SecureML: A System for Scalable Privacy-Preserving Machine Learning

Payman Mohassel*   Yupeng Zhang†

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 amount 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.

1 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

*Visa Research. **Email:** `pmohasse@visa.com`.

†University of Maryland. **Email:** `zhangyp@umd.edu`. This work was partially done when the author was interning at Visa Research.

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.[1]

We focus on machine learning algorithms for training linear regression, logistic regression and neural networks models, and adopt the *two-server* model (see section 3 for more details), commonly used by previous work on privacy-preserving machine learning via MPC [36, 35, 20]. 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 [36, 20] 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.

1.1 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 [36, 20]. Moreover, as our experiments show, we significantly reduce the gap between privacy-preserving and plaintext training.

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

[1] In the more general variant of our protocols, even the model can remain private (secret shared).

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 7), 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 6.5 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., [17]) 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 [20], we also propose a much faster offline phase where clients help generate the multiplication triplets. This provides a weaker security gauarantee 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 5.

1.2 Related Work

Earlier work on privacy preserving machine learning has focused on decision trees [30], k-means clustering [27, 13], SVM classification [47, 43], linear regression [18, 19, 39] and logistic regression [41]. These papers propose solutions based on secure multiparty computation, but appear to incur high efficiency overheads and lack implementation/evaluation.

Nikolaenko et. al. [36] 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. [20] 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. [22] 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. [45]. 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. [9] 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 [40] 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. [21]. 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 [15, 42, 8]. 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., [8]). 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.

2 Preliminaries

2.1 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., [25]).

Linear regression Given n training data samples $\mathbf{x}_i$ each containing d features and the corresponding output labels y_i, *regression* is a statistical process to learn a function g such that $g(\mathbf{x}_i) \approx y_i$. 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 $\mathbf{x}_i$ with the coefficient vector $\mathbf{w}$: $g(\mathbf{x}_i) = \sum_{j=1}^{d} x_{ij} w_j = \mathbf{x}_i \cdot \mathbf{w}$, where x_{ij} (resp. w_j) is the jth value in vector $\mathbf{x}_i$ (resp. $\mathbf{w}$), and $\cdot$ denotes the inner product of two vectors.[2]

To learn the coefficient vector $\mathbf{w}$, a *cost function* $C(\mathbf{w})$ is defined and $\mathbf{w}$ is calculated by the optimization $\operatorname{argmin}_{\mathbf{w}} C(\mathbf{w})$. In linear regression, a commonly used cost function is $C(\mathbf{w}) = \frac{1}{n} \sum C_i(\mathbf{w})$, where $C_i(\mathbf{w}) = \frac{1}{2}(\mathbf{x}_i \cdot \mathbf{w} - y_i)^2$. [3]

The solution for this optimization problem can be computed by solving the linear system $(\mathbf{X}^T \times \mathbf{X}) \times \mathbf{w} = \mathbf{X}^T \times \mathbf{Y}$, where $\mathbf{X}$ is a $n \times d$ matrix representing all the input data, and $\mathbf{Y}$ is a $n \times 1$ matrix for the output labels. However, the complexity of the matrix multiplication $\mathbf{X}^T \times \mathbf{X}$ is $O(nd^2)$ 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.

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: $\mathbf{w}$ is initialized as a vector of random values or all 0s. In each iteration, a sample $(\mathbf{x}_i, y_i)$ is selected randomly and a coefficient w_j is updated as

$$w_j := w_j - \alpha \frac{\partial C_i(\mathbf{w})}{\partial w_j}. \quad (1)$$

[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 $\lambda ||\mathbf{w}||^2$ is added to the cost function to avoid overfitting where λ is the regularization parameter. This is supported in an obvious way by the protocols in this paper, and is omitted for simplicity.

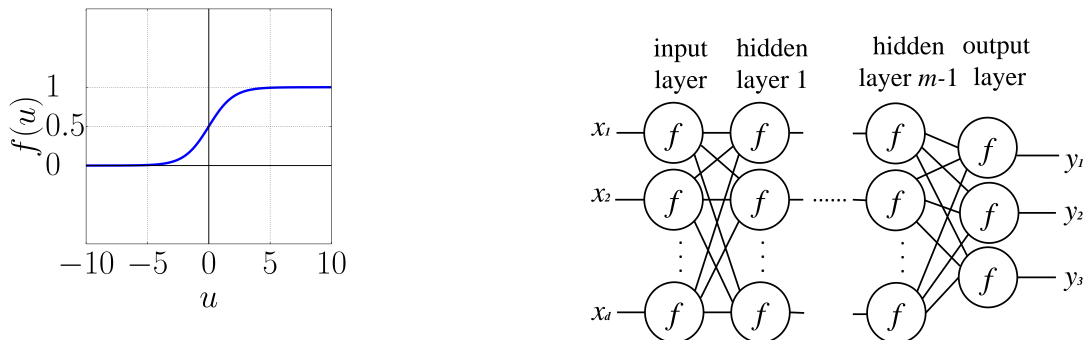

Figure 1: (a) Logistic function. (b) An example of neural network.

where α 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 $w_j := w_j - \alpha(\mathbf{x}_i \cdot \mathbf{w} - y_i)x_{ij}$. The phase to calculate the predicted output $y_i^* = \mathbf{x}_i \cdot \mathbf{w}$ 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 $\mathbf{w}$ is updated by averaging the partial derivatives of all samples on the current $\mathbf{w}$. We denote the set of indices selected in a mini-batch by B. This is called a *mini-batch* SGD and $|B|$ denotes the mini-batch size, usually ranging from 2 to 200. The benefit of mini-batch is that vectorization libraries can be used to speed up the computation such that the computation time for one mini-batch is much faster than running $|B|$ iterations without mini-batch. Besides, with mini-batch, $\mathbf{w}$ converges smoother and faster to the minimum. With mini-batch, the update function can be expressed in a vectorized form:

$$\mathbf{w} := \mathbf{w} - \frac{1}{|B|}\alpha \mathbf{X}_B^T \times (\mathbf{X}_B \times \mathbf{w} - \mathbf{Y}_B). \tag{2}$$

$\mathbf{X}_B$ and $\mathbf{Y}_B$ are $B \times d$ and $B \times 1$ submatrices of $\mathbf{X}$ and $\mathbf{Y}$ selected using indices in B, representing $|B|$ samples of data and labels in an iteration. Here $\mathbf{w}$ is viewed as a column vector.

Learning rate adjustment. If the learning rate α is too large, the result of SGD may diverge from the minimum. Therefore, a testing dataset is used to test the accuracy of the current $\mathbf{w}$. The inner product of $\mathbf{w}$ 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 correct predictions on the testing dataset. If the accuracy is decreasing, the learning rate is reduced and the training starts over with the new learning rate. To balance the overhead spent on testing, the common practice is to shuffle all the training samples and select the mini-batch in each iteration sequentially, until all the samples are used once. This is referred to as one *epoch*. After one epoch, the accuracy of the current $\mathbf{w}$ is tested. At this point, if the accuracy decreases, the learning rate is reduced by half and the training starts over; otherwise the data is reshuffled and the next epoch of training is executed.

Termination. When the difference in accuracy compared to the previous epoch is below a small threshold, $\mathbf{w}$ is viewed as having converged to the minimum and the algorithm terminates. We denote the number of epochs to train a model as E and denote the total number of iterations as t. Note that we have the following relationship: $n \cdot E = |B| \cdot t$.

Logistic Regression In classification problems with two classes, the output label y is binary. E.g., given some medical features, we are interested to predict whether the patient is healthy or sick. In this case, it is better to bound the output of the prediction between 0 and 1. Therefore, an *activation function* f is applied on top of the inner product and the relationship is expressed as: $g(\mathbf{x}_i) = f(\mathbf{x}_i \cdot \mathbf{w})$. In logistic regression, the activation function is defined as the logistic function $f(u) = \frac{1}{1+e^{-u}}$. As shown in Figure 1(a), the two tails of the logistic function converge to 0 and 1.

With this activation function, the original cost function for linear regression is no longer convex, thus applying SGD may give a local minimum instead of the global minimum. Therefore, the cost function is changed to the *cross entropy* function $C_i(\mathbf{w}) = -y_i \log y_i^* - (1 - y_i)\log(1 - y_i^*)$ and $C(\mathbf{w}) = \frac{1}{n}\sum C_i(\mathbf{w})$, where $y_i^* = f(\mathbf{x}_i \cdot \mathbf{w})$.

The mini-batch SGD algorithm for logistic regression updates the coefficients in each iteration as follows:

$$\mathbf{w} := \mathbf{w} - \frac{1}{|B|}\alpha \mathbf{X}_B^T \times (f(\mathbf{X}_B \times \mathbf{w}) - \mathbf{Y}_B). \tag{3}$$

Notice that the backward propagation of logistic regression has exactly the same form as linear regression, yet it is derived using a different activation and cost function. The only difference in the SGD for logistic regression is to apply an extra logistic function on the inner product in the forward propagation.

Neural Networks. Neural networks are a generalization of regression to learn more complicated relationships between high dimensional input and output data. It is extensively used in a wide range of areas such as image processing, voice and text recognition, often leading to breakthroughs in each area. Figure 1(b) shows an example of a neural network with $m - 1$ hidden layers. Each node in the hidden layer and the output layer is an instance of regression and is associated with an activation function and a coefficient vector. Nodes are also called *neurons*. Popular activation functions include the logistic and the RELU function ($f(u) = \max(0, u)$).

For classification problems with multiple classes, usually a *softmax* function $f(u_i) = \frac{e^{-u_i}}{\sum_{i=1}^{d_m} e^{-u_i}}$ is applied at the output layer, where d_m denotes the total number of neurons in the output layer. The insight is that the output after the softmax function is always a probability distribution: each output is between 0 and 1 and all the outputs sum up to 1.

