QUOTIENT: Two-Party Secure Neural Network Training and Prediction

Nitin Agrawal[*]
nitin.agrawal@cs.ox.ac.uk
University of Oxford

Ali Shahin Shamsabadi[*]
a.shahinshamsabadi@qmul.ac.uk
Queen Mary University of London

Matt J. Kusner
mkusner@turing.ac.uk
University of Oxford
The Alan Turing Institute

Adrià Gascón
agascon@turing.ac.uk
University of Warwick
The Alan Turing Institute

Abstract

Recently, there has been a wealth of effort devoted to the design of secure protocols for machine learning tasks. Much of this is aimed at enabling secure *prediction* from highly-accurate Deep Neural Networks (DNNs). However, as DNNs are trained on data, a key question is how such models can be also *trained* securely. The few prior works on secure DNN training have focused either on designing custom protocols for existing training algorithms, or on developing tailored training algorithms and then applying generic secure protocols. In this work, we investigate the advantages of designing training algorithms alongside a novel secure protocol, incorporating optimizations on both fronts. We present QUOTIENT, a new method for discretized training of DNNs, along with a customized secure two-party protocol for it. QUOTIENT incorporates key components of state-of-the-art DNN training such as layer normalization and adaptive gradient methods, and improves upon the state-of-the-art in DNN training in two-party computation. Compared to prior work, we obtain an improvement of 50X in WAN time and 6% in absolute accuracy.

Keywords

Secure multi-party computation, Privacy-preserving deep learning, Quantized deep neural networks

1 Introduction

The field of secure computation, and in particular Multi-Party Computation (MPC) techniques such as garbled circuits and lower level primitives like Oblivious Transfer (OT) have undergone very impressive developments in the last decade. This has been due to a sequence of engineering and theoretical breakthroughs, among which OT Extension [26] is of special relevance.

However, classical generic secure computation protocols do not scale to real-world Machine Learning (ML) applications. To overcome this, recent works have combined different secure computation techniques to design custom protocols for specific ML tasks. This includes optimization of linear/logistic regressors and neural networks [14, 40, 41, 44], matrix factorization [43], constrained optimization [29], and k-nearest neighbor classification [51, 52]. For example, Nikolaenko et. al. [44] propose a protocol for secure distributed ridge regression that combines additive homomorphic encryption and garbled circuits, while previous works [14, 41] rely in part on OT for the same functionality.

Practical ML and Secure Computation: Two Ships Passing in the Night. While there have been massive practical developments in both cryptography and ML (including the works above), most recent works for model training [22, 40, 41, 54] and prediction [7, 15, 50] on encrypted data are largely based on optimizations for *either* the ML model or the employed cryptographic techniques in isolation. In this work, we show that there is a benefit in taking a holistic approach to the problem. Specifically, our goal is to design an optimization algorithm alongside a secure computation protocol customized for it.

Secure Distributed Deep Neural Network Training. So far there has been little work on training Deep Neural Networks (DNNs) on encrypted data. The only works that we are aware of are ABY3 [40] and SecureML [41]. Different from this work, ABY3 [40] designs techniques for encrypted training of DNNs in the 3-party case, and a majority of honest parties. The work most similar to ours is SecureML [41]. They propose techniques based on secret-sharing to implement a stochastic gradient descent procedure for training linear/logistic regressors and DNNs in two-party computation. While the presented techniques are practical and general, there are three notable downsides: 1. They require an "offline" phase, that while being data-independent, takes up most of the time (more than 80 hours for a 3-layer DNN on the MNIST dataset in the 2-Party Computation (2PC) setting); 2. Their techniques are not practical over WAN (more than 4277 hours for a 3-layer DNN on the MNIST dataset), restricting their protocols to the LAN setting; 3. The accuracy of the obtained models are lower than state-of-the-art deep learning methods. More recently, secure training using homomorphic encryption has been proposed [22]. While the approach limits the communication overhead, it's estimated to require more than a year to train a 3-layer network on the MNIST dataset, making it practically unrealizable. Furthermore, the above works omit techniques necessary for modern DNNs training such as normalization & adaptive gradient methods, instead relying on vanilla SGD with constant step size. In this paper we argue that significant changes are needed in the way ML models are trained in order for them to be suited for practical evaluation in MPC. Crucially, our results show that securely-trained DNNs do not need to take a big accuracy hit if secure protocols and ML models are customized *jointly*.

---

[*] Both authors contributed equally to the paper. This work was partly done during an internship at the Alan Turing Institute.

Our Contributions: In this work we present QUOTIENT, a new method for secure two-party training and evaluation of DNNs. Alongside we develop an implementation in secure computation with semi-honest security, which we call 2PC-QUOTIENT. Our main insight is that recent work on training deep networks in fixed-point for embedded devices [57] can be leveraged for secure training. Specifically, it contains useful primitives such as repeated quantization to low fixed-point precisions to stabilize optimization. However, out of the box this work does not lead to an efficient MPC protocol. To do so, we make the following contributions, both from the ML and the MPC perspectives:

1. We ternarize the network weights: $\mathbf{W} \in \{-1, 0, 1\}$ during the forward and backward passes. As a result, ternary matrix-vector multiplication becomes a crucial primitive for training. We then propose a specialized protocol for ternary matrix-vector multiplication based on Correlated Oblivious Transfer that combines Boolean-sharing and additive-sharing for efficiency.
2. We further tailor the backward pass in an MPC-aware way, by replacing operations like quantization and normalization by alternatives with very efficient implementations in secure computation. We observe empirically in Section 6 that this change has no effect on accuracy. Alongside these changes, we extend the technique to residual layers [21], a crucial building block for DNNs.
3. We design a new fixed-point optimization algorithm inspired by a state-of-the-art floating-point adaptive gradient optimization procedure [47].
4. We implement and evaluate our proposal in terms of accuracy and running time on a variety of real-world datasets. We achieve accuracy nearly matching state-of-the-art *floating point accuracy* on 4 out of 5 datasets. Compared to state of the art 2PC secure DNN training [41], our techniques obtain ~6% absolute accuracy gains and > 50× speedup over WAN for both training and prediction.

The rest of the paper is organized as follows. In the next section we introduce notation and necessary background in ML and MPC. In Section 3 we present our proposed neural network primitives, and corresponding training procedure. In Section 4 we present a 2PC protocol for it. Finally, we present our experimental evaluation in Section 5 and conclude with a short discussion.

2 Overview and Problem Description

Here, we introduce concepts in DNN training and prediction, fixed-point encodings, and MPC applied to ML.

2.1 Deep Neural Networks

All DNNs are defined by a core set of operations, called a *layer*. These layers are repeatedly applied to an input $\mathbf{a}^0$ to produce a desired output $\mathbf{a}^L$, where the number of repetitions (or layers) L is called the *depth* of the network. Every layer consists of a linear operation and a non-linear operation and, depending on the type of network, each layer takes on different forms. We describe three popular layer types that make up a large portion of state-of-the-art DNNs: *fully-connected, convolutional,* and *residual.*

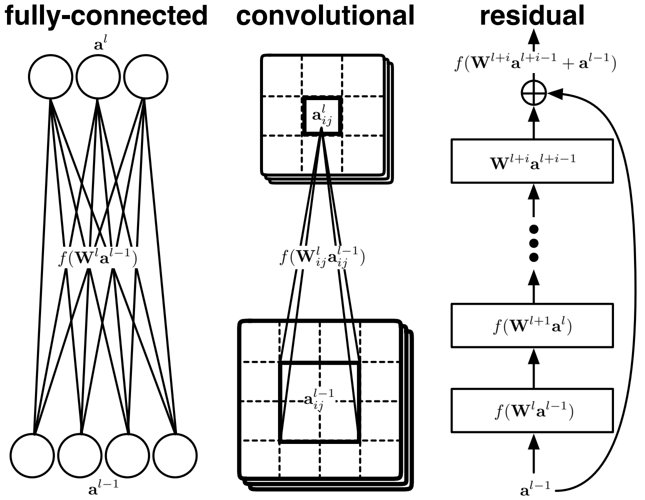

fully-connected
convolutional
residual
$\mathbf{a}^l$
$f(\mathbf{W}^l\mathbf{a}^{l-1})$
$\mathbf{a}^{l-1}$
$\mathbf{a}^l_{ij}$
$f(\mathbf{W}^l_{ij}\mathbf{a}^{l-1}_{ij})$
$\mathbf{a}^{l-1}_{ij}$
$f(\mathbf{W}^{l+i}\mathbf{a}^{l+i-1} + \mathbf{a}^{l-1})$
$\mathbf{W}^{l+i}\mathbf{a}^{l+i-1}$
$f(\mathbf{W}^{l+1}\mathbf{a}^l)$
$f(\mathbf{W}^l\mathbf{a}^{l-1})$
$\mathbf{a}^{l-1}$

Figure 1: We have protocols for three popular deep neural network layers: *fully-connected, convolutional,* and *residual.*

Fully-Connected Layers. To define a layer we simply need to define the linear and non-linear operations they use. In fully-connected layers these operations are as follows. Given an input $\mathbf{a}^{l-1} \in \mathbb{R}^{h_{l-1}}$ of any fully-connected layer $l \in \{1, \dots, L\}$, the layer performs two operations (1) a multiplication: $\mathbf{W}^l\mathbf{a}^{l-1}$ with a weight matrix $\mathbf{W}^l \in \mathbb{R}^{h_l \times h_{l-1}}$; and (2) a non-linear operation, most commonly the Rectified Linear Unit: $\textsc{ReLU}(x) = \max(x, 0)$. So the full layer computation is $\mathbf{a}^l = \textsc{ReLU}(\mathbf{W}^l\mathbf{a}^{l-1})$, where $\mathbf{a}^l$ is called the *activation* of layer l. Note that 'fully-connected' refers to the fact that any entry of $\mathbf{a}^{l-1}$ is 'connected' to the output $\mathbf{a}^l$ via the weight matrix $\mathbf{W}^l$, shown schematically in Figure 1 (*left*). Note that, more generic representations involve an intermediate step; adding a bias term b^l to compute $(\mathbf{W}^l\mathbf{a}^{l-1} + b^l)$ before performing the non-linear operation ($\mathbf{a}^l = \textsc{ReLU}(\mathbf{W}^l\mathbf{a}^{l-1} + b^l)$). In practice, this can be handled by suitably modifying $\mathbf{W}^l$ and $\mathbf{a}^{l-1}$ prior to performing operation (1).

Convolutional Layers. Convolutional layers are essentially fully-connected layers with a very particular connectivity structure. Whereas the input and output of a fully-connected layer are vectors, the input and output of a convolutional layer are 3rd-order tensors (i.e., an array of matrices). Specifically, the input $\mathbf{a}^{l-1} \in \mathbb{R}^{h_{l-1} \times w_{l-1} \times c_{l-1}}$ can be thought of as an image with height h_{l-1}, width w_{l-1}, and channels c_{l-1} (e.g., $c_{l-1} = 3$ for RGB images).

To map this to an output $\mathbf{a}^l \in \mathbb{R}^{h_l \times w_l \times c_l}$, a convolutional layer repeatedly looks at small square regions of the input, and moves this region from left-to-right, from top-to-bottom, until the entire image has been passed over. Let $\mathbf{a}^{l-1}_{ij} \in \mathbb{R}^{k_{l-1} \times k_{l-1} \times c_{l-1}}$ be the $k_{l-1} \times k_{l-1}$ region starting at entry (i, j). This is then element-wise multiplied by a set of weights $\mathbf{W}^l \in \mathbb{R}^{k_l \times k_l \times c_{l-1} \times c_l}$ and summed across the first three dimensions of the tensor. This is followed by a non-linear operation, also (most often) the $\textsc{ReLU}$ to produce an entry of the output $\mathbf{a}^l_{ij} \in \mathbb{R}^{1 \times 1 \times 1 \times c_l}$. Note that $\mathbf{a}^{l-1}_{ij}, \mathbf{a}^l_{ij}, \mathbf{W}^l$ can all be vectorized such that $\mathbf{a}^l_{ij} = f(\mathbf{W}^l\mathbf{a}^{l-1}_{ij})$. This operation is shown in Figure 1 (*center*). Apart from the dimensionality of the layers, other hyperparameters of these networks include: the size of square region, the number of 'pixels' that square regions jump in successive iterations (called the *stride*), and whether additional pixels with value 0 are added around the image to adjust the output image size (called *zero-padding*).

A common variant of a convolutional layer is a *pooling layer*. In these cases, the weights are always fixed to 1 and the non-linear function is simply $f(\cdot) \triangleq \max\{\cdot\}$ (other less popular $f(\cdot)$ include simple averaging or the Euclidean norm).

Residual Layers. The final layer type we consider are residual layers [21]. Residual layers take an activation from a prior layer $\mathbf{a}^{l-1}$ and add it to a later layer $l+i$ before the non-linear function $f(\cdot)$ (usually ReLU) is applied: $f(\mathbf{W}^{l+i}\mathbf{a}^{l+i-1} + \mathbf{a}^{l-1})$. The intermediate layers are usually convolutional but may be any layer in principle. Figure 1 (*right*) shows an example of the residual layer. DNNs with residual layers were the first to be successfully trained with more than 50 layers, achieving state-of-the-art accuracy on many computer vision tasks, and are now building blocks in many DNNs.

