Towards Deep Neural Network Training on Encrypted Data

Karthik Nandakumar
IBM Research
Singapore
nkarthik@sg.ibm.com

Nalini Ratha, Sharath Pankanti
IBM Research
Yorktown Heights, NY 10598
{ratha,sharat}@us.ibm.com

Shai Halevi
Algorand Foundation
Boston, MA
shaih@alum.mit.edu

Abstract

While deep learning is a valuable tool for solving many tough problems in computer vision, the success of deep learning models is typically determined by: (i) availability of sufficient training data, (ii) access to extensive computational resources, and (iii) expertise in selecting the right model and hyperparameters for the selected task. Often, the availability of data is the hard part due to compliance, legal, and privacy constraints. Cryptographic techniques such as fully homomorphic encryption (FHE) offer a potential solution by enabling processing on encrypted data. While prior work has been done on using FHE for inferencing, training a deep neural network in the encrypted domain is an extremely challenging task due to the computational complexity of the operations involved. In this paper, we evaluate the feasibility of training neural networks on encrypted data in a completely non-interactive way. Our proposed system uses the open-source FHE toolkit HElib to implement a Stochastic Gradient Descent (SGD)-based training of a neural network. We show that encrypted training can be made more computationally efficient by (i) simplifying the network with minimal degradation of accuracy, (ii) choosing appropriate data representation and resolution, and (iii) packing the data elements within the ciphertext in a smart way so as to minimize the number of operations and facilitate parallelization of FHE computations. Based on the above optimizations, we demonstrate that it is possible to achieve more than $50\times$ *speed up while training a fully-connected neural network on the MNIST dataset while achieving reasonable accuracy* (96%). *Though the cost of training a complex deep learning model from scratch on encrypted data is still very high, this work establishes a solid baseline and paves the way for relatively simpler tasks such as fine-tuning of deep learning models based on encrypted data to be implemented in the near future.*

1. Introduction

Deep neural networks are a powerful tool with a wide range of applications, from speech to vision and much more. Solutions that use deep neural networks consists of two main phases, namely training and inference: After appropriate datasets are identified and curated, a network architecture is established, and then the identified corpus of data is used to train it, i.e., to learn the weights for the network. Once the network weights are stable and provide meaningful results for the application at hand, the network can be used for inferencing, where it renders predictions on new data. While the training time may run into days, inferencing is expected to be fast.

There are many scenarios where the data needed for training deep neural networks is extremely sensitive. For example, credit card transaction information is available with the credit card company but not for external developers. Similarly healthcare data related to patients is available in a hospital but not for a researcher to find patterns in the data for understanding cancer progression. Moreover, privacy concerns (such as the new European data privacy regulation GDPR) may restrict the availability of data. The data owner may often lack the knowledge and proficiency to build deep learning models on their own to derive the benefits from their data. On the other hand, confidentiality and privacy constraints prevent sharing of the data with external service providers.

Cryptographic techniques for fully homomorphic encryption (FHE) offer an appealing approach for resolving the conundrum between usefulness and sensitivity of data. However, the current wisdom is that such techniques are too slow to handle the training of commonly used deep neural network models. The original proposal of fully homomorphic encryption was touted as a revolutionary technology [11], with potential far-reaching implications to cloud computing and beyond. Though only a theoretical plausibility result at first, the last decade saw major algorithmic improvements (e.g., [6, 5, 13]), resulting in many research prototypes that implement this technology and attempt to use it in different settings (e.g., [4, 12, 17, 8, 14, 19], among others).

In this work, we propose using FHE to enable training neural network models on encrypted data. This would allow the users to encrypt the data using their private (secret) key and share the encrypted data to the service provider. The service provider can train the model without ever seeing the underlying data. Since the resulting model will also be encrypted, the service provider learns nothing about the data or the learned model parameters. Moreover, the resulting model is useful only to the users with access to the private key (used for encrypting the training data) and cannot be shared with anyone else. Thus, the proposed approach is applicable to scenarios where the data owner wishes to outsource the deep learning computations to an external service provider, who has the expertise (e.g., suitable model architecture) and computational resources to execute the learning task, while ensuring that the service provider does not derive any undue benefits from the data or the model.