To train a neural network using SGD, Equation 1 is applied in every iteration to update all coefficients of all neurons where each neuron is treated similar to a regression. In particular, let d_i be the number of neurons in layer i and $d_0 = d$ be the number of features in the input data. d_m is the dimension of the output. We denote the coefficient matrix of the ith layer as a $d_{i-1} \times d_i$ matrix $\mathbf{W}_i$, and the values as a $|B| \times d_i$ matrix $\mathbf{X}_i$. $\mathbf{X}_0$ is initialized as $\mathbf{X}_B$. In the forward propagation for each iteration, the matrix $\mathbf{X}_i$ of the ith layer is computed as $\mathbf{X}_i = f(\mathbf{X}_{i-1} \times \mathbf{W}_i)$. In the backward propagation, given a cost function such as the cross entropy function, the update function for each coefficient in each neuron can be expressed in a closed form. To calculated it, we compute the vectors $\mathbf{Y}_i = \frac{\partial C(\mathbf{W})}{\partial \mathbf{U}_i}$ iteratively, where $\mathbf{U}_i = \mathbf{X}_{i-1} \times \mathbf{W}_i$. $\mathbf{Y}_m$ is initialized to $\frac{\partial C}{\partial \mathbf{X}_m} \odot \frac{\partial f(\mathbf{U}_m)}{\partial \mathbf{U}_m}$, where $\frac{\partial f(\mathbf{U}_m)}{\partial \mathbf{U}_m}$ is simply the derivative of the activation function, and $\odot$ is the element-wise product. By the chain rule, $\mathbf{Y}_i = (\mathbf{Y}_{i+1} \times \mathbf{W}_i^T) \odot \frac{\partial f(\mathbf{U}_i)}{\partial \mathbf{U}_i}$. Finally, the coefficients are updated by letting $\mathbf{W}_i := \mathbf{W}_i - \frac{\alpha}{|B|} \cdot \mathbf{X}_i \times \mathbf{Y}_i$.

Parameters: Sender $\mathcal{S}$ and Receiver $\mathcal{R}$.
Main: On input $(SELECT, sid, b)$ from $\mathcal{R}$ and $(SEND, sid, x_0, x_1)$ from $\mathcal{S}$, return $(RECV, sid, x_b)$ to $\mathcal{R}$.

Figure 2: $\mathcal{F}_{ot}$ Ideal Functionality

2.2 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 $\mathcal{S}$ has two inputs x_0 and x_1, and a receiver $\mathcal{R}$ has a selection bit b and wants to obtain x_b without learning anything else or revealing b to $\mathcal{S}$. Figure 2 describes the ideal functionality realized by such a protocol. We use the notation $(\perp; x_b) \leftarrow \mathsf{OT}(x_0, x_1; b)$ to denote a protocol realizing this functionality.

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 [38], but it requires public-key operations by both parties. OT extension [26, 10] minimizes this cost by allowing the sender and receiver to perform m OTs at the cost of λ base OTs (with public-key operations) and $O(m)$ fast symmetric-key ones, where λ 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 [10]. In this variant which we denote by COT, the sender's two inputs to each OT are not independent. Instead, the two inputs to each OT instance 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 [46]. A garbling scheme consists of a garbling algorithm that takes a random seed σ and a function f and generates a garbled circuit F and a decoding table dec; the encoding algorithm takes input x and the seed σ and generates garbled input $\widehat{x}$; the evaluation algorithm takes $\widehat{x}$ and F as input and returns the garbled output $\widehat{z}$; and finally, a decoding algorithm that takes the decoding table dec and $\widehat{z}$ and returns $f(x)$. We require the garbling scheme to satisfy the standard security properties formalized in [12].

Given such a garbling scheme, it is possible to design a secure two-party computation protocol as follows: Alice generates a random seed σ and runs the garbling algorithm for function f to obtain a garbled circuit GC. She also encodes her input $\widehat{x}$ using σ and x as inputs to the encoding algorithm. Alice sends GC and $\widehat{x}$ to Bob. Bob obtains his encoded (garbled) input $\widehat{y}$ using an oblivious transfer for each bit of y[4]. He then runs the evaluation algorithm on $GC, \widehat{x}, \widehat{y}$ to obtain the garbled output $\widehat{z}$. 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 [31] for a more detailed description and proof of security against a semi-honest adversary. In our protocols, we denote this garbled circuit 2PC by $(z_a, z_b) \leftarrow \mathsf{GarbledCircuit}(x; y, f)$

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,

[4] While and OT-based encoding is not a required property of a garbling scheme, all existing constructions permit such interacting encodings

Boolean sharing and Yao sharing. We briefly review these schemes but refer the reader to [17] for more details.

To additively share ($\mathsf{Shr}^A(\cdot)$) an ℓ-bit value a, the first party P_0 generates $a_0 \in \mathbb{Z}_{2^\ell}$ uniformly at random and sends $a_1 = a - a_0 \bmod 2^\ell$ 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 ($\mathsf{Rec}^A(\cdot,\cdot)$) 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 $\langle c \rangle_i = \langle a \rangle_i + \langle b \rangle_i \bmod 2^\ell$. We overload the addition operation to denote the addition protocol by $\langle a \rangle + \langle b \rangle$.

To multiply ($\mathsf{Mul}^A(\cdot,\cdot)$) 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 $\mathbb{Z}_{2^\ell}$ and $z = uv \bmod 2^\ell$. 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 $\mathsf{Rec}(\langle e \rangle_0, \langle e \rangle_1)$ and $\mathsf{Rec}(\langle f \rangle_0, \langle f \rangle_1)$, 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 ($\mathsf{AND}(\cdot,\cdot)$). 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 [33] 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 $\langle a \rangle_0^Y = K_0^w, K_1^w$ and $\langle a \rangle_1^Y = K_a^w$. 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 $\langle a \rangle_0^Y = K_0^w, K_1^w$ and $\langle a \rangle_1^Y = K_a^w$ to a Boolean sharing, P_0 lets $\langle a \rangle_0^B = K_0^w[0]$ and P_1 lets $\langle a \rangle_1^B = \langle a \rangle_1^Y[0]$. 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 $\mathsf{Y2B}(\cdot,\cdot)$. 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 $\mathsf{Y2B}$ conversion to convert the garbled output to a boolean sharing.

3 Security Model

3.1 Architecture

We consider a set of clients $\mathcal{C}_1, \ldots, \mathcal{C}_m$ 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 $\mathcal{S}_0$ and $\mathcal{S}_1$. Server-aided MPC has been formalized and used in various previous work (e.g. see [28]). It has also been utilized in prior work on privacy-preserving machine learning [36, 35, 20]. 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) [36], or the evaluator and a *cloud service provider* who maintains the data [23]. 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.

3.2 Security Definition

Recall that the involved parties are m clients $\mathcal{C}_1, \ldots, \mathcal{C}_m$ and two servers $\mathcal{S}_0, \mathcal{S}_1$. We assume a *semi-honest* adversary $\mathcal{A}$ 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 5), 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 $\mathcal{A}$ who corrupts $\mathcal{C}_1, \mathcal{C}_2$ and $\mathcal{S}_1$ should not learn any information about $\mathcal{C}_3$'s data beyond the trained model. We define security using the framework of Universal Composition (UC) [14]. We give a brief overview of the definition here, but refer the reader to [14] for the details. The target ideal functionality $\mathcal{F}_{ml}$ for our protocols is described in Figure 3.

An execution in the UC framework involves a collection of (non-uniform) interactive Turing machines. In this work we consider an admissible and semi-honest adversary $\mathcal{A}$ as discussed above. The *parties* exchange messages according to a protocol. Protocol inputs of uncorrupted parties are chosen by an *environment* machine. Uncorrupted parties also report their protocol outputs to the environment. At the end of the interaction, 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 $\textsc{real}[\mathcal{Z}, \mathcal{A}, \pi, \lambda]$ denote the final (single-bit) output of the environment $\mathcal{Z}$ when interacting with adversary $\mathcal{A}$ 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

Parameters: Clients $\mathcal{C}_1, \ldots, \mathcal{C}_m$ and servers $\mathcal{S}_0, \mathcal{S}_1$.
Uploading Data: On input x_i from $\mathcal{C}_i$, store x_i internally.
Computation: On input f from $\mathcal{S}_0$ or $\mathcal{S}_1$, compute $(y_1, \ldots, y_m) = f(x_1, \ldots, x_m)$ and send y_i to $\mathcal{C}_i$. This step can be repeated multiple times with different functions.

Figure 3: Ideal Functionality $\mathcal{F}_{ml}$

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 $\mathcal{F}_{ml}$ for protocols is described in Figure 3. Let $\textsc{ideal}[\mathcal{Z}, \mathcal{S}, \mathcal{F}_{ml}, \lambda]$ denote the output of the environment $\mathcal{Z}$ when interacting with adversary $\mathcal{S}$ and honest parties who run the dummy protocol in presence of functionality $\mathcal{F}$ on security parameter λ.

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

$$\left| \Pr\left[\textsc{real}[\mathcal{Z}, \mathcal{A}, \pi, \lambda] = 1\right] - \Pr\left[\textsc{ideal}[\mathcal{Z}, \mathcal{S}, \mathcal{F}_{ml}, \lambda] = 1\right] \right|.$$

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.

4 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 4.1, 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 4.2. We then generalize our techniques to support logistic regression and neural networks training in Sections 4.3 and 4.4. Finally, techniques to support predication, learning rate adjustment and termination determination are presented in Section 4.5.

4.1 Privacy Preserving Linear Regression

Recall that we assume the training data is secret shared between two servers $\mathcal{S}_0$ and $\mathcal{S}_1$. We denote the shares by $\langle \mathbf{X} \rangle_0, \langle \mathbf{Y} \rangle_0$ and $\langle \mathbf{X} \rangle_1, \langle \mathbf{Y} \rangle_1$. In practice, the clients can distribute the shares between the two servers, or encrypt the first share using the public key of $\mathcal{S}_0$, upload both the first encrypted share and the second plaintext share to $\mathcal{S}_1$. $\mathcal{S}_1$ then passes the encrypted shares to $\mathcal{S}_0$ to decrypt. In our protocol, we also let the coefficients $\mathbf{w}$ be secret shared between the two servers. It is initialized to random values simply by setting $\langle \mathbf{w} \rangle_0, \langle \mathbf{w} \rangle_1$ 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 2.1, the update function for linear regression is $w_j := w_j - \alpha(\sum_{k=1}^{d} x_{ik} w_k - y_i) x_{ij}$, 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 $\langle w_j \rangle := \langle w_j \rangle - \alpha \mathsf{Mul}^A\big(\sum_{k=1}^{d} \mathsf{Mul}^A(\langle x_{ik} \rangle, \langle w_k \rangle) - \langle y_i \rangle, \langle x_{ij} \rangle\big)$. 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 4.2.

