mathsf{max}(\{x_i\}_{i\in[n]})$ (steps 6 &7 in the published version). Computing the index of max value is required while doing prediction as well as to compute the derivative of Maxpool activation function needed for backpropagaton during training.

5.6 Derivative of Maxpool

The derivative of the Maxpool function (functionality $\mathcal{F}_{\text{DMAXPOOL}}$) is defined as the unit vector with a 1 only in the position with the maximum value. Here, we describe the more efficient Algorithm 10 that works for the special (and often-used) case of 2×2 Maxpool, where $n = 4$. In general, this algorithm works when n divides L. For the more general case, we provide the algorithm in the full version.

Algorithm 10 Efficient Derivative of $n_1 \times n_2$ Maxpool $\Pi_{\mathsf{n_1xn_2DMP}}(P_0, P_1, P_2)$ with $n \mid L, n = n_1 n_2$:

Input: P_0, P_1 hold shares of a vector $\{x_i\}_{i\in[n]}$.
Output: P_0, P_1 get shares of a vector $\{z_i\}_{i\in n]}$ where $z_i = 1$ if $i = \text{argmax}_{i\in[n]}\{x_i\}$ and $z_i = 0$ otherwise.
Common Randomness: P_0 and P_1 hold a random $r \in \mathbb{Z}_L$.

1: P_0, P_1, P_2 run Π_{MP} with shares of $\{x_i\}_{i\in[n]}$ to obtain shares of $\text{index} = \text{argmax}_{i\in[n]}\{x_i\}$.
2: P_0, P_1 blind shares of index with r and send $(\text{index} + r)$ to P_2.
3: P_2 reconstructs $\text{index} + r$ and computes $k = \text{index} + r \pmod n$.
4: P_2 constructs shares $\{e_i\}_{i\in[n]}$ such that $e_i = 1$ if $i = k$ and $e_i = 0$ otherwise.
5: P_0 and P_1 locally reverse "cyclic-shift" their shares by $g = r \pmod n$ to obtain a vector $\{z_i\}_{i\in[n]}$.
6: P_0, P_1 output the cyclically shifted vector $\{z_i\}_{i\in[n]}$.

Intuition: The key observation behind this protocol is that when n divides L (i.e., $n \mid L$), we have that $a \bmod n = (a \bmod L) \bmod n$. The first step that P_0 and P_1 run is Π_{MP} that gives them shares of the index $\mathsf{ind} \in [n]$ with the maximum value. These shares are over L and must be converted into shares of the unit vector E_{ind} which is a length n vector with 1 in position ind and 0 everywhere else. P_0 and P_1 share a random $r \in \mathbb{Z}_n$ and have P_2 reconstruct $k = (\mathsf{ind} + r) \bmod n$. P_2 then creates shares of E_k and sends the shares back to P_0 and P_1 who "left-shift" these shares by r to obtain shares of E_{ind}. This works because $a \bmod n = (a \bmod L) \bmod n$ is true when $n|L$.

5.7 End-to-end Protocols

Our main protocols can be easily put together to execute training on a wide class of neural networks. For example, consider Network A, 3-layer neural network from SecureML that consists of a fully connected layer, followed by a ReLU, followed by another fully connected layer, followed by another ReLU, followed by the function $\mathsf{ASM}(u_i) =$

$\frac{\mathsf{ReLU}(u_i)}{\sum \mathsf{ReLU}(u_i)}$ (for further details on this network, we refer the reader to [31]). To implement this, we first invoke Π_{MatMul}, followed by Π_{ReLU}, then again followed by Π_{MatMul} and Π_{ReLU} and finally we invoke Π_{DIV} to compute $\mathsf{ASM}(\cdot)$[8]. Backpropagaton is computed by making calls to Π_{MatMul} as well and Π_{DReLU} with appropriate dimensions[9]. Similarly, we can also do a general convolutional neural network with other activations such as Maxpool. We remark that we can put together these protocols easily since our protocols all maintain the invariant that parties begin with arithmetic shares of inputs and complete the protocol with arithmetic shares of the output.

6 Malicious Adversary

In this section, we show that all our protocols described in Sections 4 and 5 as well as protocols for general neural networks obtained by putting these together satisfy stronger security requirement than semi-honest security, namely, privacy against a malicious server in the client-server model (formalized by [7]). As was already pointed out by Araki et al. [7], this can only be achieved when the servers receive no information about the output of the protocol. Formally, we show that, for any malicious server, for any two inputs of the honest clients (holding the data) the view of the server is indistinguishable.

First, we show that views are identical with information theoretic correlated randomness. This holds because in all our protocol, the incoming messages to a server are either a fresh share of a value or can be generated using a uniformly random value (e.g., incoming messages of P_2 in private-compare protocol). Thus, irrespective of what the adversary sends in each round, the view of a malicious server can be simulated using uniform randomness and is completely independent of the inputs being used by the clients.

Second, in the case when correlated randomness is generated using shared PRF keys, to argue security against malicious P_0, we rely on security of the PRF key shared between P_1, P_2 that is unknown to P_0. Using this, we show that incoming messages of P_0 are computationally close to uniform. It is critical that to argue security against a malicious P_0 we do not rely on security of PRF keys known to P_0, i.e. shared keys between P_0, P_1 or P_0, P_2. Hence, we do not need to use a malicious secure coin-tossing protocol to generate secure keys between an adversary and an honest server. We only rely on the security of the PRF key shared between two honest servers. Therefore, the exact same protocol gives privacy against a single malicious server. Similar arguments can be made to argue security against a malicious P_1 or malicious P_2.

