TOWARDS SECURE MACHINE LEARNING: CHALLENGES AND SOLUTIONS

Zhixue Chen
Institute for Interdisciplinary Information Sciences
Tsinghua University

Wentao Weng
Institute for Interdisciplinary Information Sciences
Tsinghua University

January 10, 2020

1 Introduction

Machine Learning has achieved astonishing performance in recent years and has already been deeply involved in real world applications, ranging from face recognition to anomaly detection. There is no denying that the boost of machine learning benefits a lot from the huge scale of real data. For example, in Natural Language Processing, the newest BERT model was proposed last year, which used hundreds of GB data for pre-training.

However, getting data and protecting data is another huge challenge. For example in face recognition, it seems necessary that the model can access the raw images, which are simply lots of people's faces. Another example is advertisement with machine learning. By training with users' private data, such as location, device, operation habits, companies could figure out a more profitable way to tailor advertisement for users. While these data could also be seen as serious privacy.

So ideally we wish that we could somehow encrypt the training data, so that our machine learning algorithms could still generate a good result but can not identify what's the original data.

Inspired by this ideal case, Fully Homomorphic Encryption seems to be a perfect solution. What Fully Homomorphic Encryption(FHE) states is basically that suppose we have some arbitrary original data, namely $m_1, m_2, \ldots, m_k$ and we have an arbitrary function f. Then we have an encryption and decryption method E, D such that

$$D(f(E(m1), E(m2), \ldots, E(m_k))) = f(m_1, m_2, \ldots, m_k).$$

Meanwhile we require that E, D satisfy a few properties in Cryptography so that the data encryption is secure to some extent.

Although the definition seems pretty straightforward, we currently still don't have a way of implementation. Some weaker methods are proposed like Partially Homomorphic Encryption, where only one type of operation, such as addition or multiplication, is supported on the encrypted data. Some previous work like [1] and [2], proposed a method including two operations, however on a finite field, which also puts serious restrictions on its real world applications.

In this paper we will summarize how two secure machine learning models —— Logistic Regression and Neural Network are implemented. We will introduce how to build up a complex network based on additions and multiplications that FHE could sopport, and what optimization techniques are needed to boost the efficiency.

2 Secure Logistic Regression

2.1 Theory

Logistic Regression is a very popular and powerful model. Suppose that we have n samples, each with d binary attributes and a binary label, of the form $(\vec{x}_i, y_i)$. Our goal is to estimate the probability

$$\Pr(y = 1|\vec{x}).$$

To fit into the easiest machine learning model, linear regression, this goal is equivalent to finding a set of parameters $\vec{w} = (w_1, w_2, \ldots, w_d, w_{d+1})$, such that

$$\Pr(y = 1|\vec{x}) = \langle(1|\vec{x}), w\rangle.$$

However we want to make sure that all the predicted probabilities are between 0 and 1, so we add a sigmoid function to the final output, which gives the form of logistic regression:

$$\Pr(y = 1|\vec{x}) = \frac{1}{1 + \exp(-\langle(1|\vec{x}), w\rangle)}.$$

With this form, we could use cross entropy as loss function and train the model with stochastic gradient descent, which is what we do today with plain data available.

However, according to the current limit of FHE, we can not calculate things like sigmoid, gradients directly on encrypted data. So we will introduce a simple approximation here and then explain how to run the whole algorithm with FHE.

2.2 Approximation

Apparently calculating gradients and literally "training" the model in this case seems to be very complicated. So we have some tradeoff. A Taylor expansion is used to make the calculation feasible and we can directly get the optimal solution while some accuracy is lost.