Our main contributions in this paper are three fold:

- To the best of our knowledge, this is **the first paper** to attempt training a deep neural network in a non-interactive way on data encrypted using fully homomorphic encryption. Given that this is a hitherto unexplored problem setting, our primary objective is to identify the critical bottlenecks and establish a solid baseline.
- In the context of distributed neural network training, several studies have shown that errors introduced due to fixed point encodings are tolerable. However, none of these studies perform the complete training process in the fixed-point domain. They typically employ floating point operations for some intermediate computations and truncate them. While such tricks are possible in interactive protocols previously attempted in the literature, is not feasible for non-interactive training addressed in our work. In the absence of careful parameter selection for data encoding, the errors can quickly accumulate (over multiple layers and training iterations) and prevent convergence. Thus, our second contribution is to demonstrate that it is possible to fully train a neural network in the fixed-point domain and achieve convergence and reasonable accuracy. Without this result, non-interactive FHE training is not possible at all. At the same time, this result is independent of FHE, and may have its own applications (e.g., more efficient training of neural networks in plaintext).
- Our final contribution relates to speeding up FHE computations through smart implementation of ciphertext packing. While ciphertext packing is a well-known technique in FHE, we use it intelligently in this work both to minimize the number of bootstrapping operations required and to enable the parallelization of computations at each neuron, thereby providing significant reduction in the computational complexity.

2. Related Work

While privacy-preserving machine learning has been studied for nearly twenty years [21, 1], not much work has been done on specifically using homomorphic encryption in the context of neural networks. The only noteworthy prior work that we found using *non-interactive homomorphic encryption* for neural networks is the Crypto-Nets work of Gilad-Bachrach et al. [14]. That work demonstrated a carefully-designed neural network that can run the inference phase on encrypted data, with 99% accuracy on the MNIST optical character recognition tasks, achieving amortized rate of almost 60,000 predictions/hour. More recently, protocols for secure face matching [3] and secure k-nearest neighbor search [27] have been proposed based on fully homomorphic encryption.

There has been more work about using homomorphic encryption in conjunction with interactive secure-computation protocols in the context of neural networks. An early work along these lines is due to Barni et al. and Orlandi et al. [2, 25], that combined additively-homomorphic encryption with an interactive protocol, and were able to run the inference part of a small network in about ten seconds. Many more interactive protocols for the inference phase were suggested recently, including SecureML of Mohassel and Zhang [23], MiniONN of Liu et al. [22], Chameleon of Riazi et al. [26], and GAZEELE of Juvekar et al. [28]. The last of these can perform the inference phase of MNIST as fast as 30ms, and the CIFAR-10 benchmark in just 13 seconds.

All these works address only the inference phase of using the network, none of them addresses the training phase. In fact we were not able to find *any prior work that deals with private training of neural networks*. Presumably, this is due to the perception that trying to train homomorphically will be so slow as to render it unusable. Furthermore, training complex machine learning models sometimes requires conditionals (comparison and selection), which until recently were considered infeasible using only homomorphic encryption [7]. In the current work, we take the first step toward dispelling the slowness perception, showing that *even non-interactive homomorphic encryption* can be used for training, in some restricted cases.

Some prior work described how to preform training and inference for other types of models on encrypted data, specifically linear-regression models [24] and even logistic-regression models [29, 15, 19, 18, 10].

2.1. Fully Homomorphic Encryption

Nearly all contemporary FHE schemes come with two components: The basic underlying scheme is *somewhat homomorphic* (SWHE), where the parameters are set depending on the complexity of the required homomorphic operations, and the resulting instance can only support computations up to that complexity. The reason is that ciphertexts

are "noisy", with the noise growing throughout the computation, and once the noise grows beyond some (parameter-dependent) threshold the ciphertext can no longer be decrypted. This can be solved using Gentry's bootstrapping technique (at the cost of relying on circular security). In this technique the scheme is augmented with a *recryption* operation to "refresh" the ciphertext and reduce its noise level. The augmented scheme is thus *fully homomorphic*(FHE), meaning that a single instance with fixed parameters can handle arbitrary computations. But FHE is expensive, as the computation must be peppered with expensive recryption operations. So, it is often cheaper to settle for a SWHE scheme with larger parameters (that admit larger noise).

Indeed, with very few exceptions, almost all prior attempts at practical use of HE used only the SWHE component, fixing the target computation and then choosing parameters that can handle that computation and no more. But SWHE has its limits: as the complexity of the function grows, the SWHE parameters become prohibitively large. In this work, we set out to investigate the practical feasibility of using FHE to compute a complex non-linear optimization function such as neural network training.

3. Proposed Solution

In this section, we describe the deep learning model and the components of the solution needed to implement model learning in the encrypted domain using fully homomorphic encryption. In this work, we primarily focus on supervised deep learning, where the broad objective is to learn a non-linear mapping between the inputs (training samples) and the outputs (class labels of the training samples). The learning scenario considered in this work is illustrated in Figure 1. Suppose that a data owner having access to private training data would like to outsource the model learning to a service provider, while preserving the confidentiality of the training data. The data (including class labels) is encrypted is by the data owner and shared with the service provider, who defines the model architecture (denoted by function $g_\theta(.)$). The goal is learn the parameters (θ) in order to minimize a pre-defined loss function $\mathcal{L}$. The service provider learns the model in the encrypted domain and returns the model architecture along with the encrypted model parameters to the data owner. The data owner can decrypt the model parameters and use them for inferencing. Alternatively, the service provider may retain the encrypted model parameters and use them to perform inferencing on the encrypted test (query) samples provided by the data owner. In the latter scenario, the service provider returns the encrypted inference result to the data owner, who can decrypt the result.

