Client-Aided Privacy-Preserving Machine Learning

Peihan Miao[1], Xinyi Shi[1(✉)], Chao Wu[2], and Ruofan Xu[3]

[1] Brown University, Providence, USA
{peihan_miao,xinyi_shi}@brown.edu
[2] University of California, Riverside, USA
chao.wu@email.ucr.edu
[3] University of Illinois Urbana-Champaign, Urbana, USA
ruofan4@illinois.edu

Abstract. Privacy-preserving machine learning (PPML) enables multiple distrusting parties to jointly train ML models on their private data without revealing any information beyond the final trained models. In this work, we study the client-aided two-server setting where two non-colluding servers jointly train an ML model on the data held by a large number of clients. By involving the clients in the training process, we develop efficient protocols for training algorithms including linear regression, logistic regression, and neural networks. In particular, we introduce novel approaches to securely computing inner product, sign check, activation functions (e.g., ReLU, logistic function), and division on secret shared values, leveraging lightweight computation on the client side. We present constructions that are secure against semi-honest clients and further enhance them to achieve security against malicious clients. We believe these new client-aided techniques may be of independent interest.

We implement our protocols and compare them with the two-server PPML protocols presented in SecureML (Mohassel and Zhang, S&P'17) across various settings and ABY2.0 (Patra et al., Usenix Security'21) theoretically. We demonstrate that with the assistance of untrusted clients in the training process, we can significantly improve both the communication and computational efficiency by orders of magnitude. Our protocols compare favorably in all the training algorithms on both LAN and WAN networks.

Keywords: Privacy-Preserving Machine Learning · Secure Multi-Party Computation · Client-Aided Protocols

1 Introduction

In recent years, we have witnessed machine learning (ML) emerge as one of the most influential technologies and rapidly expanding research domains. Its applications span a diverse spectrum, ranging from recommendation systems to self-driving cars, large language models, and even medical prediction and diagnosis. This is in part due to increasing amount of data being collected and available

C. Galdi and D. H. Phan (Eds.): SCN 2024, LNCS 14973, pp. 207–229, 2024.
https://doi.org/10.1007/978-3-031-71070-4_10

in the Big Data era. Meanwhile, as these machine learning algorithms and applications are deployed in various real-world scenarios, data privacy is becoming increasingly critical, especially in domains dealing with sensitive or confidential data such as healthcare, finance, and government. In cases where entities are hesitant or restricted from sharing their data due to privacy regulations, the significance of protecting data privacy is further emphasized.

Addressing these concerns, privacy-preserving machine learning (PPML) has become a crucial approach to training ML models in a distributed manner, which enables multiple distrusting parties to collaboratively train ML models on their private data while maintaining data privacy. The most commonly considered setting in PPML, as proposed by Mohassel and Zhang [28], involves data owners (e.g., clients) secret sharing their data among two non-colluding parties (e.g., servers), who then jointly perform training on the secret-shared data.

At a high level, this approach can be conceptualized as two servers engaging in secure two-party computation to train the ML model on secret-shared data. Importantly, the servers learn nothing beyond the final trained model, ensuring the privacy of individual data points. Nevertheless, prior work [15,20,26,28,29,31,32] has overlooked the fact that the data was initially owned by the clients in the clear. In this work, we show that actively involving clients in the training process can yield significant improvements in both communication and computational efficiency of the overall protocol.

1.1 Our Contributions

We study two-server PPML training where the data is held by a large number of clients. Since the clients initially hold the training data in the clear, they can assist in certain computations based on their clear data to achieve better efficiency than computing on shared data. Additionally, we can leverage techniques from secure two-party computation with the assistance of an untrusted third party by treating the clients as the untrusted third party. This approach introduces a novel way of computing activation functions as well as division in the training algorithms, which proves to be much more efficient than the garbled circuit-based approaches commonly used in PPML. We believe these client-aided techniques may be of independent interest.

Our Contributions. In this work, we

- develop a new *client-aided inner product* protocol that enables a client and two servers to jointly compute the inner product of two private vectors $\langle \mathbf{x}, \mathbf{y} \rangle$, where $\mathbf{x}$ is secret shared among the two servers and $\mathbf{y}$ is held by the client;
- develop a series of *client-aided* protocols that, with the assistance of an untrusted client, allow two servers
  - to determine if their secret shared value is positive or not,
  - to compute activation functions (e.g., ReLU, logistic function) on their secret shared value, and
  - to compute divisions on their secret shared values (for softmax);

– put these techniques all together into PPML training protocols for linear regression, logistic regression, and neural networks, which are secure against *semi-honest* servers and clients;
– present techniques to enhance our security guarantees to protect against *malicious clients*;
– implement our protocols and demonstrate performance improvement compared with prior work.

Experimental Results. We implement our two-server PPML protocols for both semi-honest and malicious clients. We compare our performance with SecureML [28] in various settings and compare with the state-of-the-art ABY2.0 [29] theoretically (their code is not available). For linear regression, we achieve an improvement of 6.12−1047× over [28] in the LAN setting and 3.63−73.5× in the WAN setting. For logistic regression, we achieve an improvement of 4.85−723× on LAN and 2.71−44.3× on WAN. For neural networks, we achieve an improvement of 3.19× on LAN and 3.92× on WAN. When enhancing our security guarantees to malicious clients, we incur a small constant (2.55−4.95×) overhead compared to our semi-honest variant. This is orders of magnitude more efficient than the OT- and LHE-based variants of [28]. We also give comprehensive comparisons for the communication costs as well as the offline/online efficiency. See Sect. 5 for more details.

1.2 Related Work

Privacy-Preserving Machine Learning. In the PPML domain, secure multiparty computation has been used for various ML algorithms such as decision trees [23], k-means clustering [5,14], and SVM classification [34,38]. However, these solutions are far from practical due to the high overheads that they incur. Mohassel and Zhang [28] introduced a practically-efficient PPML framework in the two-server setting. Since then, there has been a rich body of research in PPML that follows the same framework: data owners first secret share their data among two or more non-colluding parties who then perform training on the secret-shared data. Prior work has studied this problem in various settings, including secure training and inference, semi-honest and malicious security, with a focus on a small number of servers (e.g., two-server [15,20,26,28,29,31,32], three-server [8,19,20,27,30,35], and four-server [6,9,19]) where the adversary can corrupt at most one of them. In this work, we focus on the two-server setting for ML training, and we anticipate that the client-aided techniques developed here can be applied to ML inference.

Federated Learning. As a similar setting of PPML, federated learning (FL) [3,4,10,16,18,24] enables multiple entities (e.g., mobile devices) to collaboratively train a model under the coordination of a central server (e.g., service provider) while keeping the training data decentralized, protecting the privacy of the individual users. The two-server setting has also been studied in FL [1,10].

Most of the existing FL frameworks rely on a key building block known as *secure aggregation* [1,3,4,10], which protects clients' raw data (in particular, individual model updates) through secure aggregation. However, they reveal the global model updates, particularly the mini-batch stochastic gradient descent, to the central server(s) as well as all the clients. Recent work has shown that this framework is vulnerable to various privacy attacks [12,25,33,36,39]. As a side product of this work, we can apply client-aided PPML to two-server FL to enhance the privacy guarantee of FL, revealing only the final model to the central servers.

1.3 Roadmap

We give a high-level overview of our new techniques in Sect. 2. We provide preliminaries including the definitions of required cryptographic building blocks and machine learning algorithms in Sect. 3. In Sect. 4, we present our new client-aided protocols for inner product, sign check, activation functions, and division over secret shared values. These building blocks can be put all together into client-aided PPML protocols that are secure against semi-honest clients, which we defer to the full version of this paper. Additionally, we enhance the security guarantees to protect against malicious clients in the full version of the paper. We discuss our performance and experimental results in Sect. 5.

2 Technical Overview

During the training process, we keep the invariant that all the intermediate values (e.g., model parameters, clients' data, etc.) are additively secret shared among the two servers. Their secret shares are only revealed to each other when the training process is finished and they would like to learn the final model. We discuss how to maintain the invariant for each type of operation in the training algorithms. First, addition is almost free, which can be computed locally by the servers. We discuss below how to deal with the operations that require more work, and how we can improve the efficiency by involving an untrusted client in the computation. In the full version of this paper, we present protocols for ML algorithms that are built on these building blocks.

Client-Aided Inner Product. One of the key steps in linear regression is to compute the inner product of two vectors, one vector $\mathbf{w}$ denoting the current model, and the other vector $\mathbf{x}$ denoting the client's data. In the existing PPML framework, the servers hold secret shares of both $\mathbf{w}$ and $\mathbf{x}$ and they perform a secure two-party computation protocol to compute secret shares of the inner product $\langle \mathbf{w}, \mathbf{x} \rangle$, e.g., by using Beaver multiplication triples [2] generated in an offline setup phase [28].