Assume that we put all our training data into 2^d categories according to their features, namely $c_1, \ldots, c_{2^d} \in \{0, 1\}^d$. For each category $\vec{c}$, we can count the positive and negative samples respectively, namely $Y_{\vec{c}}$ and $N_{\vec{c}}$. If we consider maximum-likelihood as "optimal", then we are just trying to maximize

$$\sum_{i=1}^{n} \Pr_{\vec{x}_i, y_i}(\vec{w}).$$

By taking log and expansion, we get

$$\sum_{i=1}^{n} \ln \Pr_{\vec{x}_i, y_i}(\vec{w}) = \sum_{\vec{c}} Y_{\vec{c}} \ln(p_{\vec{c}}(\vec{w})) + N_{\vec{c}} \ln(1 - p_{\vec{c}}(\vec{w})).$$

We denote

$$r_{\vec{c}}(\vec{w}) = \ln(\frac{p_{\vec{c}}(\vec{w})}{1 - p_{\vec{c}}(\vec{w})}).$$

Then we can express the optimal solution as

$$\vec{w}^* = argmax_{\vec{w}} \sum_{i=1}^{n} \ln \Pr_{\vec{x}_i, y_i}(\vec{w}) = \sum_{\vec{c}} Y_{\vec{c}} \ln(\frac{1}{1 + \exp(-r_{\vec{c}}(\vec{w}))}) + N_{\vec{c}} \ln(\frac{1}{1 + \exp(r_{\vec{c}}(\vec{w}))}).$$

Consider the function

$$f_{Y,N}(r) = Y \ln(\frac{1}{1 + e^{-r}}) + N \ln(\frac{1}{1 + e^{r}}).$$

By taking Taylor expansion around $r_0 = \ln(Y/N)$, we could get an approximation

$$f_{Y,N}(r) \approx C - \frac{YN}{2(Y + N)}(r - \ln(Y/N))^2.$$

Suppose that the approximation is valid, we can express $\vec{w}^*$ as a summation of simple polynomials and directly calculate its value.

2.3 Overview

We first give the overview of the algorithm in plain data.

STEP 1: Extract the k most correlated features from the data.

STEP 2: For every category, count the number of positive and negative samples, compute $\frac{Y_{\vec{c}} N_{\vec{c}}}{Y_{\vec{c}} N_{\vec{c}}}$ and $\ln(Y_{\vec{c}}/N_{\vec{c}})$.

STEP 3: Solve the optimal $\vec{w}^*$ with those precomputed values.

For a complete run of the algorithm with FHE on 1600 training samples with 105 features, it takes 5 hours. 125 minutes are spent on computing the 5 most correlated features, 33 minutes on extracting those 5 features, 47 minutes are spent on counters for each category, 60 minutes on computing values for the linear system and 210 minutes on bootstrapping operations. According to the author, the whole process could be shortened to one hour with multi-thread processing.

For accuracy, the best logistic model on **plain data** could give AUC result of 0.7 and their model gives AUC result of 0.65, which is fairly acceptable.

Their work has a pretty good performance, and more importantly serves as a building block for lots of future work. There are tons of low level operations they borrowed from **HElib** and writed by themselves, so we will pick and explain two of the most important tricks they used in implementation, which are also heavily used for future secure machine learning models.

2.4 Binary Arithmetic and Comparison

One main concern in building secure machine learning models is speed, since normally model training would involve tons of math calculations. So make things work in multi-thread settings is a necessary approach. In the algorithm they do all the work in binary representations. And we will give a short explanation of a fast binary addition in multi-thread setting.

Suppose we have ciphertext $(a_t, a_{t-1}, \ldots, a_0)$ and $(b_t, b_{t-1}, \ldots, b_0)$. Our goal is to compute $s = a + b$. The main obstacle in calculating sum is "carry bits". We define

$$g_i = a_i b_i, p_i = a_i + b_i.$$

We also define

$$p_{[i,j]} = \Pi_{k=i}^{j} p_k, g_{[i,j]} = g_i p_{i+1,j}$$

for all $0 \leq i \leq j < t$. Then we have

$$c_j = \sum_{i=0}^{j} g_{[i,j]}, s_j = a_j + b_j + c_{j-1}.$$

If we can calculate c_j fast, then we can get the integer addition fast.

We can do this by a dynamic programming approach, where our recursion formula is

$$p_{[i,j]} = p_{[i,k]} p_{[k+1,j]}, g_{[i,j]} = g_{[i,k]} p_{[k+1,j]}.$$

We can build a DAG, where each $p_{[i,j]}$, $q_{[i,j]}$, a_i, b_i is represented by a node. Since p, q can be calculated by two other nodes according to our recursion formulas, we can build the DAG so that each node has two parents with least possible computational complexity. We can compute all $p_{[i,i]}$ and $g_{[i,i]}$ with $O(1)$ time in multi-thread setting and by recursion, the longest p or q would only take $O(\ln t)$ time to compute.