7 Protocol Complexities

Protocol	Rounds	Communication
$\mathsf{MatMul}_{m,n,v}$	2	$2(2mn + 2nv + mv)\ell$
$\mathsf{MatMul}_{m,n,v}$ (with PRF)	2	$(2mn + 2nv + mv)\ell$
SelectShare	2	5ℓ
PrivateCompare	1	$2\ell \log p$
ShareConvert	4	$4\ell \log p + 6\ell$
Compute MSB	5	$4\ell \log p + 13\ell$

Table 1: **Round & communication complexity of *building blocks*.**

7.1 Overheads of building block protocols

The communication and round complexity of our building block protocols is provided in Table 1. $\mathsf{MatMul}_{m,n,v}$ denotes matrix multiplication of an $m \times n$ matrix with an $n \times v$ matrix. The first row states the complexity of $\mathsf{MatMul}_{m,n,v}$ using information-theoretically secure Beaver triplets. In our implementation, we generate the triplets using PRFs as follows: P_0 and P_2 share a PRF key and use it to generate $\langle A\rangle_0^L, \langle B\rangle_0^L, \langle C\rangle_0^L$ locally. Similarly, P_1 and P_2 share a PRF key and use it to generate $\langle A\rangle_1^L, \langle B\rangle_1^L$ locally. Now, P_2 sets $\langle C\rangle_1^L = \mathsf{Reconst}^L(\langle A\rangle_0^L, \langle A\rangle_1^L) \cdot \mathsf{Reconst}^L(\langle B\rangle_0^L, \langle B\rangle_1^L) - \langle C\rangle_0^L$ and send to P_1. This reduces the communication of multiplication by half. All other complexities are for single elements and use this optimized version of multiplication.

[8] $\mathsf{ASM}(\cdot)$ consists of a summation and a division. Summation is a local computation and does not require a protocol to be computed.

[9] We note that Π_{DReLU} is called as part of Π_{ReLU} in forward propagation and its value is stored for backpropagaton

Protocol	Rounds	Communication
$\mathsf{Linear}_{m,n,v}$	2	$(2mn + 2nv + mv)\ell$
$\mathsf{Conv2d}_{m,i,f,o}$	2	$(2m^2f^2i + 2f^2oi + m^2o)\ell$
DReLU	8	$8\ell \log p + 19\ell$
ReLU (after DReLU)	2	5ℓ
NORM(l_D) or DIV(l_D)	$10l_D$	$(8\ell \log p + 24\ell)l_D$
$\mathsf{Maxpool}_n$	$9(n-1)$	$(8\ell \log p + 29\ell)(n-1)$
DMP_n (after Maxpool)	2	$2(n+1)\ell$

Table 2: **Round & communication complexity of *main protocols*.**

7.2 Overheads of main protocols

The round and communication complexity of our main protocols are presented in Table 2. The function $\mathsf{Linear}_{m,n,v}$ denotes a matrix multiplication of dimension $m \times n$ with $n \times v$. $\mathsf{Conv2d}_{m,i,f,o}$ denotes a convolutional layer with input $m \times m$, i input channels, a filter of size $f \times f$, and o output channels. l_D denotes precision of bits. $\mathsf{Maxpool}_n$ and DMP_n denotes Maxpool and its derivative over n elements. For ReLU and DMP_n, the overheads in addition to DReLU and $\mathsf{Maxpool}_n$ respectively are presented as these protocols are always implemented together in a neural network. All communication is measured for $\ell-$bit inputs and p denotes the field size (which is 67 in our case). All of the complexities are presented using the optimized complexity of multiplication that used PRFs for generating correlated randomess.

Our gains mainly come from the secure evaluation of non-linear functions such as ReLU and Maxpool and their derivatives. Prior works such as SecureML [31], MiniONN [29], Gazelle [25], etc took a garbled circuit-based approach to evaluate these functions – i.e., after completion of an arithmetic (linear) computation such as matrix multiplication, they ran a protocol to convert shares of intermediary values into an encoding suitable for garbled circuits. The non-linear function was then evaluated using the garbled circuit after which shares were once again converted back to be suitable for arithmetic computation. This approach leads to a multiplicative factor communication overhead proportional to the security parameter κ, as garbled circuits require communicating encodings proportional to κ, for every bit in the circuit. Overall, this leads to a communication complexity $> 768\ell$ for every $\ell-$ bit input [20]. As shown in [20], this cost of conversion to garbled circuits is $6\kappa\ell$, and all previous works incur this cost. In our approach, we provide new protocols to compute such non-linear activation functions, while continuing to retain arithmetic shares of the output values. For example, the ReLU protocol that we construct avoids paying κ multiplicative overhead and has communication complexity of $8\ell \log p + 24\ell$, which is approximately 88ℓ (when $p = 67$ as is in our setting). This leads to $> 8\times$ improvement in the communication complexity of the protocols for non-linear functions.