Vectorization in the Shared Setting. We also want to benefit from the mini-batch and vectorization techniques discussed in Section 2.1 (see Equation 2). To achieve this, we generalize the addition and multiplication operations on share values to shared matrices. Matrices are shared by applying Shr^A to every element. Given two shared matrices $\langle \mathbf{A} \rangle$ and $\langle \mathbf{B} \rangle$, matrix addition can be computed non-interactively by letting $\langle \mathbf{C} \rangle_i = \langle \mathbf{A} \rangle_i + \langle \mathbf{B} \rangle_i$ for $i \in \{0, 1\}$. To multiply two shared matrices, instead of using independent multiplication triplets, we take shared matrices $\langle \mathbf{U} \rangle, \langle \mathbf{V} \rangle, \langle \mathbf{Z} \rangle$, where each element in $\mathbf{U}$ and $\mathbf{V}$ is uniformly random in $\mathbb{Z}_{2^l}$, $\mathbf{U}$ has the same dimension as $\mathbf{A}$, $\mathbf{V}$ has the same dimension as $\mathbf{B}$ and $\mathbf{Z} = \mathbf{U} \times \mathbf{V} \mod 2^l$. $\mathcal{S}_i$ computes $\langle \mathbf{E} \rangle_i = \langle \mathbf{A} \rangle_i - \langle \mathbf{U} \rangle_i$, $\langle \mathbf{F} \rangle_i = \langle \mathbf{B} \rangle_i - \langle \mathbf{V} \rangle_i$ and sends it to the other server. Both servers reconstruct $\mathbf{E}$ and $\mathbf{F}$ and set $\langle \mathbf{C} \rangle_i = -i \cdot \mathbf{E} \times \mathbf{F} + \langle \mathbf{A} \rangle_i \times \mathbf{F} + \mathbf{E} \times \langle \mathbf{B} \rangle_i + \langle \mathbf{Z} \rangle_i$. The idea of this generalization is that each element in matrix $\mathbf{A}$ is always masked by the same random element in $\mathbf{U}$, while it is multiplied by different elements in $\mathbf{B}$ 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 $\langle \mathbf{w} \rangle := \langle \mathbf{w} \rangle - \frac{1}{|B|} \alpha \mathsf{Mul}^A(\langle \mathbf{X}_B^T \rangle, \mathsf{Mul}^A\big(\langle \mathbf{X}_B \rangle, \langle \mathbf{w} \rangle\big) - \langle \mathbf{Y}_B \rangle)$. 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 $\langle \mathbf{U} \rangle$ is generated to mask the data samples $\langle \mathbf{X} \rangle$. At the beginning of the online phase, $\langle \mathbf{E} \rangle_i = \langle \mathbf{X} \rangle_i - \langle \mathbf{U} \rangle_i$ is computed and exchanged to reconstruct $\mathbf{E}$ through one interaction. After that, in each iteration, $\mathbf{E}_B$ 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 $\langle \mathbf{U} \rangle, \langle \mathbf{V} \rangle, \langle \mathbf{Z} \rangle, \langle \mathbf{V}' \rangle, \langle \mathbf{Z}' \rangle$ 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 $\mathbf{w}$ in one iteration (forward propagation), and $\mathbf{V}'$ is a $|B| \times t$ matrix wherein each column is used to mask the difference vector $\mathbf{Y}^* - \mathbf{Y}$ in one iteration (backward propagation). We then let $\mathbf{Z}[i] = \mathbf{U}_{B_i} \times \mathbf{V}[i]$ and $\mathbf{Z}'[i] = \mathbf{U}_{B_i}^T \times \mathbf{V}'[i]$ for $i = 1, \ldots, t$, where $\mathbf{M}[i]$ denotes the ith 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 $\mathcal{F}_{offline}$.

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 [21], or utilize 2PC for boolean circuits to perform fixed-point [20] or floating-point [34] 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 $z = z_1 \cdot 2^{l_D} + z_2$, where $0 \le z_2 < 2^{l_D}$, 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 in the least significant bit of the fractional part compared to standard fixed-point arithmetic.

We also note that if a decimal number z is negative, it will be represented in the field as $2^l - |z|$, where $|z|$ is its absolute value and the truncation operation changes to $\lfloor z \rfloor = 2^l - \lfloor |z| \rfloor$. We prove the following theorem for both positive and negative numbers.

Theorem 1. *In field $\mathbb{Z}_{2^l}$, let $x \in [0, 2^{l_x}] \cup [2^l - 2^{l_x}, 2^l)$, where $l > l_x + 1$ and given shares $\langle x \rangle_0, \langle x \rangle_1$ of x, let $\langle \lfloor x \rfloor \rangle_0 = \lfloor \langle x \rangle_0 \rfloor$ and $\langle \lfloor x \rfloor \rangle_1 = 2^l - \lfloor 2^l - \langle x \rangle_1 \rfloor$. Then with probability $1 - 2^{l_x+1-l}$, $\mathsf{Rec}^A(\langle \lfloor x \rfloor \rangle_0, \langle \lfloor x \rfloor \rangle_1) \in \{\lfloor x \rfloor - 1, \lfloor x \rfloor, \lfloor x \rfloor + 1\}$, where $\lfloor \cdot \rfloor$ denotes truncation by $l_D \le l_x$ bits.*

Proof. Let $\langle x \rangle_0 = x + r \mod 2^l$, where r is uniformly random in $\mathbb{Z}_{2^l}$, then $\langle x \rangle_1 = 2^l - r$. We decompose r as $r_1 \cdot 2^{l_D} + r_2$, where $0 \le r_2 < 2^{l_D}$ and $0 \le r_1 < 2^{l - l_D}$. We prove that if $2^{l_x} \le r < 2^l - 2^{l_x}$, $\mathsf{Rec}^A(\langle \lfloor x \rfloor \rangle_0, \langle \lfloor x \rfloor \rangle_1) \in \{\lfloor x \rfloor - 1, \lfloor x \rfloor, \lfloor x \rfloor + 1\}$. Consider the following two cases.

Case 1: If $0 \le x \le 2^{l_x}$, then $0 < x + r < 2^l$ and $\langle x \rangle_0 = x + r$, without modulo. Let $x = x_1 \cdot 2^{l_D} + x_2$, where $0 \le x_2 < 2^{l_D}$ and $0 \le x_1 < 2^{l_x - l_D}$. 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 the carry bit $c = 0$ if $x_2 + r_2 < 2^{l_D}$ and $c = 1$ otherwise. After the truncation, $\langle \lfloor x \rfloor \rangle_0 = \lfloor x + r \rfloor = x_1 + r_1 + c$ and $\langle \lfloor x \rfloor \rangle_1 = 2^l - r_1$. Therefore, $\mathsf{Rec}^A(\langle \lfloor x \rfloor \rangle_0, \langle \lfloor x \rfloor \rangle_1) = x_1 + c = \lfloor x \rfloor + c$.

Case 2: If $2^l - 2^{l_x} \le x < 2^l$, then $x + r \ge 2^l$ and $\langle x \rangle_0 = x + r - 2^l$. Let $x = 2^l - x_1 \cdot 2^{l_D} - x_2$, where $0 \le x_2 < 2^{l_D}$ and $0 \le x_1 < 2^{l_x - l_D}$. We have $x + r - 2^l = (r_1 - x_1) \cdot 2^{l_D} + (r_2 - x_2) = (r_1 - x_1 - c) \cdot 2^{l_D} + (r_2 - x_2 + c \cdot 2^{l_D})$, where the carry bit $c = 0$ if $r_2 > x_2$ and $c = 1$ otherwise. After the truncation, $\langle \lfloor x \rfloor \rangle_0 = \lfloor x + r - 2^l \rfloor = r_1 - x_1 - c$ and $\langle \lfloor x \rfloor \rangle_1 = 2^l - r_1$. Therefore, $\mathsf{Rec}^A(\langle \lfloor x \rfloor \rangle_0, \langle \lfloor x \rfloor \rangle_1) = 2^l - x_1 - c = \lfloor x \rfloor - c$.

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

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.