Based on this and other tricks, we have the following fast binary operations in multi-thread setting:

- N length-t integers addition in $O(\ln(N)\ln(t))$ time.
- Integer multiplication with $O(t)$ time.
- Binary comparison with $O(t)$ time ($O(t^2)$ time for convient implementation)

By implementing those binary operations, we can accomplish lots of necessary steps in our logistic regression algorithm. Among all of them, finding the k most correlated features is the most time consuming and untraditional step. We here give a short explanation.

To calculate correlation, they adopted a more straightforward approach. We use $HW(\vec{v})$ to denote the Hamming weight of $\vec{v}$. Then we define $Y = HW(\vec{y})$, which is the total number of positive samples. We also define $\alpha_j = HW(X_{|j})$ (the number of samples whose j features is 1), $\beta_j = HW(X_{|j,\vec{y}})$ (number of positive samples whose j feature is 1). Then we calculate j feature's correlation as

$$|n\beta_j - Y\alpha_j|.$$

After getting the binary expression for each feature's correlation, we can use binary comparison to find the max k features. They used a shift-and-MUX procedure to accelerate each extraction to $O(\ln t)$ comparisons.

2.5 Table Lookups

We are now able to handle additions and multiplications with FHE, but computing complex functions still remains a problem. Two approaches are widely adopted —— Taylor Expansion and Table Lookups. Taylor Expansion basically just tries to turn every complex function to a polynomial, which is pretty straightforward. Here we introduce Table Lookups.

Intuitively, for a function f, we pre-compute a table T_f such that $T_f(x) = f(x)$ for every x in some range. Then everytime we want to compute $f(x)$, we simple turn to the table and look up the value. However we can not build an infinite table, so for Table Lookups, we need to specify three parameters (p, s, v), namely precision, scale, and sign of the number. Also for the case where the input exceeds the limit, we simply use the maximum or minimum limit as input and return the result.

For implementation, we also hope this could work in multi-thread setting. Suppose we have an input $(\sigma_{p-1}, \sigma_{p-2}, \ldots, \sigma_0)$, we can calculate the "subset products" ρ_m for every $m \in [2^p]$, where

$$\rho_m = \Pi_{m_j=1}\sigma_j \Pi_{m_j=0}(1 - \sigma_j).$$

A similar tree graph could be applied to calculate all ρ_m in $O(\ln p))$ time. Then we can return the result for any given input.

In Logistic Regression, we use Table Lookups to calculate three functions, namely $f(x) \approx 1/x$, $f(x) \approx 1/(x+1)$ and $\ln(x)/(x+1)$.

3 Secure Neural Network

Neural networks have achieved great success for these few years. In this work, we will take classification tasks as example for ease of expressions.

We consider a producer-consumer model. The producer will provide a good prediction model, and the user will use this provided model to classify her own data. Formally, it contains two steps to conduct classification using a neural network.

- The first step is to design a good neural network, which we denote as a parametric function $f_\omega(x)$. The goal is then to find the best parameter ω^*, that could optimize the loss function l such that

  $$\omega^* \in \arg\min_{\omega} \mathbb{E}_{x,y\sim\mathcal{S}}\left[l(y, f_\omega(x))\right]$$

  , where $\mathcal{S}$ is a given training set. The This step is called *training*.
- In the second step, a user will pass its own data x' into the network f_{ω^*} to get predicted output y'. This step is called *inference*.

