A Study on Extending Homomorphic Encryptions to Machine Learning

B504: Introduction to Cryptography, Term Project Report

Paventhan Vivekanandan
University id - 0003397339

Indiana University Bloomington
pvivekan@iu.edu

Abstract. This project reports a detailed study of what happens when cryptographic frameworks such as homomorphic encryption schemes and learning with errors (LWE) problem are applied to Machine learning toolkits such as neural networks. It also builds on the motivation needed and the advantages we can get from such applications. More specifically, it examines ideas extracted from two papers: the first one is CryptDB, which employs partial homomorphic scheme like paillier cryptosystem, and the second one is FHE-DiNN which uses a fully homomorphic encryption scheme based on LWE to evaluate a fully connected neural network.

Keywords: Learning with errors, Ring learning with errors, homomorphic encryption, bootstrapping, neural networks.

1 Introduction

Machine learning, a branch of aritificial intelligence, focuses on learning computational models from huge volume of data and applying those models in decision making. While Cryptography is concerned with protecting the sensitivity of data, homomorphic encryptions provides an elegant way to perform computations on encrypted data without (or little) compromise on their security. Recent advancements in the field of Machine Learning such as the resurgence of neural networks has given us an efficient way to perform classification on colossal amounts of data. We can extend neural networks to take advantage of homomorphic encryption to perform classification on sensitive data which are in encrypted form.

To understand the importance of such synthesis between cryptography and machine learning (pattern recognition) research, lets consider an example based on Genomics [2][9]. Sensitive data such as the DNA and RNA sequences of patients are cheaper and faster to generate and have been accumulated in labs and medical institutes all over the world. Such genome sequences are a powerful tool in the study of biology, medicine, and human history. Detecting useful patterns from such enormous amount of data can significantly drive forward studies on different complex diseases. At the same time, these are sensitive data which expose crucial informations such as the identity of family, national origin, or the presence of disease in a person. On the otherhand, owners of computational models such as neural networks are unwilling to make their implementations public and instead they deploy the model to common platforms such as the cloud [3]. This motivates the direction of research which involves applying machine learning algorithms on sensitive data using homomorphic encryptions.

The focus of this project is a detailed discussion on extending machine learning algorithms namely the neural networks to perform classification on encrypted data using homomorphic encryptions. Mainly, this project investigates two papers related to homomorphic encryptions. The first one is **CryptDB** [11], which provide simple and adequate insights on applying homomorphic encryptions to real world scenarios. The second paper is **FHE-DiNN** [3], which provides a framework for homomorphic evaluation of neural networks.

Next section provides the necessary background information required to understand the work discussed in this paper. Section 3 includes discussion on CryptDB and FHE-DiNN. Section 4 includes information on the implementation details and experimental observations [1].

2 Background

In this section, we will investigate briefly the necessary background information needed to understand the work being studied in this paper.

2.1 Homomorphic encryption and Bootstrapping: A homomorphic encryption scheme allows for computation on encrypted data without the need to decrypt them. In 1978, shortly after the creation of RSA, Rivest, Adelman, and Dertouzos introduced the idea of creating a fully homomorphic encryption [13]. RSA scheme itself has a partial homomorphic property given as follows.

Definition 2.1.1: Consider a RSA scheme with public key (n, e), private key (d, n) and cipher-texts $c_1 \equiv m_1^e \bmod n, c_2 \equiv m_2^e \bmod n$. Also, let c_3 be the cipher-text obtained by multiplying c_1 and c_2 (ie $c_3 = (c_1 * c_2) \bmod n$). For the given scheme, the following homomorphic property holds true.

$$c_3 = c_1 * c_2 = (m_1 * m_2)^e \bmod n$$

If we multiply two cipher-texts c_1, c_2, the resulting cipher-text $c_3 = (c_1 * c_2) \bmod n$ is an encryption of the multiplication of plain-texts corresponding to c_1, c_2 (ie $c_3^d = (m_1 * m_2) \bmod n$). As another example of partial homomorphic cryptosystems, consider the Paillier cryptosystem [10] which has the following homomorphic property.