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 $\langle \mathbf{U} \rangle, \langle \mathbf{V} \rangle, \langle \mathbf{Z} \rangle, \langle \mathbf{V}' \rangle, \langle \mathbf{Z}' \rangle$ were already generated in the offline phase. Besides multiplication and addition of shared decimal numbers, the protocol also requires multiplying the coefficient vector by

Protocol $\mathsf{SGD_Linear}(\langle \mathbf{X} \rangle, \langle \mathbf{Y} \rangle, \langle \mathbf{U} \rangle, \langle \mathbf{V} \rangle, \langle \mathbf{Z} \rangle, \langle \mathbf{V}' \rangle, \langle \mathbf{Z}' \rangle)$:

1: $\mathcal{S}_i$ computes $\langle \mathbf{E} \rangle_i = \langle \mathbf{X} \rangle_i - \langle \mathbf{U} \rangle_i$ for $i \in \{0,1\}$. Then parties run $\mathsf{Rec}(\langle \mathbf{E} \rangle_0, \langle \mathbf{E} \rangle_1)$ to obtain $\mathbf{E}$.
2: **for** $j = 1, \ldots, t$ **do**
3: Parties select the mini-batch $\langle \mathbf{X}_{B_j} \rangle, \langle \mathbf{Y}_{B_j} \rangle$.
4: $\mathcal{S}_i$ computes $\langle \mathbf{F}_j \rangle_i = \langle \mathbf{w} \rangle_i - \langle \mathbf{V}[j] \rangle$ for $i \in \{0,1\}$. Then parties run $\mathsf{Rec}(\langle \mathbf{F}_j \rangle_0, \langle \mathbf{F}_j \rangle_1)$ to recover $\mathbf{F}_j$.
5: $\mathcal{S}_i$ computes $\langle \mathbf{Y}^*_{B_j} \rangle_i = -i \cdot \mathbf{E}_{B_j} \times \mathbf{F}_i + \langle \mathbf{X}_{B_j} \rangle_i \times \mathbf{F}_i + \mathbf{E}_{B_j} \times \langle \mathbf{w} \rangle_i + \langle \mathbf{Z}_j \rangle_i$ 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 $\langle \mathbf{F}'_j \rangle_i = \langle \mathbf{D}_{B_j} \rangle_i - \langle \mathbf{V}'_j \rangle_i$ for $i \in \{0,1\}$. Parties then run $\mathsf{Rec}(\langle \mathbf{F}'_j \rangle_0, \langle \mathbf{F}'_j \rangle_1)$ to obtain $\mathbf{F}'_j$.
8: $\mathcal{S}_i$ computes $\langle \mathbf{\Delta} \rangle_i = -i \cdot \mathbf{E}^T_{B_j} \times \mathbf{F}'_j + \langle \mathbf{X}^T_{B_j} \rangle_i \times \mathbf{F}'_j + \mathbf{E}^T_{B_j} \times \langle \mathbf{D}_{B_j} \rangle_i + \langle \mathbf{Z}'_j \rangle_i$ for $i \in \{0,1\}$.
9: $\mathcal{S}_i$ truncates its shares of Δ element-wise to get $\lfloor \langle \mathbf{\Delta} \rangle_i \rfloor$.
10: $\mathcal{S}_i$ computes $\langle \mathbf{w} \rangle_i := \langle \mathbf{w} \rangle_i - \frac{\alpha}{|B|} \lfloor \langle \mathbf{\Delta} \rangle_i \rfloor$ for $i \in \{0,1\}$.
11: Parties run $\mathsf{Rec}^A(\langle \mathbf{w} \rangle_0, \langle \mathbf{w} \rangle_1)$ and output $\mathbf{w}$.

Figure 4: The online phase of privacy preserving linear regression.

$\frac{\alpha}{|B|}$ in each iteration. To make this operation efficient, we set $\frac{\alpha}{|B|}$ to be a power of 2, i.e. $\frac{\alpha}{|B|} = 2^{-k}$. Then the multiplication with $\frac{\alpha}{|B|}$ 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.

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 $\mathcal{F}_{offline}$ hybrid model, this protocol realizes the ideal functionality $\mathcal{F}_{ml}$ of Figure 3 for the linear regression function, in presence of a semi-honest* admissible *adversary (see section 3).*

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 $\mathcal{S}_0$ and all but one of the clients, i.e. $\mathcal{C}_1, \ldots, \mathcal{C}_{m-1}$.

We describe a simulator $\mathcal{S}$ that simulates the above adversary in the ideal world. $\mathcal{S}$ 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 $\mathbf{w}$ back.

$\mathcal{S}$ then runs $\mathcal{A}$. On behalf of the honest client(s) $\mathcal{S}$ sends a random share in $\mathbb{Z}_{2^l}$ to $\mathcal{A}$ 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 $\langle \mathbf{X} \rangle, \langle \mathbf{Y} \rangle, \langle \mathbf{U} \rangle, \langle \mathbf{V} \rangle, \langle \mathbf{Z} \rangle, \langle \mathbf{V}' \rangle, \langle \mathbf{Z}' \rangle$, and play the role of the honest server in interactions with $\mathcal{A}$ using those randomly generated values.

Finally, in the very last step where $\mathbf{w}$ is to be recovered, $\mathcal{S}$ adjusts the honest servers' share of of $\mathbf{w}$ such that the recovered value is indeed the coefficient vector it received from the functionality. This concludes the simulation.

We briefly argue that the $\mathcal{A}$'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 $\mathbf{w}$ are generated using uniformly random shares in both the real protocol and

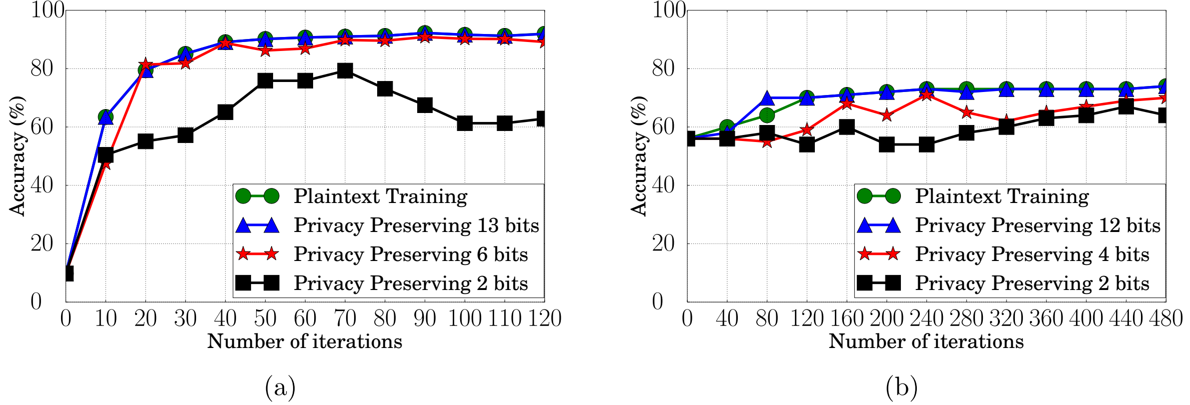

Figure 5: Comparison of accuracy of privacy preserving linear regression with truncation and plaintext training on decimal numbers. (a) MNIST dataset, $|B| = 128$, (b) Arcene dataset, $|B| = 32$.

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 $\mathbf{U}$ is sufficient to hide the data matrix $\mathbf{X}$. 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 $\mathbf{X}$ takes place without interactions between the honest and the corrupted server. □

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 $(|B|+d)\cdot t$ times and by the union bound, the probability of failure in the training is $(|B|+d)\cdot t \cdot 2^{l_x+1-l}$. For typical parameters $|B| = 128, d = 784, t = 1000, l_x = 32, l = 64$, the probability of a single failure happening during the whole training is around 2^{-12}. Moreover, even if a failure in the truncation occurs, it is unlikely to translate to a failure in training. Such a failure makes one feature in one sample invalid, yet the final model should not be affected by small changes in data, or else the trainig strategy suffers from overfitting. We confirm these observations by running experiments on two different datasets (MNIST [6] and Arcene [1]). 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.

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 $\mathbb{Z}_{2^{64}}$. 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 5a, 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 5a, 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 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 5b. 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.

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[5], 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 $n \cdot d + (|B| + d) \cdot t = nd \cdot (1 + \frac{E}{d} + \frac{E}{|B|})$ 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.

4.2 The Offline Phase

We describe how to implement the offline phase as a two-party protocol between $\mathcal{S}_0$ and $\mathcal{S}_1$ 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., [17]) 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 $\langle \mathbf{U} \rangle$ and $\langle \mathbf{V} \rangle$, the key step is to choose a $|B| \times d$ submatrix from $\langle \mathbf{U} \rangle$ and a column from $\langle \mathbf{V} \rangle$ and compute the shares of their product. This is repeated t times to generate $\langle \mathbf{Z} \rangle$. $\langle \mathbf{Z}' \rangle$ 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 $\langle \mathbf{A} \rangle$, and shares of a $d \times 1$ matrix $\langle \mathbf{B} \rangle$, we want to compute shares of a $|B| \times 1$ matrix $\langle \mathbf{C} \rangle$ such that $\mathbf{C} = \mathbf{A} \times \mathbf{B}$.

We utilize the following relationship: $\mathbf{C} = \langle \mathbf{A} \rangle_0 \times \langle \mathbf{B} \rangle_0 + \langle \mathbf{A} \rangle_0 \times \langle \mathbf{B} \rangle_1 + \langle \mathbf{A} \rangle_1 \times \langle \mathbf{B} \rangle_0 + \langle \mathbf{A} \rangle_1 \times \langle \mathbf{B} \rangle_1$. It suffices to compute $\langle \langle \mathbf{A} \rangle_0 \times \langle \mathbf{B} \rangle_1 \rangle$ and $\langle \langle \mathbf{A} \rangle_1 \times \langle \mathbf{B} \rangle_0 \rangle$ as the other two terms can be computed locally.

LHE-based generation. To compute the shares of the product $\langle \mathbf{A} \rangle_0 \times \langle \mathbf{B} \rangle_1$, $\mathcal{S}_1$ encrypts each element of $\langle \mathbf{B} \rangle_1$ using an LHE and sends them to $\mathcal{S}_0$. The LHE can be initiated using the cryptosystem of Paillier [37] or Damgard-Geisler-Kroigaard(DGK) [16]. $\mathcal{S}_0$ then performs the matrix multiplication on the ciphertexts, with additions replaced by multiplications and multiplications by exponentiations. Finally, $\mathcal{S}_0$ masks the resulting ciphertexts by random values, and sends them back to $\mathcal{S}_1$ to decrypt. The protocol can be found in Figure 6.

[5] Party $\mathcal{S}_1$ can simplify the formula to $\mathbf{E} \times (\langle \mathbf{w} \rangle - \mathbf{F}) + \langle \mathbf{X} \rangle \times \mathbf{F} + \langle \mathbf{Z} \rangle$, which has only 2 matrix multiplications.

Protocol $\mathsf{LHE_MT}(\langle \mathbf{A}\rangle_0; \langle \mathbf{B}\rangle_1)$:

(Let a_{ij} be the (i,j)th element in $\langle \mathbf{A}\rangle_0$ and b_j be the jth element in $\langle \mathbf{B}\rangle_1$.)