3.1. Deep Learning Model

Deep learning models are typically implemented as multi-layer neural networks, which allows higher-level abstract features to be computed as non-linear functions of lower-level features (starting with the raw data). Figure 2 shows a typical neural network with two hidden layers. The output of each node in the network (also known as a neuron) is computed by applying a non-linear activation function to the weighted average of its inputs, which includes a bias term that always emits value 1. The output vector of neurons in layer ℓ ($\ell = 1,\ 2,\ \cdots, L$) is obtained as:

$$\boldsymbol{a}_\ell = f(\boldsymbol{W}_\ell \boldsymbol{a}_{\ell-1}), \tag{1}$$

where f is the activation function, $\boldsymbol{W}_\ell$ is the weight matrix of layer ℓ, and L is the total number of layers in the network.

Given the training data $\{\boldsymbol{x}_i, y_i\}_{i=1}^N$, the goal is to learn the parameters (weight matrices) in order to minimize a pre-defined loss function $\mathcal{L}$. This is a non-linear optimization problem, which is typically solved using variants of gradient descent. Gradient descent starts with a random set of parameters, computes the gradient of the loss function $\mathcal{L}$ at each step, and updates the parameters so as to decrease the gradient. In this work, we use the well-known stochastic gradient descent (SGD) algorithm [30], where the gradients are averaged over a small (randomly sampled without replacement) subset (mini-batch) of the whole training dataset. One full iteration over the entire training set is referred to as the epoch. The above gradient update step is repeated until convergence to a local optimum or until the maximum number of epochs is reached. The update rule for SGD for weight matrix $\boldsymbol{W}_\ell$ is

$$\boldsymbol{W}_\ell := \boldsymbol{W}_\ell - \alpha \frac{\partial \mathcal{L}_B}{\partial \boldsymbol{W}_\ell}, \tag{2}$$

where $\mathcal{L}_B$ is the loss function computed over the mini-batch B and α is the learning rate. The error or loss value at the output layer is computed based on the forward pass, while backpropagation is used to propagate this error back through the network.

3.2. Homomorphic Implementation of Deep Learning

We use the open-source fully homomorphic encryption library called HElib [16] as our primary toolkit to implement the model learning procedure. Devising a homomorphic computation of this procedure brings up many challenges. Here we briefly discuss some of them.

Implementing the basic homomorphic operation: Most of the operations in model learning are linear, involving additions and multiplications. The current version of HElib that we use supports addition and multiplication operations of arbitrary numbers in binary representation, using encryption of the individual bits. This means that we use the underlying homomorphic encryption scheme with native plaintext space

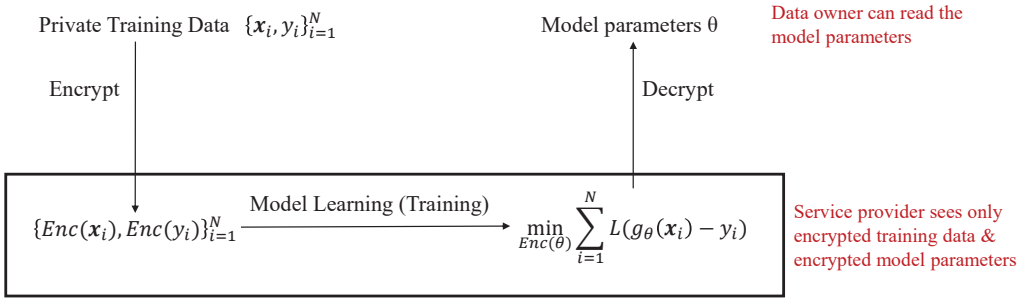


Figure 1. A typical scenario for model learning based on encrypted data.