We notice that since $\mathbf{x}$ is entirely known to the client, she can compute a masked inner product with $\mathbf{x}$ and share the masking information with the two servers. This improves both the computation and communication between the servers. Moreover, it does not require heavy computation on the client side, nor

does it require extra round of communication between the servers and the client. In particular, the client still sends secret shares of $\mathbf{x}$ to the servers, along with which she will send some extra masking values. We present the detailed protocol in Sect. 4.1.

When the vectors have dimension 1 (as a special case), this technique can be used to compute multiplication of a value shared among the servers with another value held by the client. This will be a key building block below.

Client-Aided Activation Functions. For logistic regression and neural networks, besides vector inner product (and more generally matrix multiplication), we also need to perform activation functions (e.g., logistic function, ReLU) on secret shared values. To do this, we need a way for the two servers to jointly determine whether a secret shared value is positive or not (we view the value as a two's complement representation). This is not an arithmetic operation, and the existing PPML frameworks [27–29] mainly rely on garbled circuits that compute the sum of two secret shared values to determine its highest order bit.

In this work, we present a new approach that utilizes a client as an untrusted third party. For two secret shares $[\![x]\!]_0$ and $[\![x]\!]_1$, the problem of determining if $[\![x]\!]_0 + [\![x]\!]_1 > 0$ is essentially a secure comparison problem, namely determining whether $[\![x]\!]_0 > -[\![x]\!]_1$. Instead of relying on garbled circuits [37], we reduce this problem to a special secure two-party computation problem, private set intersection cardinality (PSI-CA), via a certain encoding of the input values. In particular, each party generates a set of elements based on their input and they jointly compute the cardinality of the intersection of the two sets. $[\![x]\!]_0 > -[\![x]\!]_1$ iff the set intersection cardinality is 1, and $[\![x]\!]_0 \leq -[\![x]\!]_1$ iff the set intersection cardinality is 0. With the assistance of an untrusted third party (i.e., untrusted client), PSI-CA can be securely computed in an extremely efficient way requiring only symmetric-key cryptographic operations.

There are two issues in this approach. First, the existing client-aided PSI-CA protocols reveal the cardinality of the set intersection to either the client or one of the two servers. However, it is crucial that the result is never revealed to any party in our PPML protocols. We develop a new way to secret share the cardinality result between the client and the two servers. Another issue is that the reduction above only works if values are both positive or both negative. We observe that $[\![x]\!]_0$ and $[\![x]\!]_1$ have different signs with high probability throughout the training process, hence we can ensure the comparison is only between values of the same sign in our protocol.

To compute ReLU, we need to multiply the secret shared PSI-CA result with the secret shared value x. We can utilize the aforementioned client-aided inner product (with dimension 1) to efficiently compute the multiplication. Putting it all together, we present the client-aided ReLU protocol in Sect. 4.2. We further extend these ideas to the logistic function, see the full version of this paper for details.

Client-Aided Division. In neural network training, we additionally need to compute a softmax function on secret shared values. We use the MPC-friendly

variant of it (see Sect. 3.2) which requires division of two secret shared values. We compute the quotient bit-by-bit sequentially starting from the most significant bit. In every step, we need to compare the current dividend with the divisor, which can be done using the client-aided sign check protocol described above. The secret shared output needs to be multiplied with the secret shared divisor, which can be done using the client-aided inner product with dimension 1. The protocol is presented in Sect. 4.4.

Security Against Malicious Clients. In the aforementioned client-aided protocols, it is critical that the clients are semi-honest, namely they follow the protocol description honestly while trying to extract more information from the protocol execution. This might not be a realistic assumption in practice. Hence we further enhance the security guarantees of our protocol to protect against malicious clients. The two main building blocks we need is the client-aided inner product and client-aided sign check. To ensure security against malicious clients, we leverage the cut-and-choose technique to verify that the results are computed correctly. See the full version of this paper for details.

As it turns out, our malicious variant only incurs a small constant overhead compared to our semi-honest variant and is orders of magnitude more efficient than prior work (see the full version of this paper).

3 Preliminaries

Notation. We use λ, σ to denote the computational and statistical security parameters, respectively. We use $[\![v]\!]$ to denote an additive secret sharing of a value $v \in \mathbb{Z}_{2^\ell}$ between two servers $\mathsf{S}_0, \mathsf{S}_1$. In particular, server S_i $(i \in \{0,1\})$ holds $[\![v]\!]_i$ such that $v = [\![v]\!]_0 + [\![v]\!]_1$. To sample a random additive secret sharing of v, we use the notation $([\![v]\!]_0, [\![v]\!]_1) \leftarrow \mathsf{Sharing}(v)$. We use $\xleftarrow{\$}$ to denote random sampling from a uniform distribution. We use $[n]$ to denote the set $\{1, 2, \ldots, n\}$. For a vector $\mathbf{v}$, we use $\mathbf{v}[i]$ to denote the i-th element of the vector. By $\mathsf{negl}(\lambda)$ we denote a negligible function, i.e., a function f such that $f(\lambda) < 1/p(\lambda)$ holds for any polynomial $p(\lambda)$ and sufficiently large λ.

Fixed-Point Arithmetic. Throughout our protocols, we follow the prior work [28,29] to use the two's complement fixed-point representation to denote real numbers and keep at most ℓ_f bits in the fractional part for all intermediate values during the training process. In particular, we transform a real number x (with at most ℓ_f bits in its fractional part) into an integer in $\mathbb{Z}_{2^\ell}$ by computing $x' = 2^{\ell_f} \cdot x$. Furthermore, we assume that all intermediate values have at most ℓ_w bits in the whole number part and that $\ell_w + \ell_f \ll \ell$ (this follows from prior work [28,29]). To multiply two real numbers x and y, we multiply $x' = 2^{\ell_f} \cdot x$ with $y' = 2^{\ell_f} \cdot y$ to obtain $z' = x' \cdot y' \in \mathbb{Z}_{2^\ell}$. Note that z' has $2 \cdot \ell_f$ bits representing the fractional part of the product, so we truncate the least significant ℓ_f bits of z' such that it has ℓ_f bits in the fractional part. Since we keep the invariant that all intermediate values are additively secret shared between the two servers and

that $\ell_w + \ell_f \ll \ell$, we can truncate z' by truncating its shares $[\![z']\!]_0$ and $[\![z']\!]_1$ locally on the two servers [28].

We use the function $\mathsf{Rtol}(x)$ to denote the function of transforming a real number x to an integer in $\mathbb{Z}_{2^\ell}$, namely $\mathsf{Rtol}(x) = 2^{\ell_f} \cdot x$. We use the function $\mathsf{Trunc}(x')$ to denote the function of truncating an integer in $\mathbb{Z}_{2^\ell}$ by the lowest order ℓ_f bits, namely $\mathsf{Trunc}(x') = \lfloor x/2^{\ell_f} \rfloor$. When we compare $x \in \mathbb{Z}_{2^\ell}$ with 0, we view x as a two's complement representation and compare it with 0. When we divide x by y, which are both positive real numbers represented in $\mathbb{Z}_{2^\ell}$, we compute the quotient by $\mathsf{Quotient}(x, y) := \lfloor x \cdot 2^{\ell_f}/y \rfloor \in \mathbb{Z}_{2^\ell}$.

3.1 Specialized Two-Party Computation

Secure multi-party computation (MPC) [13,37] allows multiple parties, each holding a private input, to jointly compute a function on their private inputs without revealing anything beyond the output of the function. In this work, we consider MPC protocols for three parties with honest majority. In particular, the three parties are two servers and a client, where the adversary corrupts either the client or one of the two servers. We say an adversary is *semi-honest* if it follows the protocol description honestly while trying to extract more information from it, while a *malicious* adversary may arbitrarily deviate from the protocol specification. In our work, we assume both servers are semi-honest, and we consider both semi-honest and malicious clients. We follow the Universal Composition (UC) security definition of MPC, and refer the reader to [7] for details.

3.2 Machine Learning Algorithms

We consider a set of training data $\{\mathbf{x}_i, y_i\}_{i=1,\ldots,n}$. All the algorithms take the stochastic gradient descent (SGD) approach, which involves iteratively updating a target coefficient vector/matrix by following the gradient of a particular loss function evaluated on a random batch of training data. In the SGD method, we use B to denote the batch size, α to denote the learning rate, E to denote the number of epochs, n to denote the size of training data, and define $t = \frac{E \cdot n}{B}$ as the number of iterations. We refer the reader to prior work [28,29] on more details about these ML models.

