SecureML: A System for Scalable Privacy-Preserving Machine Learning

Payman Mohassel* and Yupeng Zhang†
*Visa Research, †University of Maryland

Abstract—Machine learning is widely used in practice to produce predictive models for applications such as image processing, speech and text recognition. These models are more accurate when trained on large amounts of data collected from different sources. However, the massive data collection raises privacy concerns.

In this paper, we present new and efficient protocols for privacy preserving machine learning for linear regression, logistic regression and neural network training using the stochastic gradient descent method. Our protocols fall in the two-server model where data owners distribute their private data among two non-colluding servers who train various models on the joint data using secure two-party computation (2PC). We develop new techniques to support secure arithmetic operations on shared decimal numbers, and propose MPC-friendly alternatives to non-linear functions such as sigmoid and softmax that are superior to prior work.

We implement our system in C++. Our experiments validate that our protocols are several orders of magnitude faster than the state of the art implementations for privacy preserving linear and logistic regressions, and scale to millions of data samples with thousands of features. We also implement the first privacy preserving system for training neural networks.

I. INTRODUCTION

Machine learning techniques are widely used in practice to produce predictive models for use in medicine, banking, recommendation services, threat analysis, and authentication technologies. Large amount of data collected over time have enabled new solutions to old problems, and advances in deep learning have led to breakthroughs in speech, image and text recognition.

Large internet companies collect users’ online activities to train recommender systems that predict their future interest. Health data from different hospitals, and government organization can be used to produce new diagnostic models, while financial companies and payment networks can combine transaction history, merchant data, and account holder information to train more accurate fraud-detection engines.

While the recent technological advances enable more efficient storage, processing and computation on big data, combining data from different sources remains an important challenge. Competitive advantage, privacy concerns and regulations, and issues surrounding data sovereignty and jurisdiction prevent many organizations from openly sharing their data. Privacy-preserving machine learning via secure multiparty computation (MPC) provides a promising solution by allowing different entities to train various models on their joint data without revealing any information beyond the outcome.†

We focus on machine learning algorithms for training linear regression, logistic regression and neural networks models, and adopt the two-server model (see section III for more details), commonly used by previous work on privacy-preserving machine learning via MPC [37], [36], [21]. In this model, in a setup phase, the data owners (clients) process, encrypt and/or secret-share their data among two non-colluding servers. In the computation phase, the two servers can train various models on the clients’ joint data without learning any information beyond the trained model.

The state of the art solutions for privacy preserving linear regression [37], [21] are many orders of magnitude slower than plaintext training. The main source of inefficiency in prior implementations is that the bulk of computation for training takes place inside a secure 2PC for boolean circuits (e.g. Yao’s garbled circuit) that performs arithmetic operation on decimal numbers represented as integers. It is well-known that boolean circuits are not suitable for performing arithmetic operations, but they seem unavoidable given that existing techniques for fixed-point or floating-point multiplication require bit-level manipulations that are most efficient using boolean circuits.

In case of logistic regression and neural networks, the problem is even more challenging as the training procedure computes many instances of non-linear activation functions such as sigmoid and softmax that are expensive to compute inside a 2PC. Indeed, we are not aware of any privacy preserving implementations for these two training algorithms.

A. Our Contributions

We design new and efficient protocols for privacy preserving linear regression, logistic regression and neural networks training in the two-server model discussed above assuming an arbitrary partitioning of the dataset across the clients.

Our privacy preserving linear regression protocol is several orders of magnitude more efficient than the state of the art solutions for the same problem. For example, for a dataset with 100,000 samples and 500 features and in a comparable setup and experimental environment, our protocol is 1100-1300× faster than the protocols implemented in [37], [21]. Moreover, as our experiments show, we significantly reduce the gap between privacy-preserving and plaintext training.

†This work was partially done when the author was interning at Visa Research.

†In the more general variant of our protocols, even the model can remain private (secret shared).
We also implement the first privacy preserving protocols for logistic regression and neural networks training with high efficiency. For example, on a dataset of size 60,000 with 784 features, our privacy preserving logistic regression has a total running time of 29s while our privacy-preserving protocol for training a neural network with 3 layers and 266 neurons runs in 21,000s.

Our protocols are naturally divided into a data-independent offline phase and a much faster online phase. When excluding the offline phase, the protocols are even more competitive with plaintext training. For instance, for a dataset with 60,000 samples and 784 features, and in the LAN setting, the linear regression protocol runs in 1.4s, the logistic regression in 8.9s, and the neural network training in 653.0s.

**Arithmetic on shared decimal numbers.** As mentioned earlier, a major bottleneck in prior work is the computation of fixed-point arithmetic inside a secure 2PC such as garbled circuits. This is prohibitively expensive, given the large number of multiplications needed for training.

Fixed-point addition is fairly straightforward. For multiplication, we show that the following strategy is very effective: represent the two shared decimal numbers as shared integers in a finite field; perform a multiplication on shared integers using offline-generated multiplication triplets; have each party truncate its share of the product so that a fixed number of bits represent the fractional part. We prove that, with high probability, the product when reconstructed from these truncated shares, is at most 1 bit off in the least significant position of the fractional part compared to fixed-point arithmetic. Our experiments on two different datasets, MNIST and Arcene [6], [1], confirm that the small truncation error has no effect on accuracy of the trained model (in fact accuracies match those of standard training) when the number of bits representing the fractional part is sufficiently large. As a result, the online phase for privacy preserving linear regression does not involve any cryptographic operations and only consists of integer multiplications and bit shifting, while the offline phase consists of generating the necessary multiplication triplets. Our microbenchmarking shows that even when considering total time (online and offline combined) our approach yields a factor of 4-8 × improvement compared to fixed-point multiplication using garbled circuits.

**MPC-friendly activation functions.** As discussed earlier, logistic regression and neural network training require computing the logistic \( \frac{1}{1+e^{-x}} \), and the softmax \( \frac{e^{x_i}}{\sum e^{x_i}} \) functions which are expensive to compute on shared values. We experimentally show that the use of low-degree polynomials to approximate the logistic function is ineffective. In particular, one needs polynomials of degree at least 10 to approach the accuracy of training using the logistic function. We propose a new activation function that can be seen as the sum of two RELU functions (see Figure 5), and computed efficiently using a small garbled circuit. Similarly, we replace the softmax function with a combination of RELU functions, additions and a single division. Our experiments using the MNIST, and Arcene datasets confirm that accuracy of the models produced using these new functions either match or are very close to those trained using the original functions.

We then propose a customized solution for switching between arithmetic sharing and Yao sharing, and back, for our particular computation, that significantly reduces the cost by minimizing rounds of interaction and number of invoked oblivious transfers (OT). Our microbenchmarking in Section G shows that the time to evaluate our new function is much faster than to approximate the logistic function with a high degree polynomial.

We use the same ideas to securely evaluate the RELU functions used in neural networks training.

**Vectorizing the protocols.** Vectorization, i.e. operating on matrices and vectors, is critical in efficiency of plaintext training. We show how to benefit from the same vectorization techniques in the shared setting. For instance, in the offline phase of our protocols which consists of generating many multiplication triplets, we propose and implement two solutions based on linearly homomorphic encryption (LHE) and oblivious transfer. The techniques are inspired by prior work (e.g., [18]) but are optimized for our vectorized scenario where we need to compute multiplication of shared matrices and vectors. As a result the complexity of our offline protocols is much better than the naive approach of generating independent multiplication triplets for each multiplication. In particular, the performance of the OT-based multiplication triplets generation is improved by a factor of 4×, and the LHE-based generation is improved by 41-66×.

In a different security model similar to [21], we also propose a much faster offline phase where clients help generate the multiplication triplets. This provides a weaker security guarantee than our standard setting. In particular, it requires the additional assumption that servers and clients do not collude, i.e. an attacker either corrupts a server or a subset of clients but not both. We discuss pros/cons of this approach and compare its performance with the standard approach in Section V and Appendix F.

**B. Related Work**

Earlier work on privacy preserving machine learning has focused on decision trees [31], k-means clustering [28], [14], SVM classification [48], [44], linear regression [19], [20], [40] and logistic regression [42]. These papers propose solutions based on secure multiparty computation, but appear to incur high efficiency overheads and lack implementation/evaluation.

Nikolaenko et. al. [37] present a privacy preserving linear regression protocol on horizontally partitioned data using a combination of LHE and garbled circuits, and evaluate it on datasets with millions of samples. Gascon et. al. [21] extend the results to vertically partitioned data and show improved performance. However, both papers reduce the problem to solving a linear system using Yao’s garbled circuit protocol, which introduces a high overhead on the training time and cannot be generalized to non-linear models. In contrast, we use the stochastic gradient descent method which enables training non-linear models such as logistic regression and neural networks. Recently, Gilad-Bachrach et. al. [23] propose a framework for secure data exchange, and support privacy preserving linear regression as an application. However, only small datasets are tested and the protocol is implemented purely using garbled circuit, which does not scale for larger datasets.
Privacy preserving logistic regression is considered by Wu et. al. [46]. They propose to approximate the logistic function using polynomials, and train the model using LHE. However, the complexity is exponential in the degree of the approximation polynomial, and as we will show in experiments, the accuracy of the model is degraded compared to using the logistic function. Aono et. al. [10] consider a different security model where an untrusted server collects and combines the encrypted data from multiple clients, and transfers it to a trusted client to train the model on the plaintext. By carefully approximating the cost function of logistic regression with a degree 2 polynomial, the optimal model can be calculated by solving a linear system. However, in this setting, the plaintext of the aggregated data is leaked to the client who trains the model. We are not aware of any prior work with a practical system for privacy preserving logistic regression in the two-server model.

Privacy preserving machine learning with neural networks is more challenging. Shokri and Shmatikov [41] propose a solution where instead of sharing the data, the two servers share the changes on a portion of the coefficients during the training. Although the system is very efficient (no cryptographic operation is needed at all), the leakage of these coefficient changes is not well-understood and no formal security guarantees are obtained. In addition, their approach only works for horizontally partitioned data since each server needs to be able to perform the training individually on its portion in order to obtain the coefficient changes. Privacy preserving predictions using neural networks were also studied recently by Gilad-Bachrach et. al. [22]. Using fully homomorphic encryption, the neural network model can make predictions on encrypted data. In this case, it is assumed that the neural network is trained on plaintext data and the model is known to one party who evaluates it on private data of another.

An orthogonal line of work considers the differential privacy of machine learning algorithms [16], [43], [9]. In this setting, the server has full access to the data in plaintext, but wants to guarantee that the released model cannot be used to infer the data used during the training. A common technique used in differentially private machine learning is to introduce an additive noise to the data or the update function (e.g., [9]). The parameters of the noise are usually predetermined by the dimensions of the data, the parameters of the machine learning algorithm and the security requirement, and hence are data-independent. Our system can be composed with such constructions given that the servers can always generate the noise according to the public parameters and add it directly onto the shared values in the training. In this way, the trained model will be differentially private once reconstructed, while all the data still remains private during the training.

II. PRELIMINARIES

A. Machine Learning

In this section, we briefly review the machine learning algorithms considered in this paper: linear regression, logistic regression and neural networks. All algorithms we present are classic and can be found in standard machine learning textbooks (e.g., [26]).

A) Linear regression: Given \( n \) training data samples \( x_i \), each containing \( d \) features and the corresponding output labels \( y_i \), regression is a statistical process to learn a function \( g \) such that \( g(x_i) \approx y_i \). Regression has many applications in real life. For example, in medical science, it is used to learn the relationship between a disease and representative features, such as age, weight, diet habits and use it for diagnosing purposes.

In linear regression, the function \( g \) is assumed to be linear and can be represented as the inner product of \( x_i \) with the coefficient vector \( w \): \( g(x_i) = \sum_{j=1}^{d} x_{ij} w_j = x_i \cdot w \), where \( x_{ij} \) (resp. \( w_j \)) is the \( j \)th value in vector \( x_i \) (resp. \( w \)), and \( \cdot \) denotes the inner product of two vectors.

To learn the coefficient vector \( w \), a cost function \( C(w) \) is defined and \( w \) is calculated by the optimization \( \arg\min_w C(w) \). In linear regression, a commonly used cost function is \( C(w) = \frac{1}{n} \sum C_i(w) \), where \( C_i(w) = \frac{1}{2} (x_i \cdot w - y_i)^2 \).