1: $\mathcal{S}_1 \to \mathcal{S}_0$: $\mathsf{Enc}(b_j)$ for $i = 1, \ldots, d$.
2: $\mathcal{S}_0 \to \mathcal{S}_1$: $c_i = \Pi_{j=0}^{d} \mathsf{Enc}(b_j)^{a_{ij}} \cdot \mathsf{Enc}(r_i)$, for $i = 1, \ldots, |B|$.
3: $\mathcal{S}_0$ sets $\langle \langle \mathbf{A}\rangle_0 \times \langle \mathbf{B}\rangle_1 \rangle_0 = \mathbf{r}$, where $\mathbf{r} = (-r_1, \ldots, -r_{|B|})^T \mod 2^l$.
4: $\mathcal{S}_1$ sets $\langle \langle \mathbf{A}\rangle_0 \times \langle \mathbf{B}\rangle_1 \rangle_1 = (\mathsf{Dec}(c_1), \ldots, \mathsf{Dec}(c_{|B|}))^T$,

Figure 6: The offline protocol based on linearly homomorphic encryption.

Here $\mathcal{S}_1$ performs d encryptions, $|B|$ decryptions and $\mathcal{S}_0$ performs $|B| \times d$ exponentiations. The cost of multiplications on the ciphertext is non-dominating and is omitted. The shares of $\langle \mathbf{A}\rangle_1 \times \langle \mathbf{B}\rangle_0$ 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 $\langle a_{ij} \cdot b_j \rangle$ for all $i = 1, \ldots, |B|$ and $j = 1, \ldots, d$. To do so, $\mathcal{S}_1$ uses each bit of b_j to select two values computed from a_{ij} using correlated OTs. In particular, for $k = 1, \ldots, l$, $\mathcal{S}_0$ sets the correlation function of COT to $f_k(x) = a_{i,j} \cdot 2^k + x \mod 2^l$ and $\mathcal{S}_0$, $\mathcal{S}_1$ run $\mathsf{COT}(r_k, f_k(x); b_j[k])$. If $b_j[k] = 0$, $\mathcal{S}_1$ gets r_k; if $b_j[k] = 1$, $\mathcal{S}_1$ gets $a_{i,j} \cdot 2^k + r_k \mod 2^l$. This is equivalent to $b_j[k] \cdot a_{ij} \cdot 2^k + r_k \mod 2^l$. Finally, $\mathcal{S}_1$ sets $\langle a_{ij} \cdot b_j \rangle_1 = \sum_{k=1}^{l} (b_j[k] \cdot a_{ij} \cdot 2^k + r_k) = a_{ij} \cdot b_j + \sum_{k=1}^{l} r_k \mod 2^l$, and $\mathcal{S}_0$ sets $\langle a_{ij} \cdot b_j \rangle_0 = \sum_{k=1}^{l} (-r_k) \mod 2^l$.

To further improve efficiency, authors of [17] 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 $\mathrm{COT}_{(l+1)/2}$. So far, all the techniques described are as discussed in [17].

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 $a_{1j}, \ldots, a_{|B|j}$ are all multiplied by b_j, which means the same selection bit $b_j[k]$ is used for all a_{ij}s. 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 $\lceil \frac{(l-k) \cdot |B|}{128} \rceil$ hash outputs. For a reasonable mini-batch size, each multiplication needs $\frac{l}{4}$ instances of COT_{128}. In this way, the total number of hashes can be reduced by a factor of 4 and the total communication can be reduced by a factor of 2.

Finally, after computing $\langle a_{ij} \cdot b_j \rangle$, the ith element of $\langle \langle \mathbf{A}\rangle_0 \times \langle \mathbf{B}\rangle_1 \rangle$ can be computed by $\langle \langle \mathbf{A}\rangle_0 \times \langle \mathbf{B}\rangle_1 \rangle[i] = \sum_{j=0}^{d} \langle a_{ij} \cdot b_j \rangle$. The shares of $\langle \mathbf{A}\rangle_1 \times \langle \mathbf{B}\rangle_0$ can be computed similarly.

In total, both parties perform $\frac{|B| \cdot d \cdot t \cdot l}{2}$ instances of COT_{128} and the total communication is

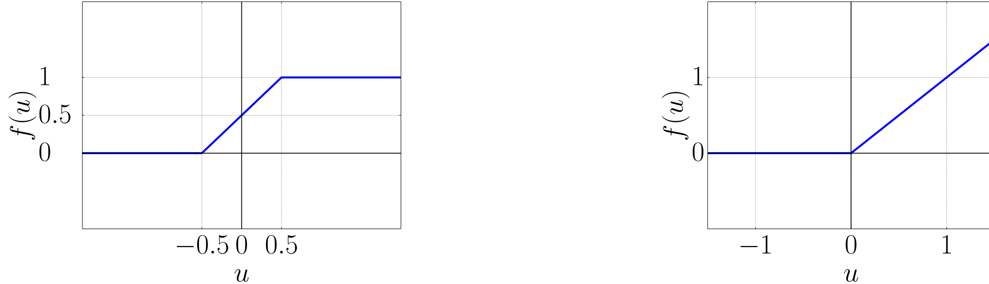

Figure 7: (a) Our new activation function. (b) RELU function.

$|B| \cdot d \cdot t \cdot l \cdot (l + \lambda)$ bits. In addition, a set of base OTs need to be performed at the beginning for OT extension. In Section 6.1 we show that the size of communication for the OT-based generation is much higher than the LHE-based generation, yet the total running time is faster. The reason is that, given OT extension, each OT operation is very cheap ($\sim 10^6$ OTs per second).

4.3 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 [9]. It can be shown that approximation using a high-degree polynomial is very accurate [32]. 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 7(a).

$$f(x) = \begin{cases} 0, & \text{if } x < -\frac{1}{2} \\ x + \frac{1}{2}, & \text{if } -\frac{1}{2} \leq x \leq \frac{1}{2} \\ 1, & \text{if } x > \frac{1}{2} \end{cases} \tag{4}$$

The intuition for this choice of activation is as follows (we also confirm its effectiveness with experiments): as mentioned in section 2.1, the main reason logistic regression works well for classification problems is that the prediction is bounded between 0 and 1. Therefore, it is very important for the two tails of the activation function to converge to 0 and 1, and both the logistic function and the function in Equation 4 have 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 (Figure 7(b)) used in neural networks. One of the justifications used for replacing logistic function by the RELU function in neural networks is that

	Logistic	Our approaches		Polynomial Approx.		
		first	second	deg. 2	deg. 5	deg. 10
MNIST	98.64	98.62	97.96	42.17	84.64	98.54
Arcene	86	86	85	72	82	86

Table 1: Accuracy (%) comparison of different approaches for logistic regression.

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 1, the performance of our approaches are 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 regression. 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 furthur 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 [17]. The observation is that as mentioned in Section 2.1, 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

Protocol $\mathsf{SGD_Logistic}(\langle \mathbf{X} \rangle, \langle \mathbf{Y} \rangle, \langle \mathbf{U} \rangle, \langle \mathbf{V} \rangle, \langle \mathbf{Z} \rangle, \langle \mathbf{V}' \rangle, \langle \mathbf{Z}' \rangle)$:

1: Do step 1–5 as in Figure 4. Both parties obtain the shares $\langle \mathbf{U}_{B_i} \rangle = \langle \mathbf{X}_{B_i} \times \mathbf{w} \rangle$ (it was defined as $\langle \mathbf{Y}^*_{B_i} \rangle$ in Figure 4).
2: **for** every element $\langle u \rangle$ in $\langle \mathbf{U}_{B_i} \rangle$ **do**
3: $(\langle b_3 \rangle^B, \langle b_4 \rangle^B) \leftarrow \mathsf{Y2B}(\mathsf{GarbledCircuit}(\langle u \rangle_0 + \frac{1}{2}, \langle u \rangle_0 - \frac{1}{2}; \langle u \rangle_1, f))$, where f sets b_1 as the most significant bit of $(\langle u \rangle_0 + \frac{1}{2}) + \langle u \rangle_1$ and b_2 as the most significant bit of $(\langle u \rangle_0 - \frac{1}{2}) + \langle u \rangle_1$. It then outputs $b_3 = \neg b_1$ and $b_4 = b_1 \wedge (\neg b_2)$.
4: $\mathcal{S}_0$ sets $m_0 = \langle b_4 \rangle_0^B \cdot \langle u \rangle_0 + r_1$ and $m_1 = (1 - \langle b_4 \rangle_0^B) \cdot \langle u \rangle_0 + r_1$. $\mathcal{S}_0$ and $\mathcal{S}_1$ run $(\perp; m_{\langle b_4 \rangle_1^B}) \leftarrow \mathsf{OT}(m_0, m_1; \langle b_4 \rangle_1^B)$. $m_{\langle b_4 \rangle_1^B}$ is equal to $(\langle b_4 \rangle_0^B \oplus \langle b_4 \rangle_1^B) \cdot \langle u \rangle_0 + r_1 = b_4 \cdot \langle u \rangle_0 + r_1$.
5: P_1 sets $m_0 = \langle b_4 \rangle_1^B \cdot \langle u \rangle_1 + r_2$ and $m_1 = (1 - \langle b_4 \rangle_1^B) \cdot \langle u \rangle_1 + r_2$. $\mathcal{S}_1$ and $\mathcal{S}_0$ run $(\perp; m_{\langle b_4 \rangle_0^B}) \leftarrow \mathsf{OT}(m_0, m_1; \langle b_4 \rangle_0^B)$. $m_{\langle b_4 \rangle_0^B}$ is equal to $b_4 \cdot \langle u \rangle_1 + r_2$.
6: $\mathcal{S}_0$ sets $m_0 = \langle b_3 \rangle_0^B + r_3$ and $m_1 = (1 - \langle b_3 \rangle_0^B) + r_3$. $\mathcal{S}_0$ and $\mathcal{S}_1$ run $(\perp; m_{\langle b_3 \rangle_1^B}) \leftarrow \mathsf{OT}(m_0, m_1; \langle b_3 \rangle_1^B)$. $m_{\langle b_3 \rangle_1^B}$ is equivalent to $b_3 + r_3$.
7: $\mathcal{S}_0$ sets $\langle y^* \rangle_0 = m_{\langle b_4 \rangle_0^B} - r_1 - r_3$ and $\mathcal{S}_1$ sets $\langle y^* \rangle_1 = m_{\langle b_4 \rangle_1^B} + m_{\langle b_3 \rangle_1^B} - r_2$.
8: **end for**
9: Both parties set $\langle \mathbf{Y}^* \rangle_i$ as a vector of all $\langle y^* \rangle_i$s computed above and continue to step 6–12 in Figure 4.