Linear Regression. In linear regression, we try to learn a coefficient vector $\mathbf{w}$ such that the following loss function is minimized: $\sum_{i=1}^{n} (\langle \mathbf{w}, \mathbf{x}_i \rangle - y_i)^2$. Applying SGD to the linear loss function gives that we update $\mathbf{w}$ in each iteration according to the following expression:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{B} \sum_{i=1}^{B} (\langle \mathbf{w}, \mathbf{x}_i \rangle - y_i) \cdot \mathbf{x}_i. \tag{1}$$

Logistic Regression. The only difference between logistic regression and linear regression is that the logistic (Sigmoid) function $f(z) = \frac{1}{1+e^{-z}}$ is applied to the inner product $z = \langle \mathbf{w}, \mathbf{x}_i \rangle$, and the loss function needs to be adjusted accordingly so that the loss function is convex and SGD still works. The SGD update step in this case is identical to linear regression except for applying the logistic function to the inner product. In particular,

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{B} \sum_{i=1}^{B} (f(\langle \mathbf{w}, \mathbf{x}_i \rangle) - y_i) \cdot \mathbf{x}_i. \tag{2}$$

The above logistic function is not MPC-friendly, and we follow the approach of [28] by considering a piecewise linear function instead, which they demonstrated yields comparable accuracy in training. We refer the reader there for more details. In particular, we approximate the logistic function by

$$f(z) = \begin{cases} 0 & \text{if } z < -1/2 \\ z + 1/2 & \text{if } z \in [-1/2, 1/2] \\ 1 & \text{if } z > 1/2 \end{cases}. \tag{3}$$

Neural Networks. Neural networks are a generalization of regression to learn more complex relationships between high dimensional input and output data. A basic neural network can be divided into m layers, each containing d_i nodes. Each node is a linear function composed with a non-linear *activation function*. One of the most popular activation functions considered in neural networks is the rectified linear unit (ReLU) function, which can be expressed as $f(x) = \max\{0, x\}$. To evaluate a neural network, the nodes at the first layer are evaluated on the input features. The outputs of these nodes are then forwarded as inputs to the next layer of the network until all layers have been evaluated in this manner. For classification problems with multiple classes, usually a *softmax* function is applied at the output layer, and we use the MPC-friendly variant [28] of the softmax function $f(u_i) = \frac{\text{ReLU}(u_i)}{\sum_{k=1}^{d_m} \text{ReLU}(u_k)}$. The training of neural networks is performed using SGD in a similar manner to logistic regression except that each layer of the network should be updated in a recursive manner, starting at the output layer and working backward.

4 Client-Aided Protocols

4.1 Client-Aided Inner Product

In this section, we present a protocol for computing the inner product of a vector $\mathbf{x} \in \mathbb{Z}_{2^\ell}^d$ that is additively secret shared among two servers and another vector $\mathbf{y} \in \mathbb{Z}_{2^\ell}^d$ held by a client. As a result, the two servers learn an additive secret sharing of the inner product $\langle \mathbf{x}, \mathbf{y} \rangle$ and the client learns nothing. The ideal functionality for our client-aided inner product is presented in Fig. 1. Looking ahead, whenever we run this protocol, the vector $\mathbf{y}$ will also be shared among the

servers (either directly shared by the client or learned from another protocol), hence in the ideal functionality we also let the servers input a secret share of $\mathbf{y}$. This will make this protocol better compile with our other protocols, especially in the case of malicious clients.

Functionality $\mathcal{F}^d_{\text{INNERPROD}}$:
Parties: Two servers $\mathsf{S}_0, \mathsf{S}_1$ and a client C.
Inputs: Each server S_i ($i \in \{0,1\}$) inputs two secret shared vectors $[\![\mathbf{x}]\!]_i, [\![\mathbf{y}]\!]_i \in \mathbb{Z}_{2^\ell}^d$. The client C inputs a vector $\mathbf{y} = [\![\mathbf{y}]\!]_0 + [\![\mathbf{y}]\!]_1$.
Functionality: On receiving $[\![\mathbf{x}]\!]_i, [\![\mathbf{y}]\!]_i$ from S_i ($i \in \{0,1\}$) and $\mathbf{y}$ from C:
- Recover $\mathbf{x} = [\![\mathbf{x}]\!]_0 + [\![\mathbf{x}]\!]_1$ and compute the inner product $v = \langle \mathbf{x}, \mathbf{y} \rangle$.
- Sample a random secret sharing of v, namely $([\![v]\!]_0, [\![v]\!]_1) \leftarrow \mathsf{Sharing}(v)$.
- Send $[\![v]\!]_i$ to each server S_i ($i \in \{0,1\}$) and send $\perp$ to the client C.

Fig. 1. Ideal functionality $\mathcal{F}^d_{\text{INNERPROD}}$ for computing the inner product.

Construction Overview. The client first samples a uniform random vector $\mathbf{r} \xleftarrow{\$} \mathbb{Z}_{2^\ell}^d$. Viewing $\mathbf{r}$ as a mask for the servers' input $\mathbf{x}$, the client generates a data-dependent multiplication triple by computing the inner product of $\mathbf{r}$ and its input vector $\mathbf{y}$, and sends the a secret share of the triple to the two servers. By using the data-dependent triple generated by the client, the two servers recover $\mathbf{x} - \mathbf{r}$ and compute a secret share of $\langle \mathbf{x}, \mathbf{y} \rangle$. Our protocol is described in Fig. 2. We state the theorem below and give the security proof in the full version of this paper.

Protocol $\Pi^d_{\text{INNERPROD}}$:
1. The client C does the following:
   (a) Sample $[\![\mathbf{r}]\!]_0, [\![\mathbf{r}]\!]_1 \xleftarrow{\$} \mathbb{Z}_{2^\ell}^d$ and compute $\mathbf{r} = [\![\mathbf{r}]\!]_0 + [\![\mathbf{r}]\!]_1$.
   (b) Compute $u = \langle \mathbf{r}, \mathbf{y} \rangle$.
   (c) Sample random secret sharings of u, namely $([\![u]\!]_0, [\![u]\!]_1) \leftarrow \mathsf{Sharing}(u)$.
   (d) Send $([\![\mathbf{r}]\!]_i, [\![u]\!]_i)$ to server S_i for $i \in \{0,1\}$.
2. Each server S_i ($i \in \{0,1\}$) does the following:
   (a) Compute $[\![\mathbf{s}]\!]_i = [\![\mathbf{x}]\!]_i - [\![\mathbf{r}]\!]_i$ and send it to the other server.
   (b) Recover $\mathbf{s} = [\![\mathbf{s}]\!]_0 + [\![\mathbf{s}]\!]_1$ and compute $[\![v]\!]_i = \langle \mathbf{s}, [\![\mathbf{y}]\!]_i \rangle + [\![u]\!]_i$ as the output.

Fig. 2. Protocol $\Pi^d_{\text{INNERPROD}}$ for computing the inner product.

Theorem 1. *The protocol* $\Pi^d_{\text{INNERPROD}}$ *(Fig. 2) securely computes the ideal functionality* $\mathcal{F}^d_{\text{INNERPROD}}$ *(Fig. 1) against a semi-honest adversary that corrupts either the client* C *or one of the two servers.*

Communication and Optimizations. In our protocol, each party computes only one inner product, so the servers and the client compute three inner products in total. The communication between the client and two servers is $(2d+2)$ ring elements in $\mathbb{Z}_{2^\ell}$ and the communication among the two servers is $2d$ ring elements. The total communication is $(4d+2)$ ring elements.

We discuss some optimizations in our implementation. In Steps 1a, 1c, 1d, the client needs to sample random vectors $[\![\mathbf{r}]\!]_0, [\![\mathbf{r}]\!]_1$ as well as random secret shares of u, and send them to the servers, leading to a total communication cost of $(2d+2)$ ring elements in $\mathbb{Z}_{2^\ell}$. To reduce this communication, we let each server share a PRF key with the client. Then the client can use the PRF keys to generate (pseudo)random vectors $[\![\mathbf{r}]\!]_0, [\![\mathbf{r}]\!]_1$ for the two servers without communication. To generate a (pseudo)random secret sharing of u, the client can use the shared PRF key with one server S_0 to generate $[\![u]\!]_0$ without communication, and send the other share $[\![u]\!]_1$ to S_1. That is, apart from sharing the PRF keys, we can reduce the communication between the client and the servers from $(2d+2)$ to 1 ring element.