The solution for this optimization problem can be computed by solving the linear system \( X^T \cdot X \cdot w = X^T \cdot Y \), where \( X \) is a \( n \times d \) matrix representing all the input data, and \( Y \) is a \( n \times 1 \) matrix for the output labels. However, the complexity of the matrix multiplication \( X^T \cdot X \) is \( O(n d^2) \) and the complexity of solving the linear system is \( O(d^3) \). Due to its high complexity, it is rarely used in practice except for small values of \( n \) and \( d \).

b) Stochastic gradient descent (SGD): SGD is an effective approximation algorithm for approaching a local minimum of a function, step by step. As the optimization function for the linear regression described above is convex, SGD provably converges to the global minimum and is typically very fast in practice. In addition, SGD can be generalized to work for logistic regression and neural network training, where no closed-form solution exists for the corresponding optimization problems. As a result, SGD is the most commonly used approach to train such models in practice and the main focus of this work.

The SGD algorithm works as follows: \( w \) is initialized as a vector of random values or all 0s. In each iteration, a sample \((x_i, y_i)\) is selected randomly and a coefficient \( w_j \) is updated as

\[
\begin{align*}
\hat{w}_j & := w_j - \alpha \frac{\partial C_i(w)}{\partial w_j}.
\end{align*}
\]

where \( \alpha \) is a learning rate defining the magnitude to move towards the minimum in each iteration. Substituting the cost function of linear regression, the formula becomes \( \hat{w}_j := w_j - \alpha (x_i \cdot w - y_i) x_{ij} \). The phase to calculate the predicted output \( y^*_i = x_i \cdot w \) is called forward propagation, and the phase to calculate the change \( \alpha (y^*_i - y_i) x_{ij} \) is called backward propagation.

Mini-batch. In practice, instead of selecting one sample of data per iteration, a small batch of samples are selected randomly and \( w \) is updated by averaging the partial derivatives of all samples on the current \( w \). We denote the set of indices selected in a mini-batch by \( B \). This is called a mini-batch SGD and \(|B|\). [2] Usually a bias \( b \) is introduced such that \( g(x_i) = x_i \cdot w + b \). However, this can be easily achieved by appending a dummy feature equal to 1 for each \( x_i \).

To simplify the notation, we assume \( b \) is already embedded in \( w \) in this paper.

[3] In ridge regression, a penalty term \( \lambda ||w||^2 \) is added to the cost function to avoid overfitting where \( \lambda \) is the regularization parameter. This is supported in an obvious way by the protocols in this paper, and is omitted for simplicity.
At this point, if the accuracy decreases, the learning rate is reduced and the training starts over; otherwise the data is shuffled and each data sample in the testing dataset is calculated as the prediction, and is compared to the corresponding label. The accuracy is the percentage of the dataset is calculated as the prediction, and is compared to the result of SGD may diverge from the minimum. Therefore, learning rate adjustment.

B. Secure Computation

Oblivious Transfer. Oblivious transfer (OT) is a fundamental cryptographic primitive that is commonly used as building block in MPC. In an oblivious transfer protocol, a sender $S$ has two inputs $x_0$ and $x_1$, and a receiver $R$ has a selection bit $b$ and wants to obtain $x_b$ without learning anything else or revealing $b$ to $S$. Figure 2 describes the ideal functionality realized by such a protocol. We use the notation $(\bot; x_b) \leftarrow \text{OT}(x_0, x_1; b)$ to denote a protocol realizing this functionality.
Parameters: Sender $S$ and Receiver $R$.
Main: On input $(SELECT, sid, b)$ from $R$ and $(SEND, sid, x_0, x_1)$ from $S$, return $(RECV, sid, x_s)$ to $R$.

![Ideal Functionality](image)

We use OTs both as part of our offline protocol for generating multiplication triplets and in the online phase for logistic regression and neural network training in order to securely compute the activation functions. One-round OT can be implemented using the protocol of [39], but it requires public-key operations by both parties. OT extension [27], [11] minimizes this cost by allowing the sender and receiver to perform $m$ OTs at the cost of $\lambda$ base OTs (with public-key operations) and $O(m)$ fast symmetric-key ones, where $\lambda$ is the security parameter. Our implementations take advantage of OT extension for better efficiency. We also use a special flavor of OT extension called correlated OT extension [11]. In this variant which we denote by COT, the sender’s two inputs to each OT are: a random value $s_0$ and a value $s_1 = f(s_0)$ for a correlation function $f$ of the sender’s choice. The communication for a COT of $l$-bit messages, denoted by COT$_l$, is $\lambda + l$ bits, and the computation consists of 3 hashing.

Garbled Circuit 2PC. Garbled Circuits were first introduced by [47]. A garbling scheme consists of a garbling algorithm that takes a random seed $\sigma$ and a function $f$ and generates a garbled circuit $F$ and a decoding table $dec$; the encoding algorithm takes input $x$ and the seed $\sigma$ and generates garbled input $\tilde{x}$; the evaluation algorithm takes $\tilde{x}$ and $F$ as input and returns the garbled output $\tilde{z}$; and finally, a decoding algorithm that takes the decoding table $dec$ and $\tilde{z}$ and returns $f(x)$. We require the garbling scheme to satisfy the standard security properties formalized in [13].

Given such a garbling scheme, it is possible to design a secure two-party computation protocol as follows: Alice generates a random seed $\sigma$ and runs the garbling algorithm for function $f$ to obtain a garbled circuit $GC$. She also encodes her input $\tilde{x}$ using $\sigma$ and $x$ as inputs to the encoding algorithm. Alice sends $GC$ and $\tilde{x}$ to Bob. Bob obtains his encoded (garbled) input $\tilde{y}$ using an oblivious transfer for each bit of $y$. He then runs the evaluation algorithm on $GC$, $\tilde{x}$, $\tilde{y}$ to obtain the garbled output $\tilde{z}$. We can have Alice, Bob, or both learn an output by communicating the decoding table accordingly. The above protocol securely realizes the ideal functionality $\mathcal{F}_f$ that simply takes the parties inputs and computes $f$ on them. See [32] for a more detailed description and proof of security against a semi-honest adversary. In our protocols, we denote this garbled circuit 2PC by $(\sigma_0, s_0) \leftrightarrow \text{GarbledCircuit}(x; y; f)$.

Secret Sharing and Multiplication Triplets. In our protocols, all intermediate values are secret-shared between the two servers. We employ three different sharing schemes: Additive sharing, Boolean sharing and Yao sharing. We briefly review these schemes but refer the reader to [18] for more details.

To additively share $(Sh^A(\cdot))$ an $l$-bit value $a$, the first party $P_0$ generates $a_0 \in \mathbb{Z}_{2^l}$ uniformly at random and sends $a_1 = a - a_0 \mod 2^l$ to the second party $P_1$. We denote the first party’s share by $(\langle a \rangle_0^A = a_0$ and the second party’s by $(\langle a \rangle_1^A = a_1$. For ease of composition we omit the modular operation in the protocol descriptions. In this paper, we mostly use the additive sharing, and denote it by $\langle \cdot \rangle$ for short. To reconstruct $(\langle a \rangle^A, \langle b \rangle^A)$ an additively shared value $\langle a \rangle$, $P_i$ sends $(\langle a \rangle_i)$ to $P_{1-i}$, who computes $(\langle a \rangle_0 + \langle a \rangle_1$.

Given two shared values $\langle a \rangle$ and $\langle b \rangle$, it is easy to non-interactively add the shares by having $P_i$ compute $(\langle c \rangle_i) = (\langle a \rangle_i + (\langle b \rangle_i) \mod 2^l$. We overload the addition operation to denote the addition protocol by $\langle a \rangle + \langle b \rangle$.

To multiply $(\langle a \rangle^A, \langle b \rangle^A)$ two shared values $\langle a \rangle$ and $\langle b \rangle$, we take advantage of Beaver’s pre-computed multiplication triplet technique. Lets assume that the two parties already share $\langle u \rangle, \langle v \rangle, \langle z \rangle$ where $u, v$ are uniformly random values in $\mathbb{Z}_{2^l}$ and $z = uv \mod 2^l$. Then $P_i$ locally computes $(\langle e \rangle_i) = (\langle a \rangle_i - (\langle u \rangle_i)$ and $(\langle f \rangle_i) = (\langle b \rangle_i - (\langle v \rangle_i)$. Both parties run Rec$(\langle e \rangle_0, \langle e \rangle_1)$ and Rec$(\langle f \rangle_0, \langle f \rangle_1)$, and $P_i$ lets $(\langle c \rangle_i) = i \cdot e \cdot f + f \cdot (\langle a \rangle_i + e \cdot (\langle b \rangle_i + (\langle z \rangle_i$.

Boolean sharing can be seen as additive sharing in $\mathbb{Z}_2$ and hence all the protocols discussed above carry over. In particular, the addition operation is replaced by the XOR operation $(\oplus)$ and multiplication is replaced by the AND operation $(\wedge)$. We denote party $P_i$’s share in a Boolean sharing by $(\langle a \rangle_i^B$.

Finally, one can also think of a garbled circuit protocol as operating on Yao sharing of inputs to produce Yao sharing of outputs. In particular, in all garbling schemes, for each wire $w$ the garbler $(P_0)$ generates two random strings $k^0_w, k^1_w$. When using the point-and-permute technique [34] the garbler also generates a random permutation bit $r_w$ and lets $K^0_w = k^0_w \| r_w$ and $K^1_w = k^1_w \| (1-r_w)$. The concatenated bits are then used to permute the rows of each garbled truth table. A Yao sharing of $a$ is $\langle \langle a \rangle_0^Y = K^0_w, K^1_w \text{ and } \langle \langle a \rangle_1^Y = K^1_w$. To reconstruct the shared value, parties exchange their shares. XOR and AND operations can be performed by garbling/evaluating the corresponding gates.

To switch from a Yao sharing $\langle \langle a \rangle_0^Y = K^0_w, K^1_w \text{ and } \langle \langle a \rangle_1^Y = K^1_w$ to a Boolean sharing, $P_0$ lets $\langle \langle a \rangle_0^B = K^0_w \| \langle 0 \rangle$ and $P_1$ lets $\langle \langle a \rangle_0^B = K^1_w \| \langle 0 \rangle$. In other words, the permutation bits used in the garbling scheme can be used to switch to boolean sharing for free. We denote this Yao to Boolean conversion by $Y2B(\cdot, \cdot)$. We note that we do not explicitly use a Yao sharing in our protocol description as it will be hidden inside the garbling scheme, but explicitly use the Y2B conversion to convert the garbled output to a boolean sharing.

### III. Security Model

#### A. Architecture

We consider a set of clients $C_1, \ldots, C_m$ who want to train various models on their joint data. We do not make any assumptions on how the data is distributed among the clients. In particular, the data can be horizontally or vertically partitioned, or be secret-shared among them as part of a previous computation.

A natural solution is to perform a secure multiparty computation where each client plays the role of one party. While this approach satisfies the privacy properties we are aiming for, it has several drawbacks. First, it requires the clients to be involved throughout the protocol. Second, unlike the two-party
case, techniques for more than two parties (and a dishonest majority) are significantly more expensive and not scalable to large input sizes or a large number of clients.

Hence, we consider a server-aided setting where the clients outsource the computation to two untrusted but non-colluding servers $S_0$ and $S_1$. Server-aided MPC has been formalized and used in various previous work (e.g. see [29]). It has also been utilized in prior work on privacy-preserving machine learning [37], [36], [21]. Two important advantages of this setting are that (i) clients can distribute (secret-share) their inputs among the two servers in a setup phase but not be involved in any future computation, and (ii) we can benefit from a combination of efficient techniques for boolean computation such as garbled circuits and OT-extension, and arithmetic computation such as offline/online multiplication triplet shares.

Depending on the application scenario, previous work refers to the two servers as the evaluator and the cryptography service provider (CSP) [37], or the evaluator and a cloud service provider who maintains the data [24]. The two servers can also be representatives of the different subsets of clients or themselves be among the clients who possess data. Regardless of the specific role assigned to the servers, the trust model is the same and assumes that the two servers are untrusted but do not collude. We discuss the security definition in detail next.