Figure 8: Privacy preserving logistic regression protocol.

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 $f(u) = (\neg b_2) + (b_2 \wedge (\neg b_1))u$. Therefore, given $\langle u \rangle$, we construct a garbled circuit that takes the bits of $\langle u + \frac{1}{2} \rangle_0$ and $\langle u \rangle_1$ 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 "$+\frac{1}{2}$" 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, "$+\frac{1}{2}$" is expressed as $\frac{1}{2} \cdot 2^{l_u}$, 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 $(\neg b_2)$ and $b_2 \wedge (\neg b_1)$. We then switch to boolean sharing of these bits and use them in two OTs to compute $\langle (\neg b_2) + (b_2 \wedge (\neg b_1))u \rangle$ and continue with the rest of the training. The detailed protocol is described in Figure 8. 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 8 and send the output to clients. Given a secure garbling scheme, in the $\mathcal{F}_{offline}$ and $\mathcal{F}_{ot}$ hybrid model, this protocol realizes the ideal functionality $\mathcal{F}_{ml}$ of Figure 3 for the logistic regression function, in presence of a semi-honest* admissible *adversary (see*

section 3).

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 4.1. 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 $\mathcal{S}_0$ can be sent simultaneously to $\mathcal{S}_1$. 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.

4.4 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 2.1. 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 $f(u_i) = \frac{e^{-u_i}}{\sum_{i=1}^{d_m} e^{-u_i}}$. 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[6]. 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 [21] 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.

[6]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.

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 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 element-wise 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 $5m \cdot t$.

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 6.3.

4.5 Predictions and Accuracy Testing

The techniques developed so far can also be used to securely make predictions, since the prediction is simply the forward propagation component of one iteration in the training. Similarly, we can also test the accuracy of the current model after each epoch securely, as the accuracy is simply an aggregated result of the predictions on the testing data. The accuracy test can be used to adjust the learning rate or decide when to terminate the training, instead of using a fixed learning rate and training the model for a fixed number of epochs.

Privacy preserving prediction. 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 show the performance of our privacy-preserving predictions in Section 6.4. 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. 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 tests 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.

5 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.

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 4.1, in the forward propagation at step 5 of Figure 4, where we compute $\mathbf{X}_B \times \mathbf{w}$, 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 $\langle y^* \rangle = \mathsf{Mul}^A(\langle \mathbf{x} \rangle, \langle \mathbf{w} \rangle)$ using independently generated multiplication triplets, and set $\langle \mathbf{Y}^* \rangle$ 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 replaced by vector inner products. Though the

total number of multiplications performed is exactly the same, matrix multiplication algorithms are in general faster using matrix libraries in modern programming languages. This is the major overhead introduced by the client-aided approach as depicted in the experiments.

The communication is also increased. Previously, the coefficient vector is masked by a single random vector to compute a single matrix multiplication, while now it is masked multiple times by different random vectors for each inner products. These masked values are transferred between the two parties in the secure computation protocol. In particular, the overhead compared to the protocols in Section 4 is $t \cdot (2|B| \cdot d - |B| - d)$ for linear and logistic regressions. this is not significant in the LAN setting but becomes important in the WAN setting.

Finally, the storage is also increased. Previously, the matrix $\mathbf{V}$ and $\mathbf{Z}$ is much smaller than the data size and the matrix $\mathbf{U}$ is of the same size as the data. Now, as the multiplication triplets are generated independently, the size of $\mathbf{V}$ becomes $|B| \cdot d \cdot t = n \cdot d \cdot E$, which is larger than the size of the data by a factor of E. The size of $\mathbf{U}$ is still the same, as each data is still masked by one random value, and the size of $\mathbf{Z}$ is still the same because the values can be aggregated once the servers collect the shares from all the clients.

Despite all these overheads, the online phase is still very efficient, while the performance of the offline phase is improved dramatically. Therefore, the privacy preserving machine learning with client-aided multiplication triplets generation is likely the most promising option for deployment in existing machine learning frameworks.

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 [20], 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.

6 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 [10], and applies free XOR [29] and fixed-key AES garbling [11] optimizations for garbled circuits. Details can be found in [44]. We use the cryptosystem of DGK [16] for LHE, implemented by Demmler et. al. in [17].

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 signficant gap between the two when using OT extension (see Table 2).

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, aquiring 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 5). 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.

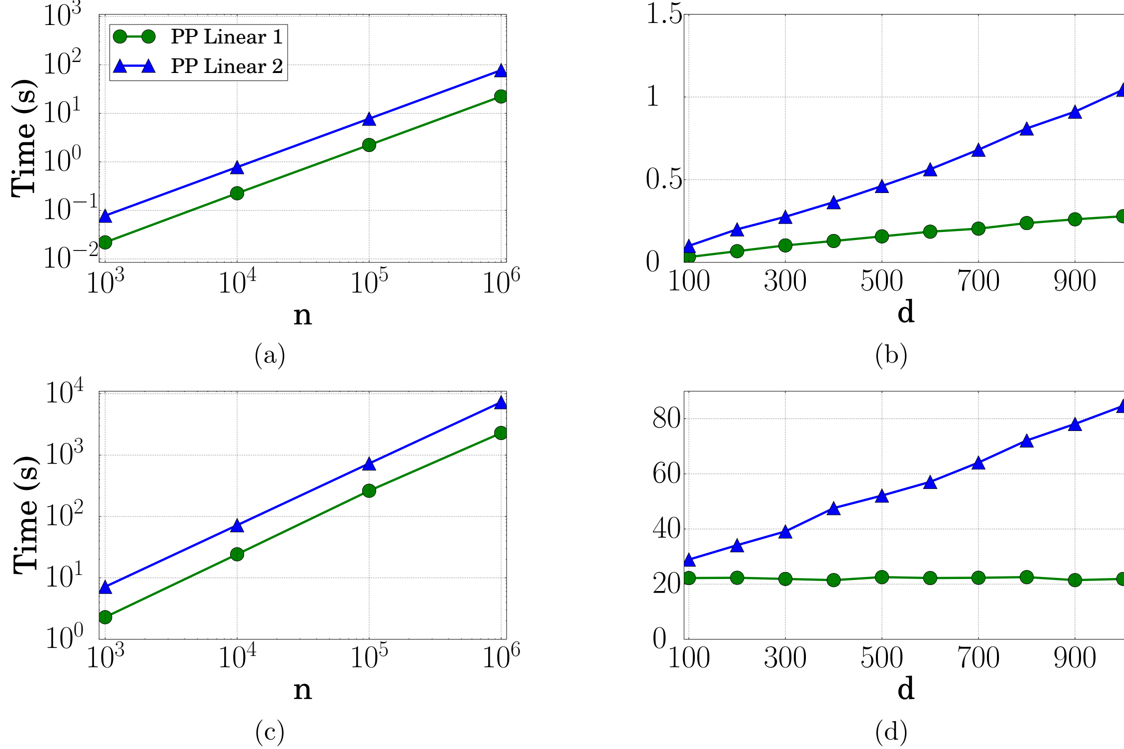

Figure 9: Online cost of privacy preserving linear regression in standard and client-aided settings. $|B|$ is set to 128. Figure (a), (b) are for LAN network and Figure (c), (d) are for WAN network. Figure (a) and (c) are in log-log scale and for $d = 784$. Figure (b) and (d) are in regular scale and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

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, 24] 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, 24]. 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. Besides these real-world datasets, we also use synthetic datasets to test the scalability of our protocols to larger sizes (e.g. a million samples).

6.1 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 MNSIT, 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 9a shows the results in the LAN setting. “PP Linear 1” denotes the online phase of our

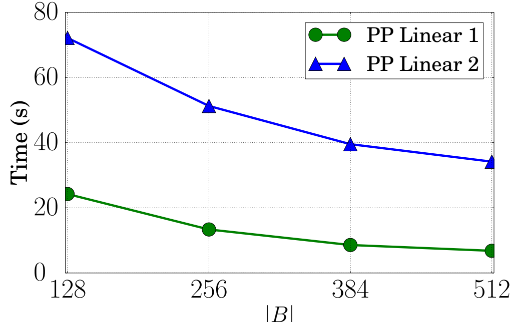

Figure 10: Performance of online phase of linear regression on WAN with different mini-batch sizes. $n = 10,000$, $d = 784$.

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 9a and 9b, 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 9c 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× 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 5. 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 9c that the training time grows linearly with the number of the samples in WAN networks. However, in Figure 9d, 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

		LHE-based			OT-based			Client aided		Dataset
n	d	LAN	WAN	Comm.	LAN	WAN	Comm.	Time	Comm.	size
1,000	100	23.9s	24.0s	2MB	0.86s	43.2s	190MB	0.028s	7MB	0.8MB
	500	83.9s	84.8s	6MB	3.8s	210.6s	1GB	0.16s	35MB	3.8MB
	1000	158.4s	163.2s	10MB	7.9s	163.2s	1.9GB	0.33s	69MB	7.6MB
10,000	100	248.4s	252.9s	20MB	7.9s	420.2s	1.9GB	0.33s	69MB	7.6MB
	500	869.1s	890.2s	60MB	39.2s	2119.1s	9.5GB	1.98s	344MB	38MB
	1000	1600.9s	1627.0s	100MB	80.0s	4097.1s	19GB	4.0s	687MB	76MB
100,000	100	2437.1s	2478.1s	200MB	88.0s	4125.1s	19GB	3.9s	687MB	76MB
	500	8721.5s	8782.4s	600MB	377.9s	20000s*	95GB	20.2s	3435MB	380MB
	1000	16000s*	16100s*	1000MB	794.0s	40000s*	190GB	49.9s	6870MB	760MB

Table 2: Performance of the offline phase. $|B| = 128$ and $E = 2$. (* means estimated via extrapolation.)