4.2 Client-Aided Sign Check

In this section, we present a protocol that allows two servers to jointly learn if a secret shared value $x \in \mathbb{Z}_{2^\ell}$ is positive or not (by viewing x as a two's complement representation), with the assistance of a client. The three parties will learn a binary secret sharing of the sign check outcome b. In particular, the servers both learn one binary share b^{S} and the client learns the other share b^{C} such that $b^{\mathsf{S}} \oplus b^{\mathsf{C}} = b$. The ideal functionality is presented in Fig. 3. Looking ahead, this protocol will be used in computing activation functions as well as divisions (for softmax). In our learning algorithms, we note that the absolute value of x is significantly less than 2^ℓ throughout the training process, hence $[\![x]\!]_0$ and $[\![x]\!]_1$ have opposite signs with overwhelming probability. In particular, we assume x has at most ℓ_f bits in the fractional part and ℓ_w bits in the whole number part, and that $\ell_w + \ell_f \ll \ell$ (this follows from prior work [27–29]). Given that $([\![x]\!]_0, [\![x]\!]_1)$ is a uniformly random share of $x \in \mathbb{Z}_{2^\ell}$, the probability that $[\![x]\!]_0$ and $[\![x]\!]_1$ have the same sign is no greater than $2^{\ell_w+\ell_f-\ell}$. The proof follows from the analysis in [28]. Therefore, we assume $[\![x]\!]_0$ and $[\![x]\!]_1$ in the ideal functionality. In addition, we let the two servers learn a secret share of b^{C} so that this protocol can be incorporated more easily into other protocols.

Functionality $\mathcal{F}_{\text{SignCheck}}$:

Parties: Two servers $\mathsf{S}_0, \mathsf{S}_1$ and a client C.

Inputs: Each server S_i $(i \in \{0,1\})$ inputs a secret shared value $[\![x]\!]_i \in \mathbb{Z}_{2^\ell}$, where $[\![x]\!]_0$ and $[\![x]\!]_1$ have opposite signs. The client C has no input.

Functionality: On receiving $[\![x]\!]_i$ from S_i $(i \in \{0,1\})$:

- Recover $x = [\![x]\!]_0 + [\![x]\!]_1$ and let $b := (x > 0)$. That is, view x as a two's complement representation and let b be the indicator of weather x is a positive number.
- Sample $b^{\mathsf{S}} \xleftarrow{\$} \{0,1\}$ and let $b^{\mathsf{C}} := b \oplus b^{\mathsf{S}}$.
- Sample a random secret sharing of b^{C}, namely $([\![b^{\mathsf{C}}]\!]_0, [\![b^{\mathsf{C}}]\!]_1) \leftarrow \mathsf{Sharing}(b^{\mathsf{C}})$.
- Send b^{S} to both servers and $[\![b^{\mathsf{C}}]\!]_i$ to each server S_i $(i \in \{0,1\})$. Send b^{C} to the client C.

Fig. 3. Ideal functionality $\mathcal{F}_{\text{SignCheck}}$ for determining if a secret shared value is positive or not.

Construction Overview. We first give an overview of our construction. The two servers hold an additive secret share of a value $x \in \mathbb{Z}_{2^\ell}$, namely each server S_i ($i \in \{0,1\}$) holds $[\![x]\!]_i$ such that $[\![x]\!]_0 + [\![x]\!]_1 = x$. We additionally assume that $[\![x]\!]_0$ and $[\![x]\!]_1$ have opposite signs. Suppose without loss of generality that $[\![x]\!]_0 \geq 0$ and $[\![x]\!]_1 < 0$, then checking whether $[\![x]\!]_0 + [\![x]\!]_1 > 0$ is equivalent to checking whether $[\![x]\!]_0 > -[\![x]\!]_1$, where both $[\![x]\!]_0$ and $-[\![x]\!]_1$ are non-negative values. We take inspiration from [22] to reduce our problem to PSI and then leverage techniques from client-aided PSI.

Let $a = \overline{a_\ell \cdots a_1}$ denote the binary representation of a non-negative value a. We denote its *0-encoding* by $\mathbf{P}_a^0 = \{\overline{a_\ell \cdots a_{i+1} 10 \cdots 0} | i \in [\ell], a_i = 0\}$ and its *1-encoding* by $\mathbf{P}_a^1 = \{\overline{a_\ell \cdots a_i 0 \cdots 0} | i \in [\ell], a_i = 1\}$. Note that all binary strings in the sets have the same length ℓ. We then pad the two sets with dummy elements to be of size ℓ each. Define two sets $\mathbf{G}_a^0$ and $\mathbf{G}_a^1$ as follows. $\mathbf{G}_a^0$ is a set of size $\ell - |\mathbf{P}_a^0|$ where all the elements are random ℓ-bit strings starting with 10, and $\mathbf{G}_a^1$ is a set of size $\ell - |\mathbf{P}_a^1|$ where all the elements are random ℓ-bit strings starting with 11. Let the corresponding *augmented 0-encoding* be defined as $\mathbf{A}_a^0 = \mathbf{P}_a^0 \cup \mathbf{G}_a^0$ and *augmented 1-encoding* be $\mathbf{A}_a^1 = \mathbf{P}_a^1 \cup \mathbf{G}_a^1$. Following the work [22], the set intersection $\mathbf{A}^1_{[\![x]\!]_0} \cap \mathbf{A}^0_{-[\![x]\!]_1}$ has size 1 if and only if $[\![x]\!]_0 > -[\![x]\!]_1$ and the intersection is empty otherwise. For the other case where $[\![x]\!]_0 < 0$ and $[\![x]\!]_1 \geq 0$, we simply swap the tasks of two parties and check whether $-[\![x]\!]_0 < [\![x]\!]_1$.

Now we reduce our problem to computing the size of the intersection of two private sets, namely PSI-CA. With the assistance of an untrusted client, we can utilize techniques from client-aided PSI-CA [17,21]. Nevertheless, we need an additional security guarantee that the servers and the client only learn a binary secret share of the PSI-CA result.

We leverage the fact that the output of our PSI-CA can only be 0 or 1, and we randomly choose to compare either $[\![x]\!]_0 > -[\![x]\!]_1$ or $[\![x]\!]_0 < -[\![x]\!]_1$. In particular, the servers randomly sample a bit b^{S} and flip the comparison if $b^{\mathsf{S}} = 1$. To be more specific, S_0 generates an augmented $(1-b^{\mathsf{S}})$-encoding of $[\![x]\!]_0$ and S_1 generates an augmented b^{S}-encoding of $-[\![x]\!]_1$. Then they perform a client-aided PSI-CA protocol using a pseudorandom function (PRF). The client-aided sign check protocol is presented in Fig. 4. We state the theorem below and give the security proof in the full version of this paper.

Protocol $\Pi_{\text{SignCheck}}$:

0. The two servers S_0 and S_1 agree on a computational security parameter λ and a pseudorandom function $F : \{0,1\}^\lambda \times \{0,1\}^\ell \to \{0,1\}^\lambda$.
1. S_0 samples a random bit $b^{\mathsf{S}} \xleftarrow{\$} \{0,1\}$ and random PRF key $k \xleftarrow{\$} \{0,1\}^\lambda$, and sends (b^{S}, k) to S_1.
2. Each server S_i $(i \in \{0,1\})$ does the following:
   (a) View $[\![x]\!]_i$ as a two's complement representation. If $[\![x]\!]_i \geq 0$, then let $b_i := 1 \oplus b^{\mathsf{S}}$; otherwise let $[\![x]\!]_i := -[\![x]\!]_i$ and let $b_i := b^{\mathsf{S}}$.
   (b) Generate an augmented b_i-encoding of $[\![x]\!]_i$ as $\mathbf{A}_i$.
   (c) Apply the PRF F_k to each element in $\mathbf{A}_i$ to obtain $\mathcal{T}_i = F_k(\mathbf{A}_i)$.
   (d) Randomly shuffle the elements in $\mathcal{T}_i$ and send the shuffled set $\widetilde{\mathcal{T}}_i$ to the client C.
3. Upon receiving $\widetilde{\mathcal{T}}_0$ and $\widetilde{\mathcal{T}}_1$ from the two servers, the client C sets $b^{\mathsf{C}} = 0$ if $\widetilde{\mathcal{T}}_0 \cap \widetilde{\mathcal{T}}_1 = \emptyset$, and sets $b^{\mathsf{C}} = 1$ otherwise.
4. The client C samples a random secret sharing of b^{C}, namely $\left([\![b^{\mathsf{C}}]\!]_0, [\![b^{\mathsf{C}}]\!]_1\right) \leftarrow \mathsf{Sharing}\left(b^{\mathsf{C}}\right)$, and sends $[\![b^{\mathsf{C}}]\!]_i$ to each server S_i $(i \in \{0,1\})$.
5. Each server S_i $(i \in \{0,1\})$ outputs $(b^{\mathsf{S}}, [\![b^{\mathsf{C}}]\!]_i)$. The client C outputs b^{C}.