Training Deep Neural Networks. The goal of any machine learning classifier is to map an input $\mathbf{a}^0$ to a desired output y (often called a *label*). To train DNNs to learn this mapping, we would like to adjust the DNNs weights $\{\mathbf{W}^l\}_{l=1}^{L}$ so that the output $\mathbf{a}^L$ after L layers is: $\mathbf{a}^L = y$. To do so, the most popular method for training DNN weights is via Stochastic Gradient Descent (SGD). Specifically, given a training dataset of input-output pairs $\{(\mathbf{a}_i^0, y_i)\}_{i=1}^{n}$, and a loss function $\ell(\mathbf{a}^L, y)$ that measures the difference between prediction $\mathbf{a}^L$ and output y (e.g., the squared loss: $(\mathbf{a}^L - y)^2$). SGD consists of the following sequence of steps: (1) The *randomization step*: sample a random single input $\mathbf{a}_i^0$, (2) The *forward pass*: pass $\mathbf{a}_i^0$ through the network to produce prediction $\mathbf{a}_i^L$, (3) The *backward pass*: compute the gradients $\mathbf{G}^l$ and $\mathbf{e}^l$ of the loss $\ell(\mathbf{a}_i^L, y_i)$ with respect to each weight $\mathbf{W}^l$ and layer activation $\mathbf{a}^l$ in the network, respectively: $\mathbf{G}^l = \frac{\partial \ell(\mathbf{a}_i^L, y_i)}{\partial \mathbf{W}^l}, \mathbf{e}^l = \frac{\partial \ell(\mathbf{a}_i^L, y_i)}{\partial \mathbf{a}^l}, \forall l \in [L]$, (4) The *update step*: update each weight by this gradient: $\mathbf{W}_l = \mathbf{W}_l - \eta \mathbf{G}^l$, where η is a constant called the *learning rate*. These four steps, called an *iteration* are repeated for each input-output pair in the training set. A pass over the whole training set is called an *epoch*. The first step is often generalized to sample a set of inputs, called a *batch*, as this exploits the parallel nature of modern GPU hardware and has benefits in terms of convergence and generalization.

2.1.1 State-Of-The-Art Training: Normalization & Adaptive Step–Sizes. While the above "vanilla" gradient descent can produce reasonably accurate classifiers, alone they do not produce state-of-the-art results. Two critical changes are necessary: (1) normalization; and (2) adaptive step-sizes. We describe each of these in detail.

Normalization. The first normalization technique introduced for modern DNNs was *batch normalization* [25]. It works by normalizing the activations $\mathbf{a}^l$ of a given layer l so that across a given batch of inputs $\mathbf{a}^l$ has roughly zero mean and unit standard deviation. There is now a general consensus in the machine learning community that normalization is a key ingredient to accurate DNNs [5]. Since batch normalization, there have been a number of other successful normalization schemes including *weight normalization* [49] and *layer normalization* [35]. The intuition behind why normalization helps is because it prevents the activations $\mathbf{a}^l$ from growing too large to destabilize the optimization, especially for DNNs with many layers that have many nested multiplications. This means that one can increase the size of the learning rate η (see the *update step* above for how the learning rate is used in SGD), which speeds up optimization [5]. Normalization has been shown to yield speedups of an order of magnitude over non-normalized networks. Further, without normalization, not only is convergence slower, in some cases one cannot even reach lower minima [35].

Adaptive Step-Sizes. While normalization allows one to use larger learning rates η, it is still unclear how to choose the correct learning rate for efficient training: too small and the network takes an impractical amount of time to converge, too large and the training diverges. To address this there has been a very significant research effort into designing optimization procedures that adaptively adjust the learning rate during training [13, 31, 47]. So-called *adaptive gradient methods* scale the learning rate by the magnitude of gradients found in previous iterations. This effectively creates a per-dimension step-size, adjusting it for every entry of the gradient $\mathbf{G}^l$ for all layers l. Without adaptive gradient methods, finding the right step-size for efficient convergence is extremely difficult. A prominent state-of-the-art adaptive gradient method is AMSgrad [47]. It was developed to address pitfalls with prior adaptive gradient methods. The authors demonstrate that it is able to converge on particularly difficult optimization problems that prior adaptive methods cannot. Even on datasets where prior adaptive methods converge, AMSgrad can cut the time to converge in half.

2.1.2 Training and Inference with Fixed-Point Numbers. Motivated by the need to deploy DNNs on embedded and mobile devices with limited memory and power, significant research effort has been devoted to model quantization and compression. Often the goal is to rely solely on fixed-point encoding of real numbers. In fact, Tensorflow offers a lightweight variant to address this goal [19, 27]. These developments are useful for secure prediction. This is because cryptographic techniques scale with the circuit representation of the function being evaluated, and so a floating-point encoding and subsequent operations on that encoding are extremely costly. However, for the task of training, there are few works that perform all operations in fixed-point [20, 23, 33, 39, 57]. We start by reviewing fixed-point encodings, and the MPC techniques that we will consider. Then, in Section 3, we describe how this method can be modified and improved for secure training.

Notation for Fixed-Point Encodings. We represent fixed-point numbers as a triple $x = (a, \ell, p)$, where $a \in \{-2^{\ell-1} - 1, \ldots, 2^{\ell-1}\}$ is the fixed-point integer, ℓ is its *range* or *bit-width*, and p is its *precision*. The rational number encoded by x is $a/2^p$. For simplicity, we will often make ℓ implicit and write the rational $a/2^p$ as $a_{(p)}$. As our computations will be over ℓ-bit values in 2's complement, overflows/underflows can happen, resulting in large big errors. This requires that our training procedure is very stable, within a very controlled range of potential values.

2.2 MPC for Machine Learning

In this section, we introduce MPC techniques, highlighting the trade-offs that inspire our design of secure training and prediction protocols. The goal of MPC protocols is to compute a public function $f(\cdot)$ on private data held by different parties. The computation is done in a way that reveals the final output of the computation to intended parties, and nothing else about the private inputs. MPC protocols work over a finite discrete domain, and thus the function $f(\cdot)$ must be defined accordingly. Generally, MPC protocols can be classified depending on (i) a type of structure used to represent $f(\cdot)$ (generally either Boolean or integer-arithmetic circuits) and (ii) a scheme to secretly share values between parties, namely Boolean-sharing, additive-sharing, Shamir-secret-sharing, and more complex variants (for more information see [11]). The choices (i) and (ii) define the computational properties of an MPC protocol.

Additive-sharing protocols are very efficient for computations over large integral domains that do not involve comparisons. This includes sequences of basic linear algebra operations, such as matrix-vector multiplications. Specifically, additions are extremely cheap as they can be performed locally, while multiplications are more expensive. On the other hand, computations involving comparisons require computing a costly bit-decomposition of the values.

In contrast to additive-sharing, protocols based on Boolean-sharing are well-suited for computations easily represented as Boolean circuits, such as division/multiplication by powers of two (via bit shifting), comparisons, and `sign()`. They are slower at addition and multiplication which require adder and multiplier circuits.

These trade-offs lead to a natural idea recently exploited in several works in the secure computation for ML: one should design protocols that are customized to full algorithms, such as the training of linear/logistic regressors and DNNs [14, 41, 44], matrix factorization [43], or k-nearest neighbor classification [51, 52]. Moreover, custom protocols alternate between different secret-sharing schemes as required by the specific computation being implemented. Of course, the transformations between secret-sharing schemes must themselves be implemented by secure protocols[1]. This point is especially relevant to DNN training, as it amounts to a sequence of linear operations (which are naturally represented as arithmetic circuits) interleaved with evaluations of non-linear activation functions such as the ReLU (which are naturally represented as Boolean circuits).

Some MPC frameworks work in the so-called *pre-processing model* (see [41]), where computation is split into a data-independent offline phase and a data-dependent online phase. Random values useful for multiplication can be generated offline, and then used for fast secure multiplication online. In this paper we consider total time, removing the assumption of an offline phase. This is not a fundamental limitation, as we have variants of our protocols that work in the pre-processing model as explained later.

Notation for Secret-Sharing. In this work, we focus on the two-party computation setting, which excludes solutions that rely on a majority of honest parties [6, 40]. Inspired by work on function-specific protocols, in this work we employ both Boolean sharing and additive sharing. We start by fixing two parties P_1 and P_2. We denote the *Boolean-share* of $x \in \{0, 1\}$ held by P_1 as $\langle x \rangle_1$, and $\langle x \rangle_2$ for P_2. In Boolean-sharing, the shares satisfy $\langle x \rangle_1 = x \oplus \langle x \rangle_2$, where $\langle x \rangle_2$ is a random bit, and $\oplus$ signifies the `XOR` operation. We denote the *additive-share* of integer $y \in \mathbb{Z}_q$ held by P_1 as $[\![y]\!]_1$, and $[\![y]\!]_2$ for P_2. Here $[\![y]\!]_1 = y - [\![y]\!]_2$, with random $[\![y]\!]_2 \in \mathbb{Z}_q$. In practice, q is 2^σ, for $\sigma \in \{8, 16, 32, 64, 128\}$, as these are word lengths offered in common architectures.

Garbled Circuits. In many of our protocols, we use Yao's Garbled Circuits [58] as a subprotocol. In this protocol, the computation is represented as a Boolean circuit. We will not introduce the protocol in detail here, and instead, refer the reader to [37] for a detailed presentation and security analysis. The relevant observation to our work is the fact that the running time of a Garbled Circuits protocol is a function of the number of non-XOR gates in a circuit (this is thanks to the Free-XOR technique [32]). Consequently, designing efficient, largely XOR circuits is crucial in this setting. In fact, it is so important that previous works have used digital synthesizers for this task [10].

Oblivious Transfer. OT is a cryptographic primitive involving two parties: a *Chooser*, and a *Sender*. The Sender holds two messages m_0, m_1, and the Chooser holds a Boolean value b. After the execution of OT, the Chooser learns m_b, i.e. the Sender's message corresponding to their Boolean value. From the privacy perspective, an OT protocol is correct if it guarantees that (i) the Chooser learns nothing about m_{1-b}, and (ii) the Sender learns nothing about b. As common in MPC, this is formalized in the simulation framework (see [17] for details).

OT is a basic primitive in MPC. In fact, any function can be evaluated securely using only an OT protocol [18] and, moreover OT is a crucial component of Yao's Garbled Circuits. A remarkable advancement in the practicality of OT protocols was the discovery of OT extension [26]. This protocol allows one to compute a small number of OTs, say 128, and then bootstrap them to execute many fast OTs. Since then, optimizations in both the base OTs and the OT extension procedure [3] have led to implementations that can perform over ten million OT executions in under one second [55].

In our protocols, we employ a more efficient primitive—Correlated Oblivious Transfer (COT). COT was introduced in [3] alongside with an efficient COT Extension protocol that uses roughly half the communication than the general OT Extension. COT is a particular case of OT where the Sender does not get to choose its messages, but instead chooses a function f relating the two messages m_0 and m_1, as $m_0 = f(m_1)$. This functionality is enough for important applications of OT in MPC such as Garbled Circuits and OT-based triplet generation (see [11]). Below we define the flavour of COT that we need in our application, following the notation from [3].

Definition 2.1 ($m \times COT_\ell$). Let $\mathbf{f} = (f_i)_{i \in [m]}$, be a sequence of *correlation functions*, each with signature $f_i : \{0,1\}^\ell \mapsto \{0,1\}^\ell$ held by party P_1 (the Sender), and let $\mathbf{w} \in \{0,1\}^\ell$ be a sequence of *choice bits* held by party P_2 (the Chooser). After an execution of $m \times \mathrm{COT}_\ell(\mathbf{f}, \mathbf{w})$, P_1 obtains m *random* vectors $(\mathbf{x}_i \in \{0,1\}^\ell)_{i=1}^m$, and P_2 obtains m vectors $(\mathbf{y}_i \in \{0,1\}^\ell)_{i=1}^m$ such that $\forall i \in [m] : \mathbf{y}_i = (\neg \mathbf{w}_i) \cdot \mathbf{x}_i + \mathbf{w}_i \cdot f_i(\mathbf{x}_i)$.

2.3 Threat Model

In the two-party model of MPC, the training procedure is outsourced to two servers. This framework has been used in several previous works [14, 29, 41–44]. It works by first secret-sharing

---

[1] This aspect was thoroughly investigated in [11].

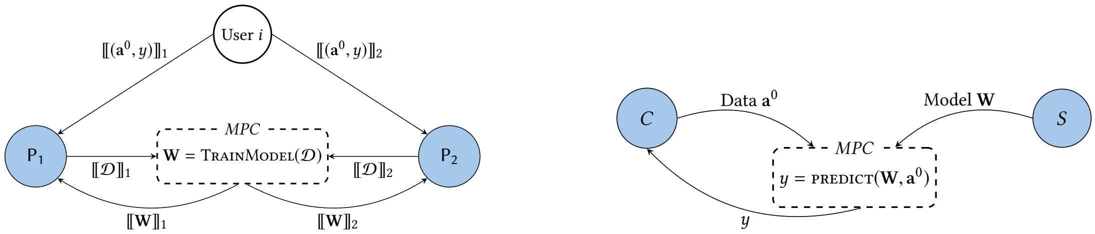


Figure 2: (Left) Training in the two-server model of MPC: (i) Each user i shares their labeled data (a_i^0, y_i) across two servers P_1 and P_2, by giving one share $[\![(\mathrm{a}_i^0, y_i)]\!]_j$ to each P_j. (ii) Each P_j compiles their share of a training dataset $\mathcal{D}$, by simply accumulating all shares received from users. Finally, (iii) P_1 and P_2 engage in a multi-party computation, in which $\mathcal{D}$ is securely reconstructed, and subsequently used to train a model W, from which each server gets a share. (Right) Private prediction using MPC: A client C and a server S engage in an multi-party computation protocol for the client to obtain y (the prediction of the server's model W for their client's data a^0) without the parties disclosing anything about W and a^0 to each other.

the training dataset $\mathcal{D}$ across the servers. This is depicted in Figure 2 (Left): users secret-share their values $(\mathbf{a}_i^0, y_i)$ across two non-colluding servers, which run the training procedure and obtain the resulting secret-shared model: $[\![\mathbf{W}]\!]_1, [\![\mathbf{W}]\!]_2$. Note that the scenario where two organizations collaborate to build a model of their respective private data in a privacy-preserving way is a particular case of this setting: this corresponds to the stage after the users have shared their data. We present fast MPC protocols to implement $\textsc{TrainModel}(\mathcal{D})$ from Figure 2 (Left). Thus, all our protocols are presented as two-party protocols between parties P_1 and P_2 via input and output additive-shares. Our protocols are secure in the semi-honest model for secure computation, as in [41].

Alongside private training, an important related problem is private prediction. This is depicted in Figure 2 (Right). Specifically, a client C has private data $\mathbf{a}^0$ for which they wish to have a private prediction y, via the private weights $\mathbf{W}$ of server S. In our setting, a fast protocol for two-party training in MPC immediately yields a fast protocol for private prediction, as prediction corresponds to the forward pass in training (i.e., Algorithm 1).