[1] The implementation I did as part of this project is available at `https://github.iu.edu/pvivekan/B504_Introduction-to-Cryptography`

Definition 2.1.2: Consider a Paillier cryptosystem with public key (n, g), private key (λ, μ) and cipher-texts $c_1 \equiv g^{m_1} \,.\, r^n \bmod n^2, c_2 \equiv g^{m_2} \,.\, r^n \bmod n^2$ (for random r where $0 < r < n$ and $r \in \mathbb{Z}_n^*$). Also, let c_3 be the cipher-text obtained by multiplying c_1 and c_2 (ie $c_3 = (c_1 * c_2) \bmod n^2$). For the given scheme, the following homomorphic property holds true.

$$c_3 = c_1 * c_2 = g^{(m_1+m_2) \bmod n} \,.\, r^n \bmod n^2$$

It allows us to compute the sum of two plaintexts through multiplication of their corresponding ciphertext (Pailler-Enc$_k(m_1)$. Pailler-Enc$_k(m_2)$ = Pailler-Enc$_k(m_1+m_2)$). This can be invaluable especially for applications requiring outsourced computation and storage such as the commercial cloud environments.

In 2009, Craig Gentry proposed the first viable fully homomorphic encryption scheme [7]. His scheme can perform both the addition and multiplication on the cipher-texts. Using this construction, one could build a circuit to perform any arbitrary computations. Gentry also noted that in his construction the noise on the cipher-text grows with each homomorphic evaluation. He observed that when the noise on the cipher-text grows beyond a certain threshold, the decryption algorithm fails. In order to address the noise problem, Gentry proposed a method called *bootstrapping* according to which when the noise on the cipher-text grows beyond the limit, it can be refreshed to allow for further homomorphic computations. The cipher-text is refreshed by *homomorphically decrypting* it which means the decryption algorithm is evaluated homomorphically. During this process, it will reencrypt the cipher-text using a different key and in doing so, it will reduce the noise on the cipher-text. Schemes that supports homomorphic evaluation of lower degree polynomials while keeping the noise at manageble levels are called *somewhat homomorphic encryption* (SWHE) schemes. The bootstrapping [8][6] is defined as follows.

Definition 2.1.3: Consider a somewhat homomorphic encryption scheme $\mathcal{E}$. Assume we have a cipher-text c which is obtained by encryption of message m under public key pk_1. Also, assume we have a public key pk_2 with which we encrypt the decryption (secret) key sk_1 corresponding to pk_1. Let $\langle sk_1 \rangle_{pk_2}$ denotes the encrypted version of sk_1 under pk_2 and $\langle c \rangle_{pk_2}$ denotes the encryption of c under pk_2. For decryption function $D_{\mathcal{E}}$, we have the following construction.

$$c' \leftarrow \textbf{Eval}\ (pk_2, D_{\mathcal{E}}, \langle sk_1 \rangle_{pk_2}, \langle c \rangle_{pk_2})$$

The above function produces a fresh encryption of m under pk_2 (ie $c' = \langle D_{\mathcal{E}}(sk_1, c) \rangle_{pk_2} = \langle m \rangle_{pk_2}$). Note that to decrypt c' we need the secret key sk_2 corresponding to public key pk_2. During the above process, the noise in the cipher-text will be reduced allowing for more homomorphic evaluations. Adding the above process to a SWHE scheme will result in *fully homomorphic encryption* (FHE) scheme. The disadvantage is that once the noise level goes beyond the acceptable limit, the above process has to be repeated at every subsequent computations performed. This will massively slow down the computation process leading to delayed (possibly unacceptable) execution time.

2.2 Learning with errors: Learing with errors (LWE), a lattice-based cryptosystem introduced by Regev in [12], is one of the candidate for leading the gaurd against attacks using quantum computers. The LWE problem is to distinguish random linear equations, of which some of them are perturbed with small amount of noise, from truely random ones. The following is the construction of LWE.