B. Security Definition

Recall that the involved parties are $m$ clients $C_1, \ldots, C_m$ and two servers $S_0, S_1$. We assume a semi-honest adversary $A$ who can corrupt any subset of the clients and at most one of the two servers. This captures the property that the two servers are not colluding, i.e. if one is controlled by the adversary, the second one behaves honestly. Note that we do not put any restrictions on collusion among the clients and between the clients and the servers. We call such an adversary an admissible adversary. In one particular scenario (see Section V), we weaken the security model by requiring that servers do not collude with the clients.

The security definition should require that such an adversary only learns the data of the clients it has corrupted and the final output but nothing else about the remaining honest clients’ data. For example, an adversary $A$ who corrupts $C_1, C_2$ and $S_1$ should not learn any information about $C_3$’s data beyond the trained model. We define security using the framework of Universal Composition (UC) [15]. We give a brief overview of the definition in Appendix A, and refer the reader to [15] for the details. The target ideal functionality $F_{ml}$ for our protocols is described in Figure 3.

IV. PRIVACY PRESERVING MACHINE LEARNING

In this section, we present our protocols for privacy preserving machine learning using SGD. We first describe a protocol for linear regression in Section IV-A, based solely on arithmetic secret sharing and multiplication triplets. Next, we discuss how to efficiently generate these multiplication triplets in the offline phase in Section IV-B. We then generalize our techniques to support logistic regression and neural networks training in Sections IV-C and IV-D. Finally, techniques to support predication, learning rate adjustment and termination determination are presented in Section IV-E.

A. Privacy Preserving Linear Regression

Recall that we assume the training data is secret shared between two servers $S_0$ and $S_1$. We denote the shares by $(X)_0, (Y)_0$ and $(X)_1, (Y)_1$. In practice, the clients can distribute the shares between the two servers, or encrypt the first share using the public key of $S_0$, upload both the first encrypted share and the second plaintext share to $S_1$. $S_1$ then receives the encrypted shares to $S_0$ to decrypt. In our protocol, we also let the coefficients $w$ be secret shared between the two servers. It is initialized to random values simply by setting $(w)_0, (w)_1$ to be random, without any communication between the two servers. It is updated and remains secret shared after each iteration of SGD, until the end when it is reconstructed.

As described in Section II-A, the update function for linear regression is $w_j := w_{j-1} - \alpha \langle \sum_{k=1}^d x_{ik} w_k - y_i \rangle x_{ij}$, only consisting of additions and multiplications. Therefore, we apply the corresponding addition and multiplication algorithms for secret shared values to update the coefficients, which is \( (w_j) := (w_j) - \alpha \langle \langle \sum_{k=1}^d \langle \langle \langle x_{ik} \rangle, \langle w_k \rangle \rangle - \langle y_i \rangle, \langle x_{ij} \rangle \rangle \rangle \). We separate our protocol into two phases: online and offline. The online phase trains the model given the data, while the offline phase consists mainly of multiplication triplet generation. We focus on the online phase in this section, and discuss the offline phase in Section IV-B.

Vectorization in the Shared Setting. We also want to benefit from the mini-batch and vectorization techniques discussed in Section II-A (see Equation 2). To achieve this, we generalize the addition and multiplication operations on share values to shared matrices. Matrices are shared by applying $\text{Shr}^A$ to every element. Given two shared matrices $\langle A \rangle$ and $\langle B \rangle$, matrix addition can be computed non-interactively by letting $\langle C \rangle = \langle A \rangle + \langle B \rangle$ for $i \in \{0, 1\}$. To multiply two shared matrices, instead of using independent multiplication triplets, we take shared matrices $\langle U \rangle, \langle V \rangle, \langle Z \rangle$, where each element in $U$ and $V$ is uniformly random in $\mathbb{Z}_{2^l}$. $U$ has the same dimension as $A$, $V$ has the same dimension as $B$ and $Z = U \times V \mod 2^l$. $S_1$ computes $\langle E \rangle = \langle A \rangle - \langle U \rangle$, $\langle F \rangle = \langle B \rangle - \langle V \rangle$ and sends it to the other server. Both servers reconstruct $\langle E \rangle$ and $\langle F \rangle$ and set $\langle C \rangle = i \cdot E \times F + \langle A \rangle_i \times F + E \times \langle B \rangle_i + \langle Z \rangle_i$. The idea of this generalization is that each element in matrix $A$ is always masked by the same random element in $U$, while it is multiplied by different elements in $B$ in the matrix multiplication. Our security proof confirms that this does not affect security of the protocol, but makes the protocol significantly more efficient due to vectorization.

Applying the technique to linear regression, in each iteration, we assume the set of mini-batch indices $B$ is public, and perform the update $\langle w \rangle := \langle w \rangle - \alpha \langle \langle \langle X^B_0 \rangle, \langle X^B_1 \rangle \rangle, \langle w \rangle \rangle - \langle Y_B \rangle$. We further observe that one data sample will be used several times in
different epochs, yet it suffices to mask it by the same random multiplication triplet. Therefore, in the offline phase, one shared $n \times d$ random matrix $(\mathbf{U})$ is generated to mask the data samples $(\mathbf{X})$. At the beginning of the online phase, $(\mathbf{E})_i = (\mathbf{X})_i - (\mathbf{U})_i$ is computed and exchanged to reconstruct $\mathbf{E}$ through one interaction. After that, in each iteration, $\mathbf{E}_D$ is selected and used in the multiplication protocol, without any further computation and communication. In particular, in the offline phase, a series of min-batch indices $B_1, \ldots, B_t$ are agreed upon by the two servers. This only requires the knowledge of $n, d, t$ or an upperbound, but not any real data. Then the multiplication triplets $(\mathbf{U}), (\mathbf{V}), (\mathbf{Z}), (\mathbf{V}'), (\mathbf{Z}')$ are precomputed with the following property: $\mathbf{U}$ is an $n \times d$ matrix to mask the data $\mathbf{X}, \mathbf{V}$ is a $d \times t$ matrix, each column of which is used to mask a vector $\mathbf{Y}' - \mathbf{Y}$ in one iteration (backward propagation). We then set $\mathbf{Z}[i] = \mathbf{U}_{B_i} \times \mathbf{V}[i]$ and $\mathbf{Z}'[i] = \mathbf{U}_{B_i}^T \times \mathbf{V}'[i]$ for $i = 1, \ldots, t$, where $\mathbf{M}[i]$ denotes the $i$th column of the matrix $\mathbf{M}$. Using the multiplication triplets in matrix form, the computation and communication in both the online and the offline phase are reduced dramatically. We will analyze the cost later.

We denote the ideal functionality realizing the generation of these matrices in the offline phase by $\mathcal{F}_{\text{offline}}$.

**Arithmetic Operations on Shared Decimal Numbers.** As discussed earlier, a major source of inefficiency in prior work on privacy preserving linear regression stems from computing on shared/encrypted decimal numbers. Prior solutions either treat decimal numbers as integers and preserve full accuracy after multiplication by using a very large finite field [22], or utilize 2PC for boolean circuits to perform fixed-point [21] or floating-point [35] multiplication on decimal numbers. The former can only support a limited number of multiplications, as the range of the result grows exponentially with the number of multiplications. This is prohibitive for training where the number of multiplications is large. The latter introduces high overhead, as the boolean circuit for multiplying two $l$-bit numbers has $O(l^2)$ gates, and such a circuit needs to be computed in a 2PC (e.g. Yao’s garbled circuits) for each multiplication performed.

We propose a simple but effective solution to support decimal arithmetics in an integer field. Consider the fixed-point multiplication of two decimal numbers $x$ and $y$ with at most $l_D$ bits in the fractional part. We first transform the numbers to integers by letting $x' = 2^{l_D}x$ and $y' = 2^{l_D}y$ and then multiply them to obtain the product $z = x'y'$. Note that $z$ has at most $2l_D$ bits representing the fractional part of the product, so we simply truncate the last $l_D$ bits of $z$ such that it has at most $l_D$ bits representing the fractional part. Mathematically speaking, if $z$ is decomposed into two parts $z = z_1 . 2^{l_D} + z_2$, where $0 \leq z_2 < 2^{l_D}$, then the truncation results is $z_1$. We denote this truncation operations by $\lfloor z \rfloor$.

We show that this truncation technique also works when $z$ is secret shared. In particular, the two servers can truncate their individual shares of $z$ independently. In the following theorem we prove that for a large enough field, these truncated shares when reconstructed, with high probability, are at most 1 off from the desired $\lfloor z \rfloor$. In other words, we incur a small error

---

**Protocol SGD Linear** $(\langle \mathbf{X}, \mathbf{Y}, \mathbf{U}, \mathbf{V}, \mathbf{Z}, \mathbf{V}', \mathbf{Z}' \rangle)$:

1. $\mathcal{S}_i$ computes $\langle \mathbf{E}_i \rangle = (\mathbf{X})_i - (\mathbf{U})_i$ for $i \in \{0, 1\}$. Then parties run $\text{Rec}(\langle \mathbf{E}_0 \rangle, \langle \mathbf{E}_1 \rangle)$ to obtain $\mathbf{E}$.
2. for $j = 1, \ldots, t$
   3. Parties select the mini-batch $\langle \mathbf{X}_{B_j}, \mathbf{Y}_{B_j} \rangle$.
   4. $\mathcal{S}_i$ computes $\langle \mathbf{F}_j \rangle_i = (\mathbf{w})_i - (\mathbf{Y}[j])$ for $i \in \{0, 1\}$. Then parties run $\text{Rec}(\langle \mathbf{F}_0 \rangle_j, \langle \mathbf{F}_1 \rangle_j)$ to recover $\mathbf{F}_j$.
   5. $\mathcal{S}_i$ computes $\langle \mathbf{Y}_{B_j} \rangle_i = i \cdot \mathbf{E}_B \times \mathbf{F}_i + \langle \mathbf{X}_{B_j} \rangle_i \times \mathbf{F}_{j} + \mathbf{E}_B \times (\mathbf{w}_i) + \langle \mathbf{Z} \rangle_i$, for $i \in \{0, 1\}$.
   6. $\mathcal{S}_i$ compute the difference $\langle \mathbf{D}_{B_j} \rangle_i = \langle \mathbf{Y}_{B_j} \rangle_i - \langle \mathbf{Y}_{B_j} \rangle_i$, for $i \in \{0, 1\}$.
   7. $\mathcal{S}_i$ computes $\langle \mathbf{F}_j \rangle_i = \langle \mathbf{D}_{B_j} \rangle_i - (\mathbf{Y}_{B_j})$, for $i \in \{0, 1\}$. Parties then run $\text{Rec}(\langle \mathbf{F}_0 \rangle_j, \langle \mathbf{F}_1 \rangle_j)$ to obtain $\mathbf{F}_j$.
   8. $\mathcal{S}_i$ computes $\langle \Delta \rangle_i = i \cdot \mathbf{E}_B \times \mathbf{F}_j + \langle \mathbf{X}_{B_j} \rangle_i \times \mathbf{F}_j + \mathbf{E}_B \times (\mathbf{D}_{B_j})_i + (\mathbf{Z})_i$, for $i \in \{0, 1\}$.
   9. $\mathcal{S}_i$ truncates its shares of $\Delta$ element-wise to get $\langle \lfloor \Delta \rfloor \rangle_i$.
   10. $\mathcal{S}_i$ computes $\langle \mathbf{w}_i \rangle = (\mathbf{w})_i - (\Delta)_i$, for $i \in \{0, 1\}$.
11. Parties run $\text{Rec}^A(\langle \mathbf{w}_0 \rangle, \langle \mathbf{w}_1 \rangle)$ and output $\mathbf{w}$.