We now introduce two types of security schemes for neural networks. Throughout the whole descriptions, we assume that the user is using a public key encryption scheme to encrypt her data.

definition 1 *Define the user's encryption algorithm and decryption algorithm as*

$$E : \mathcal{M} \to \mathcal{C}, D : \mathcal{C} \to \mathcal{M}.$$

definition 2 (Secure Inference) *In a secure inference scheme, the producer has a trained model f_{ω^*}. The user feeds the encrypted input $E(x)$ to the producer, and received the encrypted output $E(f_{\omega^*}(x))$. During the whole process, the producer knows nothing about x and $f_{\omega^*}(x)$. The user gains no information of f_{ω^*}.*

An illustration of secure inference is provided as figure 1.

definition 3 (Secure Training and Inference) *In a secure training and inference scheme, the user would first send an encrypted training data set $E(\mathcal{S}) = \{(E(x), E(y)) \mid (x, y) \in \mathcal{S}\}$ to the producer. Based on the encrypted data, the producer will train a model f_{ω^*}. Finally, the user would query $E(x)$ to the producer, the corresponding prediction for this query is $D(f_{\omega^*}(E(x)))$. Throughout the whole process, the producer would have no information of the training set, all queries and even the real model as what she gets will be an "encrypted" model.*

Intuitively, a secure training and inference scheme is much stronger than a secure inference scheme. However, it is also much harder to train. Although other techniques such as secure multi-party computation are applied to build a secure neural networks as well, We restrict our scope to secure neural networks which utilize FHE as a key component. In particular, we describe methods to use FHE to ensure security for every operation. As long as operations will never calculate wrong (we will explicitly explain why operations in FHE may sometimes get wrong), the whole process will be correct and secure.

A line of researches has been devoted into these two directions. We first summarize challenges when applying FHE in neural networks. Then we explain main ideas of current approaches. Finally we provide more details of methods to achieve secure neural networks.

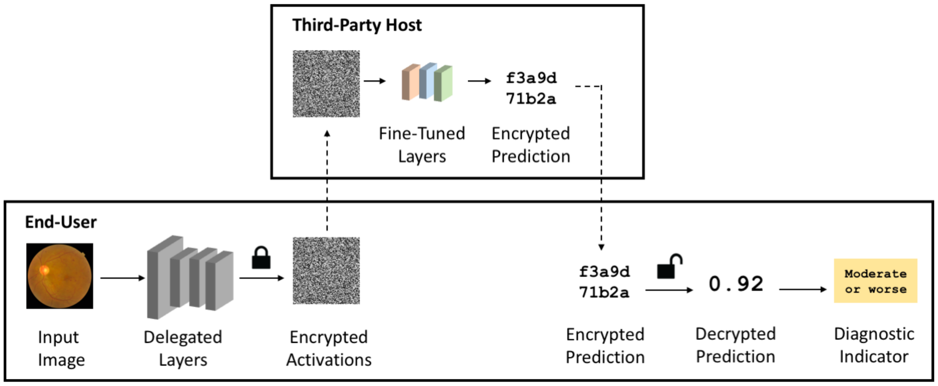

Figure 1: Illustration of secure inference

3.1 Challenges to apply FHE

To apply FHE into neural networks, we need to conquer two key obstacles, correctness and efficiency.

- Correctness. To the best of our knowledge, current implementations of FHE, such as [1],[2], [3], only support unlimited times of additions and multiplications on a finite field. From computation theory, we could construct all circuits with only XOR and AND, correspondingly additions and multiplications in $\mathbb{Z}_2$. Nevertheless, the only "correct" functions we could construct from FHE are polynomials.

  On the other hands, if we use FHE, we need to ensure the outcome of an operation is expected. To be specific, consider we are using FHE defined on $\mathcal{M} = \mathbb{Z}_{17}, \mathcal{C} = \mathbb{Z}_{17}$. Let $\oplus$ be the corresponding plus operation for encrypted data. We would have
  $$D(E(2) \oplus E(5)) = 7,$$
  but
  $$D(E(10) \oplus E(20)) \neq 30,$$
  as the decrypted result must belong to $\mathcal{M} = \mathbb{Z}_{17}$. Therefore, throughout the whole computation, we need to guarantee no result will overflow.
- Efficiency. Another challenge to apply FHE is its inefficiency. In the design of FHE, the length of a cipher text is usually much longer than that of a plain text. Inherently, we bring extra time and memory complexity to computation. Furthermore, if we follow Gentry's solution of FHE [1], [2], we need to bootstrap data from time to time to cancel piled up errors during computation. This step is extremely time-consuming.