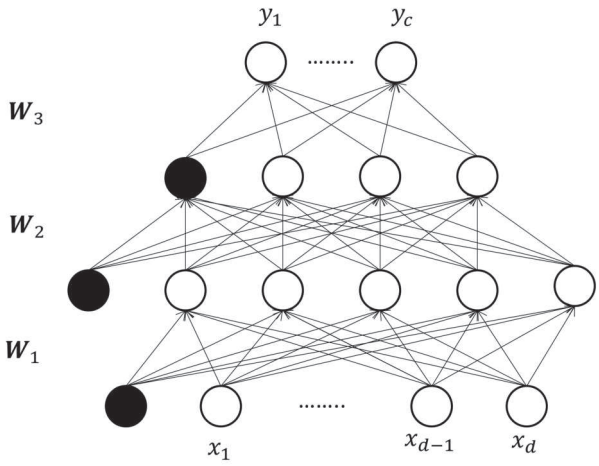


Figure 2. A typical neural network with two hidden layers. Here, the black circles denote bias nodes that always emits value 1. Weight matrices $\boldsymbol{W}_\ell$ determine the contribution of each input signal to the activation function at a node.

modulo 2, which we use to encrypt the bits of the input. While computing the addition and multiplication operations homomorphically is mostly a matter of implementing textbook routines (e.g., carry look ahead for addition), in this context we are extremely sensitive to the computation depth, which is not typical in other implementations. Therefore, several optimizations are required to efficiently implement these operations and these optimization tricks are described in [9].

Two key steps in the algorithm require computing "complex" functions (such as exponentiation, etc.). These two steps are (i) computation of the activation function f and its derivative, and (ii) computation of the loss function $\mathcal{L}$ and its derivative. The "natural" approaches for computing these functions homomorphically, are either to approximate them by low-degree polynomials (e.g., using their Taylor expansion), or by pre-computing them in a table and performing homomorphic table lookup. Namely, for a function f that we need to compute, we pre-compute (in the clear) a table T_f such that $T_f[x] = f(x)$ for every x in some range. Subsequently, given the encryptions of the (bits of) x, we perform homomorphic table lookup to get the (bits of) value $T_f[x]$. Following Crawford et al. [10], we adopt the second approach here. This is faster and shallower when it is applicable, but it can only be used to get a low-precision approximation of these functions. In order to avoid the use of too many table lookups, we will use sigmoid activation function and quadratic loss function, which have simpler derivatives.

Parameters and Bootstrapping: We adopted the parameter setting used in [10], which means that the "production version" of our solution uses the cyclotomic ring $\mathbb{Z}[X]/(\Phi_m(X))$, with $m = 2^{15} - 1$, corresponding to lattices of dimension $\phi(m) = 27000$. This native plaintext space yields 1800 plaintext slots, each holding a bit. (Each slot can actually hold an element of $GF(2^{15})$, but we only use the slots to hold bits in our implementation). The other parameters are chosen so as to get security level of about 80 bits, and this parameter setting allows bootstrapping, so we can evaluate the deep circuits that are required for training.

Most of our development and testing was done on a toy setting of parameters, with $m = 2^{10} - 1$ (corresponding to lattices of dimension $\phi(m) = 600$). For that setting we have only 60 plaintext slots per ciphertext (each capable of holding an element of $GF(2^{10})$, but only used to hold a single bit).

3.3. Data Representation and Encoding

All the operations in the proposed solution are applied to integers in binary representation (i.e., using encryption of the individual bits).

Input & Output: We use 8-bit signed integer representation for the inputs to the network. The outputs of the network are the weight matrices and each element in the weight matrix is represented as a 16-bit signed integer. To deal with negative integers, we use the 2s-complement representation wherever necessary.

Ciphertext packing: We set the mini-batch size during training to be the same as the number of slots in the plaintext space. Note that for our final implementation, $m = 2^{15} - 1$ and the number of slots is 1800. The ciphertexts are represented as 2-dimensional arrays, i.e., encryptedInput[i][0] contains the encryption of the least significant bits of all the 1800 numbers in the i-th dimension of the input. Similarly, encryptedInput[i][7] contains the encryptions of the most significant bits.

Matrix Multiplication: One of the critical and time-consuming operations in the encrypted domain, especially in the context of mini-batch SGD, is matrix multiplication. Since computation of dot products is not straightforward due to the way in which the inputs are packed in a ciphertext, we adopt the following simple approach for matrix multiplication in the encrypted domain. Suppose $\boldsymbol{A} = [a_{ij}]$ and $\boldsymbol{B} = [b_{jk}]$ are two matrices, where $i = 1, \cdots, d_i$, $j = 1, \cdots, d_j$ and $k = 1, \cdots, d_k$. Let $\boldsymbol{C} = [c_{ik}]$ be the product of $\boldsymbol{A}$ and $\boldsymbol{B}$. Then,

$$\boldsymbol{c}_{i\cdot} = \sum_{j=1}^{d_j} \alpha_{ij} \boldsymbol{b}_{j\cdot}, \tag{3}$$

where $\boldsymbol{c}_{i\cdot}$ is a ciphertext packing all the elements in the i^{th} row of $\boldsymbol{C}$, α_{ij} is a ciphertext containing the encryption of value a_{ij} in all the slots, and $\boldsymbol{b}_{j\cdot}$ is a ciphertext packing all the elements in the j^{th} row of $\boldsymbol{B}$. Thus, each matrix multiplication involves $d_i \times d_j$ ciphertext multiplications.