Fig. 4. Protocol $\Pi_{\text{SignCheck}}$ for determining if a secret shared value is positive or not.

Theorem 2. *Assuming F is a secure PRF, the protocol $\Pi_{\text{SignCheck}}$ (Fig. 4) securely computes the ideal functionality $\mathcal{F}_{\text{SignCheck}}$ (Fig. 3) against a semi-honest adversary that corrupts either the client C or one of the two servers.*

Communication and Optimizations. In our protocol, each server computes ℓ PRF operations. The total communication cost is $(2\lambda\ell + \lambda + 1)$ bits with 2 ring elements. We can apply the same optimization as in Sect. 4.1 to reduce communication by using shared PRF keys to generate random values. In particular, in Step 1 the servers can use a shared PRF key to generate (pseudo)random values (b^{S}, k) together without communication; in Step 4 the client C can use the shared PRF key to generate a (pseudo)random value with one server without communication. Then, the communication can be reduced to $2\lambda\ell$ bits with 1 ring elements.

4.3 Client-Aided ReLU

In this section, we present a protocol that allows two servers to jointly compute the ReLU function of an integer $x \in \mathbb{Z}_{2^\ell}$ that is additively secret shared among them, with the assistance of the client. Looking ahead, this protocol is a crucial component in computing activation functions. The ideal functionality for our client-aided ReLU is presented in Fig. 5.

Functionality $\mathcal{F}_{\text{ReLU}}$:

Parties: Two servers $\mathsf{S}_0, \mathsf{S}_1$ and a client C.
Inputs: Each server S_i $(i \in \{0,1\})$ inputs $[\![x]\!]_i \in \mathbb{Z}_{2^\ell}$. The client C inputs nothing.
Functionality: On receiving $[\![x]\!]_i$ from S_i $(i \in \{0,1\})$:

- Recover $x = [\![x]\!]_0 + [\![x]\!]_1$. View x as a two's complement representation and compute $y = \max\{x, 0\}$.
- Sample a random secret sharing of y, namely $([\![y]\!]_0, [\![y]\!]_1) \leftarrow \mathsf{Sharing}(y)$.
- Send $[\![y]\!]_i$ to each server S_i $(i \in \{0,1\})$ and send $\perp$ to the client C.

Fig. 5. Ideal functionality $\mathcal{F}_{\text{ReLU}}$ for computing the ReLU function.

Construction Overview. The servers hold a secret share of an integer $x \in \mathbb{Z}_{2^\ell}$ and want to jointly learn a secret share of $\text{ReLU}(x) = \max\{0, x\} = (x > 0) \cdot x$. Observe that the ReLU function simply consists of a sign check operation and a multiplication operation, which can be computed by using the protocols in Sects. 4.2 and 4.1, respectively. To combine these two protocols, the challenge is that the output of the sign check is a binary share among the servers and the client, while the input of the inner product should be additive secret shares. Observe that $x \cdot (b^{\mathsf{S}} \oplus b^{\mathsf{C}}) = x \cdot b^{\mathsf{S}} + (x \cdot b^{\mathsf{C}}) \cdot (1 - 2b^{\mathsf{S}})$ and the servers have b^{S} in clear, we only need to use the inner product protocol (with dimension 1) to compute a secret share of $x \cdot b^{\mathsf{C}}$, and then let each server S_i $(i \in \{0,1\})$ compute $[\![x \cdot (b^{\mathsf{S}} \oplus b^{\mathsf{C}})]\!]_i = [\![x]\!]_i \cdot b^{\mathsf{S}} + [\![x \cdot b^{\mathsf{C}}]\!]_i \cdot (1 - 2b^{\mathsf{S}})$. Our protocol is described in Fig. 6. We state the theorem below and defer the proof to the full version of this paper.

Protocol Π_{ReLU}:

1. $\mathsf{S}_0, \mathsf{S}_1$ and C call $\mathcal{F}_{\text{SignCheck}}$ to compute
$$\left(\left(b^{\mathsf{S}}, \left[\!\left[b^{\mathsf{C}}\right]\!\right]_0\right), \left(b^{\mathsf{S}}, \left[\!\left[b^{\mathsf{C}}\right]\!\right]_1\right), b^{\mathsf{C}}\right) \leftarrow \mathcal{F}_{\text{SignCheck}}([\![x]\!]_0, [\![x]\!]_1, \perp).$$
2. $\mathsf{S}_0, \mathsf{S}_1$ and C call $\mathcal{F}^1_{\text{InnerProd}}$ to compute
$$([\![\alpha]\!]_0, [\![\alpha]\!]_1, \perp) \leftarrow \mathcal{F}^1_{\text{InnerProd}}\left(\left([\![x]\!]_0, \left[\!\left[b^{\mathsf{C}}\right]\!\right]_0\right), \left([\![x]\!]_1, \left[\!\left[b^{\mathsf{C}}\right]\!\right]_1\right), b^{\mathsf{C}}\right).$$
3. Each server S_i $(i \in \{0,1\})$ outputs $[\![y]\!]_i = [\![x]\!]_i \cdot b^{\mathsf{S}} + [\![\alpha]\!]_i \cdot (1 - 2b^{\mathsf{S}})$.

Fig. 6. Protocol Π_{ReLU} for computing ReLU in the $(\mathcal{F}_{\text{SignCheck}}, \mathcal{F}^1_{\text{InnerProd}})$-hybrid model.

Theorem 3. *The protocol Π_{ReLU} (Fig. 6) securely computes the ideal functionality $\mathcal{F}_{\text{ReLU}}$ (Fig. 5) in the $(\mathcal{F}_{\text{SignCheck}}, \mathcal{F}^1_{\text{InnerProd}})$-hybrid model against a semi-honest adversary that corrupts either the client C or one of the two servers.*

Communication and Optimization. The communication of a ReLU function consists of the communication of a sign check and an inner product of vectors

with dimension 1. Applying the optimizations we mentioned, the communication between the client and the servers is $2\lambda\ell$ bits with 2 ring elements and the communication between the two servers is 2 ring elements. The computational cost on each server mainly contains ℓ PRF (AES) operations.

4.4 Client-Aided Division

In this section, we present a client-aided protocol that computes division of two shared values. Assume the servers hold secret shares of $x, y \in \mathbb{Z}_{2^\ell}$ such that $0 \leq x \leq y$ and $y \neq 0$, they jointly compute an additive secret share of the quotient $\mathsf{Quotient}(x, y) = \lfloor x \cdot 2^{\ell_f} / y \rfloor$ with the assistance of the client. See Fig. 7 for the ideal functionality. Looking ahead, the division protocol is used to approximate the softmax function in the output layer of neural networks.

Functionality $\mathcal{F}_{\text{DIV}}$:

Parties: Two servers $\mathsf{S}_0, \mathsf{S}_1$ and a client C.

Inputs: Each server S_i $(i \in \{0, 1\})$ inputs two secret shared values $[\![x]\!]_i \in \mathbb{Z}_{2^\ell}$ and $[\![y]\!]_i \in \mathbb{Z}_{2^\ell}$ subject to the constraint that $0 \leq x \leq y$ and $y \neq 0$. The client C inputs nothing.

Functionality: On receiving $[\![x]\!]_i$ and $[\![y]\!]_i$ from S_i $(i \in \{0, 1\})$:

- Recover $x = [\![x]\!]_0 + [\![x]\!]_1$, $y = [\![y]\!]_0 + [\![y]\!]_1$ and compute the quotient $q = \mathsf{Quotient}(x, y)$.
- Sample a random secret sharing of q, namely $([\![q]\!]_0, [\![q]\!]_1) \leftarrow \mathsf{Sharing}(q)$.
- Send $[\![q]\!]_i$ to each server S_i $(i \in \{0, 1\})$ and send $\perp$ to the client C.

Fig. 7. Ideal functionality $\mathcal{F}_{\text{DIV}}$ for computing division of two secret shared values.