---

**Theorem 1.** In field $\mathbb{Z}_{d_l}$, let $x \in [0, 2^{l_z}] \cup [2^{l_z}, 2^{l_z + 1})$ and given shares $(\langle x \rangle_0, \langle x \rangle_1)$ of $x$, let $\langle (|x|) \rangle_0 = \lfloor \frac{|x|}{2^{l_z}} \rfloor$ and $\langle (|x|) \rangle_1 = 2^{l_z} - \langle (|x|) \rangle_0$ for $|x| > 2^{l_z - 1}$. Then with probability $\frac{2^k - 2^{l_z - 1}}{2^{l_z}}$, $\text{Rec}^A(\langle (|x|) \rangle_0, \langle (|x|) \rangle_1)$ outputs $\langle |x| \rangle = |x|$, where $| \cdot |$ denotes truncation by $l_D \leq l_x$ bits.

The complete protocol between the two servers for the online phase of privacy preserving linear regression is shown in Figure 4. It assumes that the data-independent shared matrices $(\mathbf{U}), (\mathbf{V}), (\mathbf{Z}), (\mathbf{V}'), (\mathbf{Z}')$ were already generated in the offline phase. Besides multiplication and addition of shared decimal numbers, the protocol also requires multiplying the coefficient vector by $\frac{\alpha}{|D|}$ in each iteration. To make this operation efficient, we set $\alpha = 2^k$ to be a power of 2, i.e. $\frac{\alpha}{|D|} = 2^{-k}$. Then the multiplication with $\frac{1}{|D|}$ can be replaced by having the parties truncate $k$ additional bits from their shares of the coefficients.

We sketch a proof for the following Theorem on security of the online protocol in Appendix D.

**Theorem 2.** Consider a protocol where clients distribute arithmetic shares of their data among two servers who run the protocol of Figure 4 and send the output to clients. In the $\mathcal{F}_{\text{offline}}$ hybrid model, this protocol realizes the ideal functionality $\mathcal{F}_{\text{mi}}$ of Figure 3 for the linear regression function, in presence of a semi-honest admissible adversary (see section III).

**Effect of Truncation Error.** Note that when the size of the field is large enough, truncation can be performed once per iteration instead of once per multiplication. Hence in our implementations, the truncation is performed $(|B| + d) \times t$ times and by the union bound, the probability of failure in
we focus on this basic step, where given shares of a matrix, we operate on matrices. As a result the complexity of work (e.g., [18]) but are optimized for the vectorized scenario and oblivious transfer (OT). The techniques are similar to prior work, e.g., Kroigaard(DGK) [17].

We confirm these observations by running experiments on two different datasets (MNIST [6] and Arcene [11]). In particular, we show that accuracy of the models trained using privacy preserving linear regression with truncation matches those of plaintext training using standard arithmetic.

**Efficiency Discussion.** The dominating term in the computation cost of Figure 4 is the matrix multiplications in step 5 and 8. In each iteration, each party performs 4 such matrix multiplications, while in plaintext SGD training, according to Equation 2, 2 matrix multiplications are performed. Hence, the computation time for each party is only twice the time for training on plaintext data.

The total communication of the protocol is also nearly optimal. In step 1, each party sends an \( n \times d \) matrix, which is of the same size as the data. In step 4 and 7, \(|B|+d\) elements are sent per iteration. Therefore, the total communication is \( n \cdot d + (|B|+d) \cdot t = nd \cdot (1 + \frac{t}{B} + \frac{d}{B}) \) for each party. In practice, the number of epochs \( E \) is only 2-3 for linear and logistic regressions and 10-15 for neural networks, which is much smaller than \(|B| \) and \( d \). Therefore, the total communication is only a little more than the size of the data. The time spent on the communication can be calculated by dividing the total communication by the bandwidth between the two parties.

**B. The Offline Phase**

We describe how to implement the offline phase as a two-party protocol between \( S_0 \) and \( S_1 \) by generating the desired shared multiplication triplets. We present two protocols for doing so based on linearly homomorphic encryption (LHE) and oblivious transfer (OT). The techniques are similar to prior work (e.g., [18]) but are optimized for the vectorized scenario where we operate on matrices. As a result the complexity of our offline protocol is much better than the naive approach of generating independent multiplication triplets.

Recall that given shared random matrices \( \mathbf{U} \) and \( \mathbf{V} \), the key step is to choose a \( |B| \times d \) submatrix from \( \mathbf{U} \) and a column from \( \mathbf{V} \) and compute the shares of their product. This is repeated \( t \) times to generate \( \mathbf{Z} \). \((\mathbf{Z}')\) is computed in the same way with the dimensions reversed. Thus, for simplicity, we focus on this basic step, where given shares of a \( |B| \times d \) matrix \( \mathbf{A} \), and shares of a \( d \times 1 \) matrix \( \mathbf{B} \), we want to compute shares of a \( |B| \times 1 \) matrix \( \mathbf{C} \) such that \( \mathbf{C} = \mathbf{A} \times \mathbf{B} \).

We utilize the following relationship: \( \mathbf{C} = \mathbf{(A)}_0 \times \mathbf{(B)}_0 + \mathbf{(A)}_0 \times \mathbf{(B)}_1 + \mathbf{(A)}_1 \times \mathbf{(B)}_0 + \mathbf{(A)}_1 \times \mathbf{(B)}_1 \). It suffices to compute \( \langle (\mathbf{A})_0 \times \mathbf{(B)}_1 \rangle \) and \( \langle (\mathbf{A})_1 \times \mathbf{(B)}_0 \rangle \) as the other two terms can be computed locally.

**LHE-based generation.** To compute the shares of the product \( \langle \mathbf{A} \rangle_n \times \langle \mathbf{B} \rangle_1 \), \( S_1 \) encrypts each element of \( \langle \mathbf{B} \rangle_1 \) using an LHE and sends them to \( S_0 \). The LHE can be initiated using the cryptosystem of Paillier [38] or Damgard-Geisler-Kroigaard(DGK) [17]. \( S_0 \) then performs the matrix multiplication on the ciphertexts, with additions replaced by multiplications and multiplications by exponentiations. Finally, \( S_0 \) masks the resulting ciphertexts by random values, and sends them back to \( S_1 \) to decrypt. The protocol can be found in Figure 12 in the Appendix.

Here \( S_1 \) performs \( d \) encryptions, \( |B| \) decryptions and \( S_0 \) performs \(|B| \times d\) exponentiations. The cost of multiplications on the ciphertext is non-dominating and is omitted. The shares of \( \langle \mathbf{(A)}_1 \times \mathbf{(B)}_0 \rangle \) can be computed similarly.

Using this basic step, the overall computation performed in the offline phase per party is \( (|B| + d) \cdot t \) encryptions, \( (|B| + d) \cdot t \) decryptions and \( 2|B| \cdot d \cdot t \) exponentiations. The total communication is \( 2(|B| + d) \cdot t \) ciphertexts, which is much smaller than the size of the data. If we had generated the multiplication triplets independently, the number of encryptions, decryptions and the communication would increase to \( 2|B| \cdot d \cdot t \).

Finally, unlike the online phase, all communication in the offline phase can be done in one interaction.

**OT-based generation.** The shares of the product \( \langle \mathbf{A} \rangle_0 \times \langle \mathbf{B} \rangle_1 \) can also be computed using OTs. We first compute the shares of the product \( \langle a_{ij} \rangle \cdot \langle b_j \rangle \) for all \( i = 1, \ldots, |B| \) and \( j = 1, \ldots, d \). To do so, \( S_1 \) uses each bit of \( b_j \) to select two values computed from \( a_{ij} \) using correlated OTs. In particular, for \( k = 1, \ldots, l \), \( S_0 \) gets the correlation function of \( \langle \text{COT} \rangle \) to \( \langle f_k(x) \rangle = \langle a_{ij} \cdot 2^k + x \rangle \) \( \text{mod } 2^m \) and \( S_0, S_1 \) run \( \text{COT}(r_k, f_k(x); b_j[k]) \). If \( b_j[k] = 0 \), \( S_1 \) gets \( r_k \); if \( b_j[k] = 1 \), \( S_1 \) gets \( a_{ij} \cdot 2^k + r_k \) \( \text{mod } 2^m \). This is equivalent to \( b_j[k] \cdot a_{ij} \cdot 2^k + r_k \) \( \text{mod } 2^m \). Finally, \( S_1 \) gets \( \langle a_{ij} \cdot b_j \rangle = \sum_{k=1}^{l} (b_j[k] \cdot a_{ij} \cdot 2^k + r_k) = a_{ij} \cdot b_j + \sum_{k=1}^{l} r_k \) \( \text{mod } 2^m \), and \( S_0 \) gets \( \langle a_{ij} \cdot b_j \rangle = \sum_{k=1}^{l} (r_k) \) \( \text{mod } 2^m \).

To further improve efficiency, authors of [18] observe that for each \( k \), the last \( k \) bits of \( a_{ij} \cdot 2^k \) are all 0s. Therefore, only the first \( l-k \) bits need to be transferred. Therefore, the message lengths are \( l, l-1, \ldots, 1 \), instead of all being \( l \)-bits. This is equivalent to running \( l \) instances of \( \langle \text{COT}_{l+1} \rangle / 2 \). So far, all the techniques described are as discussed in [18].

The optimization described above does not improve the computation cost of OTs. The reason is that in OT, each message is XORed with a mask computed from the random oracle applied to the selection bit. In practice, the random oracle is instantiated by a hash function such as SHA256 or AES, which at least has 128 bit output. Hence, the fact that \( l \) is only 64 does not reduce time to compute the masks.

We further leverage the matrix structure to improve on this. Note that \( a_{ij}, \ldots, a_{i|B|j} \) are all multiplied by \( b_j \), which means the same selection bit \( b_j[k] \) is used for all \( a_{ij} \). Equivalently, we can view it as using \( b_j[k] \) to select messages with length \( (l-k) \cdot |B| \) bits. Therefore, they can be masked by \( \frac{(l-k) \cdot |B|}{2} \) hash outputs. For a reasonable mini-batch size, each multiplication needs \( \frac{1}{4} \) instances of \( \text{COT}_{128} \). In this way, the total number of hashes can be reduced by 4 times and the total communication can be reduced by half.

Finally, after computing \( \langle a_{ij} \cdot b_j \rangle \), the \( i \)th element of \( \langle \mathbf{A} \rangle_0 \times \langle \mathbf{B} \rangle_1 \) has only 2 matrix multiplications.
Our approaches

The intuition for this choice of activation is as follows (we might be close to the logistic function in certain intervals, but such behavior. In contrast, approximation with low degree polynomials fails to achieve this property. The polynomial might be close to the logistic function in certain intervals, but the tails are unbounded. If a data sample yields a very large input \( u \) to the activation function, \( f(u) \) will be far beyond the [0, 1] interval which affects accuracy of the model significantly in the backward propagation. Our choice of the activation function is also inspired by its similarity to the RELU function.

\[
\begin{align*}
f(x) &= \begin{cases} 
0, & \text{if } x < -\frac{1}{2} \\
\frac{1}{2}, & \text{if } -\frac{1}{2} \leq x < \frac{1}{2} \\
1, & \text{if } x \geq \frac{1}{2}
\end{cases}
\end{align*}
\] (4)

The unique polynomial passing through all these points is symmetric to the original, and evenly spread in the range [0, 1]. Thus we select as many points on the logistic function as the degree of the polynomial.

**C. Privacy Preserving Logistic Regression**

In this section, we present a protocol to support privacy preserving logistic regression. Besides issues addressed for linear regression, the main additional challenge is to compute the logistic function \( f(u) = \frac{1}{1 + e^{-u}} \) on shared numbers. Note that the division and the exponentiation in the logistic function are computed on real numbers, which are hard to support using a 2PC for arithmetic or boolean circuit. Hence, prior work proposes to approximate the function using polynomials [10]. It can be shown that approximation using a high-degree polynomial is very accurate [33]. However, for efficiency reasons, the degree of the approximation polynomial in secure computation is set to 2 or 3, which results in a large accuracy loss of the trained model compared to logistic regression.

**Secure computation friendly activation functions.** Instead of using polynomials to approximate the logistic function, we propose a new activation function that can be efficiently computed using secure computation techniques. The function is described in Equation 4 and drawn in Figure 5(a).

<table>
<thead>
<tr>
<th>Logistic</th>
<th>Our approaches</th>
<th>Polynomial Approx.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>98.64</td>
<td>98.62</td>
</tr>
<tr>
<td>Arcene</td>
<td>86</td>
<td>86</td>
</tr>
</tbody>
</table>

(Figure 5(b)) used in neural networks. One of the justifications used for replacing logistic function by the RELU function in neural networks is that the subtraction of two RELU functions with an offset yields the activation function of Equation 4 which in turn, closely imitates the logistic function.