3 Deep Learning for MPC

In this section, we describe new methods to optimize DNNs that were developed alongside our protocols (in Section 4). Our first insight is that recent work on training deep networks in fixed-point [57] can be leveraged for crypto-friendly training. Namely, while this work was originally intended for embedded devices, it contains useful primitives such as repeated quantization to low fixed-point precisions to stabilize optimization. However, out-of-the-box, this work is unsuited for privacy-preserving protocols. We make the following modifications: (a) We ternarize the network weights: $\mathbf{W} \in \{-1, 0, 1\}$ during the forward and backward passes. This will allow matrix multiplication with $\mathbf{W}$ to be phrased as repeated 1-out-of-2 oblivious transfers (described in Section 4); (b) We construct an MPC-friendly quantization function for the weight gradients in the backward pass, replacing a biased coin flip and truncation steps with a saturation-free quantization, without loss in accuracy; (c) We replace the backward pass normalization operation, division by the closest-power-of-two, with the *next*-power-of-two. While a seemingly small change, the latter operation has a very efficient circuit implementation [28] that we leverage in our secure protocols in the next section. Further, we observe empirically in Section 5 that this change also has no effect on accuracy. Ultimately, these changes will only speed up the computation of a training iteration in a secure protocol. If training is based on stochastic gradient descent, many iterations will still be necessary for convergence. To address this, we design a new fixed-point adaptive gradient algorithm. It is inspired by a state-of-the-art floating-point adaptive gradient method, AMSgrad [47]. This optimization allows us to achieve the best accuracy to date on all 5 datasets we consider for DNNs trained in fixed-point (in Figure 8). In this section, we describe the work of [57], our changes (a-c) mentioned above, and our new fixed-point adaptive gradient training procedure.

3.1 Deep Models in Fixed-Point

The work by Wu et al. [57] describes an efficient optimization procedure for DNNs operating entirely on fixed-point encodings of real numbers. We describe it in detail here.

Quantize gradients, and quantize frequently. The first idea of WAGE [57] is to introduce functions Q_w, Q_a, Q_g, Q_e that quantize the *weights* $\mathbf{W}$, *activations* $\mathbf{a}$, *weight gradients* $\mathbf{G}$, *activation gradients* $\mathbf{e}$ to a small, finite set of fixed-point numbers. While previous work [2, 4, 9, 24, 30, 36, 46, 56] had already introduced the idea of functions Q_w, Q_a to quantize weights and/or activations in the forward pass, they required the weights $\mathbf{W}$ or gradients $\mathbf{G}, \mathbf{e}$ to be represented in floating-point in the backward pass, in order to optimize accurately. In Wu et al. [57], all the quantization functions take fixed-point numbers with some precision p, i.e., $v_{(p)} = v/2^p$ and find the nearest number with another precision q, i.e., $v_{(q)}$:

$$v_{(q)} = N(v_{(p)}, q) = \frac{\lfloor \frac{v}{2^p} 2^q \rceil}{2^q} \tag{1}$$

where $\frac{a}{2^p} 2^q$ is in practice either a division of a multiplication by $2^{|p-q|}$ depending on whether $p > q$. Additionally, Wu et al. [57] introduce a saturation function $S(\cdot)$, to yield $Q(\cdot)$ as:

$$v_{(q)} = Q(v_{(p)}, q) = S(N(v_{(p)}, q), q). \tag{2}$$

where $S(x, q) = \min(\max(x, -1 + 2^{-q}), 1 - 2^{-q})$ saturates any x to be within $[-1 + 2^{-q}, 1 - 2^{-q}]$.

Weight Quantization Q_w. The first function $\mathbf{W}_{(p_w)} = Q_w(\mathbf{W}_{(p)}, p_w)$ takes as input a p-precision fixed-point weight $\mathbf{W}_{(p)}$ and returns

Algorithm 1: Fixed-Point Forward Pass (`Forward`)

```
Input: Fixed-point weights $\{\mathbf{W}^l_{(p_w)}\}_{l=1}^L$,
    Fixed-point data sample $\mathbf{a}^0_{(p_a)}$,
    Layer-wise normalizers $\{\alpha_l\}_{l=1}^L$
Output: Fixed-point activations $\{\mathbf{a}^l_{(p_a)}\}_{l=1}^L$

for $l = 1, \ldots, L$ do
    $\mathbf{a}^l = f(\mathbf{W}^l_{(p_w)} \mathbf{a}^{l-1}_{(p_a)})$        ▷ pass through layer
    $\mathbf{a}^l_{(p_a)} = Q_a(\mathbf{a}^l, p_a)$        ▷ activations precision $p_a$
```

Algorithm 2: Fixed-Point Backward Pass (`Backward`)

```
Input: Fixed-point weights: $\{\mathbf{W}^l_{(p_w)}\}_{l=1}^L$,
    Fixed-point activations (act) $\{\mathbf{a}^l_{(p_a)}\}_{l=1}^L$,
    Fixed-point label $y_{(p_a)}$,
    Activation function $f$,
    Saturation function $S$
Output: Fixed-point gradients $\{\mathbf{G}^l\}_{l=1}^L$

$\mathbf{e}^L = \frac{\partial \ell(\mathbf{a}^L_{(p_a)}, y_{(p_a)})}{\partial \mathbf{a}^L_{(p_a)}}$        ▷ act. gradient
for $l = L, \ldots, 1$ do
    $\mathbf{e}^l_{(p_e)} = Q_e(\mathbf{e}^l, p_e)$        ▷ act. gradient precision $p_e$
    $\mathbf{e}^{l-1} = \mathbf{W}^l_{(p_w)} \left( \mathbf{e}^l_{(p_e)} \circ \frac{\partial \mathbf{a}^l_{(p_a)}}{\partial f} \circ \frac{\partial \mathbf{a}^l_{(p_a)}}{\partial S} \right)$        ▷ act. gradient
    $\mathbf{G}^l = \mathbf{a}^{l-1}_{(p_a)}{}^{\top} \mathbf{e}^l_{(p_e)}$        ▷ weight gradient
```

the closest p_w-precision weights $\mathbf{W}_{(p_w)}$ using the above function:

$$\mathbf{W}_{(p_w)} = Q_w(\mathbf{W}_{(p)}, p_w) = Q(\mathbf{W}_{(p)}, p_w) \tag{3}$$

Activation quantization Q_a. The second function $\mathbf{a}_{(p_a)} = Q_a(\mathbf{a}_{(p)}, p_a)$ is almost identical except it introduces an additional scaling factor $2^{-\alpha}$ as follows:

$$\mathbf{a}_{(p_a)} = Q_a(\mathbf{a}_{(p)}, p_a) = Q(\frac{\mathbf{a}_{(p)}}{2^{\alpha}}, p_a). \tag{4}$$

Where α is a fixed integer determined by the dimensionality of $\mathbf{W}$, see [57] eq. (7). The intuition is that this is a simple form of normalization (used in the forward pass), and the authors describe a technique to set α based on the layer size.

Activation gradient quantization Q_e. For the backward pass the first gradient we must consider is that of the activations, which we call $\mathbf{e}$. The strategy to quantize $\mathbf{e}_{(p_e)} = Q_e(\mathbf{e}_{(p)}, p_e)$, is similar to quantizing the activations. Except here the scaling factor depends on the largest value of $\mathbf{e}$:

$$\mathbf{e}_{(p_e)} = Q_e(\mathbf{e}_{(p)}, p_e) = Q(\frac{\mathbf{e}_{(p)}}{2^{\mathsf{cpow}(\max\{|\mathbf{e}_{(p)}|\})}}, p_e) \tag{5}$$

where $\mathsf{cpow}(x) = 2^{\lfloor \log_2(x) \rceil}$ is the closest power of 2 to x. The intuition here is again that normalization helps, but it cannot be constant as the gradient magnitude can potentially fluctuate. Thus normalization needs to be data-specific to stabilize training.

Algorithm 3: Fixed-Point SGD Optimization (Δ_{sgd})

```
Input: Fixed-point weights $\{\mathbf{W}^l_{(\overline{p}_w)}\}_{l=1}^L$,
    Fixed-point data $\mathcal{D} = \{(\mathbf{a}^0_{(p_a),i}, y_{(p_a),i}\}_{i=1}^n$,
    Learning rate $\eta$
Output: Updated weights $\{\mathbf{W}^l_{(\overline{p}_w)}\}_{l=1}^L$

for $t = 1, \ldots, T$ do
    $(\mathbf{a}^0_{(p_a)}, y_{(p_a)}) \sim \mathcal{D}$
    $\{\mathbf{W}^l_{(p_w)} = Q_w(\mathbf{W}^l_{(\overline{p}_w)}, p_w)\}_{l=1}^L$
    $\{\mathbf{a}^l_{(p_a)}\}_{l=1}^L = \mathtt{Forward}(\{\mathbf{W}^l_{(p_w)}\}_{l=1}^L, \mathbf{a}^0_{(p_a)})$
    $\{\mathbf{G}^l\}_{l=1}^L = \mathtt{Backward}(\{\mathbf{W}^l_{(p_w)}, \mathbf{a}^l_{(p_a)}\}_{l=1}^L, y_{(p_a)})$
    $\{\mathbf{G}^l_{(p_g)} = Q_g(\mathbf{G}^l, p_g)\}_{l=1}^L$        ▷ weight gradient precision $p_g$
    $\{\mathbf{W}^l_{(\overline{p}_w)} = S(\mathbf{W}^l_{(\overline{p}_w)} - \eta \mathbf{G}^l_{(p_g)}, \overline{p}_w)\}_{l=1}^L$
```

Weight gradient quantization Q_g. The second gradient is the weight gradient $\mathbf{G}$, quantized by $\mathbf{G}_{(p_g)} = Q_g(\mathbf{G}_{(p)}, p_g)$, as follows:

$$\mathbf{G}_{(p_g)} = Q_g(\mathbf{G}^n_{(p)}, p_g) = \frac{\mathtt{sign}(\mathbf{G}^n_{(p)})}{2^{p_g - 1}} \left[\lfloor |\mathbf{G}^n_{(p)}| \rfloor + \mathrm{B}\Big(|\mathbf{G}^n_{(p)}| - \lfloor |\mathbf{G}^n_{(p)}| \rfloor \Big) \right]. \tag{6}$$

where $\mathbf{G}^n_{(p)} = \eta \mathbf{G}_{(p)} / 2^{\mathsf{cpow}(\max\{|\mathbf{G}_{(p)}|\})}$ is a normalized version of $\mathbf{G}_{(p)}$ that is also multiplied by the learning rate η, $\mathtt{sign}(a)$ returns the sign of a, and $B(p)$ draws a sample from a Bernoulli distribution with parameter p. This function normalizes the gradient, applies a randomized rounding, and scales it down. The idea behind this function is that additional randomness (combined with the randomness of selecting data points stochastically during training) should improve generalization to similar, unseen data.

3.2 *Cryto-Friendly* Deep Models in Fixed-Point

We propose three changes to the quantization functions that do not affect accuracy, but make them more suitable for MPC.

Weight quantization Q_w. We fix $p_w = 1$ and set $Q_w(\mathbf{W}_{(p)}, p_w) = 2Q(\mathbf{W}_{(p)}, p_w)$ so that our weights are ternary: $\mathbf{W} \in \{-1, 0, 1\}$ as mentioned above.

Activation gradient quantization Q_e. We make the following change to eq. (5) to $Q_e(\mathbf{e}_{(p)}, p_e) = Q(\mathbf{e}_{(p)} / 2^{\mathsf{npow}(\max\{|\mathbf{e}_{(p)}|\})}, p_e)$, where $\mathsf{npow}(x) = 2^{\lceil \log_2(x) \rceil}$ is the *next* power of 2 after x. This is faster than the closest power of 2, $\mathsf{cpow}(x)$ as the latter also needs to compute the previous power of 2 and compare them to x to find the closest power.

Weight gradient Quantization Q_g. We introduce a different quantization function than [57] which is significantly easier to implement using secure computation techniques:

$$\mathbf{G}_{(p_g)} = Q_g(\mathbf{G}_{(p)}, p_g) = N(\frac{\mathbf{G}_{(p)}}{2^{\mathsf{npow}(\max\{|\mathbf{G}_{(p)}|\})}}, p_g). \tag{7}$$

We find that the original quantization function in eq. (6) needlessly removes information and adds unnecessary overhead to a secure implementation.

Forward/Backward passes. Algorithm 1 corresponds to prediction in DNNs, and Algorithms 1, 2, 3 describe the training procedure. Apart from the quantization functions note that in the backward pass (Algorithm 2) we take the gradient of the activation with respect to the activation function: $\partial \mathbf{a}^l_{(p_a)} / \partial f$ and the saturation function: $\partial \mathbf{a}^l_{(p_a)} / \partial S$. Algorithm 3 describes a stochastic gradient descent (SGD) algorithm for learning fixed-point weights inspired by [57]. We keep around two copies of weights in different precisions p_w and $\overline{p}_w$. The weights $\mathbf{W}_{(p_w)}$ are ternary and will enable fast secure forward (Algorithm 1) and backward passes. The other weights $\mathbf{W}_{(\overline{p}_w)}$ are at a higher precision and are updated with gradient information. We get the ternary weights $\mathbf{W}_{(p_w)}$ by quantizing the weights $\mathbf{W}_{(\overline{p}_w)}$ in line 3. Note that the forward and backward passes of convolutional networks can be written using the exact same steps as Algorithms 1 and 2 where weights are reshaped to take into account weight-sharing. Similarly, for residual networks the only differences are: (a) line 2 in the forward pass changes; and (b) line 4 in the backward pass has an added term from prior layers.

3.3 A Fixed-Point Adaptive Gradient Algorithm

One of the state-of-the-art adaptive gradient algorithms is AMSgrad [47]. However, AMSgrad includes a number of operations that are possibly unstable in fixed-point: (i) the square-root of a sum of squares, (ii) division, (iii) moving average. We redesign AMSgrad in Algorithm 4 to get around these difficulties in the following ways. First, we replace the square-root of a sum of squares with absolute value in line 9, a close upper-bound for values $\mathbf{v} \in [-1, 1]$. We verify empirically that this approximation does not degrade performance.