While there are more efficient methods of ciphertext packing such as diagonal order packing for generic matrix multiplication tasks, those methods are not suitable for neural network learning. This is because multiple matrix multiplications have to be executed sequentially during the forward and backward training passes, and sophisticated packing approaches require expensive re-ordering of elements within a ciphertext after each network layer. This eventually leads to more bootstrapping operations, which further slows down the FHE computations.

4. Results

In this section, we describe the dataset used in our experiment as well as the results in terms of accuracy and timing.

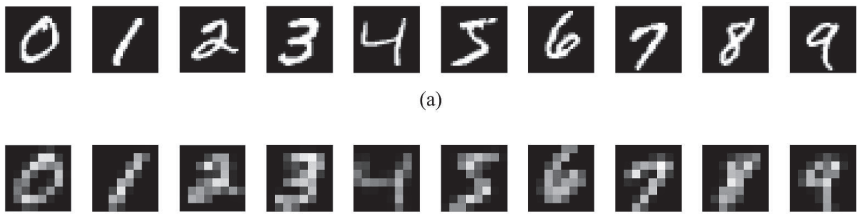
4.1. Dataset and Neural Network Parameter Selection

We conduct experiments on the standard MNIST benchmark dataset [20] for handwritten digit recognition consisting of 60,000 training examples and 10,000 test examples. Each example is a 28×28 gray-level image, with digits located at the center of the image. The architecture of the neural network used in this work is shown in Figure 4, which is simply a 3-layer fully connected network with sigmoid activation function. We normalize all the samples (both training and test) by subtracting the average and dividing by the standard deviation of training samples. This normalized image is finally vectorized to obtain a d_0-dimensional representation, which forms the input to the neural network, i.e, $\boldsymbol{x}_i \in \mathcal{R}^{d_0}$.

Since the objective is to classify the input as one of 10 possible digits within ["0"-"9"], we set the size of the output layer as 10. The desired output is represented as a 10-dimensional vector $\boldsymbol{y}_i = [y_{i,0}, \cdots, y_{i,9}]$, with value $y_{i,j} = 1$ if the sample belongs to j^{th} class and 0 otherwise. We use a quadratic loss function at the output during training, i.e., $\mathcal{L} = (||\boldsymbol{a_L} - \boldsymbol{y}_i||^2)/2$. During inferencing, the input sample is assigned to the class whose corresponding neuron has the highest activation.

We consider two different sets of parameters for the above 3-layer neural network. Firstly, we present the full 784-dimensional (28×28) input to the neural network (denoted as NN1), which contained 128 and 32 neurons in the two hidden layers. Consequently, the number of parameters to be learned is $104,938$ ($= (128 \times 785) + (32 \times 129) + (10 \times 33)$). Since learning such a large number of parameters is currently beyond the reach of most FHE schemes, we also consider a much smaller network (denoted as NN2) with $d_0 = 64$, and containing 32 and 16 neurons in the two hidden layers (see Figure 4). This is achieved by cropping only the central 24×24 pixels of each image and rescaling the image by a factor of $(1/3)$ using bicubic interpolation to obtain a 8×8 pixel representation. Figure 3 shows some examples of the raw and processed MNIST images. For the latter network, the number of parameters to be learned is only $2,778$ ($= (32 \times 65) + (16 \times 33) + (10 \times 17)$).

The weights of the network are randomly initialized by sampling from a Gaussian distribution with zero mean and a standard deviation of 0.1. Though quadratic loss function and sigmoid activation function may not be optimal choices for the selected application, we nevertheless employ them to avoid the need for complex table lookups during back-propagation. Note that the sigmoid activation function is given by $f(z) = (1 + exp(-z))^{-1}$ and its derivative can be easily computed as $f'(z) = f(z)(1 - f(z))$, without the need for any rational divisions. Similarly, the derivative of the quadratic loss function is simply $(\boldsymbol{a}_L - \boldsymbol{y}_i)$. Thus, the entire training process requires the computation of only one



(a)

(b)

Figure 3. Samples from the MNIST training dataset (a) (28×28) pixel representation before any preprocessing and (b) (8×8) pixel representation after cropping and rescaling (displayed with the same size as the top row for comparison).

```
nn.Sequential{
  [input -> (1) -> ... -> (7) -> output]
  (1): nn.Reshape(64)
  (2): nn.Linear(64->32)
  (3): nn.Sigmoid
  (4): nn.Linear(32->16)
  (5): nn.Sigmoid
  (6): nn.Linear(16->10)
  (7): nn.Sigmoid
}
```

Figure 4. Neural network architecture (NN2) used for MNIST dataset with 64 inputs. Cropping of boundary pixels and rescaling using bicubic interpolation are used to reduce the original MNIST images to (8×8) pixels. The cropping and rescaling operations are performed in the plaintext domain and the 64 inputs are then encrypted using the FHE scheme.