Definition 2.2.1: Let $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ where $\mathbb{R}$ is the real numbers, $\mathbb{Z}$ is the integers, and $\mathbb{T}$ is the torus representing additive group on real numbers modulo one. Let $\mathbf{s} \in \{0,1\}^n$ be a fixed vector for some positive integer n. Let $\mathcal{X}$ be the probability distribution over $\mathbb{R}$ for noise. Sample a vector $\mathbf{a}$ uniformly from $\mathbb{T}^n$ ($\mathbf{a} \overset{\$}{\leftarrow} \mathbb{T}^n$). Also, sample error vector $\mathbf{e}$ using distribution $\mathcal{X}$. We define the LWE distribution $\mathbf{lwe}_{n,\mathrm{s},\mathcal{X}}$ as ($\mathbf{a}$, $\mathbf{b}$) where $\mathbf{b} = \langle \mathbf{s}, \mathbf{a} \rangle + \mathbf{e} \in \mathbb{T}$. Here, we compute $\langle \mathbf{s}, \mathbf{a} \rangle$ as $\sum_{i=1}^{n} a_i s_i$. The learning with errors problem (*search* version) $\mathbf{LWE}_{n,\mathcal{X}}$ is to find $\mathbf{s}$, given access to poylnomially many samples from $\mathbf{lwe}_{n,\mathrm{s},\mathcal{X}}$. Also, the LWE assumption (for *decision* version) states that, for $s \overset{\$}{\leftarrow} \{0,1\}^n$, it is hard to distinguish between ($\mathbf{a}$, $\mathbf{b}$) and ($\mathbf{u}$, $\mathbf{v}$) where $(\mathbf{u}, \mathbf{v}) \overset{\$}{\leftarrow} \mathbb{T}^{n+1}$ and ($\mathbf{a}$, $\mathbf{b}$) sampled from $\mathbf{lwe}_{n,\mathrm{s},\mathcal{X}}$.

Consider we have a Gaussian function, with paramter $\sigma > 0$, as $\rho_\sigma(x) = e^{-\pi|x|^2/\sigma^2}$ for any $x \in \mathbb{R}$. Then we can say that a distribution $\mathcal{D}$ is *sub-Gaussian* with parameter σ if there exists an $M > 0$ such that $\forall x \in \mathbb{R}$, we have $\mathcal{D}(x) \leq M \,.\, \rho_\sigma(x)$. Let $\mathcal{X}_\sigma$ be the distribution on $\mathbb{T}$ obtained by considering $\mathbb{D}_\sigma$ modulo one. This is the distribution commonly used in most of the LWE problems represented as $\mathbf{LWE}_{n,\mathcal{X}_\sigma}$. The following example gives the necessary insights into LWE problem[2].

Example 2.2.2: For some prime number q, let $\mathbb{Z}_q$ denotes the ring of integers modulo q and $\mathbb{Z}_q^n$ denotes the set of n-vectors over $\mathbb{Z}_q$. Consider $a \in \mathbb{Z}_q^n$. For this example, we will use a more general definition of LWE where a belongs to $\mathbb{Z}_q$ instead of the torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$. The following equation shows the computation of LWE.

$$b = \begin{bmatrix} 5 & 7 & 8 & 4 \\ 12 & 6 & 1 & 5 \\ 5 & 4 & 10 & 2 \\ 3 & 10 & 0 & 11 \\ 0 & 1 & 7 & 2 \end{bmatrix} \times 5 + \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & -1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & -1 & 0 & 1 \end{bmatrix} \pmod{13} = \begin{bmatrix} 0 & 10 & 2 & 8 \\ 9 & 5 & 5 & 12 \\ 0 & 7 & 10 & 11 \\ 2 & 12 & 0 & 4 \\ 0 & 4 & 9 & 11 \end{bmatrix}$$

[2] For more examples, please check `https://asecuritysite.com/encryption/lwe2`

In the above example, the first matrix represents $a \in \mathbb{Z}_{13}^{5\times 4}$ which is multiplied by the secret key $s = 5$. The result of $a \times s$ is added with the error e. Then $\bmod 13$ is applied to give the result b. The matrix (a, b) is the public key and the scalar s is the secret key (s can also be a matrix). For message $m \in \{0, 1\}$, we compute $u = \sum a_{samples} \bmod q$ and $v = \sum b_{samples} + \frac{q}{2}m \bmod q$ and return (u, v) as the encrypted value. Here, the sampling of a, b is done randomly. For decryption, we compute $d = v - su \bmod q$ and output 0 if $d < q/2$ or 1 if $d \geq q/2$. This example follows the original idea of Oded Regev [12]. FHE-DiNN uses the same idea with minor modifications to make it easy for homomorphic evalutations of neural networks. The following LWE-based private encryption scheme is adopted by FHE-DiNN.