Second, we replace division of $\hat{\mathbf{V}}_{(p_v)}$ in line 11 with division by the next power of two. Third, we continuously quantize the weighted moving sums of lines 7 and 8 to maintain similar precisions throughout. These changes make it possible to implement AMSgrad in secure computation efficiently. In the next section we describe the protocols we design to run Algorithms 1-4 privately.

4 Oblivious Transfer for Secure Learning

In this section, we present custom MPC protocols to implement private versions of the algorithms introduced in the previous section. Our protocols rely heavily on OT, and crucially exploit characteristics of our networks such as ternary weights and fixed-point (8-bit precision) gradients. We present our MPC protocol for neural network training by describing its components separately. First, in Section 4.1 we describe our protocol for ternary matrix-vector multiplication, a crucial primitive for training used both in the forward and backward pass. Next, in Sections 4.2 and 4.3 we describe the protocols for the forward and backward passes, respectively. These protocols use our matrix-vector multiplication protocol (and a slight variant of it), in combination with efficient garbled circuit implementations for normalization and computation of ReLU.

4.1 Secure Ternary Matrix-Vector Multiplication

A recurrent primitive, both in the forward and backward passes of the fixed-point neural networks from Section 3 is the product $\mathbf{W}\mathbf{a}$, for ternary matrix $\mathbf{W} \in \{-1, 0, 1\}^{n \times m}$ and fixed-point integer vector $\mathbf{a} \in \mathbb{Z}_q^m$. Several previous works on MPC-based ML have looked specifically at matrix-vector multiplication [41, 51]. As described in Section 2.2, multiplication is a costly operation in MPC. Our insight is that if $\mathbf{W}$ is ternary, we can replace multiplications by selections, enabling much faster protocols. More concretely, we can compute the product $\mathbf{z} = \mathbf{W}\mathbf{a}$ as shown in Algorithms 5.

Algorithm 4: Fixed-Point AMSgrad Optim. (Δ_{ams})

```
Input: Fixed-point weights $\{\mathbf{W}^l_{(\overline{p}_w)}\}_{l=1}^{L}$,
Fixed-point data $\mathcal{D} = \{(\mathbf{a}^0_{(p_a),i}, y_{(p_a),i}\}_{i=1}^{n}$, learning rate $\eta$
Output: Updated weights $\{\mathbf{W}^l_{(\overline{p}_w)}\}_{l=1}^{L}$

1:  Initialize: $\{\mathbf{M}^l, \mathbf{V}^l\}_{l=1}^{L} = 0$
2:  for $t = 1, \ldots, T$ do
3:      $(\mathbf{a}^0_{(p_a)}, y_{(p_a)}) \sim \mathcal{D}$
4:      $\{\mathbf{W}^l_{(p_w)} = 2Q(\mathbf{W}^l_{(\overline{p}_w)}, p_w)\}_{l=1}^{L}$
5:      $\{\mathbf{a}^l_{(p_a)}\}_{l=1}^{L} = \mathtt{Forward}(\{\mathbf{W}^l_{(p_w)}\}_{l=1}^{L}, \mathbf{a}^0_{(p_a)})$
6:      $\{\mathbf{G}^l\}_{l=1}^{L} = \mathtt{Backward}(\{\mathbf{W}^l_{(p_w)}, \mathbf{a}^l_{(p_a)}\}_{l=1}^{L}, y_{(p_a)})$
7:      $\{\mathbf{G}^l = \frac{\mathbf{G}^l}{2^{\mathrm{npow}(\max\{|\mathbf{G}^l|\})}}\}_{l=1}^{L}$    ▷ scale gradients
8:      $\{\mathbf{M}^l = N(0.9\mathbf{M}^l, p_m) + 0.1\mathbf{G}^l\}_{l=1}^{L}$    ▷ weighted mean
9:      $\{\mathbf{V}^l = N(0.99\mathbf{V}^l, p_v) + 0.01|\mathbf{G}^l|\}_{l=1}^{L}$
10:     $\{\hat{\mathbf{V}}^l = \max(\hat{\mathbf{V}}^l, \mathbf{V}^l)\}_{l=1}^{L}$
11:     $\{\mathbf{G}^l = \frac{\mathbf{M}^l}{2^{\mathrm{npow}(|\hat{\mathbf{V}}^l| + \epsilon)}}\}_{l=1}^{L}$    ▷ history-based scaling
12:     $\{\mathbf{G}^l_{(p_g)} = N(\mathbf{G}^l, p_g)\}_{l=1}^{L}$    ▷ weight gradient precision $p_g$
13:     $\{\mathbf{W}^l_{(\overline{p}_w)} = S(\mathbf{W}^l_{(\overline{p}_w)} - \eta\mathbf{G}^l_{(p_g)}, \overline{p}_w)\}_{l=1}^{L}$
```

Algorithm 5: Ternary-Integer Matrix-Vector Product

```
Input: Matrix $\mathbf{W} \in \{-1, 0, 1\}^{n \times m}$ and vector $\mathbf{a} \in \mathbb{Z}_q^m$
    $\mathbf{z} = (0)_{i \in [n]}$
    for $i \in [n], j \in [m]$ do
        if $\mathbf{W}_{\mathbf{i},\mathbf{j}} > 0$ then $z_i$ += $\mathbf{a}_j$
        if $\mathbf{W}_{\mathbf{i},\mathbf{j}} < 0$ then $z_i$ −= $\mathbf{a}_j$
    return z
```

A natural choice for implementing the functionality in Algorithm 5 securely are MPC protocols that represent the computation as a Boolean circuit. In our two-party setting natural choices are garbled circuits and the GMW protocol [18]. In Boolean circuits, the If-Then-Else construction corresponds to a multiplexer, and a comparison with 0 is essentially free: it is given by the sign bit in a two's complement binary encoding. However, a circuit implementation of the computation above will require $|\mathbf{W}|$ additions and the same number of subtractions, which need to implemented with full-adders. Whereas, additions, if computed on additive shares, do not require interaction and thus are extremely fast.

Our proposed protocol achieves the best of both worlds by combining Boolean sharing and additive sharing. To this end, we represent the ternary matrix $\mathbf{W}$ by two Boolean matrices $\mathbf{W}^+ \in \{0, 1\}^{n \times m}$ and $\mathbf{W}^- \in \{0, 1\}^{n \times m}$, defined as $\mathbf{W}^+_{i,j} = 1 \Leftrightarrow \mathbf{W}_{i,j} = 1$ and $\mathbf{W}^-_{i,j} = 1 \Leftrightarrow \mathbf{W}_{i,j} = -1$. Now, the product $\mathbf{W}\mathbf{a}$ can be rewritten as $\mathbf{W}^+\mathbf{a} - \mathbf{W}^-\mathbf{a}$. This reduces our problem from $\mathbf{W}\mathbf{a}$ with ternary $\mathbf{W}$ to two computations of $\mathbf{W}\mathbf{a}$ with binary $\mathbf{W}$. Note that we can

```
Protocol 6: Boolean-Integer Inner Product
  Parties: $\mathsf{P}_1$ and $\mathsf{P}_2$
  Input: Arithmetic shares of integer vector $\mathbf{a} \in \mathbb{Z}_q^m$ and
         Boolean shares of binary vector $\mathbf{w} \in \{0,1\}^m$
  Output: Arithmetic shares of $z = \mathbf{w}^\top \mathbf{a}$
  1: Each $\mathsf{P}_i$ generates random values $(z_{i,j})_{j\in[m]}$.
  2: for $j \in [m]$ do
  3:     $\mathsf{P}_i$ sets
             $m_{i,0} := \langle \mathbf{w}_j \rangle_i \cdot [\![\mathbf{a}_j]\!]_i - z_{i,j}$
             $m_{i,1} := \neg\langle \mathbf{w}_j \rangle_i \cdot [\![\mathbf{a}_j]\!]_i - z_{i,j}$
  4:     $\mathsf{P}_1$ and $\mathsf{P}_2$ run $\mathrm{OT}(m_{1,0}, m_{1,1}, \langle \mathbf{w}_j \rangle_2)$, with $\mathsf{P}_1$ as Sender
         and $\mathsf{P}_2$ as Chooser, for $\mathsf{P}_2$ to obtain $z'_{1,j}$
  5:     $\mathsf{P}_1$ and $\mathsf{P}_2$ run $\mathrm{OT}(m_{2,0}, m_{2,1}, \langle \mathbf{w}_j \rangle_1)$, with $\mathsf{P}_2$ as Sender
         and $\mathsf{P}_1$ as Chooser, for $\mathsf{P}_1$ to obtain $z'_{2,j}$
  6: Each $\mathsf{P}_i$ sets $[\![z]\!]_i = \sum_{j\in[m]}(z_{i,j} + z'_{i,j})$.
```

use the same decomposition to split any matrix with inputs in a domain of size k into k Boolean matrices, and thus our protocol is not restricted to the ternary case.

Accordingly, at the core of our protocol is a two-party subprotocol for computing additive shares of $\mathbf{Wa}$, when $\mathbf{W}$ is Boolean-shared among the parties and $\mathbf{a}$ is additively shared. In turn, this protocol relies on a two-party subprotocol for computing additive shares of the inner product of a Boolean-shared binary vector and an additively shared integer vector. As a first approach to this problem, we show in Protocol 6 a solution based on oblivious transfer. We state and prove the correctness of Protocol 6 in Appendix C.

One can think of Protocol 6 as a component-wise multiplication protocol, as we will use it also for that purpose. The only modification required is in step 6, i.e. the local aggregation of additive shares of the result of component-wise multiplication. We could directly use this protocol to implement our desired matrix-vector multiplication functionality, and this leads to very significant improvements due to the concrete efficiency of OT Extension implementations. However, we can further optimize this to obtain our final protocol.

The use of OT in Protocol 6 has similarities with the GMW protocol, and is inspired by the OT-based method due to Gilboa for computing multiplication triplets, and discussed in [11]. A similar idea was used in the protocol for computing the sigmoid function by Mohassel and Zhang [41]. Moreover, concurrently to this work, Riazi et al. [48] have proposed OT-based protocols for secure prediction using DNNs. They propose a protocol called Oblivious Conditional Addition (OCA) that is analogous to Protocol 6. While their work only addresses secure prediction, our improved protocol presented in the next section is relevant in their setting as well.

4.1.1 Optimizations: Correlated OT and Packing. In the previous section, our protocol assumed a standard OT functionality, but actually, we can exploit even more efficient primitives. Our optimization of Protocol 6, presented in Protocol 7 exploits COT in a way to implement our required inner product functionality. The idea behind Protocol 7 is simple: note that in Protocol 6 parties choose their random shares $z_{i,j}$ of intermediate values in the computation, and they use them to mask OT messages. However, as we only require the $z_{i,j}$'s to be random, one could in principle let the OT choose them. Note also that the parties can choose their share of the result as a function of their inputs, which can be implemented in COT. This is done in lines 4 and 5 of Protocol 7. The next Lemma states the correctness of Protocol 7. As the protocol consists of just two executions of $m \times \mathrm{COT}_\ell$ and local additions its security is trivial, while we present the proof of correctness in Appendix D. Finally, note that, as in the case of Protocol 6, it is easy to turn Protocol 7 into a protocol for component-wise multiplication.

```
Protocol 7: Boolean-Integer Inner Product via $m\times\mathrm{COT}_\ell$
  Parties: $\mathsf{P}_1$ and $\mathsf{P}_2$
  Input: $\ell$-bit arithmetic shares of integer vector $\mathbf{a} \in \mathbb{Z}_q^m$ and
         Boolean shares of binary vector $\mathbf{w} \in \{0,1\}^m$
  Output: Arithmetic shares of $z = \mathbf{w}^\top \mathbf{a}$
  1: Each party $\mathsf{P}_i$ constructs a vector of correlation functions
     $\mathbf{f}^{\mathrm{i}} = (f_{i,j}(x))_{j\in[m]}$,
         $f_{i,j}(x) = x - [\![\mathbf{w}_j]\!]_i \cdot [\![\mathbf{a}_j]\!]_i + \neg[\![\mathbf{w}_j]\!]_i \cdot [\![\mathbf{a}_j]\!]_i$
  2: The parties run $m \times \mathrm{COT}_\ell(\mathbf{f}^{1}, [\![\mathbf{w}_j]\!]_2)$ with $\mathsf{P}_1$ acting as
     the Sender, and $\mathsf{P}_1$ obtains $\mathbf{x}$ while $\mathsf{P}_2$ obtains $\mathbf{y}$.
  3: The parties run $m \times \mathrm{COT}_\ell(\mathbf{f}^{2}, [\![\mathbf{w}_j]\!]_1)$ with $\mathsf{P}_2$ acting as
     the Sender, and $\mathsf{P}_2$ obtains $\mathbf{x}'$ while $\mathsf{P}_1$ obtains $\mathbf{y}'$.
  4: $\mathsf{P}_1$ sets $[\![z]\!]_1 = \sum_{j\in[m]}([\![\mathbf{w}_j]\!]_1 \cdot [\![\mathbf{a}_j]\!]_1 - \mathbf{x}_j + \mathbf{y'}_j)$
  5: $\mathsf{P}_2$ sets $[\![z]\!]_2 = \sum_{j\in[m]}([\![\mathbf{w}_j]\!]_2 \cdot [\![\mathbf{a}_j]\!]_2 - \mathbf{x'}_j + \mathbf{y}_j)$
```

```
Protocol 8: Ternary-Integer Matrix-Vector Product
  Parties: $\mathsf{P}_1$ and $\mathsf{P}_2$
  Input: Arithmetic shares of integer vector $\mathbf{a} \in \mathbb{Z}_q^m$ and
         Boolean shares of binary matrices
         $\mathbf{W}^+, \mathbf{W}^- \in \{0,1\}^{n,m}$
  Output: Arithmetic shares of $\mathbf{z} = \mathbf{W}^+\mathbf{a} - \mathbf{W}^-\mathbf{a}$
  1: $\mathsf{P}_1$ and $\mathsf{P}_2$ compute $[\![\mathbf{W}^+\mathbf{a}]\!]$ using $n$ executions of Protocol 7.

  2: $\mathsf{P}_1$ and $\mathsf{P}_2$ compute $[\![\mathbf{W}^-\mathbf{a}]\!]$ using $n$ executions Protocol 7.
  3: $\mathsf{P}_i$ sets $[\![\mathbf{z}]\!]_i := [\![\mathbf{W}^+\mathbf{a}]\!]_i - [\![\mathbf{W}^+\mathbf{a}]\!]_i$.
```

Lemma 4.1. *Let* $\mathbf{w}$ *and* $\mathbf{a}$ *be a Boolean and integer vector, respectively, shared among parties* $\mathsf{P}_1, \mathsf{P}_2$. *Given an* $m \times COT_\ell$ *protocol, the two-party Protocol 7 is secure against semi-honest adversaries, and computes an additive share of the inner product* $\mathbf{w}^\top\mathbf{a}$ *among* $\mathsf{P}_1, \mathsf{P}_2$.