3.2 High Level Ideas of Designing Secure Neural Network

To resolve problems of FHE, current work on secure neural network follow a very similar structure originated from [4]. The first step is to deal with correctness issue. In neural networks, several functions are common to compute in each neuron, which are listed as follows.

- Weighted Sum: we are given two vectors A, B, the goal is to calculate their dot product. We could view convolution layers and fully connected layers as series of weighted sums. It is straight forward to implement weighted sum using FHE because it only entails additions and multiplications.
- Max pooling: we take the maximum of an area of the feature map to build a new one in the neural network. It is nontrivial to implement max pooling as it is non-linear. The solution is to use polynomials to approximate max pooling.
- Mean pooling: we take the mean of an area of the feature map to build a new one. It is easy to use FHE as mean pooling is just a weighted sum.
- Activation functions: it is usual to add nonlinear transform after a convolution layer, such as the sigmoid function $\frac{1}{1+e^{-x}}$ or the relu function $\max(x, 0)$. One possible way is to approximate activation functions by polynomials. Another solution is to use polynomials as nonlinear activation functions. In general, people use x^2 as an alternative choice.

Another issue is to deal with floating-point weights in neural networks. As our encryption can only supports integer, we have to find a mapping to encode floating point with integers just like how system deals with floating points. Nevertheless, we have to carefully pick parameters for the encryption scheme to avoid overflow as we have mentioned before.

The above techniques are almost sufficient to deal with secure inference scheme. However, if we want to train with data securely, we need to enable back propagation during the training phase of neural networks. Most importantly, we need to implement the encrypted version of gradients.

Before we talk about detailed implementations, we need to recall some techniques introduced in FHE literature. In particular, we will describe the encryption scheme in [3].

3.3 Preliminaries

We adopt the encryption scheme introduced in [3]. Formally, the plain text space $\mathcal{M}$ is the ring $R_t^n = \mathbb{Z}_t[x]/(x^n+1)$. We will map this ring to another ring $R_q^n = \mathbb{Z}_q[x]/(x^n+1)$. In general, we will assume t, q are primes, and $q > t$. We then choose random polynomials $f', g \in R_q^n$, and let $f = tf' + 1$. Define the public key as a polynomial $h = tgf^{-1}$, and f is our secret key. For a message m, its encryption is

$$c = \left[\lfloor q/t \rfloor m + e + hs\right]_q,$$

where e, s are random polynomials in R_q^n serving as noise. Here $[x]_q$ means to modulo every coefficient in x by q. On the other hand, to decrypt a cipher text c, we use

$$m = \left[\lfloor \frac{t}{q} fc \rceil\right]_t,$$

here the notation $\lfloor \frac{t}{q} \rceil$ denotes rounding to the nearest integer.

To depict additions, let c_1, c_2 be two cipher texts, and m_1, m_2 be their corresponding plain texts. We have

$$c_1 + c_2 = \lfloor \frac{q}{t} \rfloor (m_1 + m_2) + e_1 + e_2 + h(s_1 + s_2),$$

which is the corresponding cipher texts of $m_1 + m_2$.

Furthermore, for multiplication of $m_1 m_2$, we could calculate

$$\lfloor \frac{t}{q} c_1 c_2 \rceil = \lfloor \frac{q}{t} \rfloor (m_1 m_2) + e' + h^2 s_1 s_2.$$

We could decrypt this cipher text to $m_1 m_2$ using initial secret key f by techniques in [3].

We could observe that noises increase significantly after each multiplication. To ensure we could decrypt the cipher text successfully, we need use bootstrapping (or refresh in different literature) from [3] to reduce noise. Another method is to use levelled homomorphic encryption. We first specify the circuit complexity, or in specific the number of multiplications during the whole computation. We can then decide the system parameters q, t to ensure no overflow occurring during the whole process.

3.4 Secure Inference

We now describe the detailed implementation to do securely inference. The framework is mostly based on [4]. We adopt the aforesaid levelled homomorphic encryption, and carefully pick the system parameters after the design of the whole neural network.

As HE can support additions and multiplications, we could easily construct polynomial functions. It is then straight forward to adopt convolution layers and fully connected layers in the encryption scheme. We could then view the whole design as following three parts: quantization of floating-point inputs, approximations of activation functions, and optimizations.