Definition 2.2.3: Let $\mu \in \{0, 1\}$ be a message and λ is the security parameter. The encryption and decryption algorithms are defined as follows.

1. **Setup**(λ): fix $n = n(\lambda)$ and return $s \overset{\$}{\leftarrow} \{0, 1\}^n$.
2. **Enc**$(s,\ \mu)$: for $a \overset{\$}{\leftarrow} \mathbb{T}^n$ and $b = \langle s, a\rangle + e + \frac{\mu}{2}$, where $e \overset{\mathcal{X}}{\longleftarrow} \mathbb{R}$, return (a, b).
3. **Dec**$(s,\ (a, b))$: return $\lfloor 2\,(b - \langle s, a\rangle)\rceil$.

where $\lfloor x\rceil$ rounds to the nearest integer (for example, if $x = 0.48$ then $\lfloor x\rceil = 0$ and if $x = 0.54$ then $\lfloor x\rceil = 1$). In the above scheme, e is the noise of the cipher-text. We say that a cipher-text is a valid encryption of μ if it decrypts to μ with overwhelming probability.

There is a variant of LWE problem called the *Ring learning with errors* (R-LWE) problem which is specialized to polynomial rings over finite fields. It is similar to LWE, except that now we are distinguishing random noisy ring equations from truely uniform pairs [5]. R-LWE provides the basis for homomorphic encryptions. FHE-DiNN uses TLWE which is a generalization of LWE and R-LWE. It is defined as follows.

Definition 2.2.4: Let $k \geq 1$ be an integer, N be a power of 2, and $\mathcal{X}$ be an error distribution over $\mathbb{R}_N[X]$ where $\mathbb{R}_N[X] = \mathbb{R}[X]/(X^N + 1)$ and $\mathbb{R}[X]$ denotes polynomials in X with coefficients in $\mathbb{R}$. A TLWE secret key $\hat{s} \in \mathbb{B}_N[X]^k$ is a vector of k polynomials over $\mathbb{Z}_N[X]$ with binary coefficients where $\mathbb{Z}_N[X] = \mathbb{Z}[X]/(X^N + 1)$. Consider a message $\mu \in \mathbb{T}_N[X]$ encoded as a polynomial where $\mathbb{T}_N[X] = \mathbb{T}[X]/(X^N + 1)$. For $a \overset{\$}{\leftarrow} \mathbb{T}_N[X]^k$ and $b = \hat{s} \,.\, a + \mu + e$, where $e \overset{\mathcal{X}}{\longleftarrow} \mathbb{R}_N[X]$, a TLWE encryption of message μ under key $\hat{s}$ will return $(a, b) \in \mathbb{T}_N[X]^k \times \mathbb{T}_N[X]$. From the TLWE ciphertext $\hat{c}$ obtained from the encryption of message μ, we can extract a LWE encryption c', which denotes the constant term of polynomial μ, using key s' extracted from key $\hat{s}$. This technique is used in the FHE-DiNN during bootstrapping as we will see.

2.3 Artificial neural networks: A neural networks consists of an input layer, one or more hidden layers, and an output layer (see fig. 1). Each layer consists of artificial neurons which are modelled based on the neurons in a biological brain.

Each neuron transmits a signal to neurons in the next layer like the synapses in a biological brain.