8 Systems Evaluation

8.1 System Details

We test our prototype by running experiments over Amazon EC2 c4.8x large instances in two environments, respectively modeling a LAN and WAN setting.

- **LAN setting:** We use 3 Amazon EC2 c4.8xlarge machines running Ubuntu in the same region. At the time of running the experiments, the average bandwidth was 625MB/s and the average ping time was 0.22ms.
- **WAN setting:** In the WAN setting, we rent machines in different geographical regions with the same machine specifications as in the LAN setting. At the time of running the experiments, the average bandwidth was 40MB/s and the average ping time was 58ms.

Our system is implemented in about 7400 lines of C++ code with the use of standard libraries. We use the Eigen Library [1] for faster matrix multiplications. The ring is set to $\mathbb{Z}_{2^{64}}$ and we use the `uint64_t` native C++ datatype for all variables. As noted in [31], compared to using a field for the underlying protocols or using dedicated number theoretic libraries such as NTL [4], this has the benefit of implementing modulo operations for free.

Neural Networks generally work over floating point numbers. To make them compatible with efficient cryptographic techniques, they must be encoded into fixed-point form. We use the methodology from SecureML to support fixed-point arithmetic in an integer ring (described in Appendix A). We use 13-bits of precision for our implementation as found sufficient in SecureML (cleartext training to get accuracy numbers is also done with these parameters). We evaluate our end-to-end implementation over 3 popular neural networks for the MNIST dataset [3]. These networks are briefly described in Fig. 4 and in detail in Appendix C.

Network A: SecureML [31]
3-Layer DNN with fully connected layers, ReLU activations and normalization, accuracy 93.4%

Network C: LeNet [26]
4-Layer CNN, larger version of Network B, ReLU & Maxpool activations, accuracy 99.15%

Network B: MiniONN [29]
4-Layer CNN with 2 convolutional layers, ReLU & Maxpool activations, accuracy 98.77%

Network D: Chameleon [35]
3-Layer CNN, ReLU & Maxpool activations (stride = 2x2), accuracy 99%

Figure 4: **Different neural networks used for evaluations in this work. Refer to Appendix C for more details.**

8.2 Summary of experiments

We develop a prototype implementation SecureNN for our three-party secure computation protocols. We test the performance of our protocols by training 3 different neural networks over the MNIST dataset [3]. We also evaluate SecureNN on secure inference benchmarks in Section 8.4. Finally, in Section 8.5, we present microbenchmarks that measure the performance of various sub-protocols implemented in SecureNN such as Linear Layer, Convolutional Layer, ReLU and Maxpool (and its derivatives) that enables the estimation of the performance cost of other networks using the above functions. Our times for secure training are extrapolated from 10 iterations and for inference all times are averaged over 10 executions. We consider overall execution time (and do not split execution time into an offline, data independent phase, and an online, data dependent phase). The learning rate is 2^{-5} in all experiments, except in the Network A (described below), where we retain prior works' learning rate of 2^{-7}.

	Epochs	Accuracy	LAN (hours)	WAN (hours)
A	15	93.4%	1.03	7.83
B	5	97.94%	5.8	17.99
	10	98.05%	11.6	35.99
	15	98.77%	17.4	53.98
C	5	98.15%	9.98	30.66
	10	98.43%	19.96	61.33
	15	99.15%	29.95	91.99

Table 3: **Secure training execution times for batch size 128.**

	Batch size	Accuracy	LAN (hours)	WAN (hours)
B	4	99.15%	9.98	112.71
	16	98.99%	8.34	36.46
	128	97.94%	5.8	17.99
C	4	99.01%	18.31	123.96
	16	99.1%	13.43	46.2
	128	98.15%	9.98	30.66

Table 4: **Secure training execution times for 5 epochs.**

8.3 Secure Training

We evaluate our protocols for secure training in both the LAN and WAN settings over the Networks A, B, and C (Fig. 4). For some of these networks, we achieve more than 99% accuracy for inference. We remark that we are the first work to show the feasibility of secure training on large and complex NNs such as CNNs that achieve high levels of accuracy. We vary the epochs between 5 and 15 for all networks except Network A which does not achieve good accuracy for smaller epochs and vary the batch size between 4 and 128 for networks B and C. Table 3 presents a summary of our results in the LAN/WAN setting as a function of the number of epochs for training (batch size fixed to 128), while Table 4 presents the results when the batch size is varied and the number of epochs is fixed to 5.

Comparison with prior work. The prior work to consider neural network training is SecureML [31] with implementation tested over Network A. They evaluate their performance for both the 2 and 3-server setting on similar hardware and network settings. We provide a comparison of our protocols with their work in Table 5. In the LAN setting, our

	Framework	LAN (hr)			WAN (hr)		
		Offline	Online	Total	Offline	Online	Total
A	SecureML 2PC	80.5	1.2	81.7	4277	59	4336
	SecureML 3PC	4.15	2.87	7.02	-	-	-
	Our 3PC	0	1.03	1.03	0	7.83	7.83

Table 5: **Training time comparison for Network A for batch size 128 and 15 epochs with SecureML [31].**