3.4.1 Quantization

Note that the inputs to the neural network are all float points. As a result, we need to design encoder and decoder to converse a float point a to a plain text m, and vice versa. Plain texts are basically polynomials in our setting. In [4],

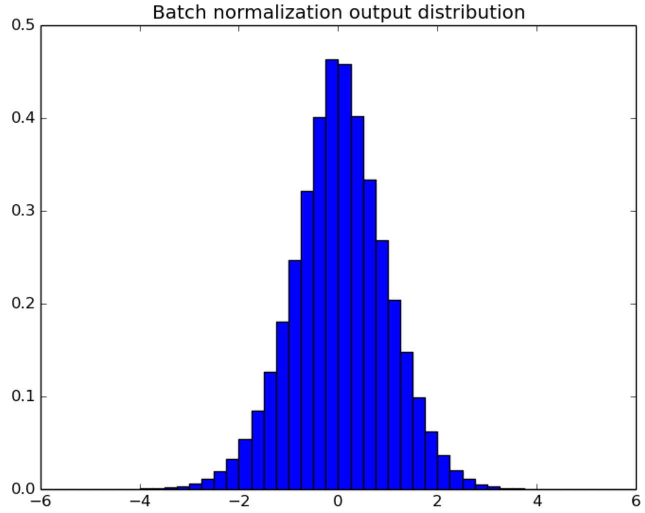

Figure 2: Batch Normalization output distribution

the authors propose two possible designs. The first one is to use a's binary expression. Suppose $a = \sum_{i=-l}^{l} a_i 2^i$, and l is a cutoff, or namely the precision. As $a_i \in \{0, 1\}$, it could fit our integer polynomial setting. The corresponding polynomial of a is $\sum_{i=0}^{2l} a_{i-l} x^i$. Later we will leverage the sparsity in binary expressions to boost the efficiency in inference. The second design is to directly view a as a constant polynomial. However, we need to truncate a to an integer, which will also bring precision problem. Nevertheless, this solution could help us compute in parallel using the technique called cipher text packing [5], which we will later discuss about.

3.4.2 Approximations of activation functions

Currently, we have two possible directions to enable activation functions in the neural networks.

One solution is to maintain a lookup table we have previously described in the case of logistic regression. We could discretize inputs to several pieces. Each piece maintains a specific encrypted value of the activation function. But we require the client to send the encrypted look up table to the server, as the server cannot encrypt the data.

Another solution is to use polynomials as an approximation. A naive way is to directly use polynomials as activation functions. Suppose we have an activation function $f(x)$, the goal is to find a polynomial $p(x)$ that could approximate $f(x)$ well.

In [4], the authors directly use a polynomial activation function $f(x) = x^2$. As a result, we could directly utilize HE to simulate this activation function. However, they also note that the derivative of x^2 is unbounded, which leads to unstable training as gradients would get unbounded if the depth of the network is large, such as five to six layers.

An intuitive way to construct polynomials from a function is to use Taylor expansion. In specific,

$$f(x_1) \approx \sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{n!}(x_1 - x_0)^i.$$

However, it requires the function f is n-time differentiable for all x. A typical counterexample is Relu $f(x) = \max(x, 0)$ whose derivative is not continuous. Furthermore, the error is hard to quantify as we are looking at the whole range of x.

Later, [6] uses an alternative way to directly approximate a Relu activation function $f(x)$. They propose to add batch normalization between a convolution layer and an activation layer. The batch normalization could reshape the input distribution of the activation function to a standard normal distribution approximately, as it is shown in figure 2. They then sample points according to the standard normal distribution, namely a set $X = \{x_i\}$. Based on their values $Y = \{f(x_i)\}$, we could use polynomials to fit these values. In contrast to previous method where we have no specific input distribution, this method could reduce the approximation error significantly. But we should note that, in [6], we use the initial activation function like Relu during the training phase. We only switch to the approximated polynomial during the inference phase.