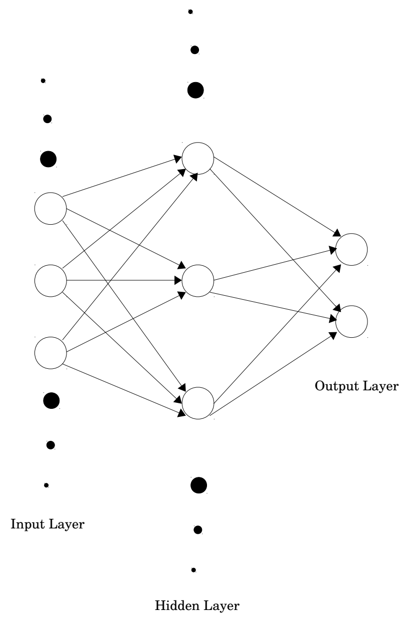

Fig. 1: Fully connected feed-forward neural network

Mathematically, a neuron is function which takes some inputs $(x_1, x_2, ..., x_n)$ and weights $(w_1, w_2, ..., w_n)$ correspnding to those inputs. It then calculates a summation based on the product of the inputs and corresponding weights $(\sum_{i=1}^{n} x_i w_i)$ and passes them through an activation function (see fig. 2). A neural network is trained on a huge amount of training dataset during which it learns certain features based on which it performs classification. The output layer gives out the class of the input.

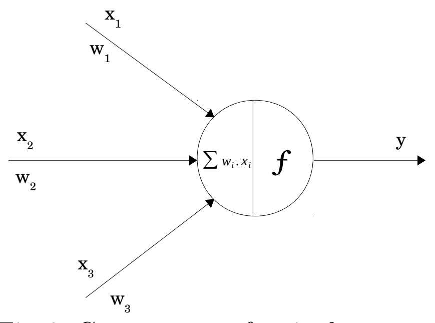

Fig. 2: Components of a single neuron

Each neuron has an activation function. There are different choices for the activation function such as the sigmoid function, relu function, and hard-sigmoid function. FHE-DiNN uses sign function for activation for ease of homomorphic evaluation. However, it uses hard-sigmoid during training. The sign function is defined as follows.

$$\text{sign}(x) = \begin{cases} -1, \text{ for } x < 0 \\ +1, \text{ for } x \geq 0 \end{cases}$$

3 Discussion

In this section, we will discuss two real-world applications of homomorphic encryptions. The first one is CryptDB, which uses partial homomorphic encryption and the second one is FHE-DiNN which implements fully homomorphic encryption.

3.1 CryptDB: CryptDB [11] provides security for applications backed by SQL data-bases. It allows execution of queries over encrypted data in the SQL database. SQL uses a well-defined set of operators which supports such query executions. More specifically, CryptDB employs *onions of encryption* which supports storing multiple ciphertexts within each other. The top layer will use random encryption which provides maximum security. The next layer uses deterministic encryption which supports equality operations over the ciphertext. The inner layer supports order-preserving encryption [1] which reveals the order relation between ciphertexts ($\text{OPE}_k(m_1) < \text{OPE}_k(m_2)$). It also implements partial homomorphic encryption to support queries requesting SUM aggregate and average using paillier cryptosystem. Based on the type of the query request (such as the equality =, order queries like <, >, or, SUM, AVG), CryptDB decrypts (remove) one or two top layers while preserving the encryption on a layer that supports execution of the requested query. CryptDB provides good insight on how to use homomorphic encryption in a practical setting and also on how some computations can be performed on the ciphertexts without the need to decrypt them. Based on this insight, we can extend our discussion to using homomorphic encryptions for classification problems based on neural networks.

3.2 FHE-DiNN: FHE-DiNN [3] implements a *Discretized Neural Networks* in which the inputs are restricted to the set $\{-1, 1\}$, and the weights are discretized, and finally the network is homomorphically evaluated. FHE-DiNN does not train the discretized network directly. It instead discretize an already trained standard neural network which works on real-valued inputs. Also, FHE-DiNN does not train the neural network on encrypted data. It instead takes weights learned from a neural network that is trained on plaintext data. It then performs a homomorphic evaluation on the network for encrypted inputs and corresponding learned (and discretized) weights (the weights are not encrypted). It implements homomorphic evaluations on the neural network operations such as calculating the multisum ($\sum_{i=1}^{n} w_i x_i$) and applying the activation function while preserving the homomorphic property. During the process, each neuron's output is also refreshed through bootstrapping.