	Framework	Runtime (s)			Communication (MB)		
		Offline	Online	Total	Offline	Online	Total
A	SecureML	4.7	0.18	4.88	-	-	-
	ABY^3	0.003	0.005	0.008	-	-	0.5
	Our 3PC	0	0.043	0.043	0	2.1	2.1
B	MiniONN	3.58	5.74	9.32	20.9	636.6	657.5
	Gazelle	0.481	0.33	0.81	47.5	22.5	70.0
	Our 3PC	0	0.13	0.13	0	8.86	8.86
C	Our 3PC	0	0.23	0.23	0	18.94	18.94
D	DeepSecure	-	-	9.67	-	-	791
	Chameleon 3PC	1.34	1.36	2.7	7.8	5.1	12.9
	Gazelle	0.15	0.05	0.20	5.9	2.1	8.0
	ABY^3	0.006	0.004	0.01	-	-	5.2
	Our 3PC	0	0.076	0.076	0	4.05	4.05

Table 6: **Single image inference time comparison of various protocols in the LAN setting.**

protocol is roughly $6.8\times$ faster than their 3-party protocol and $79\times$ faster than their 2-party protocol. In the WAN setting, our improvements are even more dramatic and we get an improvement of $553\times$ over the 2-party protocol[10]. Even comparing only the online time of SecureML with our overall 3PC time, we obtain an improvement of $1.16\times$ over their 2PC and a $2.7\times$ improvement over their 3PC (their 3PC trades off some offline cost with a larger online cost).

8.4 Secure Inference

We also evaluate our protocols for the task of secure inference for the networks A, B, C and D. These networks can either be a result of secure training using the 3PC protocol and are secret shared between P_0 and P_1, or a trained model can be secret shared between P_0 and P_1 at the beginning of the protocol.

Comparison with prior work. A number of previous works have considered a single secure inference in the LAN setting for various networks. Table 6 summarizes our comparison with these state-of-the-art protocols. Networks A and B were considered in SecureML [31], MiniONN [29] and Gazelle [25] using different techniques for secure computation between 2 parties. All these works used similar hardware and network settings as our LAN experiments and we quote experimental numbers from the respective papers.

Each of these works split their computation into an input independent offline phase and an input dependent online phase. In our protocols, we count all the overhead in our online cost. Our protocols in the 3PC setting achieve roughly $3\times$ improvement in small networks that have a small number of non-linear operations (such as Network D) and between $6-113\times$ improvements in larger networks. We are the first to evaluate on Network C (which is considerably larger in size) and the table shows our runtime and communication. Finally, for Network D, we also compare our protocols with the 3PC protocols in Chameleon [35]. Our protocol is about $35\times$ faster than prior works. In all cases, our performance gains can be attributed to much better communication complexity of our protocols compared to previous works (see comparison in Table 6).

Single vs Batch Prediction. Table 7 summarizes our results for secure inference over different networks for 1 prediction and batch of 128 predictions in both the LAN and WAN settings. Due to use of matrix-based Beaver triplets for

[10]SecureML didn't provide numbers for their 3-party protocol in the WAN setting.

	LAN (s)		WAN (s)		Comm (MB)	
Batch size →	1	128	1	128	1	128
A	0.043	0.38	2.43	2.79	2.1	29
B	0.13	7.18	3.93	21.99	8.86	1066
C	0.23	10.82	4.08	30.45	18.94	1550
D	0.076	2.6	3.06	8.04	4.05	317.7

Table 7: **Prediction timings for batch size 1 vs 128 for our protocols on Networks A-D over MNIST.**

Protocol	Dimension	LAN (ms)	WAN (ms)	Comm. (MB)
$\mathsf{Conv2d}_{m,f,i,o}$	$8, 5, 16, 50$	3.8	28.4	0.42
	$28, 3, 1, 20$	1.8	26.5	0.2
	$28, 5, 1, 20$	2.8	27.5	0.33
$\mathsf{MatMul}_{m,n,v}$	$1, 100, 1$	0.33	25.2	0.0032
	$1, 500, 100$	4.8	29.4	0.81
	$784, 128, 10$	9.7	34.3	1.69
Maxpool	$8 \times 8 \times 50, 4 \times 4$	59.7	3062.2	2.23
	$24 \times 24 \times 16, 2 \times 2$	61.1	672.6	5.14
	$24 \times 24 \times 20, 2 \times 2$	62.6	685	6.43
DMP	$8 \times 8 \times 50, 4 \times 4$	1.9	51.6	0.18
	$24 \times 24 \times 16, 2 \times 2$	4.8	54.2	0.52
	$24 \times 24 \times 20, 2 \times 2$	4.9	55.2	0.65
DReLu	64×16	11.2	161.9	0.68
	128×128	109.8	288.7	10.88
	576×20	71.5	232.9	7.65
ReLU	64×16	0.42	25.3	0.04
	128×128	2.8	27.1	0.66
	576×20	2.5	26.6	0.46

Table 8: **Microbenchmarks in LAN/WAN settings.**

secure multiplication protocol in linear and convolutional layers, and batching of communication, the time for multiple predictions grows sub-linearly. SecureML also did predictions for batch size 100 for Network A and took $14s$ and $143s$ in the LAN and the WAN settings, respectively. In contrast, we take only $0.38s$ and $2.79s$ for 128 predictions using 3PC protocol.