Protocol 8 achieves our goal of computing an arithmetic share of $\mathbf{Wa} = \mathbf{W}^+\mathbf{a} - \mathbf{W}^-\mathbf{a}$, for an $n \times m$ matrix. This is easily achieved using $2n$ calls to Protocol 7. This translates into $4n$ executions of $m \times \mathrm{COT}_\ell$, plus *local* extremely efficient additions. Note that this protocol is fully paralellizable, as all the COT executions can be run in parallel.

Overall, our approach exploits the fact that $\mathbf{W}$ is ternary without having to perform any Boolean additions in secure computations. Our experiments in Section 5 show concrete gains over the prior state-of-the-art.

Communication costs. Using the $m \times \mathrm{COT}_\ell$ Extension protocol from [3], the parties running Protocol 7 send $m(\tau + \ell)$ bits to each other to compute inner products of length m, where τ is the security parameter (128 in our implementation). This results in $nm(\tau + \ell)$

bits being sent/received by each party for the whole matrix-vector multiplication protocol. In contrast, the OT-based approaches to matrix-vector multiplication entirely based on arithmetic sharing from [41] would require at least $nm\ell(\tau+\ell)$ (assuming optimizations like packing and vectorization).

Packing for matrix-vector multiplication. While the forward pass of quotient operates over 8-bit vectors (and thus $q=8$ in Protocol 8), the value of ℓ in implementations of $m \times \mathrm{COT}_\ell$ is 128, i.e., the AES block size. However, we can exploit this to pack $128/8=16$ vector multiplications against the same matrix for the same communication and computation. This is very useful in batched gradient descent, as this results in 16x additional savings in communication and computation. This packing optimization was also used in [41] for implementing an OT-based offline phase to matrix-vector multiplications occurring in batched gradient descent.

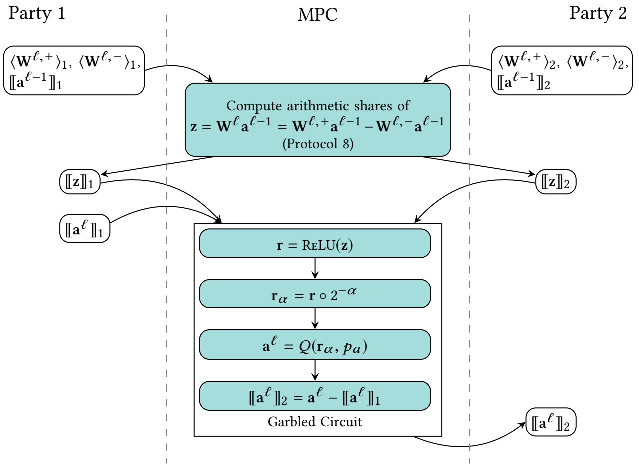
4.2 Secure Forward Pass

Our protocol for the forward pass (Algorithm 1) is a sequential composition of Protocol 8, and a garbled circuit protocol with three components: (i) Evaluation of ReLU, (ii) normalization by a public data-independent value α_l (line 3 in Algorithm 1), and (iii) quantization function $Q(\cdot)$ in eq. (2) in Section 3.1. The protocol is depicted in Figure 3. Note that we only show a forward pass for a single layer, but the protocol trivially composes sequentially with itself, as input $\mathbf{a}^{\ell-1}$ and output $\mathbf{a}^{\ell}$ are both secret-shared additively. Security follows directly from the security of the subprotocols, as their outputs and inputs are always secret-shares. For scalability, we describe an efficient circuit implementation of (i)-(iii). In our proposed circuit P_1 inputs its share $[\![\mathbf{z}]\!]_1$, and a random value chosen in advance that will become its share of the output, denoted $[\![\mathbf{a}^{\ell}]\!]_1$. In the garbled circuit, $\mathbf{z}$ is first reconstructed (this requires $|\mathbf{z}|$ parallel additions). For component (i), ReLU, we exploit the fact that the entries in $\mathbf{z}$ are encoded in binary in the circuit using two's complement as $z_i = (b_{i,k} \cdots b_{i,1})$, where $b_{i,k} = 1 \Leftrightarrow z_i < 0$. Hence $\mathrm{ReLU}(z_i) = (\neg b_{i,k}) \cdot b_{i,k}$. Note that this is very efficient, as it only requires to evaluate $k|\mathbf{z}|$ NOT and AND gates. The next two steps, (ii) normalization and (iii) quantization are extremely cheap, as they can be implemented with logical and arithmetic shifts without requiring any secure gate evaluation. Finally, to construct P_2's output we need to perform a subtraction inside the circuit to compute $[\![\mathbf{a}^{\ell}]\!]_2 = \mathbf{a}^{\ell} - [\![\mathbf{a}^{\ell}]\!]_1$. Altogether this means that our forward pass requires execution of Protocol 8 and a garbled circuit protocol to evaluate a vector addition, a vector subtraction, and a linear number of additional gates. Moreover, note that this garbled circuit evaluation can be parallelized across components of the vector $\mathbf{z}$.

4.3 Secure Backward Pass

Figure 4 shows our protocols for the backward pass (Algorithm 2). Analogous to the forward pass, we depict the pass for one layer, and observe that all its inputs and outputs are secret-shared. Hence, it can be easily composed sequentially across layers, and with the forward pass, and its security follows trivially from the security of each of the subprotocols.

The protocol works via a sequence of subprotocols. Each of them produces a result shared among the parties $\mathsf{P}_1, \mathsf{P}_2$ either as a Boolean-share or an additive share. As in the forward pass, this protocol leverages Protocol 8, as well as Protocol 7 for component-wise multiplication. Recall that the goal of the backward pass is to recompute ternary weight matrices $\{\mathbf{W}^{\ell}\}_{\ell=1}^{L}$ by means of a gradient-based procedure. As $\{\mathbf{W}^{\ell}\}_{\ell=1}^{L}$ is represented in our MPC protocol by pairs of binary matrices, the protocol to be run for each layer ℓ takes as inputs the Boolean-shares of such matrices, i.e., $\mathbf{W}^{\ell,-}$ and $\mathbf{W}^{\ell,+}$, from each party. Moreover, the parties contribute to the protocol arithmetic shares of the input to each layer a^{ℓ} computed in the forward pass, as well as the target values $\mathbf{y}$.



Party 1
MPC
Party 2
$\langle \mathbf{W}^{\ell,+}\rangle_1, \langle \mathbf{W}^{\ell,-}\rangle_1, [\![\mathbf{a}^{\ell-1}]\!]_1$
$\langle \mathbf{W}^{\ell,+}\rangle_2, \langle \mathbf{W}^{\ell,-}\rangle_2, [\![\mathbf{a}^{\ell-1}]\!]_2$
Compute arithmetic shares of
$\mathbf{z} = \mathbf{W}^{\ell}\mathbf{a}^{\ell-1} = \mathbf{W}^{\ell,+}\mathbf{a}^{\ell-1} - \mathbf{W}^{\ell,-}\mathbf{a}^{\ell-1}$
(Protocol 8)
$[\![\mathbf{z}]\!]_1$
$[\![\mathbf{z}]\!]_2$
$[\![\mathbf{a}^{\ell}]\!]_1$
$\mathbf{r} = \mathrm{ReLU}(\mathbf{z})$
$\mathbf{r}_\alpha = \mathbf{r} \circ 2^{-\alpha}$
$\mathbf{a}^{\ell} = Q(\mathbf{r}_\alpha, p_a)$
$[\![\mathbf{a}^{\ell}]\!]_2 = \mathbf{a}^{\ell} - [\![\mathbf{a}^{\ell}]\!]_1$
Garbled Circuit
$[\![\mathbf{a}^{\ell}]\!]_2$

Figure 3: Our MPC protocol for private prediction (forward pass). We compose three protocols to evaluate one layer of the form $f(\mathbf{W}\mathbf{a})$, with ternary $\mathbf{W}$, and where $f = \mathrm{ReLU}$.

The first step of the backward pass is a data-dependent normalization of the activation gradient $\mathbf{e}^{\ell}$, followed by the quantization step that we described in the forward pass, as shown in Figure 4. We now describe the design of the Boolean circuit used to compute these steps.

Normalization by the infinite norm. Our goal is to design a (small) circuit that, given $\mathbf{e}$, computes $\mathbf{e}/2^{\mathsf{npow}(\max\{|\mathbf{e}|\})}$. A naive circuit would compute the absolute value of every entry $|\mathbf{e}|$, compute the maximum value $\max\{|\mathbf{e}|\}$, compute $2^{\lceil \log_2(\max\{|\mathbf{e}|\})\rceil}$, and finally compute a division. However, computing exponentiation and logarithm in a garbled circuit would be prohibitively expensive. Such circuits are large, and we have to do this computation in each layer ℓ, as many times as the number of total iterations. To overcome this we apply two crucial optimizations: (i) approximate $\max\{|\mathbf{e}|\}$ by bitwise OR of all values $|\mathbf{e}_i|$, and (ii) compute npow with an efficient folklore procedure for obtaining the number of leading zeros in a binary string. This requires only b OR gates and $\log(b)$ arithmetic right shits and additions, where b is the bitwidth of the entries of $\mathbf{e}$ (8 in our applications) [28]. Also note that the division can be computed as an arithmetic right shift. An important remark is that the denominator is a private value in the circuit, which means that, although we can use right shift for division, our circuit needs to first compute all possible right shifts and then select the result according to the value of $\mathsf{npow}(\max\{|\mathbf{e}|\})$. This involves a subcircuit linear in b, and since in our case $b=8$ we once again benefit from having small bitwidth, by trading a small overhead for costly divisions, logarithms, and exponentiations.

Derivatives of ReLU and saturation. Computing derivatives of ReLU and saturation $S(\cdot)$ can be done efficiently in a Boolean representation, as they lie in $\{0, 1\}$. Specifically, computation only involves extracting the sign bit for ReLU, and ANDing a few bits for saturation. Ultimately we need to compute $\mathbf{e} \circ \mathbf{d}$, for $\mathbf{d} = \text{ReLU}'(\mathbf{a}) \circ S'(\mathbf{a})$ (line 4 in Algorithm 2). Not that this is a Boolean combination of integers, so we can alternate between Boolean and arithmetic shares and compute it with Protocol 7 (for component-wise product). We can further optimize the procedure by computing $\langle \mathbf{d} \rangle_i$ in the forward pass, since ReLU is already used there. This is commonly done in ML implementations in the clear.

The remainder of the backward pass involves (i) computing $\mathbf{e}^{l-1}$ (the rest of line 4 in Algorithm 2), for which we use Protocol 8, and (ii) an outer product between 8-bit vectors $\mathbf{a}^{l-1\top}\mathbf{e}$ (line 5). For (ii) we use a vectorized version of the OT-based multiplication protocol presented originally in [16], and used in [11, 41].

Overall, this makes our backward pass very efficient, involving three small garbled circuits (two can be parallelized), and relies heavily on oblivious transfer computations.

SGD. To implement Algorithm 3 we need to additionally keep higher precision 8-bit matrices $\{\mathbf{W}^{\ell}_{(\overline{p}_w)}\}_{\ell=1}^{L}$ as arithmetic shares. We ternarize these to obtain our weights $\mathbf{W}$, which we implement in the natural way with a small garbled circuit involving 2 comparisons. The same operations for quantization and normalization of $\mathbf{e}$ can be used for $\mathbf{G}$.

AMSGrad. Almost all of the operations in AMSGrad (Algorithm 4): quantization, normalization, saturation, and absolute value have been described as part of the previous protocols. The only addition is element-wise maximum (line 10), which we do via a comparison of Boolean shares.

Convolutional & residual layers. Although we have only described a fully connected layer, extending this protocol to convolutional layers is straightforward. The forward and backward passes of convolutional layers can be written using the same steps as Algorithms 1 and 2 with weight-reshaping. And max-pooling operations are simply comparisons, efficiently implemented in Boolean shares. Similarly, for residual networks, we only introduce integer additions in the forward pass (which we can perform on additive shares) and another computation of $\mathbf{d}$ for the backward pass.

5 Experiments

In this section, we present our experimental results for secure DNN training and prediction.

Experimental Settings. The experiments were executed over two Microsoft Azure Ds32 v3 machines equipped with 128GB RAM and Intel Xeon E5-2673 v4 2.3GHz processor, running Ubuntu 16.04. In LAN experiments, machines were hosted in the same region (West Europe) with an average latency of 0.3ms and a bandwidth of 1.82GB/s. For WAN, the machines were hosted in two different regions (North Europe & East US), with an average latency of 42ms and a bandwidth of 24.3 MB/sec. The machine specifications were chosen to be comparable with the ones in [41], hence enabling direct running time comparisons.