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 10 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 2. 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

	MNIST	Gisette	Arcene
Cholesky	92.02%	96.7%	87%
SGD	91.95%	96.5%	86%

Table 3: Comparison of accuracy for SGD and Cholesky.

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 4.2, 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.

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 2 that all the running times grow roughly linearly[7] with both n and d, which agrees with the asymptotic complexity derived in Section 4.2.

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 80.0s.

Comparison with prior work. As surveyed in Section 1.2, privacy preserving linear regression was also considered by [36] (NWI$^+$13) and [20] (GSB$^+$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 2.1. We show that the model trained by the SGD method can reach the same accuracy in Table 3, on the MNIST, Gisette and Arcene datasets.

The protocols in NWI$^+$13 and GSB$^+$16 can be decomposed into two steps. In the first step, the $d \times d$ matrix $\mathbf{X}^T \times \mathbf{X}$ is constructed securely, which defines a linear system. In the second step, the Cholesky algorithm or its variants are implemented using a garbled circuit. In the first step of NWI$^+$13, each client encrypts a $d \times d$ matrix using LHE. In GSB$^+$16, the first step is computed using multiplication triplets generated by the CSP, which is faster than NWI$^+$13. However, now the clients cannot collude with the CSP, which is similar to the model we consider in the client-aided setting.

[7] The number of encryptions and decryptions in the LHE-based generation is $O(|B| + d)$. As $|B|$ is fixed to 128, its running time does not grow strictly linearly with d, as reflected in Table 2.

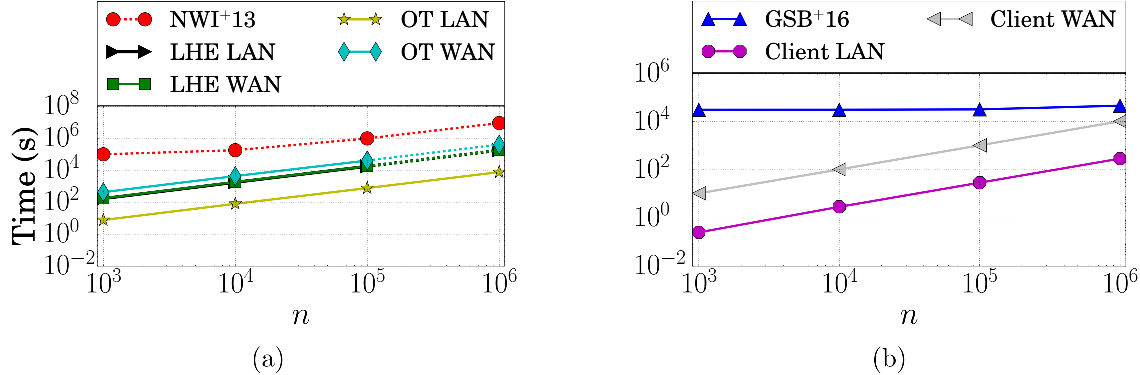

Figure 11: Efficiency comparison with prior work. Figures are in log-log scale, $d = 500$, $|B| = 128$ for our schemes.

Using garbled circuits, NWI^{+}13 implements the Cholesky algorithm while GSB^{+}16 implements CGD, an approximation algorithm.

For comparison, we use the numbers reported in [20, Table 1, Figure 6]. As the performance of the first step of NWI^{+}13 is not reported in the table, we implement it using Paillier's encryption [37] 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 [20, 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 [20, 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 11a, 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$\times$ speedup. The OT-based protocol is 1270$\times$ faster in the LAN setting and 24$\times$ 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 [9].

We further compare the performance of the scheme in GSB^{+}16 and our scheme with client-generated multiplication triplets in Figure 11b, 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$\times$ speedup in WAN setting and a 1110$\times$ 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.

[8] For $n = 1,000,000, d = 500$, since the data point is missing in [20, Table 1], we extrapolate assuming a quadratic complexity in d.

[9] The running time of our scheme using OT-based offline in the WAN setting for $n = 100,000$ and $n = 1,000,000$, using LHE-based offline in LAN and WAN for $n = 1,000,000$ are also estimated (dotted in the figure). The running time using OT-based offline in LAN for $n = 1,000,000$ is from real execution, though the number was not reported in Table 2 due to page limit. Similarly, we were also able to run the client-aided offline for $n = 1,000,000$.

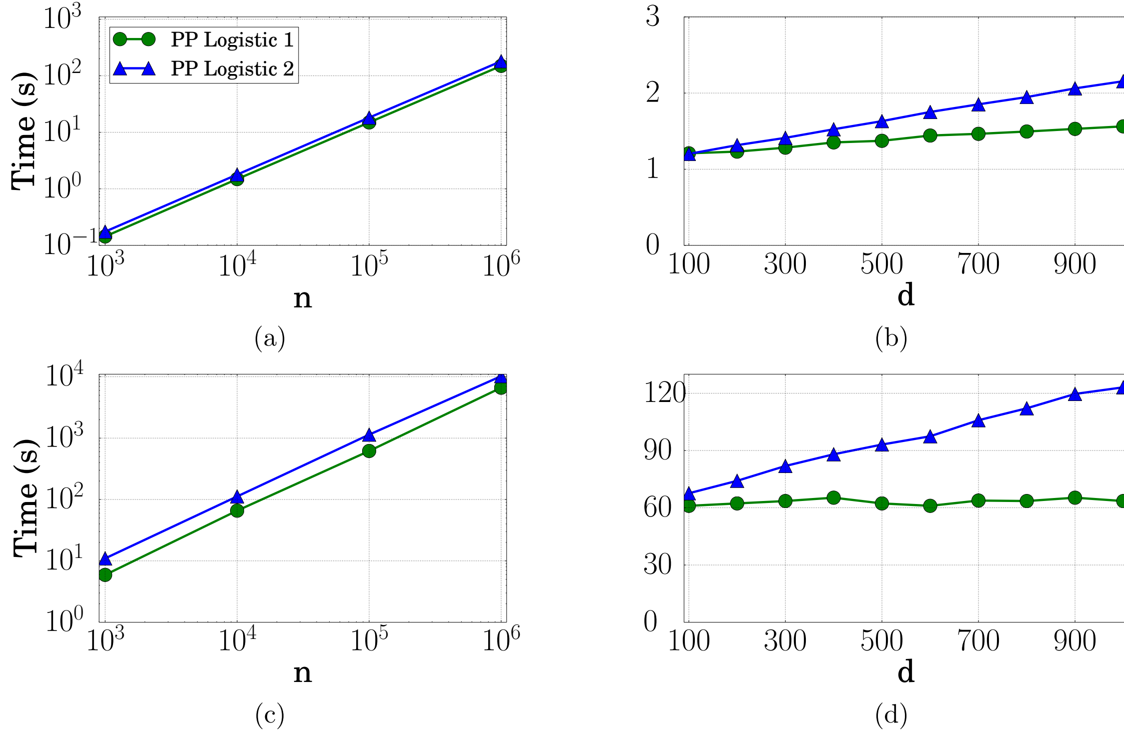

Figure 12: Online cost of privacy preserving logistic regression in the standard and client-aided setting. $|B|$ is set to 128. Figure (a), (b) are for LAN network and Figure (c), (d) are for WAN network. Figure (a) and (c) are in log-log scale, $d = 784$. Figure (b) and (d) are in regular scale, $n = 10,000$.

The reason why the cost of NWI$^+$13 and GSB$^+$16 are 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 work were obtained on a network with 1 Gbps bandwidth [20] which is close to our LAN setting. Indeed, the garbled circuit introduces a huge communication and storage overhead. As reported in [20, Figure 4], the garbled circuits for $d = 500$ in both schemes have more than 10^{11} gates, which is 3000GB. The communication time to transfer such a circuit would be at least 330000s on a WAN network, which implies the speedup for our scheme could be more significant in the WAN setting.

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 $2 \sim 5$ 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.

	Protocol 1		Protocol 2	
	Offline	Online	Offline	Online
RELU	290,000s*	4239.7s	14951.2s	10332.3s
Square	320,000s*	653.0s	16783.9s	4260.3s

Table 4: Performance of privacy preserving neural networks training on MNIST in LAN setting. $n = 60,000$, $d = 784$.

6.2 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 12, our privacy preserving logistic regression introduces some overhead on top of the linear regression. Specifically, in Figure 12a, 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 12a and 12b.

Figure 12c and 12d 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 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.

6.3 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 4. $|B|$ is set to 128 and the training converges after 15 epochs.

As shown in the table, when RELU function is used, the online phase of our first protocol takes 4239.7s, while the offline phase using OT takes around $2.9\times10^5 s$. When the square function is used, the performance of the online phase is improved significantly, as most of the garbled circuits are replaced by multiplications on secret shared values. In particular, it only takes 653.0s for the online phase of our first protocol. The running time of the offline phase is increased, showing a trade-off between the two phases. Using client-aided multiplication triplets, the offline phase is further reduced to about 1.5×10^4s, with an overhead on the online phase.

Due to high number of interactions and high communication, the neural network training on WAN setting is not yet practical. To execute one round of forward and backward propagation in the neural network, the online phase takes 30.52s using RELU function and the offline phase takes around 2200s using LHE-based approach. The total running time is linear in the number of rounds, which is around 7000 in this case.

In terms of the accuracy, the model trained by our protocol can reach 93.4% using RELU and 93.1% using the square function. In practice, there are other types of neural networks that can reach better accuracy. For example, the *convolutional* neural networks are believed to work better for image processing tasks. In such neural networks, the neurons are not fully connected and the inner product between the data and the coefficients is replaced by a 2-D convolution. In principle, we can also support such neural networks, as the convolution can be computed using additions and multiplications. However, improving the perfomance using techniques such as Fast Fourier Transform inside secure computation is interesting open questions. Experimenting with various MPC-friendly activations is another avenue for research.