8.5 Microbenchmarks

Table 8 presents microbenchmark timings for our various ML functionality protocols varied across different dimensions. All timings are average timings. While we have reduced the timings of non-linear function computations significantly, as can be seen from the table, they do have much higher cost than computation of linear layers and convolutional layers. The overheads for DMP and ReLU are additional over the costs of Maxpool and DReLU respectively as these pairs of protocols are always used together in training.

9 Limitations, Optimizations and Future Work

9.1 Scope and Assumptions

Though we rely on standard assumptions widely accepted in the research community, we acknowledge the limitations of our work. We consider honest-but-curious setting and do guarantee correctness against malicious adversaries that can arbitrarily tamper with the protocol execution. We also restrict ourselves to a 3-server set-up and its not immediate how to generalize the protocol to different settings. We do not handle denial of service where one of the server

refuses to co-operate with the computation. We do not prevent leakage via side-channels such as timing or communication patterns between servers during the protocol execution. We assume each server has access to an untampered random number generator, for example using the AES-NI instruction in the processor. Barring the use of such cryptographic random number generators and secure point-to-point communication channels between servers, the protocols in this work are information theoretically secure. Finally, the protocols are oblivious to the underlying neural network application and hence they can be used to convert such NN deployments to a 3PC secure version.

9.2 Optimizations and Future Work

We use a number of optimizations in our work which we briefly descibe here. We use a standard trick from literature to reduce the communication overhead of Beavers' triplets by using PRF's i.e., 5 out of the 6 triplet shares are generated locally using PRF's and the remaining share is sent over the network. We store certain variables from computations in the forward pass and use them in the backward pass. For instance, the output of the DReLU function that is computed in the forward pass during ReLU computation is used as-is in the backward pass. In certain networks, "theoretically swapping" the order of ReLU and Maxpool reduces the communication overhead (since Maxpool reduces dimensions of vectors). We use 4 cores to parallelize the computations in Algorithm 3. We do not optimize finding argmax over a large array or division (using "binary sort") since they're used primarily over smaller arrays. We would like to explore the general techniques for these as future work.

10 Related Work

In recent years, privacy-preserving machine learning has received considerable attention from the research community. We first discuss the most closely related works that consider neural network inference and training, and then provide an overview of other related works.

Neural Network Inference and Training. Perhaps the first work to consider secure neural network prediction was the work of Gilad-Barach *et al.* [23] who used homomorphic encryption techniques to provide secure prediction. For efficiency reasons, they approximated non-linear functions, such as the ReLU activation function to a quadratic function. Since this approximation results in loss in accuracy, there have been works that approximate ReLU using higher degree polynomials [16], but incur higher cost.

The work of SecureML [31] provided secure protocols for neural network training and prediction with non-linear activations, using a combination of arithmetic and Yao's garbled circuit techniques. They provided computational security against a single semi-honest adversary in both the 2 and 3-server models. The work of MiniONN [29] further optimized the protocols of SecureML [31] (specifically reducing the offline cost of matrix multiplications by increasing the online cost) for the case of prediction in the 2-server model. They also provided computational security against a semi-honest adversary. Concurrently and independently to this work, the works of Chameleon [35] and Gazelle [25] provide secure inference protocols in the 3-server and 2-server models, respectively. Chameleon remove expensive oblivious transfer protocols (needed for secure multiplications) by using the third party as a dealer, while Gazelle focusses on making the linear layers (such as matrix multiplication and convolution) more communication efficient by providing specialized packing schemes for additively homomorphic encryption schemes. Both these works are also computationally secure against one semi-honest adversary. All of the above protocols [31, 29, 35, 25] use garbled circuits for non-linear activations. In contrast, we provide protocols for non-linear activation functions by avoiding garbled circuits and dramatically reducing communication complexity.

Other related works. General purpose MPC protocols such as [9, 10, 14, 30, 7, 22, 27, 17] are not optimized for complex tasks such as NN training/inference and hence prior work in the domain [31, 29, 25, 35] all tend to outperform them. In this work, we restrict our comparison to works specifically optimized towards NN training or inference. Bost *et al.* [11] propose a number of building block functionalities to perform secure inference for linear classifiers, decision trees and naive bayes in the two-party setting. Later, [37] gave an improved protocol for decision trees. Perhaps the first work to consider secure training was that of Lindell and Pinkas [28] who provided algorithms to execute decision tree based training over shared data. Nikolaenko et al. [33] implemented a secure matrix factorization to train a movie recommender system. Shokri and Smatikov [36] considered the problem of privacy in neural network training when data is horizontally partitioned. Here, the parties run the training on their data individually, and exchange the changes in coefficients obtained during training – the goal is to minimize leakage and provide privacy to users using various techniques such as differential privacy [21].

11 Conclusions

We develop new 3-party secure computation protocols for a variety of neural network training and prediction algorithms such that no single party learns any information about the data. We implement our protocols and run experiments over Amazon EC2 to showcase the benefits of our approach. We are the first work to enable secure training of large neural networks such as CNNs that have accuracy of $> 99\%$ over the MNIST dataset. We propose novel protocols for non-linear functions that are significantly more communication efficient - thereby obtaining significantly faster performance compared to prior work. Finally, our protocols provide both full semi-honest security as well as privacy against malicious adversaries.