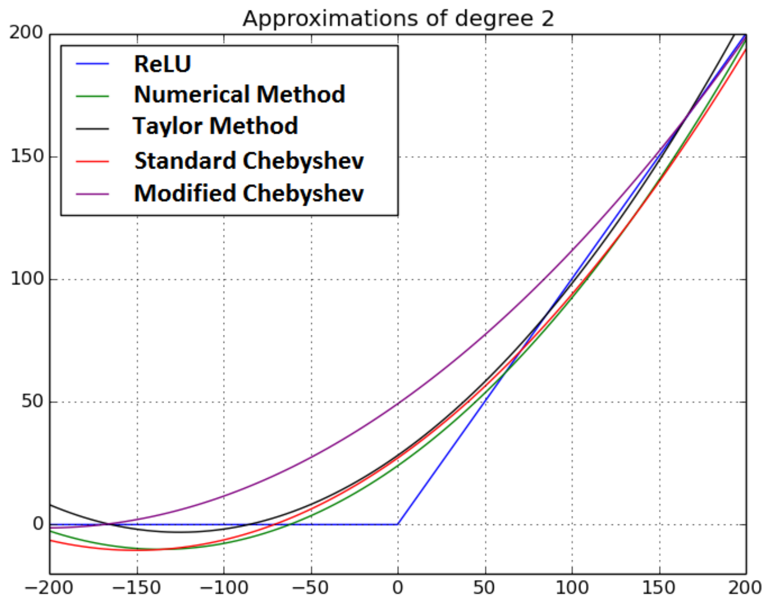

Figure 3: Approximations of Relu

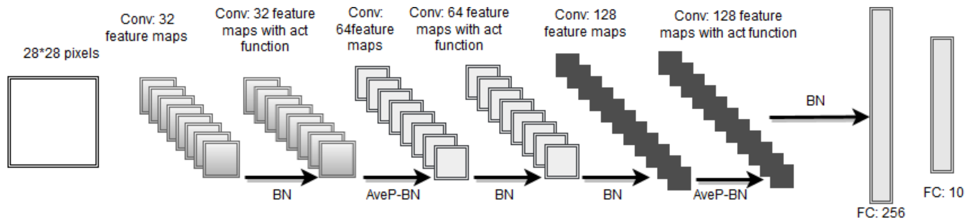
28*28 pixels
Conv: 32 feature maps
Conv: 32 feature maps with act function
Conv: 64feature maps
Conv: 64 feature maps with act function
Conv: 128 feature maps
Conv: 128 feature maps with act function
BN
AveP-BN
BN
BN
AveP-BN
BN
FC: 256
FC: 10

Figure 4: Network structure in [7]

In the work [7], the authors compare multiple methods to approximate activation functions like Sigmoid, Relu, and Tanh. Motivated by the theory of numerical analysis, if the difference between derivatives of f, g is bounded, the difference between f, g is also bounded. The authors then propose a method which uses polynomials to approximate the gradient of the activation function, instead of the initial function. An integral will then turn the approximated gradient into an approximated activation function. Figure 3 shows how different methods could approximate Relu.

In contrast to [6], [7] also train the neural network with the approximated activation function. They also incorporate batch normalization in their network. Finally, they could achieve a very similar performance as the initial activation functions with their approximated polynomials. Their network can be viewed as figure 4.

3.4.3 Optimizations

However, above techniques work extremely slow in practice. To improve the efficiency or the throughput during the inference phase, we will introduce large integer encoder in [4], cipher text packing techniques in [5], and advance pruning and quantization techniques from [8].

As we note before, the plain text space is R_t^n, where each coefficient of polynomials is from $\mathbb{Z}_t$. We need to choose a t of the same order as results throughout the whole network to ensure no overflow happens during the computation. However, it means that t would be almost exponential of the number of multiplication in the computation. The efficiency will then degrade significantly. The solution is to use Chinese Remainder Theorem (CRT). If we write $t = \prod t_i^{p_i}$, where t_i are co-prime with each other, we could calculate the network with plain text $R^n_{t_i^{p_i}}$. Finally, we could recover the initial plain text in R_t^n by CRT. This way is much more efficient way since our plain texts get shorter.