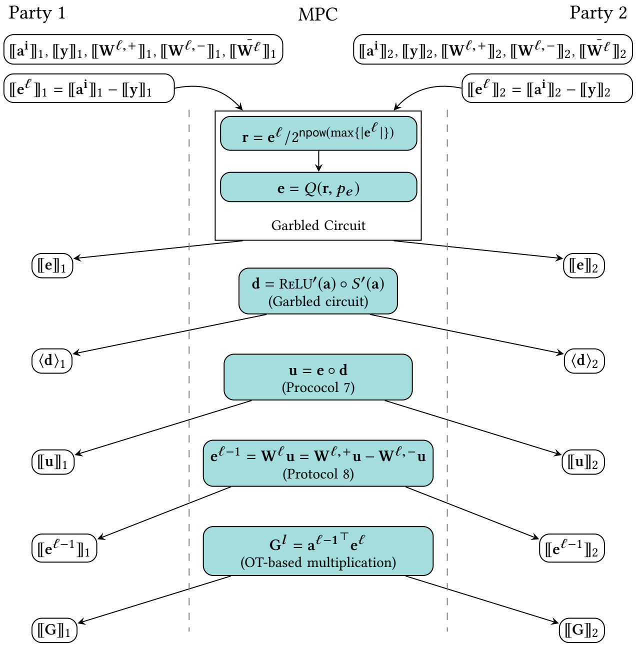

Party 1
MPC
Party 2
$[\![\mathbf{a^i}]\!]_1, [\![\mathbf{y}]\!]_1, [\![\mathbf{W}^{\ell,+}]\!]_1, [\![\mathbf{W}^{\ell,-}]\!]_1, [\![\bar{\mathbf{W}}^{\ell}]\!]_1$
$[\![\mathbf{a^i}]\!]_2, [\![\mathbf{y}]\!]_2, [\![\mathbf{W}^{\ell,+}]\!]_2, [\![\mathbf{W}^{\ell,-}]\!]_2, [\![\bar{\mathbf{W}}^{\ell}]\!]_2$
$[\![\mathbf{e}^{\ell}]\!]_1 = [\![\mathbf{a^i}]\!]_1 - [\![\mathbf{y}]\!]_1$
$[\![\mathbf{e}^{\ell}]\!]_2 = [\![\mathbf{a^i}]\!]_2 - [\![\mathbf{y}]\!]_2$
$\mathbf{r} = \mathbf{e}^{\ell}/2^{\mathsf{npow}(\max\{|\mathbf{e}^{\ell}|\})}$
$\mathbf{e} = Q(\mathbf{r}, p_e)$
Garbled Circuit
$[\![\mathbf{e}]\!]_1$
$[\![\mathbf{e}]\!]_2$
$\mathbf{d} = \text{ReLU}'(\mathbf{a}) \circ S'(\mathbf{a})$
(Garbled circuit)
$\langle \mathbf{d} \rangle_1$
$\langle \mathbf{d} \rangle_2$
$\mathbf{u} = \mathbf{e} \circ \mathbf{d}$
(Prococol 7)
$[\![\mathbf{u}]\!]_1$
$[\![\mathbf{u}]\!]_2$
$\mathbf{e}^{\ell-1} = \mathbf{W}^{\ell}\mathbf{u} = \mathbf{W}^{\ell,+}\mathbf{u} - \mathbf{W}^{\ell,-}\mathbf{u}$
(Protocol 8)
$[\![\mathbf{e}^{\ell-1}]\!]_1$
$[\![\mathbf{e}^{\ell-1}]\!]_2$
$\mathbf{G}^{l} = \mathbf{a}^{\ell-1\top}\mathbf{e}^{\ell}$
(OT-based multiplication)
$[\![\mathbf{G}]\!]_1$
$[\![\mathbf{G}]\!]_2$

Figure 4: Our protocol for the backward pass, corresponding to Algorithms 2 from Section 3.

Implementation. We use two distinct code bases. We use the EMP-toolkit [55] to implement our secure protocols for forward and backward passes, as described in Section 4. EMP is written in C++ and offers efficient implementations of OT and COT extension [3]. We extended the semi-honest COT implementation to the functionality required for Protocol 7, as it is currently limited to correlation functions of the form $f(x) = x \oplus \Delta$ (the ones required by Yao's garbled circuits protocol). This code base was used for timing results. We developed a more versatile insecure Python implementation based on Tensorflow[1] for accuracy experiments. While this implementation does not use MPC, it mirrors the functionality implemented using the EMP-toolkit.

Evaluations. For training over QUOTIENT, we use two weight variables as described in Section 3: (i) ternary (2-bit) weights for the forward and backward passes, and (ii) 8-bit weights for the SGD and AMSgrad algorithms. We use 8-bits for the quantized weight gradients (g), activations (a) and activation gradients (e).

As our protocols are online, to compare with other approaches employing an offline phase, we take a conservative approach: we compare their offline computation time with our total computation time using similar computational resources. We adopt this strategy because online phases for these approaches are relatively inexpensive and could potentially involve a set-up overhead. Additionally, our model could easily be divided into offline/online phases, but we omit this for simplicity.

We employ a naive parallelization strategy: running independent processes over a mini-batch on different cores. This speeds up the

Network	k	QUOTIENT (s)	GC (s)	SecureML (OT) (s)	SecureML (LHE) (s)
LAN	10^3	0.08	0.025	0.028	5.3
	10^4	0.08	0.14	0.16	53
	10^5	0.13	1.41	1.4	512
	10^6	0.60	13.12	14^*	5000^*
	10^7	6.0	139.80	140^*	50000^*
WAN	10^3	1.7	1.9	1.4	6.2
	10^4	1.7	3.7	12.5	62
	10^5	2.6	20	140	641
	10^6	7.3	148	1400^*	6400^*
	10^7	44	1527	14000^*	64000^*

Table 1: Comparison of our COT-based component-wise multiplication of k-dimensional vectors with ternary fixed-point multiplication using garbled circuits (GC) and SecureML [41] (OT, LHE). One of the vectors hold only ternary values.

Network	n	QUOTIENT (s)	SecureML (OT Vec) (s)	SecureML (LHE Vec) (s)
LAN	100	0.08	0.05	1.6
	500	0.1	0.28	5.5
	1000	0.14	0.46	10
WAN	100	1.7	3.7	2
	500	2	19	6.2
	1000	2.7	34	11

Table 2: Performance comparison of our matrix-vector multiplication approach with the vectorized approaches of SecureML [41] (OT, LHE). Here we multiply a $128 \times n$ ternary matrix with an n-dimensional vector.

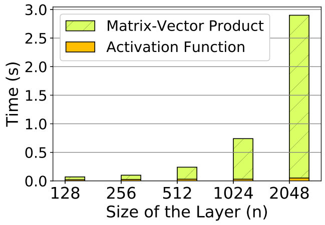


Figure 5: Forward pass time for single prediction over an $n \times n$ fully connected layer.

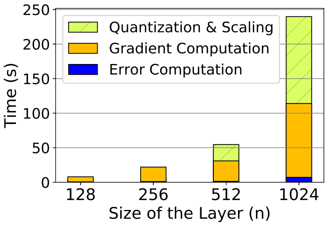


Figure 6: Forward and Backward pass time over an $n \times n$ fully-connected layer for 1 batch. Here batch size = 128.

computation on average by 8-15x over LAN and by about 10-100x over WAN depending on the number of parallelizable processes. We leave more involved parallelization strategies to future work.

5.1 Data-independent benchmarking

In this section, we present the running times of the basic building blocks that will be used for DNN prediction and training.

Component-wise Multiplication. Table 1 compares the running times of our COT-based approach from Protocol 7 for computing component-wise product with (i) an implementation of Algorithm 5 in a garbled circuit and (ii) two protocols proposed in SecureML [41] for the offline phase. Their first protocol corresponds to Gilboa's method for oblivious product evaluation (the OT-based variant implemented with a packing optimization). Their second is a sequence of Paillier encryptions, decryptions, and homomorphic additions (the LHE variant). For QUOTIENT and GC, one vector is ternary and the other holds 128-bit values, while for OT and LHE both vectors hold 32-bit values. Although, our protocols for multiplication are independent and suitable for parallelization, here we benchmark without parallelization. QUOTIENT clearly outperforms all other approaches as soon as we move past the set-up overhead of the base OTs. Note that most DNN layers involve greater than 10^4 multiplications, which makes our approach more suitable for those applications.

Matrix-Vector Product. As discussed in Section 4, our component-wise multiplication directly translates into matrix-vector products with local additions as described in Protocol 8. However, the protocols from [41] benefit greatly from a vectorization optimization, and thus a comparison of the matrix-vector multiplication tasks is important. Table 2 compares the performance of implementation of Protocol 8, for a ternary $128 \times n$ matrix and an n-dimensional vector. Similar to Table 1, we populate the matrix with ternary values (2-bit values) and the vector with up to 128-bit values. Our approach is at least 5x faster than the vectorized LHE protocol from [41] on LAN, and is roughly 10x faster than the OT protocol from [41] on WAN for $n \geq 500$. In general, the speedup increases as we increase the number of computations.

Layer Evaluation. Furthermore, we benchmark the basic building blocks of secure DNN training framework—forward and backward pass for a variety of different layer sizes. Figure 5 shows the running time of QUOTIENT for the forward pass as we increase the layer size. We split the total time into time spent on the matrix-vector product (Protocol 6) and computation of activation function (ReLU) using garbled circuits. Figure 6 shows the running time of the forward and backward pass, over a single batch of size 128. We report the running time of each of the three required functionalities: quantization & scaling, gradient computation, and error computation. As we increase the size of the layers, the garbled circuit for the quantization phase starts to dominate over the COT based matrix-matrix product required for the gradient computation. This can primarily be attributed to the quantization and scaling of the gradient matrix. Our COT-based matrix-vector multiplication shifts the bottleneck from the multiplication to the garbled circuits based scaling phase. Finding efficient protocols for that task is an interesting task for future work. In particular, parallelized garbled circuits and optimization of the matrix-matrix multiplication in the gradient computation phase could be explored.

5.2 Experiments on Real-World Data

In this section, we evaluate our proposed QUOTIENT on real-world datasets. We show that: (i) QUOTIENT incurs only a small accuracy loss with respect to training over floating point, and (ii) it is more accurate than the state-of-the-art in fixed-point NN training, namely WAGE [57]. Both (i) and (ii) hold for several state-of-the-art architectures including fully connected, convolutional, and residual layers, and for different types of data (for residual layer results see the Appendix B). For our 2PC protocols, we show that (iii) 2PC-QUOTIENT outperforms the existing secure training approach for

DNNs SecureML [41], both in terms of accuracy and running time, and both in the LAN and WAN settings. We first report the accuracy across a variety of datasets to show (i) and (ii) above. Then we report running times for 2PC-QUOTIENT training and prediction to argue (iii).

5.2.1 Datasets and Deep Neural Network Architectures. We evaluate QUOTIENT on six different datasets. Five of these are privacy sensitive in nature, as they include health datasets—Thyroid [45], Breast cancer [8], MotionSense [38], Skin cancer MNIST [53] and a financial credit dataset—German credit [12]. We also evaluate our approach on MNIST [34] for the purpose of benchmarking against prior work.

MNIST contains 60K training and 10K test grayscale (28 × 28) images of 10 different handwritten digits. We adopt the state of the art floating point convolutional neural network LeNet [34] (32C5-BN-32C5-BN-MP2-64C5-BN-64C5-BN-MP2-512FC-BN-DO-10SM)[2] into a fixed-point equivalent of the form 32C5-MP2-64C5-MP2-512FC-10MSE for secure training & inference. In addition, we explore a variety of fully-connected neural networks 2×(128FC)-10MSE, 3×(128FC)-10MSE, 2×(512FC)-10MSE & 3×(512FC)-10MSE. We set the learning to 1 for both SGD and AMSgrad optimizers.

MotionSense contains smartphone accelerometer and gyroscope sensor data for four distinct activities namely walking, jogging, upstairs, and downstairs. For each subject, the dataset contains about 30 minutes of continuously recorded data. We use a rolling window, of size 2.56 seconds each for extracting around 50K samples for training and 11K for testing. As proposed in [38] we use floating point convolutional neural network 64C3-BN-MP2-DO-64C3-BN-MP2-DO-32C3-BN-MP2-DO-32C3-BN-MP2-DO-256FC-BN-DO-64FC-BN-DO-4SM and it's fixed-point analogue of the form 64C3-MP2-64C3-MP2-32C3-MP2-32C3-MP2-256FC-64FC-4MSE. We furthermore explore a fully connected architecture of the form 3×(512FC)-4MSE.

Thyroid contains 3.7K training sample and 3.4K test samples of 21 dimensional patient data. The patients are grouped into three classes namely normal, hyperfunction and subnormal based on their thyroid functioning. We use a fully-connected neural network of the form 2×(100FC)-3SM for this dataset and its analogue fixed-point network 2×(100FC)-3MSE.

Breast cancer contains 5547 breast cancer histopathology RGB (50 × 50) images segregated into two classes—invasive ductal carcinoma and non-invasive ductal carcinoma. We use the 90:10 split for training and testing. We use a convolutional neural network with 3×(36C3-MP2)-576FC-2SM and a fully-connected network of the form 3×(512FC) along with their fixed point analogues.

Skin Cancer MNIST contains 8K training and 2K (28 × 28) dermatoscopic RGB images. They have been grouped into seven skin lesion categories. We use floating point network ResNet-20 [21]. ResNet-20 is made up of batch-normalisation, dropout, SoftMax layers and employs cross-entropy loss for training in addition to the residual layers. For the fixed-point version of ResNet-20 [21], we exclude batch normalisation, average pooling, and SoftMax layers. In addition, we use a fully connected architecture of the form 2×(512FC)-7MSE for secure training.

Dataset	QUOTIENT (%)	Floating point (%)
MNIST	99.38	99.48
MotionSense	93.48	95.65
Thyroid	97.03	98.30
Breast cancer	79.21	80.00
German credit	79.50	80.50

Table 3: Accuracy comparison of training over state of the art floating point neural networks their fixed point equivalents using QUOTIENT.

German credit contains 1k instances of bank account holders. they have been divided into two credit classes—Good or Bad. Each individual has 20 attributes (7 quantitative and 13 categorical). As a preprocessing step, we normalize the quantitative variables and encode the categorical variables using one-hot encoding. This amounts to 60 distinct feature for each individual in the dataset. We use the 80:20 split for training and testing. We use a fully-connected neural network of the form 2×(124FC)-2SM and its fixed point analogue for this dataset.