References

[1] Eigen Library. `http://eigen.tuxfamily.org/`. Version: 3.3.3.

[2] Fixed-point data type. `http://dec64.com`. Last Updted: 2018-01-20.

[3] MNIST database. `http://yann.lecun.com/exdb/mnist/`. Accessed: 2017-09-24.

[4] NTL Library. `http://www.shoup.net/ntl/`. Accessed: 2017-09-26.

[5] Stanford CS231n: Convolutional Neural Networks for Visual Recognition. `http://cs231n.github.io/convolutional-networks/`.

[6] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (GDPR). *Official Journal of the European Union*, L119, May 2016.

[7] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *ACM CCS*, 2016.

[8] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991.

[9] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206, 2008.

[10] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11(6):403–418, 2012.

[11] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. In *NDSS*, 2015.

[12] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42Nd IEEE Symposium on Foundations of Computer Science*, FOCS '01, pages 136–, 2001.

[13] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY*, 13(1):143–202, 2000.

[14] Octavian Catrina and Sebastiaan De Hoogh. Improved primitives for secure multiparty integer computation. In *SCN*, pages 182–199, 2010.

[15] Centers for Medicare & Medicaid Services. The Health Insurance Portability and Accountability Act of 1996 (HIPAA). Online at http://www.cms.hhs.gov/hipaa/, 1996.

[16] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. Privacy-preserving classification on deep neural network. Cryptology ePrint Archive, Report 2017/035, 2017. `https://eprint.iacr.org/2017/035`.

[17] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority mpc for malicious adversaries. In *Annual International Cryptology Conference*, pages 34–64. Springer, 2018.

[18] Benny Chor and Eyal Kushilevitz. A zero-one law for boolean privacy. *SIAM J. Discrete Math.*, 4(1), 1991.

[19] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Homomorphic encryption and secure comparison. In *IJACT*, 2008.

[20] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY – A framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[21] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. 2014.

[22] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *IACR Eurocrypt*, 2017.

[23] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, 2016.

[24] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *ACM STOC*, 1987.

[25] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *Usenix Security*, 2018.

[26] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[27] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *ACM CCS*, 2017.

[28] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. In *Annual International Cryptology Conference*, pages 36–54. Springer, 2000.

[29] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In *ACM CCS*, 2017.

[30] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In *ACM CCS*, 2015.

[31] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *ieee-oakland*, 2017.

[32] Michael A. Nielsen. Neural networks and deep learning. In *Determination Press*, 2015.

[33] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. In *ACM CCS*, pages 801–812, 2013.

[34] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *PKC*, 2007.

[35] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *AsiaCCS*, 2018.

[36] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1310–1321. ACM, 2015.

[37] David J. Wu, Tony Feng, Michael Naehrig, and Kristin E. Lauter. Privately evaluating decision trees and random forests. *PoPETs*, 2016, 2016.

[38] Xiangxin Zhu, Carl Vondrick, Charless Fowlkes, and Deva Ramanan. Do we need more training data? In *International Journal of Computer Vision*, 2016.

A Fixed-Point Encoding and Decimal Arithmetic

In order for neural network algorithms to be compatible with cryptographic applications, they must typically be encoded into integer form (most neural network algorithms work over floating point numbers). Now, decimal arithmetic must be performed over these values in an integer ring which requires careful detail. We follow [31] and describe details below. We use fixed point arithmetic to perform all computations. In other words, all numbers are represented as integers in the native C++ datatype `uint64_t`. We use a precision of $l_D = 13$ bits for representing numbers. For instance, an integer 2^{15} in this encoding corresponds to the float 4 and an integer $2^{64} - 2^{13}$ corresponds to a float -1. Since we use unsigned integers for encoding, $\mathsf{ReLU}(\cdot)$ compares its argument with 2^{63}. Such encoding is gaining popularity in the systems community with the introduction of fixed-point data types [2].

To perform decimal arithmetic in an integer ring, we use the same solution as is used in [31]. Addition of two fixed point decimal numbers is straightforward. To perform multiplication, we multiply the two decimal numbers and *truncate* the last l_D bits of the product. Theorem 1 in [31] shows that this above truncation technique also works over shared secrets (2-out-of-2 shares) i.e., the two parties can simply truncate their shares locally preserving correctness with an error of at most 1 bit with high probability. Denoting an arithmetic shift by $\Pi_{\mathsf{AS}}(a, \alpha)$, truncation of shares i.e., dividing shares by a power of 2 is described in Algorithm 11. We refer the reader to [31] for further details.

Algorithm 11 ShareTruncate $\Pi_{\mathsf{ST}}(P_0, P_1)$

Input: P_0, P_1 hold shares of a in $\mathbb{Z}_L$ and a positive integer n.
Output: P_0, P_1 get shares of $a/2^n$ in $\mathbb{Z}_L$.

1: $\Pi_{\mathsf{AS}}(x, m)$:= Arithmetic shift of x by m.
2: **for** $i \in \{0, 1\}$ **do**
3: $\quad P_i$ computes $\langle b \rangle_i^L = (-1)^i \cdot \Pi_{\mathsf{AS}}\left((-1)^i \cdot \langle a \rangle_i^L, n\right)$.
4: **end for**
5: **return** P_0, P_1 output $\langle b \rangle_i^L$.