Once we use the new activation function, we have two choices when computing the backward propagation. We can either use the same update function as the logistic function (i.e. continue to compute the partial derivative using the logistic function), or compute the partial derivative of the new function and substitute it into the update function. We test both options and find out that the first approach yields better accuracy matching that of using the logistic function. Therefore, we will use the first approach in the rest of the paper. We believe one reason for lower accuracy of the second approach is that by replacing the activation function, the cross entropy cost function is no longer convex; using the first approach, the update formula is very close to training using the distance cost function, which might help produce a better model. Better theoretical analysis of these observations is an interesting research direction.

To justify our claims, we compare the accuracy of the produced model using our approaches with logistic regression, and polynomial approximation with different degrees. For the polynomial approximation, we fix the constant to \( \frac{1}{2} \) so that \( f(0) = \frac{1}{2} \). Then we select as many points on the logistic function as the degree of the polynomial. The points are symmetric to the original, and evenly spread in the range of the data value (e.g., [0,1] for MNIST, [0,1000] for Arcene). The unique polynomial passing through all these points is selected for approximation. The test is run on the MNIST data with mini-batch size \(|B| = 128\). The series of random mini-batches are the same for all approaches. Here we train the models on plaintext data only. As shown in Table I, the performance of our approaches is much better than polynomial approximation. In particular, our first approach reaches almost the same accuracy (98.62\%) as logistic regression, and our second approach performs slightly worse. On the contrary, when a degree 3 polynomial is used to approximate the logistic function, the accuracy can only reach 42.17\%, which is even worse than a linear activation function. The reason is that the tails diverge even faster than a linear activation function. When the degree is 5, the accuracy can reach 84\%; when the degree is 10, the accuracy finally matches that of logistic regression. However, computing a polynomial of degree 10 in secure computation introduces a high overhead. Similar effects are also verified by experiments on the Arcene dataset.

Nevertheless, we suggest further work to explore more MPC-friendly activation functions that can be computed efficiently using simple boolean or arithmetic circuits.

**The privacy preserving protocol.** The new activation function proposed above is circuit friendly. It only involves testing whether the input is within the \([-1/2, 1/2]\) interval. However,
applying Yao’s garbled circuit protocol naively to the whole logistic regression is very inefficient. Instead, we take advantage of techniques to switch between arithmetic sharing and Yao sharing proposed in [18]. The observation is that as mentioned in Section II-A, the only difference between the SGD for logistic regression and linear regression is the application of an extra activation function in each forward propagation. Therefore, following the same protocol for privacy preserving linear regression, after computing the inner product of the input data and the coefficient vector, we switch the arithmetic sharing to a Yao sharing and evaluate the activation function using a garbled circuit. Then, we switch back to arithmetic sharing and continue the backward propagation.

Here, we propose a more involved protocol to further optimize the circuit size, the number of interactions and the number of multiplication triplets used. Note that if we let \( b_1 = 0 \) if \( u + \frac{1}{2} \geq 0 \), \( b_1 = 1 \) otherwise, and \( b_2 = 0 \) if \( u - \frac{1}{2} \geq 0 \), \( b_2 = 1 \) otherwise, then the activation function can be expressed as 
\[
    f(u) = (-b_2) + (b_2 \land (-b_1))u.
\]
Therefore, given \( \langle u \rangle \), we construct a garbled circuit that takes the bits of \( \langle u + \frac{1}{2} \rangle \) and \( \langle u \rangle \) as input, adds them and sets \( b_1 \) as the most significant bit (msb) of the result (the msb indicates whether a value is positive or negative). To be more precise, the \( \langle u + \frac{1}{2} \rangle \) value is represented in the field and scaled to have the same number of bit representing the fractional part as \( u \). In particular, since \( u \) is the product of two values before truncation, \( \langle u \rangle \) is expressed as \( \frac{1}{2} \cdot 2^u \), where \( l_u \) is the sum of bit-length of the decimal part in the data \( x \) and the coefficient \( w \), but we use \( +\frac{1}{2} \) for ease of presentation. \( b_2 \) is computed in a similar fashion. Instead of computing the rest of the function in the garbled circuit which would require a linear number of additional AND gates, we let the garbled circuit output the Yao sharing (output labels) of the bits \( (-b_2) \) and \( b_2 \land (-b_1) \). We then switch to boolean sharing of these bits and use them in two OTs to compute \( \langle (-b_2) + (b_2 \land (-b_1))u \rangle \) and continue with the rest of the training. The detailed protocol is described in Figure 13 in the Appendix. The following theorem states the security of privacy-preserving logistic regression. The proof is omitted due to lack of space but we note that it is implied by the security of the secret sharing scheme, the garbling scheme, and OT.

**Theorem 3.** Consider a protocol where clients distribute arithmetic shares of their data among two servers who run the protocol of Figure 13 and send the output to clients. Given a secure garbling scheme, in the \( F_{\text{offline}} \) and \( F_{\text{ot}} \) hybrid model, this protocol realizes the ideal functionality \( F_{\text{nl}} \) of Figure 3 for the logistic regression function, in presence of a semi-honest admissible adversary (see section III).

**Efficiency Discussion.** The additional overhead of the logistic regression is very small. Most of the steps are exactly the same as the linear regression protocol in Section IV-A. In addition, one garbled circuit protocol and 3 extra OTs are performed in each forward propagation. The garbled circuit performs two additions and one AND, yielding a total \( 2l - 1 \) AND gates for each value \( u \). The base OT for OT extension can be performed in the offline phase. Therefore, the total communication overhead is \( |B| \cdot t \cdot ((2l - 1) \cdot 2\lambda + 3l) \) for each party. Note that the garbled circuit and the messages in OTs from \( S_0 \) can be sent simultaneously to \( S_1 \). Thus, the logistic regression only introduces one more interaction per iteration, and yields a total of \( 3t \) interactions between the two parties. No extra multiplication triplets are required since we do away with arithmetic operations for the activation function.

**D. Privacy Preserving Neural Network Training**

All techniques we proposed for privacy preserving linear and logistic regression naturally extend to support privacy preserving neural network training. We can use the RELU function as the activation function in each neuron and the cross entropy function as the cost function. The update function for each coefficient in each neuron can be expressed in a closed form as discussed in Section II-A. All the functions in both forward and backward propagation, other than evaluating the activation function and its partial derivative, involve only simple additions and multiplications, and are implemented using the same techniques discussed for linear regression. To evaluate the RELU function \( f(u) = (u > 0) \cdot u \) and its derivative \( f'(u) = (u > 0) \), we use the same approach as for logistic regression by switching to Yao sharing. The garbled circuit simply adds the two shares and outputs the most significant bit, which is even simpler than the circuit we needed for our new logistic function. Note that both the RELU function and its derivative can be evaluated together in one iteration, and the result of the latter is used in the backward propagation.

We also propose a secure computation friendly alternative to the softmax function \( f(u_i) = \frac{e^{-u_i}}{\sum_{j=1}^{m} e^{-u_j}} \). We first replace the exponentiations in the numerator with RELU functions such that the results remain non-negative as intended by \( e^{-u_i} \). Then, we compute the total sum by adding the outputs of all RELU functions, and divide each output by the total sum using a division garbled circuit. In this way, the output is guaranteed to be a probability distribution. In the experiment section we show that using an example neural network and training on the MNIST dataset, the model trained by Tensorflow (with softmax) can reach 94.5% accuracy on all 10 classes, while we reach 93.4% using our proposed function. We omit a detailed description of the protocol due to space limits.

As we observe in our experiments, the time spent on garbled circuits for the RELU functions dominates the online training time. Therefore, we also consider replacing the activation function with the square function \( f(u) = u^2 \), as recently proposed in [22] but for prediction only. (We still use RELU functions for approximating softmax.) With this modification, we can reach 93.1% accuracy. Now a garbled circuit computing a RELU function is replaced by a multiplication on shared values, thus the online efficiency is improved dramatically. However, this approach consumes more multiplication triplets and increases cost of the offline phase.

**Efficiency Discussion.** In the online phase, the computation complexity is twice that of the plaintext training for the matrix arithmetic operations, plus the overhead of evaluating the RELU functions and divisions using garbled circuits and OTs. In our experiments, we use the division circuit from

\( \text{If the sum is 0, which means all the results of RELU functions are 0s, we assign the same probability to each output. This is done with a garbled circuit.} \)
the EMP toolkit [3], which has $O(l^2)$ AND gates for $l$-bit numbers. The total communication is the sum of the sizes of all matrices involved in the matrix multiplication and elementwise multiplication, which is $O(t \cdot \sum_{i=1}^{m} (|B| \cdot d_{i-1} + d_{i-1} \cdot d_{i})).$

The total number of iterations is $\sum_{i=1}^{m} t_i$.

In the offline phase, the total number of multiplication triplets is increased by a factor of $O(\sum_{i=1}^{m} d_m)$ compared to regression, which is exactly the number of neurons in the neural network. Some of the multiplication triplets can be generated in the matrix form for online matrix multiplication. Others need to be generated independently for element-wise multiplications. We show the cost experimentally in Section VI-C.

### V. Client-Aided Offline Protocol

As expected and shown by the experiments, the main bottleneck in our privacy preserving machine learning protocols is the offline phase. It involves a large number of cryptographic operations such as OT or LHE, which are much slower than simple addition and multiplication in a finite field in the online phase. This motivates us to explore an alternative way of generating multiplication triplets. In particular, we can let the clients generate the multiplication triplets. Since the clients need to secretly share their data in the first place, it is natural to further ask them to secretly share some extra multiplication triplets. Now, these multiplication triplets can be generated in a trusted way with no heavy cryptographic operations, which improves the efficiency significantly. However, despite its benefits, it changes the trust model and introduces some overhead for the online phase. A detailed discussion of the client-aided triplet generations and the analysis of the overhead can be found in Appendix F.

**The new security model.** The security model also changes with the client-aided offline phase. We only informally sketch the differences here. Previously, a client is only responsible to upload his own data, and thus the server clearly cannot learn any extra information when he colludes with a subset of clients. Now, as the clients are also generating multiplication triplets, if a subset of clients are colluding with one server, they may reconstruct the coefficient vector in an iteration, which indirectly leaks information about the data from honest clients. Therefore, in the client-aided scenario, we change the security model to not allow collusion between a server and a client. Similar models have appeared in prior work. E.g., in [21], the CSP provides multiplication triplets to the clients to securely compute inner products of their data. If a client is colluding with the CSP, he can immediately learns others’ data. Our client-aided protocols are secure under the new model, because the clients learn no extra information after uploading the data and the multiplication triplets. As long as the multiplication triplets are correct, which is the case for semihonest clients we consider, the training is correct and secure.

### VI. Experimental Results

We implement a privacy preserving machine learning system based on our protocols and show the experimental results in this section.

**The Implementation.** The system is implemented in C++. In all our experiments, the field size is set to $2^{64}$. Hence, we observe that the modulo operations can be implemented using regular arithmetics on the unsigned long integer type in C++ with no extra cost. This is significantly faster than any number-theoretic library that is able to handle operations in arbitrary fields. E.g., we tested that an integer addition (multiplication) is $100 \times$ faster than a modular addition (multiplication) in the same field implemented in the GMP [5] or the NTL [7] library. More generally, any element in the finite field $\mathbb{Z}_{2^l}$ can be represented by one or several unsigned long integers and an addition (multiplication) can be calculated by one or several regular additions (multiplications) plus some bit operations. This enjoys from the same order of speedup compared to using general purpose number theoretic libraries. We use the Eigen library [2] to handle matrix operations. OTs and garbled circuits are implemented using the EMP toolkit [3]. It implements the OT extension of [11], and applies free XOR [30] and fixed-key AES garbling [12] optimizations for garbled circuits. Details can be found in [45]. We use the cryptosystem of DGK [17] for LHE, implemented by Demmler et. al. in [18].