6.4 Experiments for predictions.

Table 5 summarizes the cost of predictions. We use samples from the evaluation dataset of MNIST with $d = 784$. The neural network we use is the same as described in Section 6.3. It has 2 hidden layers with 128 neurons each, and outputs a hot vector with 10 elements. We assume that the models remain secret shared between the two servers. As shown in the table, 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.

	k	Linear (ms)		Logistic (ms)		Neural (s)	
		Online	Offline	Online	Offline	Online	Offline
LAN	1	0.20	2.5	0.59	2.5	0.18	4.7
	100	0.22	51	3.9	51	0.20	13.8
WAN	1	72	620	158	620	0.57	17.8
	100	215	2010	429	2010	1.2	472

Table 5: Online and offline performances for privacy preserving predicition. $d = 784$. The neural network is the same as the one in Section 6.3.

6.5 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.

		Total (OT)	Total (LHE)	GC
LAN	$k = 1000$	0.028s	5.3s	0.13s
	$k = 10,000$	0.16s	53s	1.2s
	$k = 100,000$	1.4s	512s	11s
WAN	$k = 1000$	1.4s	6.2s	5.8s
	$k = 10,000$	12.5s	62s	68s
	$k = 100,000$	140s	641s	552s

Table 6: Comparison of our decimal multiplication and the fixed-point multiplication using garbled circuit.

Arithmetic on shared decimal numbers. Table 6 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, 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.

	d	Online	Online Vec	OT	OT Vec	LHE	LHE Vec
LAN	100	0.37ms	0.22ms	0.21s	0.05s	67s	1.6s
	500	1.7ms	0.82ms	1.2s	0.28s	338s	5.5s
	1000	3.5ms	1.7ms	2.0s	0.46s	645s	10s
WAN	100	0.2s	0.09s	14s	3.7s	81s	2s
	500	0.44s	0.20ss	81s	19s	412s	6.2s
	1000	0.62s	0.27s	154s	34s	718s	11s

Table 7: Speedup from vectorization. $|B| = 128$.

Vectorization. Table 7 shows the speedup gained from vectorization. As a basic operation in our training, we need to multiply a shared $|B| \times d$ 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 7, 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.

	New Logistic	Poly Total Client-aided	Poly Total OT	Poly Total LHE
LAN	0.0045s	0.0005s	0.025s	6.8s
WAN	0.2s	0.69s	2.5s	8.5s

Table 8: Performance of our new logistic function and polynomial approximation.

New logistic function. We compare the cost of calculating our new logistic function with approximating the logistic function using a degree 10 polynomial. For the polynomial approximation, we use our scheme for decimal multiplications and compute it using 9 sequential multiplications using the Horner's rule. Table 8 shows the running time for 128 parallel evaluations of the function (just to amortize the effect of network delay). As shown in the table, unless using client-aided multiplication triplets in LAN networks, which weakens the security model, our new logistic function is dramatically faster than using polynomial approximation ($3.5 - 1511\times$).

Acknowledgements

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 #1514261 and #1526950.

References

[1] Arcene data set. `https://archive.ics.uci.edu/ml/datasets/Arcene`. Accessed: 2016-07-14.

[2] Eigen library. `http://eigen.tuxfamily.org/`.

[3] EMP toolkit. `https://github.com/emp-toolkit`.

[4] Gisette data set. `https://archive.ics.uci.edu/ml/datasets/Gisette`. Accessed: 2016-07-14.

[5] GMP library. `https://gmplib.org/`.

[6] MNIST database. `http://yann.lecun.com/exdb/mnist/`. Accessed: 2016-07-14.

[7] NTL library. `http://www.shoup.net/ntl/`.

[8] Abadi, M., Chu, A., Goodfellow, I., McMahan, H. B., Mironov, I., Talwar, K., and Zhang, L. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 308–318.

[9] Aono, Y., Hayashi, T., Trieu Phong, L., and Wang, L. Scalable and secure logistic regression via homomorphic encryption. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (2016), ACM, pp. 142–144.

[10] Asharov, G., Lindell, Y., Schneider, T., and Zohner, M. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the ACM CCS 2013* (2013).

[11] Bellare, M., Hoang, V. T., Keelveedhi, S., and Rogaway, P. Efficient garbling from a fixed-key blockcipher. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 478–492.

[12] Bellare, M., Hoang, V. T., and Rogaway, P. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 784–796.

[13] Bunn, P., and Ostrovsky, R. Secure two-party k-means clustering. In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 486–497.

[14] Canetti, R. Universally composable security: A new paradigm for cryptographic protocols. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on* (2001), IEEE, pp. 136–145.

[15] Chaudhuri, K., and Monteleoni, C. Privacy-preserving logistic regression. In *Advances in Neural Information Processing Systems* (2009), pp. 289–296.

[16] Damgard, I., Geisler, M., and Kroigard, M. Homomorphic encryption and secure comparison. *International Journal of Applied Cryptography 1*, 1 (2008), 22–31.

[17] Demmler, D., Schneider, T., and Zohner, M. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS* (2015).

[18] Du, W., and Atallah, M. J. Privacy-preserving cooperative scientific computations. In *csfw* (2001), vol. 1, Citeseer, p. 273.

[19] Du, W., Han, Y. S., and Chen, S. Privacy-preserving multivariate statistical analysis: Linear regression and classification. In *SDM* (2004), vol. 4, SIAM, pp. 222–233.

[20] Gascon, A., Schoppmann, P., Balle, B., Raykova, M., Doerner, J., Zahur, S., and Evans, D. Secure linear regression on vertically partitioned datasets.

[21] Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., and Wernsing, J. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of The 33rd International Conference on Machine Learning* (2016), pp. 201–210.

[22] Gilad-Bachrach, R., Laine, K., Lauter, K., Rindal, P., and Rosulek, M. Secure data exchange: A marketplace in the cloud. Cryptology ePrint Archive, Report 2016/620, 2016. `http://eprint.iacr.org/2016/620`.

[23] Gilad-Bachrach, R., Laine, K., Lauter, K., Rindal, P., and Rosulek, M. Secure data exchange: A marketplace in the cloud.

[24] Guyon, I., Gunn, S., Ben-Hur, A., and Dror, G. Result analysis of the nips 2003 feature selection challenge. In *Advances in neural information processing systems* (2004), pp. 545–552.

[25] Hastie, T., Tibshirani, R., and Friedman, J. The elements of statistical learning – data mining, inference, and prediction.

[26] Ishai, Y., Kilian, J., Nissim, K., and Petrank, E. Extending oblivious transfers efficiently. *Advances in Cryptology-CRYPTO 2003* (2003), 145–161.

[27] Jagannathan, G., and Wright, R. N. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining* (2005), ACM, pp. 593–599.

[28] Kamara, S., Mohassel, P., and Raykova, M. Outsourcing multi-party computation. *IACR Cryptology ePrint Archive* (2011), 272.

[29] Kolesnikov, V., and Schneider, T. Improved garbled circuit: Free xor gates and applications. In *International Colloquium on Automata, Languages, and Programming* (2008), Springer, pp. 486–498.

[30] Lindell, Y., and Pinkas, B. Privacy preserving data mining. In *Annual International Cryptology Conference* (2000), Springer, pp. 36–54.

[31] Lindell, Y., and Pinkas, B. A proof of security of yaos protocol for two-party computation. *Journal of Cryptology 22*, 2 (2009), 161–188.

[32] Livni, R., Shalev-Shwartz, S., and Shamir, O. On the computational efficiency of training neural networks. In *Advances in Neural Information Processing Systems* (2014), pp. 855–863.

[33] Malkhi, D., Nisan, N., Pinkas, B., Sella, Y., et al. Fairplay-secure two-party computation system.

[34] Nayak, K., Wang, X. S., Ioannidis, S., Weinsberg, U., Taft, N., and Shi, E. Graphsc: Parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 377–394.

[35] Nikolaenko, V., Ioannidis, S., Weinsberg, U., Joye, M., Taft, N., and Boneh, D. Privacy-preserving matrix factorization. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 801–812.

[36] Nikolaenko, V., Weinsberg, U., Ioannidis, S., Joye, M., Boneh, D., and Taft, N. Privacy-preserving ridge regression on hundreds of millions of records. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 334–348.

[37] Paillier, P. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques* (1999), Springer, pp. 223–238.

[38] Peikert, C., Vaikuntanathan, V., and Waters, B. A framework for efficient and composable oblivious transfer. *Advances in Cryptology–CRYPTO 2008* (2008), 554–571.

[39] Sanil, A. P., Karr, A. F., Lin, X., and Reiter, J. P. Privacy preserving regression modelling via distributed computation. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining* (2004), ACM, pp. 677–682.

[40] Shokri, R., and Shmatikov, V. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1310–1321.

[41] Slavkovic, A. B., Nardi, Y., and Tibbits, M. M. " secure" logistic regression of horizontally and vertically partitioned distributed databases. In *Seventh IEEE International Conference on Data Mining Workshops (ICDMW 2007)* (2007), IEEE, pp. 723–728.

[42] Song, S., Chaudhuri, K., and Sarwate, A. D. Stochastic gradient descent with differentially private updates. In *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE* (2013), IEEE, pp. 245–248.

[43] Vaidya, J., Yu, H., and Jiang, X. Privacy-preserving svm classification. *Knowledge and Information Systems 14*, 2 (2008), 161–178.

[44] Wang, X., Malozemoff, A. J., and Katz, J. Faster two-party computation secure against malicious adversaries in the single-execution setting. Cryptology ePrint Archive, Report 2016/762, 2016. `http://eprint.iacr.org/2016/762`.

[45] Wu, S., Teruya, T., Kawamoto, J., Sakuma, J., and Kikuchi, H. Privacy-preservation for stochastic gradient descent application to secure logistic regression. *The 27th Annual Conference of the Japanese Society for Artificial Intelligence 27* (2013), 1–4.

[46] Yao, A. C. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on* (1982), IEEE, pp. 160–164.

[47] Yu, H., Vaidya, J., and Jiang, X. Privacy-preserving svm classification on vertically partitioned data. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (2006), Springer, pp. 647–656.