5.2.2 Accuracy. We evaluate the accuracy of QUOTIENT on different datasets and architectures. Also, we judge the impact of our MPC-friendly modifications on accuracy. To do so we consider four variants of QUOTIENT: (i) Our proposed QUOTIENT, with secure AMSgrad optimizer (`QUOTIENT[AMSgrad=Ours, Norm=Ours]`) (ii) QUOTIENT with the standard AMSgrad optimizer (`QUOTIENT[AMSgrad=Std, Norm=Ours]`) (described in Appendix A) (iii) QUOTIENT with our proposed AMSgrad with the closest power of 2 (C-Pow2) functionality instead of next power of 2 for quantization (`QUOTIENT[AMSgrad=Ours, Norm=WAGE]`) (iv) QUOTIENT with the standard SGD optimizer (`QUOTIENT[SGD=Std, Norm=Ours]`).

Comparison with Floating Point Training. As a baseline evaluation of our training using `QUOTIENT[AMSgrad=Ours, Norm=Ours]`, we compare its performance (upon convergence) with the floating point training counterparts in Table 3. For MNIST and Breast cancer datasets `QUOTIENT[AMSgrad=Ours, Norm=Ours]` training achieves near state of the art accuracy levels for MNIST and Breast Cancer, while we differ by at most ~2% for German credit, MotionSense and Thyroid datasets.

Secure AMSgrad vs SGD. Figure 7 shows the training curves for QUOTIENT over all the datasets. In particular, we compare `QUOTIENT[AMSgrad=Ours, Norm=Ours]` and `QUOTIENT[SGD=Std, Norm=Ours]`. The secure AMSgrad optimizer converges faster than the secure SGD optimizer especially for convolutional neural networks (used for MNIST, MotionSense, and Breast cancer).

Effects of our Optimizations. In order to evaluate the impact of our optimizations, we compare training with floating point and WAGE [57] with the first three variations of QUOTIENT in Figure 8. For comparison with WAGE, we use the same fixed-point architecture as the ones used with QUOTIENT with an exception of the choice of optimizer— we employ our proposed secure AMSgrad,

---

[2] BN, DO, and SM are batch-normalization, DropOut, and SoftMax respectively.

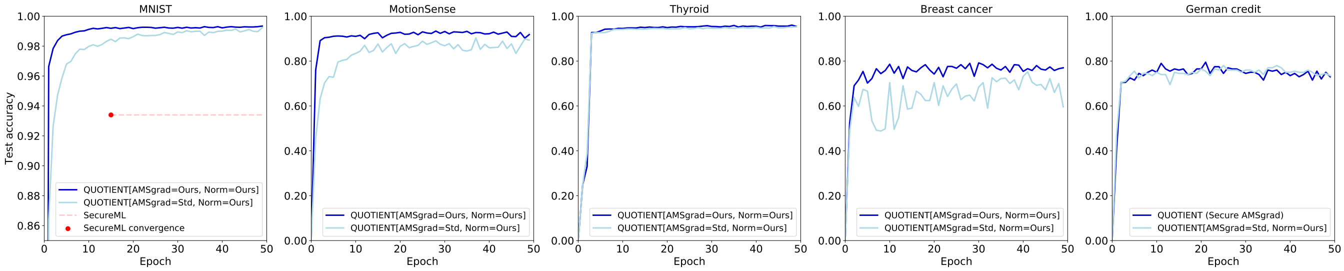


Figure 7: Performance comparison of secure AMSgrad and secure SGD for QUOTIENT. The plots compare training curves over MNIST (using CNN), MotionSense, Thyroid, Breast cancer and German credit datasets.

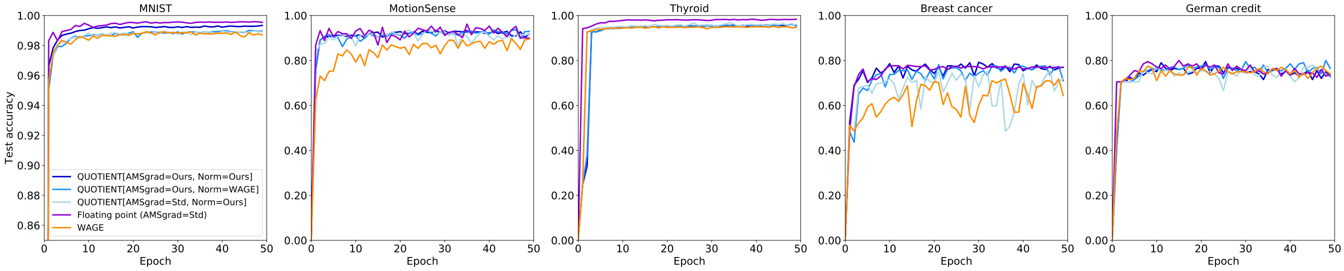


Figure 8: Performance comparison of three different variants of QUOTIENT training with floating point and WAGE [57] training on MNIST, MotionSense, Thyroid, Breast cancer and German credit datasets. `QUOTIENT[AMSgrad=Ours, Norm=WAGE]` and `QUOTIENT[AMSgrad=Std, Norm=Ours]`) differ from (`QUOTIENT[AMSgrad=Ours, Norm=Ours]` in using next power of 2 vs the Closet Power of 2 and using standard AMSgrad by changing lines 9 and 11 of secure AMSgrad in Algorithm 4, respectively.

while WAGE uses a SGD optimizer in conjunction with randomized rounding. Moreover, the floating point analogues employ (a) batch normalization and dropout in each layer, (b) softmax for the last layer, and (c) cross-entropy loss instead of MSE. We observe that `QUOTIENT[AMSgrad=Ours, Norm=Ours]` outperforms all the other variants and is very close to the floating point training accuracy levels. We believe this is due to our modified AMSgrad, our normalization scheme, and use of the next power of two (used in `QUOTIENT[AMSgrad=Ours, Norm=Ours]`) which may act as an additional regularization compared with the closest power of two.

5.2.3 End-to-End Running Times. In the previous section, we demonstrated that large networks can match and improve upon the state-of-the-art in fixed-point deep networks. However, often one can use much simpler networks that are much faster and only sacrifice little accuracy. In order to balance accuracy and run-time, we design practical networks for each dataset and evaluate them here.

2PC-QUOTIENT Training. Table 4 shows the running time of 2PC-QUOTIENT for practical networks across all datasets over LAN and WAN. We report accuracy and timings at 1,5 and 10 epochs. We note that the training time grows roughly linearly with the number of epochs. In all cases except the largest MNIST model, 10 epochs finish in under 12 days. Note that standard large deep models can easily take this long to train.

Our training protocols port nicely to the WAN settings. On average, our networks are only about 5x slower over WAN than over LAN. This is is due to the low communication load and round-trip of our protocols. As a result, we present the first 2PC pragmatic neural network training solution over WAN.

2PC-QUOTIENT Prediction. In addition to secure training, forward pass of QUOTIENT can be used for secure prediction. Table 5 presents prediction timings for all datasets over LAN. In addition to fully connected networks, we also report the prediction timings over convolutional neural networks (we report residual networks in Appendix B). The timings have been reported for a single point as well as 128 parallel batched predictions (this corresponds to classifying many data points in one shot). Except for the multi-channel Breast cancer dataset, all single predictions take less than 1s and all batched predictions take less than 60s.

Table 6 presents an equivalent of Table 5 under WAN settings. Here we limit ourselves to only fully connected architectures. While still practical, per prediction costs over WAN are, on an average, 20x slower than over LAN. This is due to the high initial setup overhead over WAN. However, batched predictions are only about 4-6x slower over WAN.

Comparison with Previous Work. The only prior work for 2PC secure training of neural networks that we are aware of is SecureML [41]. They report the results for only fully connected neural network training on the MNIST dataset. 2PC-QUOTIENT is able to achieve its best accuracy levels in less than 16 hours over LAN and less than 90 hours over WAN. This amounts to a speedup of 5x over LAN and a speedup of 50x over WAN. In particular, QUOTIENT is able to make 2PC neural network training over WAN practical. Moreover, our 2PC-QUOTIENT training is able to achieve near state of the art

LAN																
	MNIST								MotionSense		Thyroid		Breast cancer		German credit	
	2 × (128FC)		3 × (128FC)		2 × (512FC)		3 × (512FC)		2 × (512FC)		2 × (100FC)		3 × (512FC)		2 × (124FC)	
Epoch	Time	Acc	Time	Acc	Time	Acc	Time	Acc	Time	Acc	Time	Acc	Time	Acc	Time	Acc
1	8.72h	0.8706	10.05h	0.9023	27.13h	0.9341	38.76h	0.9424	10.07h	0.8048	0.08h	0.2480	14.51h	0.4940	0.03h	0.4100
5	43.60h	0.9438	50.25h	0.9536	135.65h	0.9715	193.80h	0.9745	50.35h	0.8847	0.40h	0.9341	72.55h	0.7360	0.15h	0.725
10	87.20h	0.9504	100.50h	0.9604	271.30h	0.9772	387.60h	0.9812	100.70h	0.8855	0.80h	0.9453	145.10h	0.7600	0.30h	0.7900
WAN																
	MNIST								MotionSense		Thyroid		Breast cancer		German credit	
	2 × (128FC)		3 × (128FC)		2 × (512FC)		3 × (512FC)		2 × (512FC)		2 × (100FC)		3 × (512FC)		2 × (124FC)	
Epoch	Time	Acc	Time	Acc	Time	Acc	Time	Acc	Time	Acc	Time	Acc	Time	Acc	Time	Acc
1	50.74h	0.8706	57.90h	0.9023	139.71h	0.9341	190.10h	0.9424	44.43h	0.8048	0.52h	0.2480	74.10h	0.4940	0.15h	0.4100
5	253.7h	0.9438	289.5h	0.9536	698.55h	0.9715	950.5h	0.9745	222.15h	0.8847	2.6h	0.9341	370.5h	0.7360	0.75h	0.725
10	507.4h	0.9504	579h	0.9604	1397.1h	0.9772	1901h	0.9812	444.3h	0.8855	5.2h	0.9453	741h	0.7600	1.5h	0.7900

Table 4: Training time and accuracy values for various datasets and architectures after 1, 5 & 10 training epochs using QUOTIENT over LAN and WAN.

	MNIST					MotionSense		Thyroid	Breast cancer		German credit
	2 × (128FC)	3 × (128FC)	2 × (512FC)	3 × (512FC)	Conv	2 × (512FC)	Conv	2 × (100FC)	3 × (512FC)	Conv	2 × (124FC)
Single Prediction(s)	0.356	0.462	0.690	0.939	192	0.439	134	0.282	3.58	62	0.272
Batched Prediction (s)	2.24	2.88	4.79	6.50	2226	2.91	1455	1.83	44.02	447	1.77

Table 5: Prediction time for various datasets and architectures using QUOTIENT over LAN. Here batch size = 128.

	MNIST				MotionSense	Thyroid	Breast cancer	German credit
	2 × (128FC)	3 × (128FC)	2 × (512FC)	3 × (512FC)	2 × (512FC)	2 × (100FC)	3 × (512FC)	2 × (124FC)
Single Prediction(s)	6.8	8.8	14.4	19.9	9.46	5.99	33.3	5.1
Batched Prediction (s)	8.3	10.9	22.6	29.9	12.08	6.89	69.1	7.3

Table 6: Prediction time for various datasets and architectures using QUOTIENT over WAN. Here batch size = 128.

accuracy of 99.38% on MNIST dataset, amounting to an absolute improvement of 6% over SecureML's error rates upon convergence. In terms of secure prediction, 2PC-QUOTIENT is 13x faster for single prediction and 7x faster for batched predictions over LAN in comparison to SecureML. Furthermore, 2PC-QUOTIENT offers a 3x speed-up for a single prediction and 50x for batched predictions versus SecureML over WAN.

6 Conclusion

In this paper, we introduced QUOTIENT, a new method to train deep neural network models securely that leverages oblivious transfer (OT). By simultaneously designing new machine learning techniques and new protocols tailored to machine learning training, QUOTIENT improves upon the state-of-the-art in both accuracy and speed. Our work is the first we are aware of to enable secure training of convolutional and residual layers, key building blocks of modern deep learning. However, training over convolutional networks is still slow, and incurs on a large communication load with our methods. Finding dedicated MPC protocols for fast evaluation of convolutional layers is an interesting venue for further work.

Acknowledgments

Adrià Gascón and Matt J. Kusner were supported by The Alan Turing Institute under the EPSRC grant EP/N510129/1, and funding from the UK Government's Defence & Security Programme in support of the Alan Turing Institute.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*. 265–283.

[2] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*. 1709–1720.

[3] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2013. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM conference on Computer & communications security*. ACM, 535–548.

[4] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. 2018. SIGNSGD: Compressed Optimisation for Non-Convex Problems. In *International Conference on Machine Learning*. 559–568.

[5] Johan Bjorck, Carla Gomes, Bart Selman, and Kilian Q. Weinberger. 2018. Understanding Batch Normalization. *arXiv preprint arXiv:1806.02375* (2018).

[6] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*. Springer, 192–206.

[7] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. 2018. Fast homomorphic evaluation of deep discretized neural networks. In *Annual International Cryptology Conference*. Springer, 483–512.

[8] Angel Cruz-Roa, Ajay Basavanhally, Fabio González, Hannah Gilmore, Michael Feldman, Shridar Ganesan, Natalie Shih, John Tomaszewski, and Anant Madabhushi. 2014. Automatic detection of invasive ductal carcinoma in whole slide images with convolutional neural networks. In *Medical Imaging 2014: Digital Pathology*, Vol. 9041. International Society for Optics and Photonics, 904103.

[9] Christopher De Sa, Megan Leszczynski, Jian Zhang, Alana Marzoev, Christopher R Aberger, Kunle Olukotun, and Christopher Ré. 2018. High-accuracy low-precision training. *arXiv preprint arXiv:1803.03383* (2018).

[10] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. 2015. Automated Synthesis of Optimized Circuits for Secure Computation. In *ACM Conference on Computer and Communications Security*. ACM, 1504–1517.

[11] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*. The Internet Society.