**Experimental settings.** The experiments are executed on two Amazon EC2 c4.8xlarge machines running Linux, with 60GB of RAM each. For the experiments on a LAN network, we host the two machines in the same region. The average network delay is 0.17ms and the bandwidth is 1GB/s. The setting is quite representative of the LAN setting, as we further tested that two computers connected by a cable have similar network delay and bandwidth. For the experiments on a WAN network, we host the two machines in two different regions, one in the US east and the other in the US west. The average network delay is 72ms and the bandwidth is 9MB/s. We collected 10 runs for each data point in the results and report the average.

Our experiments in the LAN setting capture the scenario where the two servers in our protocols have a high-bandwidth/low-latency network connection, but otherwise are not administered/controlled by the same party. The primary reason for reporting experiments in the LAN setting is more accurate benchmarking and comparison as the majority of prior work, including all previous MPC implementations for machine learning only report results in the LAN setting. Moreover, contrasting our results in the LAN and WAN setting highlights the significance of network bandwidth in our various protocols. For example, as our experiments show, the total time for the...
offline phase in the LAN and WAN setting are very close when using LHE techniques to generate multiplication triplets while there is a significant gap between the two when using OT extension (see Table II).

Furthermore, while the LAN setting is understandably not always a realistic assumption, there are scenarios where a high bandwidth link (or even a direct dedicated link) between the two servers is plausible. For example, in payment networks, it is not uncommon for the various involved parties (issuing Banks, acquiring Banks, large merchants, and payment networks) to communicate over fast dedicated links connecting them. Similarly, in any international organization that needs to abide by different privacy regulations and data sovereignty restrictions, the two servers may indeed be connected using a high bandwidth direct link but be administered in different countries. In such scenarios, the logical, administrative, or legal separation of the two servers plays a more significant role.

**Offline vs. Online.** We report experimental numbers for both the offline and the online phase of our protocols separately, but only use total costs (online + offline) when comparing to related work. The offline phase includes all computation and communication that can be performed without presence of data, while the online phase consists of all data-dependent steps of the protocol. Optimizing the online cost is useful for application scenarios where a fast turn-around is required. In particular, when using our protocols for privacy-preserving prediction (e.g. fraud detection), new data needs to be classified with low latency and high throughput. Indeed, we run a set of experiments to demonstrate that online cost of privacy-preserving prediction can be made fast enough to run for latency critical applications (See Table IV). Similarly, when training small models dynamically and on a regular basis, it is important to have high online efficiency. In contrast, when training large models (e.g. a large neural networks), the separation of the offline and the online costs is less important.

**Data sets.** In our experiments, we use the following datasets. The MNIST dataset [6] contains images of handwritten digits from “0” to “9”. It has 60,000 training samples, each with 784 features representing 28×28 pixels in the image. Each feature is a grayscale between 0~255. The Gisette dataset [4], [25] contains images of digits “4” and “9”. It has 13,500 samples and 5,000 features between 0~1.000. We also use the Arcene dataset [1], [25]. It contains mass-spectrometric data and is used to determine if the patient has cancer. There are 200 data samples with 10,000 features. Each value is between 0 and 1000. All of the datasets are drawn from the real world.

**A. Experiments for Linear Regression**

We start with the experimental results for our privacy preserving linear regression protocols in different settings, and compare it with previous privacy preserving solutions.

**Online phase.** To examine how the the online phase scales, we run experiments on datasets with size \( (n) \) from 1,000 to 1,000,000 and \( d \) from 100 to 1,000. When \( n \leq 60000 \) and \( d \leq 784 \), the samples are directly drawn from the MNIST dataset. When \( n \) and \( d \) are larger than that of MNIST, we duplicate the dataset and add dummy values for missing features. Note that when \( n, d, E \) are fixed, the actual data used in the training does not affect the running time.

Figure 6a shows the results in the LAN setting. “PP Linear 1” denotes the online phase of our privacy preserving linear regression with multiplication triplets in matrix form, and “PP Linear 2” denotes the online phase of the client-aided variant. The running time reported is the total online time when two servers are running simultaneously and interacting with each other. The two parties take roughly the same time based on our experiments. The learning rate is predetermined and we do not count the time to find an appropriate learning rate in the figures. The number of features is fixed to 784 and \( n \) varies from 1,000 to 1,000,000.

As shown in the figure, the online time of our linear regression is very fast in the LAN setting. In particular, it only takes 22.3s to train a linear model securely on 1 million data samples with 784 features each. From 22.3s needed for privacy preserving training, only a small portion, namely less than 2s, is spent on the network delay for the interactions. The communication time to transfer the data is negligible given the high bandwidth of the LAN network. Our second protocol using client-generated multiplication triplets has an overhead of roughly 3.5×. In particular, it takes 77.6s to train the model with \( n = 1,000,000 \) and \( d = 784 \). As shown in Figure 6a and 6b, the running time of our protocol scales linearly with both \( n \) and \( d \). We also observe that the SGD for linear and logistic regressions on all the datasets we tested always converges within the first epoch, and terminate after the second epoch, which confirms that the SGD is very effective and efficient in practice.

Figure 6c shows the corresponding performance on a WAN network. The running time of our privacy preserving protocols increase significantly. In particular, our first protocol takes 2291.8s to train the model when \( n = 1,000,000 \) and \( d = 784 \). The reason is that now the network delay is the dominating factor in the training time. The computation time is exactly the same as the LAN setting, which is around 20s; the
communication time is still negligible even under the bandwidth of the WAN network. The total running time is almost the same as the network delay times the number of iterations. Our second protocol is still roughly $3.3 \times$ slower than the first protocol, but the reason is different from the LAN setting. In the WAN setting, this overhead comes from the increment of the communication, as explained in Section V. Even under this big network delay in the WAN network, as we will show later, the performance of our privacy preserving machine learning is still orders of magnitude faster than the state of the art. Besides, it is also shown in Figure 6c that the training time grows linearly with the number of the samples in WAN networks. However, in Figure 6d, when fixing $n = 10,000$, the training time of our first protocol only grows slightly when $d$ increases, which again has to do with the fact that number of interactions is independent of $d$. The overhead of our second protocol compared to the first one is increasing with $d$, because the communication grows linearly with $d$ in the second protocol. When $d = 100$, the training time is almost the same, as it is dominated by the interaction; when $d = 1000$, the training time is $4 \times$ slower because of the overhead of communication.

We also show that we can improve the performance in the WAN setting by increasing the mini-batch size, in order to balance the computation time and the network delay. Figure 7 shows the result of this parameter tweaking. We let $n = 10,000$ and $d = 784$ and increases $|B|$ to measure its effect on performance. As shown in the figure, the running time of the online phase is decreasing when we increase the mini-batch size. In particular, it takes 6.8s to train the model in our first protocol when $|B| = 512$, which is almost 4 times faster than the time needed when $|B| = 128$. This is because when the number of epochs is the same, the number of iterations (or interactions) is inverse proportional to the mini-batch size. When the mini-batch size is increasing, the computation time remains roughly unchanged, while the time spent on interaction decreases. However, the running time cannot always keep decreasing. When the computation time becomes dominating, the running time will remain unchanged. Furthermore, if $|B|$ is set too large, the number of iterations is too small in an epoch such that the model may not reach the optimum as fast as before, which may result in an increase in the number of necessary epochs $E$ which itself can affect the performance. Mini-batch size is usually determined considering the speed up of vectorization, parallelization and robustness of the model in plaintext training.

In the privacy preserving setting, we suggest that one should also take the network condition into consideration and find an appropriate mini-batch size to optimize the training time.

**Offline phase.** The performance of the offline phase is summarized in Table II. We report the running time on LAN and WAN networks and the total communication for OT-based and LHE-based multiplication triplets generation. For the client-aided setting, we simulate the total computation time by generating all the triplets on a single machine. We report its total time and total communication, but do not differentiate between the LAN and WAN settings, since in practice the data would be sent from multiple clients with different network conditions. As a point of reference, we also include the dataset size assuming each value is stored as 64-bit decimal number. We vary $n$ from 1000 to 100,000 and $d$ from 100 to 1000. The mini-batch size is set to 128 and the number of epochs is set to 2, as we usually only need 2 epochs in the online phase. If more epochs are needed, all the results reported in the table clearly grow linearly with the number of epochs.

As shown in the table, the LHE-based multiplication triplets generation is the slowest among all approaches. In particular, it takes 1600.9s for $n = 10,000$ and $d = 1000$. The reason is that each basic operation in LHE, i.e., encryption, and decryption are very slow, which makes the approach impractical. E.g., one encryption takes 3ms, which is around $10,000 \times$ slower than one OT (when using OT extension). However, the LHE-based approach yields the best communication. As calculated in Section IV-B, the asymptotic complexity is much smaller than the dataset size. Taking the large ciphertext (2048 bits) into consideration, the overall communication is still on the same order as the dataset size. This communication introduces almost no overhead when running on both LAN and WAN networks. Unlike the online phase, the offline phase only requires 1 interaction and hence the network delay is negligible.

The performance of the OT-based multiplication triplets generation is much better in the LAN setting. In particular, it only takes 80.0s for $n = 10,000$ and $d = 1000$. It introduces a huge overhead on the communication, namely 19GB while the data is only 76MB. This communication overhead makes the running time much slower on WAN networks. Because of this communication overhead, which is the major cost of OT, the total running time is even slower than the LHE-based generation on WAN networks.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$d$</th>
<th>LAN</th>
<th>WAN</th>
<th>Communication</th>
<th>LAN</th>
<th>WAN</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>100</td>
<td>23.8s</td>
<td>24.0s</td>
<td>2MB</td>
<td>0.86s</td>
<td>24.2s</td>
<td>190MB</td>
</tr>
<tr>
<td>500</td>
<td>500</td>
<td>83.9s</td>
<td>84.8s</td>
<td>6MB</td>
<td>3.8s</td>
<td>210.6s</td>
<td>1GB</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>158.4s</td>
<td>163.2s</td>
<td>10MB</td>
<td>7.9s</td>
<td>163.2s</td>
<td>1.9GB</td>
</tr>
<tr>
<td>$n$</td>
<td>$d$</td>
<td>LAN</td>
<td>WAN</td>
<td>Communication</td>
<td>LAN</td>
<td>WAN</td>
<td>Communication</td>
</tr>
<tr>
<td>10,000</td>
<td>100</td>
<td>248.4s</td>
<td>252.9s</td>
<td>20MB</td>
<td>7.9s</td>
<td>420.2s</td>
<td>1.9GB</td>
</tr>
<tr>
<td>500</td>
<td>500</td>
<td>869.1s</td>
<td>890.2s</td>
<td>60MB</td>
<td>39.2s</td>
<td>2119.1s</td>
<td>9.5GB</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>1600.9s</td>
<td>1627.0s</td>
<td>100MB</td>
<td>80.8s</td>
<td>4097.1s</td>
<td>19GB</td>
</tr>
<tr>
<td>$n$</td>
<td>$d$</td>
<td>LAN</td>
<td>WAN</td>
<td>Communication</td>
<td>LAN</td>
<td>WAN</td>
<td>Communication</td>
</tr>
<tr>
<td>100,000</td>
<td>100</td>
<td>2437.1s</td>
<td>2478.1s</td>
<td>200MB</td>
<td>88.0s</td>
<td>4125.1s</td>
<td>19GB</td>
</tr>
<tr>
<td>500</td>
<td>500</td>
<td>8721.5s</td>
<td>8782.4s</td>
<td>600MB</td>
<td>577.9s</td>
<td>20006*</td>
<td>95GB</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>16000s*</td>
<td>161000s*</td>
<td>1000MB</td>
<td>794.0s</td>
<td>40000s*</td>
<td>190GB</td>
</tr>
</tbody>
</table>

**TABLE II:** Performance of the offline phase. $|B| = 128$ and $E = 2$. * means estimated visa extrapolation.

![Fig. 7: Performance of online cost of linear regression on WAN with different mini-batch sizes. $n = 10,000$, $d = 784$.](image)
Finally, the client-aided multiplication triplets generation is the fastest because no cryptographic operation is involved. It only takes 4.0s for \(n = 10,000\) and \(d = 1000\). The overhead on the total communication is only around 9 times the dataset size which is acceptable in practice.