B Neural Networks

Our main focus in this work is on Deep and Convolutional Neural Network (DNN and CNN) training algorithms. At a very high level, every layer in the forward propagation comprises of a linear operation (such as matrix multiplication in the case of fully connected layers and convolution in the case of Convolutional Neural Networks, where weights are multiplied by the activation), followed by a (non-linear) activation function f. One of the most popular activation functions is the Rectified Linear Unit (ReLU) defined as $\mathsf{ReLU}(x) = \mathsf{max}(0, x)$. The backpropagaton updates the weights appropriately making use of derivative of the activation function (in this case $\mathsf{ReLU}'(x)$, which is defined to be 1 if $x > 0$ and 0 otherwise) and matrix multiplication. Cross entropy is used as the loss function and stochastic gradient descent is used for minimizing the loss.

A large class of networks can be represented using the following functions: matrix multiplication, convolution, ReLU, Maxpool (which is defined as the maximum of a set of values, usually in a sub-matrix), normalization (which is defined to be $\frac{x_i}{\sum x_i}$ for a given set of values $\{x_1, \cdots, x_n\}$) and their derivatives. We consider three networks that perform training over MNIST dataset for hand-written digit recognition. This dataset contains 60,000 training samples of handwritten digits. Each image is a 28-by-28 pixel square image, with each pixel represented using 1 byte. The inference set contains 10,000 images. We use these networks for training as well as inference. Below, we describe the different networks used in detail.

- **Network A.** This is a 3-layer Deep Neural Network (DNN) from [31] comprising of fully connected (linear) layers followed by ReLU as the activation function. During training of this network, $\mathsf{ASM}(u_i) = \frac{\mathsf{ReLU}(u_i)}{\sum \mathsf{ReLU}(u_i)}$ is applied to the output of the last layer to convert the output values into a probability distribution before doing the backpropagaton. The loss function is cross entropy and stochastic gradient descent (SGD) is used. This network, after training for 15 epochs, has a prediction accuracy of 93.4% as illustrated in [31].
- **Network B.** Next is the Convolutional Neural Network (CNN) from [29]; while [29] used this network for prediction, we use the network for training. This is a 4-layer convolutional neural network that has the following structure. First layer is a 2-dimensional convolutional layer with 1 input channel, 16 output channels and a 5×5 filter, followed by a ReLU activation, followed by a 2×2 Maxpool. The second layer is a 2-dimensional convolutional layer with 16 input channels, 16 output channels and a 5×5 filter. The activation functions following this layer are once again ReLU and a 2×2 Maxpool. The third layer is an 256×100 fully-connected layer. The activation function is ReLU. Finally, the layer is a 100×10 linear layer and this is normalized using $\mathsf{ASM}(\cdot)$ to get a probability distribution. The loss function is cross entropy and SGD is used. Backpropagaton equations are computed appropriately. We show that this network achieves an accuracy of 98.77% on the MNIST dataset after training for 15 epochs.
- **Network C.** Finally, we also run our protocols over the LeNet network [26], which is a larger version of the network from [29]. This is a 4-layer CNN with similar structure as above but more number of output channels and bigger linear layers. First layer is a 2-dimensional convolutional layer with 1 input channel, 20 output channels and a 5×5 filter. The activation functions following this layer are ReLU, followed by a 2×2 Maxpool. The second layer is a 2-dimensional convolutional layer with 20 input channels, 50 output channels and another 5×5 filter. The activation functions following this layer are once again ReLU and a 2×2 Maxpool. The third layer is an 800×500 fully-connected layer. The next activation function is ReLU. The final layer is a 500×10 linear layer and this is normalized using $\mathsf{ASM}(\cdot)$ to get a probability distribution. We show that this network, after training for 15 epochs, has a prediction accuracy of 99.15%.
- **Network D.** In addition to these networks for training, for the case of secure inference, we also consider a network from Chameleon [35] for comparison in the 3-party setting. The first layer of this network is a 2-dimensional convolutional layer with a 5×5 filter, stride of 2, and 5 output channels. The activation function is ReLU. The second layer is a fully connected layer from a vector of size 980 to a vector of size 100 followed

by another ReLU activation. The last layer is a fully connected layer from a vector of size 100 to a vector of size 10. The arg max function is used to pick among the 10 values for predicting the digit. This network gives inference accuracy of 99%.

B.1 Neural Network Training Algorithms

We detail the forward and backward propagation equations for Network A. The equations for the other networks can be derived similarly.

We follow the notation from [32] closely. At a very high level, every layer in the forward propagation comprises of a linear operation (such as matrix multiplication in the case of fully connected layers and convolution in the case of Convolutional Neural Networks, where weights are multiplied by the activation), followed by a (non-linear) activation function f. One of the most popular activation functions is the Rectified Linear Unit (ReLU) defined as $\mathsf{ReLU}(x) = \mathsf{max}(0, x)$. Usually, the *softmax* function, defined as $\mathsf{SM}(u_i) = \frac{e^{-u_i}}{\sum e^{-u_i}}$ is applied to the output of the last layer. This function, being hard to compute cryptographically in a secure manner, is approximated by the function $\mathsf{ASM}(u_i) = \frac{\mathsf{ReLU}(u_i)}{\sum \mathsf{ReLU}(u_i)}$ – this is similar to what is done in the work of [31]. The idea behind the SM function is to convert the output values into a probability distribution - the same effect being also achieved by the ASM function. The backpropagaton updates the weights appropriately making use of derivative of the activation function (in this case $\mathsf{ReLU}'(x)$, which is defined to be 1 if $x > 0$ and 0 otherwise) and matrix multiplication.