Construction Overview. Inspired by the division protocol of SecureNN [35], we compute the quotient bit by bit. Let k_i $(i \in \{\ell_f, \ldots, 0\})$ be every bit of the quotient. In our protocol, the servers compute a secret share of each bit step by step and then combine them together to get a secret share of the quotient. In particular, the servers store a secret share of an intermediate variable $u \in \mathbb{Z}_{2^\ell}$ (the dividend) initiated to be x. We start with the most significant bit k_{ℓ_f} by computing the sign check of $u - y$. Afterwards, we replace u by $u = 2 \cdot (u - k_{\ell_f} \cdot y)$. Then we can compute the next bit in exactly the same way, i.e., $k_{\ell_f - 1}$ is equal to the sign of $u - y$. We can simply repeat the above two steps for the remaining bits. The main idea is that when we compute the i-th bit k_i $(i \in \{\ell_f - 1, \ldots, 0\})$ after getting $k_{\ell_f}, \ldots, k_{i+1}$, we are actually computing the sign of $x \cdot 2^{\ell_f} - y \cdot \sum_{j=i+1}^{\ell_f} k_j \cdot 2^j - y \cdot 2^i$. Our protocol is described in Fig. 8. We state the theorem below and give the security proof in the full version of this paper.

Protocol Π_{Div}:

1. Each server S_i $(i \in \{0,1\})$ sets $[\![u_{\ell_f+1}]\!]_i = [\![x]\!]_i$.
2. For j from ℓ_f downto 0:
   (a) Each server S_i $(i \in \{0,1\})$ computes $[\![z_j]\!]_i = [\![u_{j+1}]\!]_i - [\![y]\!]_j + i$.
   (b) S_0, S_1 and C call $\mathcal{F}_{\text{SignCheck}}$ to compute

$$\left(\left(b_j^{\mathsf{S}}, \left[\!\left[b_j^{\mathsf{C}}\right]\!\right]_0\right), \left(b_j^{\mathsf{S}}, \left[\!\left[b_j^{\mathsf{C}}\right]\!\right]_1\right), b_j^{\mathsf{C}}\right) \leftarrow \mathcal{F}_{\text{SignCheck}}([\![z_j]\!]_0, [\![z_j]\!]_1, \perp).$$

   (c) S_0, S_1 and C call $\mathcal{F}^1_{\text{InnerProd}}$ to compute

$$([\![v_j]\!]_0, [\![v_j]\!]_1, \perp) \leftarrow \mathcal{F}^1_{\text{InnerProd}}\left(\left([\![y]\!]_0, \left[\!\left[b_j^{\mathsf{C}}\right]\!\right]_0\right), \left([\![y]\!]_1, \left[\!\left[b_j^{\mathsf{C}}\right]\!\right]_1\right), b_j^{\mathsf{C}}\right).$$

   (d) Each server S_i computes

$$[\![k_j]\!]_i = i \cdot b_j^{\mathsf{S}} + \left[\!\left[b_j^{\mathsf{C}}\right]\!\right]_i \cdot (1 - 2b_j^{\mathsf{S}}), \quad [\![v_j^*]\!]_i = [\![y]\!]_i \cdot b_j^{\mathsf{S}} + [\![v_j]\!]_i \cdot (1 - 2b_j^{\mathsf{S}}),$$
$$[\![u_j]\!]_i = 2 \cdot \left([\![u_{j+1}]\!]_i - [\![v_j^*]\!]_i\right).$$

3. Each server S_i $(i \in \{0,1\})$ outputs $[\![q]\!]_i = \sum_{j=0}^{\ell_f} 2^j \cdot [\![k_j]\!]_i$.

Fig. 8. Protocol Π_{Div} for computing division in the $(\mathcal{F}_{\text{SignCheck}}, \mathcal{F}^1_{\text{InnerProd}})$-hybrid model.

Theorem 4. *The protocol Π_{Div} (Fig. 8) securely computes the ideal functionality $\mathcal{F}_{\text{Div}}$ (Fig. 7) in the $(\mathcal{F}_{\text{SignCheck}}, \mathcal{F}^1_{\text{InnerProd}})$-hybrid model against a semi-honest adversary that corrupts either the client C or one of the two servers.*

Communication. Considering all the computation among secret shared values, the servers and the client jointly compute (ℓ_f+1) sign checks and multiplications in our division protocol. We can naturally use the protocols proposed in Sects. 4.1 and 4.2. The total communication is $2(\ell_f + 1) \cdot \lambda \cdot \ell$ bits with $4 \cdot (\ell_f + 1)$ ring elements.

5 Performance Evaluation

We implement our two-server PPML protocols for training algorithms including linear regression, logistic regression, and neural networks, against both semi-honest and malicious clients. We report our performance in comparison with SecureML [28] in the semi-honest model in this section and defer the performance against malicious clients to the full version of the paper. We did not compare the concrete performance with the state-of-the-art ABY2.0 [29] because their code for ML training is not available, but we did theoretical comparisons with their work.

We did not compare our protocols to prior works on PPML with three or more non-colluding servers because we believe our model differs from theirs in several key aspects. While the clients in our model could be considered as an additional server, the requirements on them are much weaker. Specifically, in

prior works with three or more non-colluding servers, all the servers jointly hold secret shares of all the intermediate values. They participate in every step of the computation throughout the entire protocol. Nevertheless, our approach does not require the clients to stay online or hold any secret state. In client-aided sign check, activation functions, and division protocols, each time the servers may choose an arbitrary client for assistance while other clients are offline. After each iteration, the client may completely go offline without having to keep any secret state. Furthermore, the clients initially hold their data in the clear, which can be leveraged in client-aided inner product.

5.1 Implementation Details

We implement our protocols in C++. The only cryptographic primitive we need is PRF, which is instantiated with AES. We set the computational security parameter $\lambda = 128$ and statistical security parameter $\sigma = 40$.

Experiment Settings. Our experiments are performed on a single Amazon Web Services (AWS) Elastic Compute Cloud (EC2) c4.8xlarge virtual machine with 18-core 2.9 GHz Intel Xeon CPU and 60 GB of RAM which is the same as [28]. We simulate the network connection using the Linux `tc` command. For the experiments on a LAN network, we set the round-trip time (RTT) latency to be 0.34 ms and network bandwidth to be 8192 Mbps, same as [28]. For the experiments on a WAN network, we set the RTT latency to be 60 ms and the network bandwidth to be 60 Mbps.

Dataset and Parameters. In our experiments, we train our algorithms on the MNIST dataset [11], which contains images of handwritten digits from 0 to 9. Each training sample has 784 features representing 28×28 pixels in the image.

In our training protocols, we have the number of features $d = 784$; we set the mini-batch size $B = 128$ and the number of epochs $E = 2$ (all the samples are used twice in training). The total number of training samples n varies between 10,240 and 100,352. The total number of training iterations is $t = \frac{E \cdot n}{B}$.

For fixed-point arithmetic, we set $\ell = 64, \ell_f = 13, \ell_w = 6$. That is, all the values are represented in $\mathbb{Z}_{2^{64}}$, where the lowest order 13 bits are the fractional part, and we assume there are at most 32 bits in the whole number part. These parameters are taken from [28].

Offline vs. Online. In the protocols of [28], there is an offline and an online phase, where the offline phase includes all the computation and communication that can be done without presence of data while the online phase consists of all *data-dependent* steps of the protocols. In the offline phase, they proposed three different approaches, one based on oblivious transfer (OT), one based on linearly homomorphic encryption (LHE), and one client-aided. The OT-based and LHE-based offline protocols are performed among the two servers to generate multiplication triples, while the client-aided offline protocol relies on a client to generate the triples.

Our protocols, on the other hand, only have an online phase, where the clients are heavily involved in the protocol execution. In particular, they will generate

data-dependent triples in the client-aided inner product protocol. Getting rid of the offline phase allows us to reduce the offline storage on the servers as well as the amount of communication between the servers. In our experiments below, we give comprehensive comparisons to [28] in both offline and online phases.

5.2 Linear Regression

In this section, we compare the performance of our semi-honest linear regression training to [28] instantiated with an OT-based, LHE-based, or client-aided offline phase. We report the running time in both LAN and WAN settings in Table 1 and the communication costs in Table 2.