It is also shown in Table II that all the running times grow roughly linearly\(^7\) with both \(n\) and \(d\), which agrees with the asymptotic complexity derived in Section IV-B.

Combining the results presented for both the online and the offline phase, our system is still quite efficient. E.g., in the LAN setting, when client-aided multiplication triplets are used, it only takes 1.0s for our privacy preserving linear regression in the online phase, with \(n = 10,000\) and \(d = 1000\). The total time for the offline phase is only 4.0s, which would be further distributed to multiple clients in practice. When OT-based generation is used, the online phase takes 0.28s and the offline phase takes 8.0s.

**Comparison with prior work.** As surveyed in Section I-B, privacy preserving linear regression was also considered by [37] (NWI\(^+\)13) and [21] (GSB\(^+\)16) in a similar two-server setting. Instead of using the SGD method, these two papers propose to calculate the optimum by solving a linear system we described in Section II-A. We show that the model trained by the SGD method can reach the same accuracy in Table V in the Appendix, as reflected in Table II. Besides, the offline phase of our protocols is data independent.

The reason why the cost of NWI\(^+\)13 and GSB\(^+\)16 is so high when \(n\) is small is that the size of the garbled circuit to solve the linear system only depends on \(d\). Even if there is only 1 data sample, the time of the second step for \(d = 500\) is around 90,000s in NWI\(^+\)13 and 30,000s in GSB\(^+\)16.

Note that the gap between our scheme and prior work will become even larger as \(d\) increases, as the running time is linear in \(d\) in our schemes and quadratic or cubic in the two prior schemes. In addition, all the numbers reported for the two prior works were obtained on a network with 1 Gbps bandwidth [21] which is close to our LAN setting. Indeed, the garbled circuit work were obtained on a network with 1 Gbps bandwidth [21].

For comparison, we use the numbers reported in [21, Table 1, Figure 6] . As the performance of the first step of NWI\(^+\)13 is not reported in the table, we implement it on our own using Paillier’s encryption [38] with batching, which is the same as used in the protocol of NWI\(^+\)13. For the first step in GSB\(^+\)16, we use the result of the total time for two clients only in [21, Table 1] with \(d = 500\), which is the fastest\(^8\), for the second step in GSB\(^+\)16, we use the result for CGD with 15 iterations in [21, Figure 6] with \(d = 500\). We sum up the running time of our offline and online phase, and sum up the running time of the first and the second step in NWI\(^+\)13 and GSB\(^+\)16, and report the total running time of all parties in all the schemes.

In Figure 8a, we compare the performance of the scheme in NWI\(^+\)13 and our schemes with OT-based and LHE-based multiplication triplets generation, executed in both LAN and WAN settings. As shown in the figure, the performance is improved significantly. For example, when \(n = 100,000\) and \(d = 500\), even our LHE-based protocol in both LAN and WAN settings has a 54× speedup. The OT-based protocol is 1270× faster in the LAN setting and 24× faster in the WAN setting. We could not execute the first step of NWI\(^+\)13 for \(n \geq 10,000\) and the dotted line in the figure is our extrapolation.

We further compare the performance of the scheme in GSB\(^+\)16 and our scheme with client-generated multiplication triplets in Figure 8b, as they are both secure under the assumption that servers and clients do not collude. As shown in the figure, when \(n = 100,000\) and \(d = 500\), our scheme has a 31× speedup in WAN setting and a 1110× speedup in LAN setting. As the figure is in log-log scale, the larger slope of the growth of the running time for our schemes does not mean we will be slower eventually with large enough \(n\). It means that the relative speedup is decreasing, but, in fact, the absolute difference between the running time of our scheme and GSB\(^+\)16 keeps increasing.

Finally, NWI\(^+\)13 only supports horizontally partitioned data, where each client holds one or multiple rows of the data matrix; GSB\(^+\)16 only supports vertically partitioned data with 2 ~ 5 clients, where each client holds one entire column of the data. Our schemes can support arbitrary partitioning of the data. Besides, the offline phase of our protocols is data independent. The servers and the clients can start the offline phase with basic knowledge on the bounds of the dataset size, while the bulk of the computation in the two prior work need to be performed after obtaining the data.
We are not aware of any prior work in this security model with an implementation. We are the first to implement a scalable system for privacy preserving logistic regression.

C. Experiments for Neural Networks

We also implemented our privacy preserving protocol for training an example neural network on the MNIST dataset. The neural network has two hidden layers with 128 neurons in each layer. We experiment with both the RELU and the square function as the activation function in the hidden layers and our proposed alternative to softmax function in the output layer. The neural network is fully connected and the cost function is the cross entropy function. The labels are represented as *hot vectors* with 10 elements, where the element indexed by the digit is set to 1 while others are 0s. We run our system on a LAN network and the performance is summarized in Table III. |

<table>
<thead>
<tr>
<th>Protocol 1</th>
<th>Protocol 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offline</td>
<td>Online</td>
</tr>
<tr>
<td>RELU</td>
<td>290,000s*</td>
</tr>
<tr>
<td>Square</td>
<td>320,000s*</td>
</tr>
</tbody>
</table>

Fig. 9: Online cost of privacy preserving logistic regression in the standard and client-aided setting. $|B|$ is set to 128. Figure (a), (b) are for LAN network and Figure (c), (d) are for WAN network. Figure (a) and (c) are in log-log scale, $d = 784$. Figure (b) and (d) are in regular scale, $n = 10,000$.

B. Experiments for Logistic Regression

In this section, we review experimental results for our privacy preserving logistic regression protocol. Since this protocol does not require any additional multiplication triplets, the offline phase has the exact same cost as linear regression.

As shown in Figure 9, our privacy preserving logistic regression introduces some overhead on top of the linear regression. Specifically, in Figure 9a, when $n = 1,000,000$ and $d = 784$, our protocol 1 using OT-based or LHE-based multiplication triplets takes 149.7s in the online phase. This overhead is introduced purely by the extra garbled circuit to compute our logistic function. The fact that a small additional garbled circuit introduces a $7 \times$ overhead, serves as evidence that the running time would be much larger if the whole training was implemented in garbled circuits. Our protocol 2, using client-generated multiplication triplets, takes 180.7s as no extra multiplication triplet is used in logistic regression and the garbled circuit is an additive overhead, no matter which type of multiplication triplet is used. The training time grows linearly with both $n$ and $d$, as presented in Figure 9a and 9b.

Figure 9c and 9d shows the result on a WAN network. The time spent on the interactions is still the dominating factor. When $n = 1,000,000$ and $d = 784$, it takes around 6623s for our first protocol, and 10213s for the second. Compared to privacy preserving linear regression, one extra interaction and extra communication for the garbled circuit is added per iteration. We can also increase the mini-batch size $|B|$ to balance the computation and interactions and improve the performance. We omit the result due to page limit.

To further show the scalability of our system, we run the online part of our privacy preserving logistic regression on the Gisette dataset with 5000 features and up to 1,000,000 samples on a LAN network. It takes 268.9s using our first protocol and 623.5s using the second one. The trained model can reach an accuracy of 97.9% on the testing dataset.

We thank Jing Huang from Visa Research for helpful discussions on machine learning, and Xiao Wang from University of Maryland for his help on the EMP toolkit. The work was partially supported by NSF grants #5245250 and #5246010.

ACKNOWLEDGEMENTS
the environment outputs a single bit. The adversary can also interact arbitrarily with the environment — without loss of generality the adversary is a dummy adversary which simply forwards all received protocol messages to the environment and acts in the protocol as instructed by the environment.

Security is defined by comparing a real and ideal interaction. Let REAL[Z, A, π, λ] denote the final (single-bit) output of the environment Z when interacting with adversary A and honest parties who execute protocol π on security parameter λ. This interaction is referred to as the real interaction involving protocol π.

In the ideal interaction, parties simply forward the inputs they receive to an uncorruptable functionality machine and forward the functionality’s response to the environment. Hence, the trusted functionality performs the entire computation on behalf of the parties. The target ideal functionality \( F_{ml} \) for protocols is described in Figure 3. Let IDEAL[Z, S, F_{ml}, \lambda] denote the output of the environment Z when interacting with adversary S and honest parties who run the dummy protocol in presence of functionality F on security parameter \( \lambda \).

We say that a protocol π securely realizes a functionality \( F_{ml} \) if for every admissible adversary A attacking the real interaction (without loss of generality, we can take A to be the dummy adversary), there exists an adversary \( S \) (called a simulator) attacking the ideal interaction, such that for all environments \( Z \), the following quantity is negligible (in \( \lambda \)):

\[
P_T[\text{REAL}[Z, A, \pi, \lambda] = 1] - P_T[\text{IDEAL}[Z, S, F_{ml}, \lambda] = 1] = \text{negligible}.
\]

Intuitively, the simulator must achieve the same effect (on the environment) in the ideal interaction that the adversary achieves in the real interaction. Note that the environment’s view includes (without loss of generality) all of the messages that honest parties sent to the adversary as well as the outputs of the honest parties.

**APPENDIX B**

**PROOF OF SMALL TRUNCATION ERROR**

**Theorem.** In field \( \mathbb{Z}_p \), let \( x \in \{0, 2^l\} \cup \{2^l - 2^l, 2^l\} \), where \( l > l_x + 1 \) and given shares \( (x)_0, (x)_1 \) of \( x \), let \( (\{x\})_0 = \{\{x\}_0\} \) and \( (\{x\})_1 = 2^l - (2^l - \{x\}_1) \). Then with probability \( 1 - 2^{2l_x + 1 - 4} \), \( \text{Rec}^A((\{x\})_0, (\{x\}_1)) \in \{\{x\} - 1, \{x\}, \{x\} + 1\} \), where \( \lfloor \cdot \rfloor \) denotes truncation by \( l_D \leq l_x \) bits.

**Proof.** Let \( (x)_0 = x + r \mod 2^l \), where \( r \) is uniformly random in \( \mathbb{Z}_p \), then \( \{x\}_1 = 2^l - r \). We decompose \( x \) as \( x = r_1 \cdot 2^{l_D} + r_2 \), where \( 0 \leq r_2 < 2^{l_D} \) and \( 0 \leq r_1 < 2^{l_D} \). We prove that if \( 2^l - r < 2^l - 2^l \), \( \text{Rec}^A((\{x\}_0, (\{x\}_1)) \in \{\{x\} - 1, \{x\}, \{x\} + 1\} \). Consider the following two cases.

**Case 1:** If \( 0 \leq x < 2^{l_x} \), then \( 0 < x + r < 2^l \) and \( (x)_0 = x + r \), without modulo. Let \( x = x_1 \cdot 2^{l_D} + x_2 \), where \( 0 \leq x_2 < 2^{l_D} \) and \( 0 \leq x_1 < 2^{l_x - l_D} \). Then we have \( x + r = (x_1 + r_1) \cdot 2^{l_D} + (x_2 + r_2) = (x_1 + r_1 + c) \cdot 2^{l_D} + (x_2 + r_2 + c \cdot 2^{l_D}) \), where \( c \) is \( 0 \) if \( x_2 + r_2 < 2^{l_D} \) and \( 1 \) otherwise. After the truncation, \( (\{x\}_1 = x_1 + c \cdot \{x\}_1 \) and \( (\{x\}_1 = 2^l - r_1 - c \cdot \{x\}_1 \). Therefore, \( \text{Rec}^A((\{x\}_0, (\{x\}_1)) = x + c \).

**Case 2:** If \( 2^l - 2^l \leq x < 2^l \), then \( x + r \geq 2^l \) and \( (x)_0 = x + r - 2^l \). Let \( x = 2^l - x_1 \cdot 2^{l_D} - x_2 \), where \( 0 \leq x_2 < 2^{l_D} \) and \( 0 \leq x_1 < 2^{l_x - l_D} \). We have \( x + r = (x_1 + r_1) \cdot 2^{l_D} + (r_2 - x_2) = (r_1 - x_1 - c) \cdot 2^{l_D} + (r_2 - x_2 + c \cdot 2^{l_D}) \), where the carry \( c = 0 \) if \( r_2 > x_2 \) and \( c = 1 \) otherwise. After the truncation, \( (\{x\}_0 = x_1 + r + c \) and \( (\{x\}_1 = 2^l - r_1 - c \cdot \{x\}_1 \). Therefore, \( \text{Rec}^A((\{x\}_0, (\{x\}_1)) = x + c \).