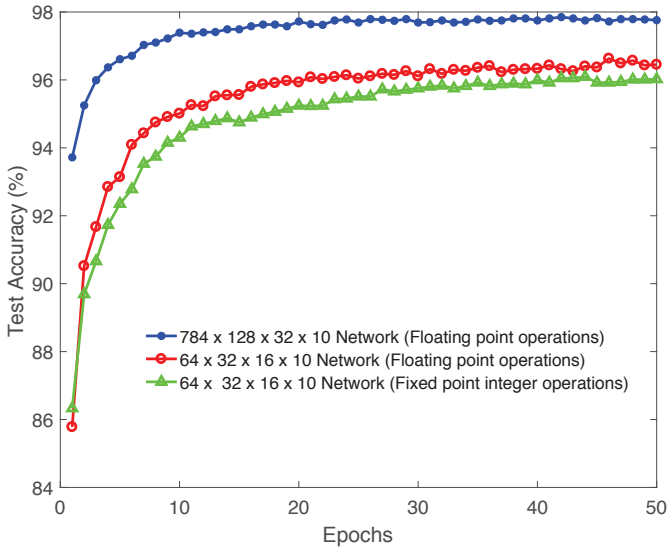


Figure 5. Classification accuracy of the proposed neural networks on the MNIST test dataset.

complex function, namely, the sigmoid function, which is implemented as a 8-bit table lookup as described in Section 3.2. However, it must be noted that the data owner needs to encrypt the look-up table required by the service provider. Therefore, in principle, other activation functions such as ReLU can indeed be implemented using lookup tables in the encrypted learning context.

4.2. Classification Accuracy

Both the networks (NN1 and NN2) described in the previous section are trained using mini-batch SGD with a batch size of 60 samples. When these networks are trained for 50 epochs using full floating point operations, they achieve an overall classification accuracy of 97.8% (for NN1) and 96.4% (for NN2). The evolution of test accuracy over multiple epochs is shown in Figure 5. This shows that reasonable classification accuracy can be achieved on the MNIST dataset with much fewer number of parameters.

Next, to estimate the classification accuracy of the proposed FHE-NIT solution, we quantize all the values into fixed-point signed integers. As described earlier, 8-bit representations are used for all the input and loss values, while 16-bit representations are used for the weights and gradients. It can be observed from Figure 5 that the above quantized network (trained in the plaintext domain) can achieve a classification accuracy of 96%. Finally, we verify the gradient computations in the encrypted domain for a single mini-batch of data. Using the exact same weight initializations and sample set in both the plaintext and encrypted domains, we confirmed that the computations performed in both the domains are identical. Thus, it can be claimed that the classification accuracy of the model learned using homomorphically encrypted data will be the same as that of the quantized version of NN2, which is 96%.

4.3. Computational Complexity

Testing of encrypted domain processing was done on an Intel Xeon E5-2698 v3 (which is a Haswell processor), with two sockets and sixteen cores per socket, running at 2.30GHz. The machine has 250GB of main memory, the compiler was GCC version 7.2.1, and we used NTL version 10.5.0 and GnuMP version 6.0.

We primarily worked with the cyclotomic ring $\mathbb{Z}[X]/\Phi_m(X)$ with $m = 2^{10} - 1 = 1023$ (so $\phi(m) = 600$) for most of the development tasks. Since these parameters do not provide sufficient security, we also attempted to compare the time complexity when $m = 2^{15} - 1 = 32767$ (so $\phi(m) = 27000$), which corresponds to about 80 bits of security.

The computational complexity of executing the SGD algorithm (including both the forward path and back propagation) in the FHE domain based on one mini-batch of size 60 training samples is summarized in Table 1. There are four main factors that decide the computational complexity of the training algorithm in the encrypted domain.

- **Size of the neural network**: As described in Section 4.1, we consider two different neural networks, namely, NN1 and NN2. While the full-scale network NN1 has 954 nodes ($784 + 128 + 32 + 10$), the reduced-scale network NN2 has only 122 nodes ($64 + 32 + 16 + 10$). From the first two rows of Table 1, we observe that reducing the network size speeds up the computations by approximately a factor of 4.