Table 1. Running time of semi-honest linear regression on LAN and WAN networks comparing our protocol to [28] instantiated with different offline approaches. * indicates estimated running time.

	n	Offline Time (s)		Online Time (s)		Total Time (s)	
		LAN	WAN	LAN	WAN	LAN	WAN
Our work	10,240	**0**	**0**	**1.32**	**52.1**	**1.32**	**52.1**
	100,352	**0**	**0**	**13.2**	**505**	**13.2**	**505**
OT-based [28]	10,240	266	3,733	2.42	57.8	268	3,791
	100,352	2,667	36,600*	25.8	557	2,692	37,157*
LHE-based [28]	10,240	1,414	1,435	2.42	57.8	1,416	1,493
	100,352	13,800*	14,000*	25.8	557	13,826*	14,557*
Client-aided [28]	10,240	4.69	94.9	3.39	94.4	8.08	189
	100,352	52.0	749	35.3	1,126	87.3	1,875

Table 2. Communication cost of semi-honest linear regression comparing our protocol to [28] instantiated with different offline approaches. "S − S" and "C − S" denote the communication between the two servers and the communication between the clients and servers, respectively. * indicates estimated communication.

	n	Offline Comm (MB)		Online Comm (MB)			Total
		S − S	C − S	S − S	C − S	Total	Comm
Our work	10,240	**0**	**0**	**2.23**	185	**187**	**187**
	100,352	**0**	**0**	**21.8**	1,814	**1,836**	**1,836**
OT-based [28]	10,240	24,151	0	125	**123**	248	24,399
	100,352	236,607	0	1,224	**1,204**	2,428	239,034
LHE-based [28]	10,240	115	0	125	**123**	248	362
	100,352	1,120*	0	1,224	**1,204**	2,428	3,548*
Client-aided [28]	10,240	0	614	368	**123**	491	1,105
	100,352	0	6,016	3,609	**1,204**	4,813	10,829

In Table 1, we report the running time for both the offline and online phases in [28] as well as the total time. Our protocol does not incur any offline cost, and our online phase is also more efficient as our computation overhead is lower. In particular, in the online phase we achieve 1.83−2.67× improvement over [28] in the LAN setting and 1.11–2.23× improvement in the WAN setting. For the total running time (offline + online), we achieve 6.12−1047× improvement in the LAN setting and 3.63−73.5× improvement in the WAN setting.

In Table 2, we report both the communication between the two servers and the communication between the clients and servers, which are denoted by "$\mathsf{S}-\mathsf{S}$" and "$\mathsf{C}-\mathsf{S}$" respectively in the table. Again, our protocol does not incur any offline cost. In the online phase, our communication cost between the two servers is significantly lower than [28]. In particular, our $\mathsf{S}-\mathsf{S}$ online communication is $2(B+d)\cdot t$ ring elements. The $\mathsf{S}-\mathsf{S}$ online communication of the OT-based and LHE-based protocols in [28] is $2n\cdot d+2(B+d)\cdot t$ ring elements, and that of the client-aided variant is $2n\cdot d+2(Bd+B)\cdot t$ ring elements. Although our online communication between the clients and servers is higher than [28], the total communication is still much lower than [28]. In particular, in the online phase our $\mathsf{S}-\mathsf{S}$ communication achieves 56.1−165× improvement over [28], and our total online communication achieves 1.32−2.63× improvement. For the total communication (offline + online), we achieve 1.93−130× improvement.

Increasing Mini-Batch Size. If we increase the mini-batch size B, we can achieve lower communication, and hence the performance also improves especially in the WAN setting. This is because some part of the communication grows with the number of iterations. If the number of epochs and n remain the same and the mini-batch size is increased, then the number of iterations decreases and the communication is lowered as well. See Fig. 9 for the performance of the online phase in the WAN setting with different mini-batch sizes. We only compare with the client-aided variant of [28] because they achieve the most comparable overall running time.

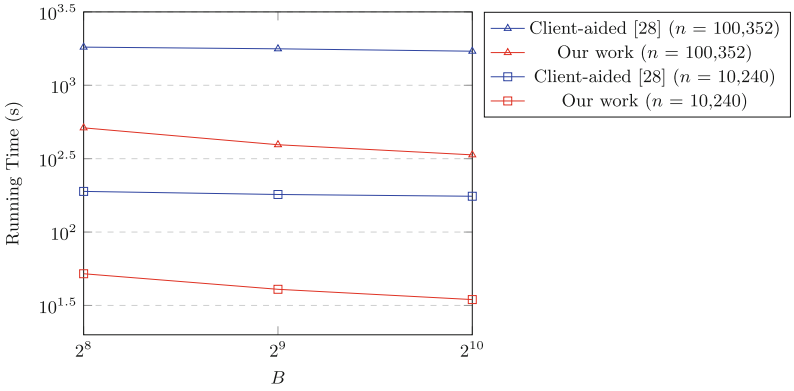

Fig. 9. Total running time of semi-honest linear regression over WAN with different mini-batch sizes.

Comparison with ABY2.0 [29]**.** We compare our performance with [29] theoretically as their code is not available. Since [29] uses the same OT-based and LHE-based multiplication triples generation as [28] in the offline phase, as seen in Tables 1 and 2, their offline time and communication are already much higher than our total time and communication.

Although not mentioned in their paper, we observe that a similar client-aided approach can be applied to [29] to improve the efficiency of the offline phase while introducing some overhead in the online phase. Nevertheless, we expect our work to still outperform [29] in that case. With a client-aided offline phase, the total communication (offline + online) of [29] is $(12Bd + 6B + 4d) \cdot t$ ring elements. If using all the optimizations we mentioned, its communication can be reduced to $n \cdot (3d + 1) + (Bd + 3B + 2d) \cdot t$ ring elements. In comparison, our total communication is $n \cdot (d + 1) + (Bd + 3B + 2d) \cdot t$ ring elements and our computation cost is roughly half of [29]. This is because each server computes two matrix multiplications for a private inner product in [28,29], while they each compute one matrix multiplication in our protocol.

5.3 Logistic Regression

In this section, we compare the performance of our semi-honest logistic regression training to [28] in Tables 3 and 4. Compared to linear regression, the only overhead of logistic regression is the cost of the activation function. In each iteration, each client and the two servers run Π_{ReLU} twice.

Our computation cost of one ReLU mainly consists of ℓ PRF operations and sorting these ℓ PRF results for each server. As shown in Table 3 for the running time, in the online phase we achieve 1.97−2.45× improvement over [28] in the LAN setting and 1.23−1.64× improvement in the WAN setting. For the total running time (offline + online), we achieve 4.85−723× improvement in the LAN setting and 2.71−44.3× improvement in the WAN setting.

Table 3. Running time of semi-honest logistic regression on LAN and WAN networks comparing our protocol to [28] instantiated with different offline approaches. * indicates estimated running time.

	n	Offline Time (s)		Online Time (s)		Total Time (s)	
		LAN	WAN	LAN	WAN	LAN	WAN
Our work	10,240	**0**	**0**	**1.96**	**87.6**	**1.96**	**87.6**
	100,352	**0**	**0**	**19.8**	**850**	**19.8**	**850**
OT-based [28]	10,240	266	3,733	3.86	108	270	3,841
	100,352	2,667	36,600*	40.0	1,056	2,707	37,656*
LHE-based [28]	10,240	1,414	1,435	3.86	108	1,418	1,543
	100,352	13,800*	14,000*	40.0	1,056	13,840*	15,056*
Client-aided [28]	10,240	4.71	95.4	4.81	142	9.52	237
	100,352	55.9	941	46.3	1,398	102	2,339

Table 4. Communication cost of semi-honest logistic regression comparing our protocol to [28] instantiated with different offline approaches. "S − S" and "C − S" denote the communication between the two servers and the communication between the clients and servers, respectively. * indicates estimated communication.

	n	Offline Comm (MB)		Online Comm (MB)			Total
		S − S	C − S	S − S	C − S	Total	Comm
Our work	10,240	**0**	**0**	**2.85**	266	**269**	**269**
	100,352	**0**	**0**	**28.0**	2,605	**2,633**	**2,633**
OT-based [28]	10,240	24,151	0	257	**123**	380	24,531
	100,352	236,607	0	2,524	**1,204**	3,728	240,335
LHE-based [28]	10,240	115	0	257	**123**	380	495
	100,352	1,120*	0	2,524	**1,204**	3,728	4,848*
Client-aided [28]	10,240	0	614	502	**123**	625	1,239
	100,352	0	6,016	4,424	**1,204**	5,628	11,644

In terms of communication, our total communication overhead in Π_{ReLU} is $4B \cdot t \cdot \lambda \cdot \ell$ bits with $8B \cdot t$ ring elements, while the communication overhead in [28] is $2B \cdot t \cdot (2\lambda \cdot (2\ell - 1) + 3\ell)$ bits. In [29], the online communication for one ReLU is $3\ell + 230$ bits and the offline communication is $1337\lambda + 5\ell + 1332$ bits, so its total communication overhead for logistic regression is $2B \cdot t \cdot (1337\lambda + 8\ell + 1562)$ bits. As shown in Table 4, in the online phase our S − S communication achieves 90.1−176× improvement over [28], and our total online communication achieves 1.41−2.32× improvement. For the total communication (offline + online), we achieve 1.84−91.3× improvement.