[12] Dua Dheeru and Efi Karra Taniskidou. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[13] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12, Jul (2011), 2121–2159.

[14] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. 2017. Privacy-preserving distributed linear regression on high-dimensional data. *Proceedings on Privacy Enhancing Technologies* 2017, 4 (2017), 345–364. https://doi.org/10.1515/popets-2017-0053

[15] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *ICML (JMLR Workshop and Conference Proceedings)*, Vol. 48. JMLR.org, 201–210.

[16] Niv Gilboa. 1999. Two party RSA key generation. In *Annual International Cryptology Conference*. Springer, 116–129.

[17] Oded Goldreich. 2004. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press.

[18] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*. ACM, 218–229.

[19] Google. 2018. TensorFlow Lite. https://www.tensorflow.org/mobile/tflite.

[20] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *International Conference on Machine Learning*. 1737–1746.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[22] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N Wright. 2018. Privacy-preserving machine learning as a service. *Proceedings on Privacy Enhancing Technologies* 2018, 3 (2018), 123–142.

[23] Lu Hou, Ruiliang Zhang, and James T Kwok. 2019. Analysis of Quantized Models. In *International Conference on Learning Representations*.

[24] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Advances in neural information processing systems*. 4107–4115.

[25] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[26] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending oblivious transfers efficiently. In *Annual International Cryptology Conference*. Springer, 145–161.

[27] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *CVPR*. IEEE Computer Society, 2704–2713.

[28] Henry S. Warren Jr. 2013. *Hacker's Delight, Second Edition*. Pearson Education. http://www.hackersdelight.org/

[29] Niki Kilbertus, Adrià Gascón, Matt Kusner, Michael Veale, Krishna P Gummadi, and Adrian Weller. 2018. Blind Justice: Fairness with Encrypted Sensitive Attributes. In *International Conference on Machine Learning*. 2635–2644.

[30] Minje Kim and Paris Smaragdis. 2016. Bitwise neural networks. *arXiv preprint arXiv:1601.06071* (2016).

[31] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[32] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved garbled circuit: Free XOR gates and applications. In *International Colloquium on Automata, Languages, and Programming*. Springer, 486–498.

[33] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al. 2017. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In *Advances in neural information processing systems*. 1742–1752.

[34] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[35] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[36] Hao Li, Soham De, Zheng Xu, Christoph Studer, Hanan Samet, and Tom Goldstein. 2017. Training quantized nets: A deeper understanding. In *Advances in Neural Information Processing Systems*. 5811–5821.

[37] Yehuda Lindell and Benny Pinkas. 2009. A proof of security of Yao's protocol for two-party computation. *Journal of cryptology* 22, 2 (2009), 161–188.

[38] Mohammad Malekzadeh, Richard G Clegg, Andrea Cavallaro, and Hamed Haddadi. 2018. Mobile Sensor Data Anonymization. *arXiv preprint arXiv:1810.11546* (2018).

[39] Daisuke Miyashita, Edward H Lee, and Boris Murmann. 2016. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025* (2016).

[40] Payman Mohassel and Peter Rindal. 2018. ABY 3: a mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM Conference on Computer and Communications Security*. ACM, 35–52.

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

[42] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 377–394.

[43] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. 2013. Privacy-preserving matrix factorization. In *ACM Conference on Computer and Communications Security*. ACM, 801–812.

[44] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. 2013. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 334–348.

[45] J. Ross Quinlan. 1987. Simplifying decision trees. *International journal of man-machine studies* 27, 3 (1987), 221–234.

[46] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.

[47] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. 2018. On the Convergence of Adam and Beyond. In *International Conference on Learning Representations*.

[48] M Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. 2019. XONN: XNOR-based Oblivious Deep Neural Network Inference. *arXiv preprint arXiv:1902.07342* (2019).

[49] Tim Salimans and Durk P Kingma. 2016. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*. 901–909.

[50] Amartya Sanyal, Matt J. Kusner, Adrià Gascón, and Varun Kanade. 2018. TAPAS: Tricks to Accelerate (encrypted) Prediction As a Service. In *International Conference on Machine Learning*. 4497–4506.

[51] Phillipp Schoppmann, Adrià Gascón, and Borja Balle. 2018. Private Nearest Neighbors Classification in Federated Databases. *IACR Cryptology ePrint Archive* 2018 (2018), 289.

[52] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2015. Compacting privacy-preserving k-nearest neighbor search using logic synthesis. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference*. IEEE, 1–6.

[53] Philipp Tschandl. 2018. The HAM10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions. https://doi.org/10.7910/DVN/DBW86T

[54] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proceedings on Privacy Enhancing Technologies* 1 (2019), 24.

[55] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.

[56] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*. 1509–1519.

[57] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. 2018. Training and inference with integers in deep neural networks. *arXiv preprint arXiv:1802.04680* (2018).

[58] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science*. IEEE, 162–167.

A Standard AMSgrad Optimizer

Algorithm 9 describes the steps for training deep neural networks using standard AMSgrad optimizer. This can be summarised as: (i) Sampling the input pair (a^0, y) from the dataset $\mathcal{D}$; (ii) Using

Algorithm 9: Standard AMSgrad Optim. (Δ_{ams})

```
Input: Weights $\{\mathbf{W}^l\}_{l=1}^L$,
Data $\mathcal{D} = \{(\mathbf{a}_i^0, y_i)\}_{i=1}^n$, learning rate $\eta$
Output: Updated weights $\{\mathbf{W}^l\}_{l=1}^L$
Initialize: $\{\mathbf{M}^l, \mathbf{V}^l\}_{l=1}^L = 0$
for $t = 1, \ldots, T$ do
    $(\mathbf{a}^0_{(p_a)}, y_{(p_a)}) \sim \mathcal{D}$
    $\{\mathbf{a}^l\}_{l=1}^L = \texttt{Forward}(\{\mathbf{W}^l\}_{l=1}^L, \mathbf{a}^0)$
    $\{\mathbf{G}^l\}_{l=1}^L = \texttt{Backward}(\{\mathbf{W}^l, \mathbf{a}^l\}_{l=1}^L, y)$
    $\{\mathbf{M}^l = 0.9\mathbf{M}^l + 0.1\mathbf{G}^l\}_{l=1}^L$    ▷ weighted mean
    $\{\mathbf{V}^l = 0.99\mathbf{V}^l + 0.01(\mathbf{G}^l)^2\}_{l=1}^L$    ▷ weighted variance
    $\{\hat{\mathbf{V}}^l = \max(\hat{\mathbf{V}}^l, \mathbf{V}^l)\}_{l=1}^L$
    $\{\mathbf{G}^l = \frac{\mathbf{M}^l}{\sqrt{\hat{\mathbf{V}}^l+\epsilon}}\}_{l=1}^L$    ▷ history-based scaling with square root
    $\{\mathbf{W}^l = \mathbf{W}^l - \eta\mathbf{G}^l\}_{l=1}^L$
```

	LAN		WAN
	2 × (512FC)	ResNet	2 × (512FC)
Single Prediction(s)	1.269	1982	17
Batched Prediction (s)	14.31	18122	39.9

Table 7: Prediction time for Residual and fully-connected neural networks on Skin cancer MNIST using QUOTIENT over LAN. Here batch size = 128.

	2 × (512FC)		
Epoch	LAN	WAN	Acc
1	8.57h	38.44h	0.2078
5	42.85h	192.2h	0.7026
10	85.70h	384.4h	0.7157

Table 8: Training time and accuracy values of QUOTIENT on Skin cancer MNSIT dataset after 1, 5 & 10 training epochs using fully-connected neural network over both LAN and WAN.

the input ($\mathbf{a}^0$) to obtain the prediction $\{\mathbf{a}^l\}_{l=1}^L$; (iii) Computing the loss between the prediction $\{\mathbf{a}^l\}_{l=1}^L$ and the target output (y), and computing the gradient $\mathbf{G}^l$ for the loss $\ell(\mathbf{a}_i^L, y_i)$ with respect to each of the weights $\mathbf{W}_l$ in the network; (iv) Updating the weights using a weighted average of the past gradients, specifically, their first and second moments $\mathbf{M}^l$ and $\mathbf{V}^l$.

B Experiments on Residual Layers

In addition to fully-connected and convolutional layers, we also evaluate 2PC-QUOTIENT on residual neural networks. Table 7 shows the prediction time of 2PC-QUOTIENT on Skin cancer MNIST dataset using for both fully-connected and residual neural networks, while we show its training timing using practical fully-connected neural networks in Table. 8.

C Proof of Protocol 6

Lemma C.1. *Let* $\mathbf{b}$ *and* $\mathbf{a}$ *be a Boolean and integer vector, respectively, shared among parties* $\mathsf{P}_1, \mathsf{P}_2$. *Given a secure OT protocol, the two-party Protocol 6 is secure against semi-honest adversaries, and computes an additive share of the inner product* $\mathbf{b}^\top\mathbf{a}$ *among* $\mathsf{P}_1, \mathsf{P}_2$.

Proof. For the correctness, note that, for all $j \in [m]$, $b_j a_j = b_j\llbracket a_j\rrbracket_1 + b_j\llbracket a_j\rrbracket_2$, and that each of the OTs in steps 4 and 5 of the algorithm are used to compute an additive share of each of the terms of the sum. Concretely, in the OT in line 4 is used to compute an additive share of $b_j\llbracket a_j\rrbracket_1$ as $z_{1,j} + z'_{1,j}$, and the value received by party P_2 is $m_{1,\llbracket b_j\rrbracket_2} = -z_{1,j}$ if $\llbracket b_j\rrbracket_1 \oplus \llbracket b_j\rrbracket_2 = 0$ and $m_{1,\llbracket b_j\rrbracket_2} = \llbracket a_j\rrbracket_i - z_{1,j}$ otherwise. Security follows easily from the fact that all messages in the OTs are masked with fresh randomness $z_{i,j}$ and thus constructing simulators from a simulator for OT is straightforward. ◻

D Proof of Protocol 7

Proof. Privacy follows directly from the correctness of the COT subprotocol. To see that the protocol computes the right value, we argue for a $j \in [m]$, and distinguish cases according to all possible values of the shares of w_j.

Case I ($w_j = 0$; $\llbracket w_j\rrbracket_1 = 1$, $\llbracket w_j\rrbracket_2 = 1$): In this case $f^1(x) = x - \llbracket a_j\rrbracket_1$ and $f^2(x') = x' - \llbracket a_j\rrbracket_2$. Upon execution of step 2, P_1 obtains x and P_2 obtains $y = x - \llbracket a_j\rrbracket_1$. Upon execution of step 3, P_2 obtains x' and P_1 obtains $y' = x' - \llbracket a_j\rrbracket_2$. P_1 accumulates $\llbracket z_j\rrbracket_1 = \llbracket a_j\rrbracket_1 - x + x' - \llbracket a_j\rrbracket_2$ into $\llbracket z\rrbracket_1$ and P_2 accumulates $\llbracket z_j\rrbracket_2 = \llbracket a_j\rrbracket_1 - x' + x - \llbracket a_j\rrbracket_1$ into $\llbracket z\rrbracket_2$. Then $\llbracket z_j\rrbracket_1 + \llbracket z_j\rrbracket_2 = 0$, which is $w_j a_j$ as $w_j = 0$.

Case II ($w_j = 0$; $\llbracket w_j\rrbracket_1 = 0$, $\llbracket w_j\rrbracket_2 = 0$): In this case $f^1(x) = x + \llbracket a_j\rrbracket_1$ and $f^2(x') = x' + \llbracket a_j\rrbracket_2$. Upon execution of step 2, P_1 obtains x and P_2 obtains $y = x$. Upon execution of step 3, P_2 obtains x' and P_1 obtains $y' = x'$ P_1 accumulates $\llbracket z_j\rrbracket_1 = -x + x'$ into $\llbracket z\rrbracket_1$ and P_2 accumulates $\llbracket z_j\rrbracket_2 = -x' + x$ into $\llbracket z\rrbracket_2$. Then $\llbracket z_j\rrbracket_1 + \llbracket z_j\rrbracket_2 = 0$, which is $w_j a_j$ as $w_j = 0$.

Case III ($w_j = 1$; $\llbracket w_j\rrbracket_1 = 0$, $\llbracket w_j\rrbracket_2 = 1$): In this case $f^1(x) = x + \llbracket a_j\rrbracket_1$ and $f^2(x') = x' - \llbracket a_j\rrbracket_2$. Upon execution of step 2, P_1 obtains x and P_2 obtains $y = x + \llbracket a_j\rrbracket_1$. Upon execution of step 3, P_2 obtains x', P_1 obtains $y' = x'$. P_1 accumulates $\llbracket z_j\rrbracket_1 = -x + x'$ into $\llbracket z\rrbracket_1$ and P_2 accumulates $\llbracket z_j\rrbracket_2 = \llbracket a_j\rrbracket_2 - x' + x + \llbracket a_j\rrbracket_1$ into $\llbracket z\rrbracket_2$. Then $\llbracket z_j\rrbracket_1 + \llbracket z_j\rrbracket_2 = a_j$, which is $w_j a_j$ as $w_j = 1$.

Case IV ($w_j = 1$; $\llbracket w_j\rrbracket_1 = 1$, $\llbracket w_j\rrbracket_2 = 0$): In this case $f^1(x) = x - \llbracket a_j\rrbracket_1$ and $f^2(x') = x' + \llbracket a_j\rrbracket_2$. Upon execution of step 2, P_1 obtains x and P_2 obtains $y = x$. Upon execution of step 3, P_2 obtains x' and P_1 obtains $y' = x' + \llbracket a_j\rrbracket_2$. P_1 accumulates $\llbracket z_j\rrbracket_1 = \llbracket a_j\rrbracket_1 - x + x' + \llbracket a_j\rrbracket_2$ into $\llbracket z\rrbracket_1$, P_2 accumulates $\llbracket z_j\rrbracket_2 = -x' + x$ into $\llbracket z\rrbracket_2$. Then, $\llbracket z_j\rrbracket_1 + \llbracket z_j\rrbracket_2 = a_j$, which is $w_j a_j$ as $w_j = 1$. ◻