The idea of cipher text packing is again to utilize CRT, but on a different perspective. Assume that we carefully choose t such that $x^n + 1 = \prod(x - \alpha_i) \pmod{t}$. We could use CRT to show the ring isomorphism

$$R_t^n \cong \mathbb{Z}_t^{\times n}.$$

As a result, we could encode n values in $\mathbb{Z}_t$ into a single polynomial in R_t^n. An operation in R_t^n is equivalent to n independent operations on these values. We then use polynomials to achieve parallel computation.

Recently, [8] employs the idea of network pruning into the design of secure neural network. They first eliminate useless weights in the network with the network surgery approach proposed in [9]. Another main contribution of this work is to leverage sparsity properties of polynomials. Recall that when doing multiplications, we are indeed doing polynomial multiplications whose complexity are about $O(n^2)$. Now suppose we use binary expressions to encode a value. For example, the value $\frac{1}{4}$ with be encoded as x^{126} if we have a base 2^{128}. Assume we want to multiple $\frac{1}{4}$ by another value which is encoded as a polynomial p. We only need to multiply p by x^{126}. It largely only shifts the polynomial, which is a much cheaper operation. Based on this observation, the idea of [9] is to find a sparse representation of our activation functions. Formally, suppose we have a activation function $f(x)$, we would like to find a polynomial $g(x) = \sum_{i=0}^{l} a_i x^i$, such that

$$a_i \in \{2^p : p \in \{-k, -k+1, \cdots, k\}\},$$

and the error is minimized. Here k is some predefined integer. Although this approximation may reduce the accuracy by a little fraction, it improves the efficiency of the network a lot.

3.5 Secure Learning and Inference

Finally, we address the problem to achieve secure learning and inference. The structure is very similar to secure inference. During the training phase, basically, we need to implement a forwarding of the network, and a back propagation of the network. We could note that the forwarding of the network is almost identical to the inference phase, In [10], a FHE is used to implement the network training phase because unknown weights make levelled homomorphic encryption hard to implement. The authors propose to use a square loss function

$$l(x, y) = (y - f_{w^*}(x))^2.$$

However, this square loss function is not suitable when it comes to classification tasks, where typically we use cross entropy as the loss function.

The other difference between training and inference is that, we need to update parameters by gradient descent in back propagation. Note that we are facing with encrypted data, and encrypted weights. We therefore need to obtain encrypted versions of gradients. In [10], the authors implement a look up table for required gradients. The process of back propagation still consists of multiple matrix multiplications, well-suited for FHE.

4 Conclusion

In this work, we address the issue of building a machine learning system which could support secure of either inference or both training and inference. However, we note that the currently implementation is still inefficient, and problems such as overflow prevent a deep neural network to be supported. According to [10], it takes about one and a half day to train a three-layer neural networks with about 10M parameters on the MNIST data set. Furthermore, currently all operations are on CPU. To boost the efficiency of network training, techniques to compute FHE operations on GPU are still waited. Smart approximations and alternatives of well-used activation functions are in great need to design a more robust and efficient neural network.

References

[1] Craig Gentry et al. Fully homomorphic encryption using ideal lattices. In *Stoc*, volume 9, pages 169–178, 2009.

[2] Craig Gentry, Shai Halevi, and Nigel P Smart. Fully homomorphic encryption with polylog overhead. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 465–482. Springer, 2012.

[3] Joppe W Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In *IMA International Conference on Cryptography and Coding*, pages 45–64. Springer, 2013.

[4] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

[5] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

[6] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. Privacy-preserving classification on deep neural network. *IACR Cryptology ePrint Archive*, 2017:35, 2017.

[7] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. Cryptodl: Deep neural networks over encrypted data. *arXiv preprint arXiv:1711.05189*, 2017.

[8] Edward Chou, Josh Beal, Daniel Levy, Serena Yeung, Albert Haque, and Li Fei-Fei. Faster cryptonets: Leveraging sparsity for real-world encrypted inference. *arXiv preprint arXiv:1811.09953*, 2018.

[9] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *Advances In Neural Information Processing Systems*, pages 1379–1387, 2016.

[10] Karthik Nandakumar, Nalini Ratha, Sharath Pankanti, and Shai Halevi. Towards deep neural network training on encrypted data. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 0–0, 2019.