Forward/Backward Prop Equations. We denote by l a generic layer of the network, where $1 \leq l \leq L$. We use w^l_{jk} to denote the weight of the connection from k^{th} neuron in the $(l-1)^{\text{th}}$ layer to neuron j^{th} in the l^{th} layer. We use a^l_j, b^l_j for the activation and bias of the j^{th} neuron in the l^{th} layer. We also define $z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_k$ for notational convenience. We use y_j to denote the output.

We drop the lower indices to denote the corresponding vector/matrix – for instance, w^l denotes the weight matrix between the $(l-1)^{\text{th}}$ and l^{th} layer, whereas w^l_{jk} denote individual values. The cost function used is the cross entropy function and is given by:

$$C = -\frac{1}{n} \sum_s \sum_j \left(y_j \ln a^L_j + (1 - y_j) \ln(1 - a^L_j) \right) \tag{1}$$

where n is the number of samples and s is a generic sample. The forward propagation is governed by the following equation:

$$a^l_j = \sigma(z^l_j) = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_k \right) \tag{2}$$

where σ is the non-linear operation, in our case $\mathsf{ReLU}(\cdot)$. Using $\odot$ to denote Hadamard product (element wise product) we define δ^l_j as the error of neuron j in layer l and is given by $\partial C / \partial z^l_j$. The backpropagaton equations are an approximation of actual gradients given that the forward pass contains $\mathsf{ASM}(\cdot)$. The backpropagaton equations are faithful to sigmoid function as the last layer activation function[11] and are given by the following 4 equations:

$$\delta^L = a^L - y \tag{3a}$$

$$\delta^l = (w^{l+1})^T \delta^{l+1} \odot \mathsf{ReLU}'(z^l) \tag{3b}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{3c}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{3d}$$

Eq. (3a) computes the error of the last layer, Eq. (3b) gives a way of computing the errors for layer l in terms of the errors for layer $l+1$, the weights w^{l+1} and z^l. Finally, Eq. (3c) and (3d) give compute the gradients of the biases and weights respectively.

Stochastic Gradient Descent (SGD). SGD is an iterative algorithm to minimize a function. We use SGD to train our DNN by initializing the weights to random values. In the *forward pass*, the network propagates from the inputs a^1 to compute the output y and in the *backward pass* the gradients are computed and the weights are updated. For efficiency reasons, instead of computing the forward and backward pass on each data sample, frequently a small set of samples

[11] sigmoid function is given by $f(x) = 1/(1 + e^{-x})$

are chosen randomly (called a *mini-batch*) and propagated together. The size of the mini-batch is denoted by B, set to 128 in this work. The complete algorithm for the 3-layer neural network is described in Algorithm 12.

Algorithm 12 Network A Π_{ML}:

Input: Inputs are read into a^1 one mini-batch at a time.

1: **for** $l = 2 : L$ **do**
2: $\quad z^{x,l} = w^l a^{x,l-1} + b^l$
3: $\quad a^{x,l} = \sigma(z^{x,l})$
4: **end for**
5: $\mathsf{ASM}(a_i^L) = \frac{\mathsf{ReLU}(a_i^L)}{\sum \mathsf{ReLU}(a_i^L)}$
6: $\delta^{x,L} = \mathsf{ASM}(a^{x,L}) - y^x$
7: **for** $l = L - 1 : 2$ **do**
8: $\quad \delta^{x,l} = w^{l+1}\delta^{x,l+1} \odot \mathsf{ReLU}'(z^{x,l})$
9: **end for**
10: **for** $l = L : 2$ **do**
11: $\quad b^l \to b^l - \frac{\alpha}{|B|} \sum_x \delta^{x,l}$
12: $\quad w^l \to w^l - \frac{\alpha}{|B|} \sum_x \delta^{x,l}(a^{x,l-1})^T$
13: **end for**

The entire algorithm can be broken down into the following phases:

– **Forward Pass** (Lines 1 to 4): This is the forward propagation phase resulting in the output a^L for each sample x.

– **Normalization** (Line 5): The final layer output a^L is normalized according to the $\mathsf{ASM}(\cdot)$ function.

– **Final Layer Error** (Line 6): This step computes the error for the final layer.

– **Error Backprop** (Lines 7 to 9): These equations back-propagate the error from the final layer to the previous layers.

– **Update Equations** (Lines 10 to 13): These equations compute the gradients in terms of the errors and update the weights, biases accordingly.

As can be seen from the description of the training algorithm, the main functions that we would need to compute securely are: matrix multiplication, $\mathsf{ReLU}(\cdot)$, division, $\mathsf{ASM}(\cdot)$ (which can be computed from $\mathsf{ReLU}(\cdot)$ and division) and $\mathsf{ReLU}'(\cdot)$. With these functions, one can run a secure 3PC protocol for Algorithm 12 by piecing together individual sub-protocols.