- **Ciphertext packing**: We consider two different approaches for ciphertext packing. In the naive approach, we packed different dimensions of a input sample/weight vector into a single ciphertext. In the optimized approach, we packed the same dimension of multiple input samples into a single ciphertext and replicate each weight parameter into all the slots of a weight vector. Our experiments indicate that the optimized packing technique reduces the computational time by a factor of 1.5 on a single threaded machine (see rows 2 and 3 of Table 1). For example, training (one mini-batch) the reduced network NN2 using the optimized packing takes 9 hours and 24 minutes on a single-threaded machine, compared to approximately 14 hours required by the naive approach.

  With optimized packing, almost 80% of the computational time is consumed by the three matrix multiplication tasks, namely, computation of the weight average input to a layer (requires multiplication of the input to a layer with its corresponding weight matrix), loss propagation to the previous layer (requires multiplication of the loss at the previous layer with the weight matrix), and the gradient computation. Furthermore, one complete mini-batch requires 6 bootstrapping operations (one after each layer during both the forward pass and backpropagation).

- **Number of threads**: Multi-threading was very effective in reducing the computation time because it is well-known that the weight updates in SGD are independent. In other words, it is possible to process each neuron independently and in parallel. Even within a single neuron, the multiplication operations across multiple input dimensions can be parallelized. From Table 2, it can be observed that by parallelizing computations within a single neuron across multiple threads, it is possible to achieve almost linear speedup. Specifically, with 30 threads we observed about a $15\times$ speed-up (see rows 2 and 4 of Table 1), which means that the execution of one mini-batch would take only 40 minutes for $m = 1023$. Note that such a linear speed-up is not possible without the optimized ciphertext packing technique. The naive packing approach speeded-up the computation by only a factor of two even with 30 threads (see rows 3 and 5 of Table 1).

- **Security parameter**: For $m = 1023$, a single thread execution of one mini-batch of size 60 training samples, required approximately 9 hours and 24 minutes. It was also observed that when $m = 32767$, almost all the operations slowed down by approximately 40-60 times on a single threaded machine. However, it must be noted that $m = 32767$ can accommodate $1,800$ slots as compared to 60 slots for $m = 1023$. Thus, it is possible to compensate for the increased computational complexity by packing more input samples into a single ciphertext and reducing the number of batches to be processed.

Further improvements: We consider the results above as an encouraging start. We would like to emphasize that this is just a first attempt, and many other avenues of optimizations are still available. In particular, we only started exploring the options for batching/packing. We currently batch the inputs, but maintain a separate ciphertext for each weight in the network.

5. Conclusions and Future Work

Due to privacy and compliance requirements, users with large amounts of data can not make use of deep learning methods available as a service. We have made the first attempt to build a fully homomorphic deep learning service for training. The selection of representation, scaling, and key strength impacts both the accuracy and computational complexity of learning. After properly choosing our parameters, we demonstrate the results on MNIST dataset. It must be

Table 1. Computational time taken for executing one complete mini-batch (containing 60 training samples) of the SGD algorithm in the FHE domain with $m = 1023$.

S. No.	Network Size	Ciphertext Packing Method	No. of CPU Threads	Computational Time
1	NN1	Optimized	1	$\approx$ 1.5 days
2	NN2	Optimized	1	9 hours 24 minutes
3	NN2	Naive	1	$\approx$ 14 hours
4	NN2	Optimized	30	**40 minutes**
5	NN2	Naive	30	$\approx$ 6.5 hours

Table 2. Time taken (in seconds) to process one neuron at the first layer.

# threads:	1	4	8	16	30
$m = 1023$	213	60	33	20	14
$m = 32767$	-	-	-	-	919

highlighted that the current implementation of FHE training is still extremely slow for practical use. We estimate that training in the encrypted domain is still about four or five orders of magnitude slower than training in the plaintext domain.

It is worth mentioning that implementation of the proposed system is based on HElib, authored by the inventors of FHE and the entire system was carefully developed after in-depth consultation with the pioneers of FHE technology. Because of these reasons, the system is likely to serve as a credible baseline for the research community to calibrate their related contributions. Our baseline implementation, which we can share with the community, is expected to foster further research and innovation among the researchers interested in problems jointly spanning pattern recognition and cryptography. In summary, we believe we have created the alphabet and words for learning in encrypted domain for others to write their literary work.