FHE-DiNN uses a LWE-based encryption scheme which is modified in a way to allow for the output of the multisum to fall in the message space. This encryption scheme is defined as follows.

Definition 3.2.1: Let B be a positive integer and let $m \in [-B, B]$ be a message. Split the torus $\mathbb{T}$ in $2B + 1$ slices, one for each possible message. The encryption and decryption algorithm is given as follows.

1. **Setup**(λ): fix $n = n(\lambda)$, $\sigma = \sigma(\lambda)$ and return $s \overset{\$}{\leftarrow} \mathbb{T}^n$.
2. **Enc**(s, m): for $a \overset{\$}{\leftarrow} \mathbb{T}^n$, $b = \langle s, a\rangle + e + \frac{m}{2B+1}$, where $e \overset{\mathcal{X}_\sigma}{\longleftarrow} \mathbb{R}$, return (a, b).
3. **Dec**(s, (a, b)): return $\lfloor (b - \langle s, a\rangle) \,.\, (2B + 1) \rceil$.

The scaling factor $\frac{1}{2B+1}$ maps an input message to its corresponding torus slice during encryption while the scaling factor $2B+1$ is applied during decryption. The above encryption scheme has the following homomorphic property for a multisum.

Definition 3.2.2: For message $m_1, m_2 \in [-B, B]$, secret key s, ciphertext $c_1 = (a_1, b_1) \leftarrow \text{Enc}(s, m_1), c_2 = (a_2, b_2) \leftarrow \text{Enc}(s, m_2)$, and constant $w \in \mathbb{Z}$, the following homomorphic property holds as long as $m_1 + w \,.\, m_2 \in [-B, B]$ and the noise on the ciphertext did not grow beyond limit.

$$\text{Dec}(s, c_1 + w \,.\, c_2) = \text{Dec}(s, (a_1 + w \,.\, a_2, b_1 + w \,.\, b_2)) = m_1 + w \,.\, m_2$$

FHE-DiNN uses **sign** function as the activation function. During training, the standard neural network employs hard sigmoid function for activation. Due to this shift some error is incurred during classification. The bootstrapping process and the activation function are applied at the same time. The bootstrapping starts by mapping the torus $\mathbb{T}$ to an object called *wheel*. The wheel is split into $2N$ "ticks" where $[-N, N]$ is the range of the multisum ciphertext.

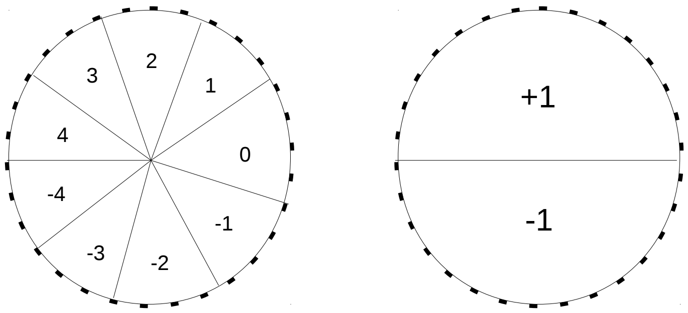

Fig. 3: Bootstrapping using wheel

The bootstrapping then maps the multisum value to the closest (rightmost) tick in the wheel. The top half of the wheel corresponds to an encryption of $+1$ by the activation function and the bottom half corresponds to an encryption of -1 (see fig. 3). The bootstrapping is implemented in three steps: Bootstrap =

Extract $\circ$ BlindRotate $\circ$ KeySwitch. The KeySwitch is first applied to reduce the dimension of a LWE ciphertext. This is followed by BlindRotate which implements a homomorphic evaluation of the expression $(\sum_i x_i X^i) . (\sum_i w_i X^{-i})$ (multiplication of two polynomials in X, encoding inputs x and weights w, which is a TLWE ciphertext). Finally, Extract is applied to obtain the constant term $\sum_i w_i x_i$ from the above expression. The resulting constant term which corresponds to the encrypted (bootstrapped) multisum is then mapped to the corresponding slice in the wheel which is then evaluted by the activation function resulting in -1 or +1 (encrypted).