Finally, the probability that our assumption holds, i.e. the probability of a random \( r \) being in the range \( [2^{l_x}, 2^l - 2^l] \) is \( 1 - 2^{2l_x + 1 - 4} \).

Theorem 1 can be extended to a prime field \( \mathbb{Z}_p \) in a natural way by replacing \( 2^l \) with \( p \) in the proof. We also note that the truncation does not affect security of the secret sharing as the shares are truncated independently by each party without any interaction.

**APPENDIX C**

**EFFECT OF TRUNCATION ON TRAINING FOR MNIST AND AUREC Dataset**

We run our privacy preserving linear regression protocol with the truncation technique on the MNIST dataset [6] consisting of images of handwriting digits and compare accuracy of the trained model to plaintext training with standard decimal numbers operations. The mini-batch size is set to \( |B| = 128 \) and the learning rate is \( \alpha = 2^{-7} \). The input data has 784 features, each a gray scale of a pixel scaled between 0 and 1, represented using 8 decimal bits. We set the field to \( \mathbb{Z}_{2^6} \). For a fair comparison, coefficients are all initialized to 0s and the same sequence of the mini-batch indices are used for all trainings. To simplify the task, we change the labels to be 0 for digit “0” and 1 for non-zero digits. In Figure 10, the x-axis is the number of iterations of the SGD algorithm and the y-axis is the accuracy of the trained model on the testing dataset. Here we reconstruct the coefficient vector after every iteration in our protocol to test the accuracy. As shown in Figure 10, when we use 13 bits for the fractional part of \( \mathbf{w} \), the privacy preserving training behaves almost exactly the same as the plaintext training. This is because we only introduce a small error on the 13th bit of the decimal part of \( \mathbf{w} \). Our experiments never triggered the failure condition in theorem 1. However, when we use 6 bits for the decimal part of \( \mathbf{w} \), the accuracy of our protocol oscillates during the training. This is because now the error is on the 6th bit which has a larger effect and may push the model away from the optimum. When the distance

---

**Fig. 10:** Comparison of accuracy of privacy preserving linear regression with truncation and plaintext training on decimal numbers.
to the optimum is large enough, the SGD will move back towards the optimum again. Finally, when we use 2 bits for the fractional part, the oscillating behavior is more extreme. We observe a similar effect when training on another dataset called Arcene [1] as shown in Figure 11. In other words, when sufficient bits are used to represent the fractional part of the coefficients, our new approach for fixed-point multiplication of shared decimal numbers has little impact on accuracy of the trained model.

APPENDIX D
PROOF OF SECURITY FOR PRIVACY PRESERVING LINEAR REGRESSION

We repeat the theorem of security for privacy preserving linear regression here and provide a proof sketch.

Theorem. Consider a protocol where clients distribute arithmetic shares of their data among two servers who run the protocol of Figure 4 and send the output to clients. In the \( F_{\text{offline}} \) hybrid model, this protocol realizes the ideal functionality \( F_{\text{ml}} \) of Figure 3 for the linear regression function, in presence of a semi-honest admissible adversary (see section III).

Sketch. An admissible adversary in our model can corrupt one server and any subset of the clients. Given that the protocol is symmetric with respect to the two servers, we simply need to consider the scenario where the adversary corrupts \( S_0 \) and all but one of the clients, i.e. \( C_1, \ldots, C_{m-1} \).

We describe a simulator \( S \) that simulates the above adversary in the ideal world. \( S \) submits the corrupted clients’ inputs data to the functionality and receives the final output of the linear regression i.e. the final value of the coefficients \( w \) back.

\( S \) then runs \( A \). On behalf of the honest client(s) \( S \) sends a random share in \( \mathbb{Z}_q^m \) to \( A \) for each value in the held by that client. This is the only message where clients are involved. In the remainder of the protocol, generate random matrices and vectors corresponding to the honest server’s shares of \( \langle X, Y, U, V, Z, V', Z' \rangle \), and play the role of the honest server in interactions with \( A \) using those randomly generated values.

Finally, in the very last step where \( w \) is to be recovered, \( S \) adjusts the honest servers’ share of \( w \) such that the recovered value is indeed the coefficient vector it received from the functionality. This concludes the simulation.

We briefly argue that the \( A \)’s view in the real and ideal worlds and as a result, the environment’s view in the two worlds is indistinguishable. This immediately follows from the security of the arithmetic secret sharing and the fact that the matrices/vectors generated in the offline phase are indeed random. In particular, all messages sent and received and reconstructed in the protocol (with the exception of \( w \) are generated using uniformly random shares in both the real protocol and the simulation described above, so indeed the view are both identically distributed, this concludes our argument.

We note that this argument implicitly explains why using one mask matrix \( U \) is sufficient to hide the data matrix \( X \). The reason is that the adversary only gets to see the masked value once in the first interaction and the rest of the computation on \( X \) takes place without interactions between the honest and the corrupted server.

APPENDIX E
PRIVACY PRESERVING PREDICTION AND ACCURACY TESTING

Privacy preserving prediction. As a side product, our protocols also support privacy preserving predictions. The algorithm is exactly the same as computing the predicted value \( y^* \) for linear regression, logistic regression and neural networks and the cost is only half of one iteration. We iterate that we can hide the input data, the model, the prediction result or any combinations of them, as they can all be secret shared in our protocols. Table IV summarizes the cost of predictions. Note that the online phase is extremely fast, which benefits latency critical applications as the offline phase can be precomputed independently of the data. In addition, because of vectorization, the time grows sublinearly when making multiple predictions in parallel.

<table>
<thead>
<tr>
<th>k</th>
<th>Linear (ms)</th>
<th>Logistic (ms)</th>
<th>Neural (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Online</td>
<td>Offline</td>
<td>Online</td>
</tr>
<tr>
<td>LAN</td>
<td>0.20</td>
<td>2.5</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>0.22</td>
<td>51</td>
<td>5.9</td>
</tr>
<tr>
<td>WAN</td>
<td>1</td>
<td>72</td>
<td>620</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>215</td>
<td>2010</td>
</tr>
</tbody>
</table>

TABLE IV: Online and offline performances for privacy preserving predication.

If either the input data or the model can be revealed, the efficiency can be further improved. E.g., if the model is in plaintext, the multiplications of the input data with the coefficients can be computed directly on the shares without precomputed multiplication triplets.

In classification problems, the prediction is usually rounded to the closest class. E.g. in logistic regression, if the predicted value is 0.8, the data is likely to be classified as 1, and the exact result may reveal extra information on the input. This rounding can be viewed as testing whether a secret shared value minus \( \frac{1}{2} \) is larger than 0, and can be supported by applying an extra garbled circuit, similar to how we approximated the logistic function. The garbled circuit would add the two shares and output the most significant bit.
Privacy preserving accuracy testing. A simple way to decide the learning rate is to test it on some insensitive data of the same category beforehand, and set it to a constant without any adjustment throughout training. Similarly, the number of iterations can be fixed in advance.

At the cost of some leakage, we propose an alternative solution that enables adjusting the rate and number of iteration in the same fashion as plaintext training. To do so, we need to test the accuracy of the current model after each epoch on a testing dataset. As a first step, we simply perform a privacy preserving prediction for each testing data sample. Then, we test whether it is the same as the label and aggregate the result. Again we use a simple garbled circuit to perform the equality test, in which the number of gates is linear in the bit length of the values. Finally, each party sums up all the secret-shared results of equality test as the shared accuracy. The cost of doing so is only running half of an iteration plus some extra garbled circuits for rounding and equality testing. As the size of the testing data is usually significantly smaller than the training data, the time spent on the accuracy testing is only a small portion of the training.

To adjust the learning rate, we compare the shared accuracy of two epochs using a garbled circuit and reduce the learning rate if the accuracy is decreasing. Similarly, we calculate the difference of the accuracy and test if it is smaller than a threshold using a garbled circuit, and terminate if the model converges. All these tests are done on the aggregated accuracy, which is a single value per epoch and independent of the number of the training and testing data samples, thus the overhead is negligible. Notice that in each epoch, whether or not we adjust the learning rate or whether we terminate or not leaks one extra bit of information hence providing a trade-off between the efficiency (reduced number of epochs) and security, compared to using a fixed learning rate and a fixed number of iterations.

Appendix F

Client-Aided Multiplication Triplets

We start with the linear regressions for simplicity. Note that in the whole training, each feature in each data sample is used exactly in two multiplications per epoch: one in the forward propagation and the other in the backward propagation. Therefore, it suffices for the client holding this value to generate 2E multiplication triplets. In particular, for each feature of each sample, the client possessing the data generates a random value \( u \) to mask the data, and generates random values \( v_k, v'_k \) for \( k = 1, \ldots, E \) and computes \( z_k = u \cdot v_k, z'_k = u \cdot v'_k \). Finally, the client distributes shares of \( \langle u \rangle, \langle v_k \rangle, \langle v'_k \rangle, \langle z_k \rangle, \langle z'_k \rangle \) to the two servers.

Notice that we do not assume the clients know the partitioning of the data possession when generating the triplets. This means that we can no longer utilize the vectorized equation for the online phase. For example, in Section IV-A, in the forward propagation at step 5 of Figure 4, where we compute \( \mathbf{X}_B \times \mathbf{v} \), we use precomputed matrix multiplication triplets of \( \mathbf{U} \times \mathbf{V} = \mathbf{Z} \) with exactly the same dimensions as the online phase. Now, when the multiplication triplets are generated by the clients, the data in the mini-batch \( \mathbf{X}_B \) may belong to different clients who may not know they are in the same mini-batch of the training, and thus cannot agree on a common random vector \( \mathbf{V} \) to compute \( \mathbf{Z} \).

Instead for each data sample \( \mathbf{x} \) in \( \mathbf{X}_B \), the two parties compute \( \langle y^* \rangle = \text{Mul}^A((\mathbf{x}), (\mathbf{w})) \) using independently generated multiplication triplets, and set \( \langle Y^* \rangle \) to be a vector of \( \langle y^* \rangle \)'s. Because of this, the computation, communication of the online phase and the storage of the two servers are increased.

The client-aided multiplication triplets generation significantly improves the efficiency of the offline phase, as there is no cryptographic operation involved. However, it introduces overhead to the online phase. The matrix multiplications are
our OT-based approach is faster than garbled circuits by a factor of \(5 - 8\times\) on LAN, and a factor of \(4 - 5\times\) on WAN networks. The typical number of multiplications in parallel, needed to train on our datasets is close to 100,000. Though our LHE-based approach is much slower than using garbled circuits on LAN networks, and is comparable on WAN networks, we will show in the next microbenchmark that the LHE-based approach benefits the most from vectorization, which makes it even faster than our OT-based approach on WAN networks.

Note that if the client-aided offline phase is used, the speedup is more significant. Typically it only takes milliseconds in total for \(k = 10,000\). However, as the security model is weakened when using client-aided multiplication triplets, we did not compare it directly with the fixed-point multiplication.

### Vectorization

Table VII shows the speedup gained from vectorization. As a basic operation in our training, we need to multiply a \(|B| \times d\) matrix and a \(d \times 1\) vector. \(|B|\) is set to 128 and \(d\) varies from 100 to 1000. In the unoptimized version, we compute the inner product between the vector and each row of the matrix; in the vectorized version, we compute the matrix-vector multiplication. As shown in Table VII, the online time is improved by around \(2 \times\). The OT-based offline phase is improved by \(4 \times\). The LHE-based offline phase is improved by \(41 - 66\times\) and it is faster than the OT-based offline phase on WAN networks because of the vectorization.

### Microbenchmarks

In addition to evaluating the end-to-end performance of our system, we present microbenchmarks to show the effect of our major optimizations individually in this section.

#### Arithmetic on shared decimal numbers

Table VI compares the performance of our new scheme for decimal multiplications with that of fixed-point multiplication using garbled circuit. We run \(k\) multiplications in parallel and compare the total time of our scheme (online + OT-based offline and online + LHE-based offline) to GC-based fixed-point multiplication, with a 16-bit integer part and a 16-bit decimal part. As shown in the table,