5.4 Neural Networks

We train a neural network consisting of three fully connected layers while the cross entropy function is employed as the loss function. The neural network has 128 neurons in each hidden layer and 10 in the output layers. We use the ReLU activation function for the two hidden layers and the MPC-friendly variant of the softmax function (see Sect. 3.2) for the output layer.

Table 5. Running time of semi-honest neural networks on LAN and WAN networks comparing our protocol to client-aided [28]. * indicates estimated running time.

	n	Offline Time (s)		Online Time (s)		Total Time (s)	
		LAN	WAN	LAN	WAN	LAN	WAN
Our work	10,240	**0**	**0**	257	**5875**	**257**	**5875**
	100,352	**0**	**0**	2,510*	**57,500***	**2,510***	**57,500***
Client-aided [28]	10,240	674	16,350*	**147**	6,690*	821	23,040*
	100,352	6,600*	160,200*	**1,440***	65,600*	8,040*	225,800*

Table 6. Communication cost of semi-honest neural networks comparing our protocol to client-aided [28]. "S − S" and "C − S" denote the communication between the two servers and the communication between the clients and servers, respectively. * indicates estimated communication.

	n	Offline Comm (MB)		Online Comm (MB)			Total
		S − S	C − S	S − S	C − S	Total	Comm
Our work	10,240	**0**	**0**	**664**	35,798	**36,462**	**36,462**
	100,352	**0**	**0**	**6,500***	351,000*	**357,500***	**357,500***
Client-aided [28]	10,240	0	112,114	43,659	**124**	43,783	155,897
	100,352	0	1,099,000*	427,900*	**1,220***	429,100*	1,528,000*

We compare the performance of our semi-honest neural network training to [28] in Tables 5 and 6. We only compare with the client-aided variant of [28] because it achieves the most comparable performance to ours. The neural network has two hidden layers with 128 neurons in each layer.

As shown in Table 5, in the online phase we achieve an improvement of 1.14× on WAN. For the total running time (offline + online), we achieve an improvement of 3.19× on LAN and 3.92× on WAN. As shown in Table 6, our S − S communication in the online phase achieves an improvement of 65.8× and our total online communication achieves an improvement of 1.20×. We achieve a 4.28× improvement for the total communication (offline + online).

Acknowledgments. This project is supported in part by the NSF CNS Award 2247352, Brown Data Science Seed Grant, Meta Research Award, Google Research Scholar Award, and Amazon Research Award.

References

1. Addanki, S., Garbe, K., Jaffe, E., Ostrovsky, R., Polychroniadou, A.: Prio+: privacy preserving aggregate statistics via Boolean shares. In: Galdi, C., Jarecki, S. (eds.) SCN 2022. LNCS, vol. 13409, pp. 516–539. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-14791-3_23
2. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 420–432. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-46766-1_34
3. Bell, J.H., Bonawitz, K.A., Gascón, A., Lepoint, T., Raykova, M.: Secure single-server aggregation with (poly)logarithmic overhead. In: ACM SIGSAC CCS (2020)
4. Bonawitz, K.A., et al.: Practical secure aggregation for federated learning on user-held data. CoRR (2016)
5. Bunn, P., Ostrovsky, R.: Secure two-party k-means clustering. In: CCS (2007)
6. Byali, M., Chaudhari, H., Patra, A., Suresh, A.: FLASH: fast and robust framework for privacy-preserving machine learning. Proc. Priv. Enhanc. Technol. (2020)
7. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: FOCS (2001)
8. Chaudhari, H., Choudhury, A., Patra, A., Suresh, A.: ASTRA: high throughput 3PC over rings with application to secure prediction. In: ACM SIGSAC (2019)

9. Chaudhari, H., Rachuri, R., Suresh, A.: Trident: efficient 4PC framework for privacy preserving machine learning. In: NDSS (2020)
10. Corrigan-Gibbs, H., Boneh, D.: Prio: private, robust, and scalable computation of aggregate statistics. In: USENIX NSDI (2017)
11. Deng, L.: The MNIST database of handwritten digit images for machine learning research. IEEE Signal Process. Mag. (2012)
12. Geng, J., et al.: Towards general deep leakage in federated learning. CoRR (2021)
13. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: STOC (1987)
14. Jagannathan, G., Wright, R.N.: Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In: ACM SIGKDD (2005)
15. Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: GAZELLE: a low latency framework for secure neural network inference. In: USENIX Security (2018)
16. Kairouz, P., et al.: Advances and open problems in federated learning. CoRR (2019)
17. Kamara, S., Mohassel, P., Raykova, M., Sadeghian, S.: Scaling private set intersection to billion-element sets. In: Christin, N., Safavi-Naini, R. (eds.) FC 2014. LNCS, vol. 8437, pp. 195–215. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45472-5_13
18. Konečný, J., McMahan, H.B., Ramage, D., Richtárik, P.: Federated optimization: distributed machine learning for on-device intelligence. CoRR (2016)
19. Koti, N., Pancholi, M., Patra, A., Suresh, A.: SWIFT: super-fast and robust privacy-preserving machine learning. In: USENIX Security (2021)
20. Kumar, N., Rathee, M., Chandran, N., Gupta, D., Rastogi, A., Sharma, R.: CrypTFlow: secure TensorFlow inference. In: IEEE SP (2020)
21. Le, P.H., Ranellucci, S., Gordon, S.D.: Two-party private set intersection with an untrusted third party. In: SIGSAC (2019)
22. Lin, H.-Y., Tzeng, W.-G.: An efficient solution to the millionaires' problem based on homomorphic encryption. In: Ioannidis, J., Keromytis, A., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 456–466. Springer, Heidelberg (2005). https://doi.org/10.1007/11496137_31
23. Lindell, Y., Pinkas, B.: Privacy preserving data mining. J. Cryptol. (2002)
24. McMahan, B., Moore, E., Ramage, D., Hampson, S., y Arcas, B.A.: Communication-efficient learning of deep networks from decentralized data. In: AISTATS (2017)
25. Melis, L., Song, C., De Cristofaro, E., Shmatikov, V.: Exploiting unintended feature leakage in collaborative learning. In: IEEE SP (2019)
26. Mishra, P., Lehmkuhl, R., Srinivasan, A., Zheng, W., Popa, R.A.: Delphi: a cryptographic inference service for neural networks. In: USENIX Security (2020)
27. Mohassel, P., Rindal, P.: Aby^3: a mixed protocol framework for machine learning. In: ACM SIGSAC CCS (2018)
28. Mohassel, P., Zhang, Y.: SecureML: a system for scalable privacy-preserving machine learning. In: IEEE SP (2017)
29. Patra, A., Schneider, T., Suresh, A., Yalame, H.: ABY2.0: improved mixed-protocol secure two-party computation. In: USENIX Security (2021)
30. Patra, A., Suresh, A.: BLAZE: blazing fast privacy-preserving machine learning. In: NDSS (2020)
31. Rathee, D., et al.: CryptFlow2: practical 2-party secure inference. In: ACM SIGSAC CCS (2020)
32. Sadegh Riazi, M., Weinert, C., Tkachenko, O., Songhori, E.M., Schneider, T., Koushanfar, F.: Chameleon: a hybrid secure computation framework for machine learning applications. In: AsiaCCS (2018)

33. Salem, A., Bhattacharya, A., Backes, M., Fritz, M., Zhang, Y.: Updates-leak: data set inference and reconstruction attacks in online learning. In: USENIX Security (2020)
34. Vaidya, J., Yu, H., Jiang, X.: Privacy-preserving SVM classification. Knowl. Inf. Syst. (2008)
35. Wagh, S., Gupta, D., Chandran, N.: SecureNN: 3-party secure computation for neural network training. Proc. Priv. Enhancing Technol. (2019)
36. Wang, Z., Song, M., Zhang, Z., Song, Y., Wang, Q., Qi, H.: Beyond inferring class representatives: user-level privacy leakage from federated learning. In: IEEE INFOCOM (2019)
37. Yao, A.C.-C.: How to generate and exchange secrets (extended abstract). In: FOCS (1986)
38. Yu, H., Vaidya, J., Jiang, X.: Privacy-preserving SVM classification on vertically partitioned data. In: Ng, W.-K., Kitsuregawa, M., Li, J., Chang, K. (eds.) PAKDD 2006. LNCS (LNAI), vol. 3918, pp. 647–656. Springer, Heidelberg (2006). https://doi.org/10.1007/11731139_74
39. Zhu, L., Liu, Z., Han, S.: Deep leakage from gradients. In: NeurIPS (2019)