4 Experimental results

Both CryptDB and FHE-DiNN implementations are available online. Although I experimented with them for understanding I did my own implementation using the ideas presented in those papers. This helped me to gain some important intuitions. In this section, I will discuss the details of my implementation. The code is available in the github[3].

Dataset: The dataset used for this project consists of images from the Flickr photo sharing website. The raw images are treated as numerical feature vectors on which standard machine learning techniques can be applied. Every image, which is an 8×8 pixel, is pre-processed into an $8 \times 8 \times 3 = 192$ dimensional feature vector. The neural network is trained directly on the feature vectors. Each given image has an orientation that takes values from the set $\{0°, 90°, 180°, 270°\}$. I have modified the digit recognition problem (as in FHE-DiNN) to detecting correct orientation problem.

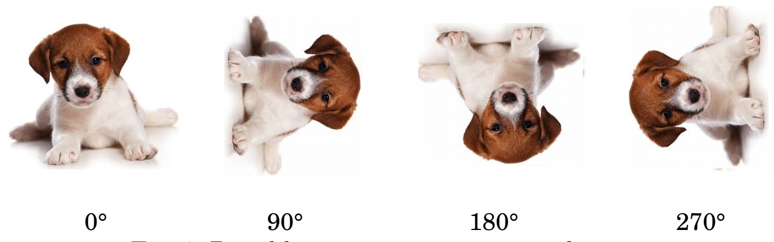

Fig. 4: Possible orientations in input dataset

In fig. 4, we can see the possible orientations of an image in the given dataset. The neural network implemented takes an input image as a vector of numbers and identifies its orientation.

Implementation: I implemented a fully connected feed-forward neural network with 12 input nodes, 12 hidden nodes, and 4 output nodes (each corresponding to a specific orientation). The input nodes corresponds to features

[3] The full implementation is available at `https://github.iu.edu/pvivekan/B504_Introduction-to-Cryptography`

(indexes in the input feature vector) learned using *decision tree*. Then I trained the network on the input dataset using the *backpropogation* algorithm. I used the `Pyfhel()` library available in python for the homomorphic evaluation of the neural network. It supports various operations on the integers and the fractions including addition and multiplication. However, the library has no support for bootstrapping. I handled this problem by choosing a large prime thereby allowing more room for the noise to grow in the ciphertext. This helped me to evaluate the entire neural network without the need for bootstrapping. But implementing bootstrapping could have improved the execution time as the code execution slowed down with increase in noise on the ciphertext. This implementation is only *somewhat homomorphic*. Adding bootstrapping to it will make it *fully homomorphic*. Initially, I discretized the network using the ideas given in FHE-DiNN. It introduced some error in the classification. To improve accuracy and also because the `Pyfhel()` library supports real valued inputs, I removed the discretization part. The FHE-DiNN maps the input space to the set $\{+1, -1\}$ (they mapped pixel with value > 128 to $+1$ and < 128 to -1). However, I didn't need this step for my implementation and so I kept the input values as it was originally. In FHE-DiNN, they encrypted only the input and not the weights. I tried classification by encrypting both the weights (including bias) and the input. The classification accuracy on the test data was excellent but the total execution time was very poor. It took more than `30 mins` where the normal execution time without homomorphic evaluations took less then `1 sec`. Later, I removed the encryption for the weights which gave much better performance related to the execution time (it took a little over `12 mins`) while maintaining the accuracy. As I allowed more room for the noise to grow on the ciphertext I was able to maintain the accuracy of the classification using homomorphic evaluations same as that of the plain execution (`663 out of 943` images were correctly classified giving an accuracy of `70.31%`). Initially, I tried hard sigmoid and relu activation functions for training and testing. But it performed poorly for my network giving classification accuracy of around `25%`. Later, I switched to sigmoid function which gave me the accuracy rate of `70.31%`. One drawback of this code is that I could not implement the homomorphic evalution of the activation function using `Pyfhel()`. It didn't have the supporting operations. So, I have to decrypt and reencrypt during every invocation of the activation function. Also, I used a max function in the output layer for which I could not implement the corresponding homomorphic version. However, I found some ideas from [4] to evalute the max function and comparison operators (used in relu and hard sigmoid) homomorphically.