References

[1] R. Agrawal and R. Srikant. Privacy-preserving data mining. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 439–450. ACM, 2000. 2

[2] M. Barni, C. Orlandi, and A. Piva. A privacy-preserving protocol for neural-network-based computation. In *Proceedings of the 8th Workshop on Multimedia and Security*, MM&Sec '06, pages 146–151, New York, NY, USA, 2006. ACM. 2

[3] V. N. Boddeti. Secure face matching using fully homomorphic encryption. arXiv:1805.00577, 2018. http://arxiv.org/abs/1805.00577. 2

[4] D. Boneh, C. Gentry, S. Halevi, F. Wang, and D. J. Wu. Private database queries using somewhat homomorphic encryption. In *ACNS*, volume 7954 of *Lecture Notes in Computer Science*, pages 102–118. Springer, 2013. 1

[5] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012. 1

[6] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory*, 6(3):13, 2014. 1

[7] D. Chialva and A. Dooms. Conditionals in homomorphic encryption and machine learning applications. Cryptology ePrint Archive, Report 2018/1032, 2018. https://eprint.iacr.org/2018/1032. 2

[8] A. Costache, N. P. Smart, S. Vivek, and A. Waller. Fixed-point arithmetic in SHE schemes. In *SAC*, volume 10532 of *Lecture Notes in Computer Science*, pages 401–422. Springer, 2016. 1

[9] J. L. Crawford, C. Gentry, S. Halevi, D. Platt, and V. Shoup. Doing real work with fhe: The case of logistic regression. In *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 1–12. ACM, 2018. 4

[10] J. L. H. Crawford, C. Gentry, S. Halevi, D. Platt, and V. Shoup. Doing real work with FHE: the case of logistic regression. *IACR Cryptology ePrint Archive*, 2018:202, 2018. 2, 4

[11] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st ACM Symposium on Theory of Computing – STOC 2009*, pages 169–178. ACM, 2009. 1

[12] C. Gentry, S. Halevi, C. S. Jutla, and M. Raykova. Private database access with he-over-oram architecture. In *ACNS*, volume 9092 of *Lecture Notes in Computer Science*, pages 172–191. Springer, 2015. 1

[13] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology - CRYPTO 2013, Part I*, pages 75–92. Springer, 2013. 1

[14] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 201–210. JMLR.org, 2016. 1, 2

[15] Z. G. T. Gu and S. Garg. Safetynets: Verifiable execution of deep neural networks on an untrusted cloud. In *Advances in Neural Information Processing Systems*, NIPS '17, 2017. 2

[16] S. Halevi and V. Shoup. HElib - An Implementation of homomorphic encryption. https://github.com/shaih/HElib/, Accessed September 2014. 3

[17] A. Khedr, P. G. Gulak, and V. Vaikuntanathan. SHIELD: scalable homomorphic implementation of encrypted data-classifiers. *IEEE Trans. Computers*, 65(9):2848–2858, 2016. 1

[18] A. Kim, Y. Song, M. Kim, K. Lee, and J. H. Cheon. Logistic regression model training based on the approximate homomorphic encryption. Cryptology ePrint Archive, Report 2018/254, 2018. https://eprint.iacr.org/2018/254. 2

[19] M. Kim, Y. Song, S. Wang, Y. Xia, and X. Jiang. Secure logistic regression based on homomorphic encryption: Design and evaluation. *JMIR Med Inform*, 6(2):e19, Apr 2018. available fromhttps://ia.cr/2018/074. 1, 2

[20] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 5

[21] Y. Lindell and B. Pinkas. Privacy preserving data mining. *J. Cryptology*, 15(3):177–206, 2002. 2

[22] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 619–631. ACM, 2017. 2

[23] P. Mohassel and Y. Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 19–38. IEEE Computer Society, 2017. 2

[24] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 334–348. IEEE, 2013. 2

[25] C. Orlandi, A. Piva, and M. Barni. Oblivious neural network computing via homomorphic encryption. *EURASIP J. Information Security*, 2007, 2007. 2

[26] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *13 ACM Asia Conference on Information, Computer and Communications Security (ASIACCS'18)*. ACM, Jun 4-8, 2018. To appear. Preliminary version: http://ia.cr/2017/1164. 2

[27] H. Shaul, D. Feldman, and D. Rus. Scalable secure computation of statistical functions with applications to k-nearest neighbors. arXiv:1801.07301, 2018. http://arxiv.org/abs/1801.07301. 2

[28] C. J. V. Vaikuntanathan and A. Chandrakasan. GAZELLE: a low latency framework for secure neural network inference. In *arXiv preprint*, 2018. 2

[29] S. Wang, Y. Zhang, W. Dai, K. Lauter, M. Kim, Y. Tang, H. Xiong, and X. Jiang. Healer: homomorphic computation of exact logistic regression for secure rare disease variants analysis in gwas. *Bioinformatics*, 32(2):211–218, 2016. 2

[30] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *NIPS*, 2010. 3