5 Observations and Conclusion

This project allowed me to investigate in two different directions. The first direction emphasize more on machine learning and the second direction revolves around cryptography. In the first direction I examined questions like what happens if the neural network is trained on ciphertext data? Does the neural network learns different set of features when trained on ciphertext compared to the features learned during training on plaintext? If so, how does that affects the efficiency and accuracy of the classification? Can similar techniques be extended to machine learning algorithms other than neural networks such as decision trees and perceptron? In the second direction, I experimented with questions such as does any data is leaked during the homomorphic evaluation of the neural network? If so, how significant it would be for an adversary? How does a fully homomorphic evaluation scales compared to a somewhat homomorphic evaluation? How accurate is the classification based on encrypted data?

I am able to find answers for some of the above questions. To name a few, training the neural network on ciphertext is bad idea. The ciphertext is noisy and we cannot extract useful features from them. If we are able to extract useful features from ciphertext, then it indicates weakness in encryption. Also, homomorphic encryption makes classification very slow. If homomorphic encryption is properly implemented then it should not leak any significant information to the adversary although security might be compromised due to *malleability* of homomorphic encryptions making them weak against choosen-ciphertext attacks (CCA). Although, somewhat homomorphic encryption seems to have shorter execution time compared to fully homomorphic encryptions (due to absence of bootstrapping), this may not be true in general. Regarding accuracy, if provided more room for the noise on the ciphertext to grow, the classification accuracy seems to be the same as that of the normal classification.

In a nutshell, this project gave me the opportunity to build on some abstract algebra concepts such as ring, field, factor ring, and ideals. Also, I am able understand potential post-quantum crytographic foundations such as learning with errors and its variant ring learning with errors problems. I am also able to build a solid understanding of homomorphic encryptions and the need for bootstrapping.

Acknowledgements

For this project, I incorporated some of my previous work and ideas that I learned from B551 - Elements of Artificial Intelligence and B555 - Machine Learning courses.

References

1. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, Paris, France (June 2004)
2. Archer, D., Chen, L., Cheon, J.H., Gilad-Bachrach, R., Hallman, R.A., Huang, Z., Jiang, X., Kumaresan, R., Malin, B.A., Sofia, H., Song, Y., Wang, S.: Applications of homomorphic encryptions. Presented in Homomorphic Encryption Standardization Workshop held by Microsoft, Redmond, Washington (2017)

3. Bourse, F., Minelli, M., Minihold, M., Paillier, P.: Fast homomorphic evaluation of deep discretized neural networks. In: In: Shacham H., Boldyreva A. (eds) Advances in Cryptology – CRYPTO 2018. Lecture Notes in Computer Science, vol 10993. Springer, Cham (September 2018)
4. Cheon, J.H., Kim, D., Kim, D., Lee, H.H., Lee, K.: Numerical method for comparison on homomorphically encrypted numbers. In: Galbraith, S.D., Moriai, S. (eds.) Advances in Cryptology – ASIACRYPT 2019. pp. 415–445. Springer International Publishing, Cham (2019)
5. Crockett, E., Peikert, C.: Challenges for ring-lwe. IACR Cryptology ePrint Archive **2016**, 782 (2016)
6. Frame, G.: Heide: An ide for the homomorphic encryption library helib (2015). https://doi.org/https://doi.org/10.15368/theses.2015.64, `https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=2523&context=theses`
7. Gentry, C.: A Fully Homomorphic Encryption Scheme. Ph.D. thesis, Stanford, CA, USA (2009)
8. Hiromasa, R., Abe, M., Okamoto, T.: Packing messages and optimizing bootstrapping in gsw-fhe. In: Katz, J. (ed.) Public-Key Cryptography – PKC 2015. pp. 699–715. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
9. Kim, M., Lauter, K.: Private genome analysis through homomorphic encryption. BMC Medical Informatics and Decision Making **15**, S3 (12 2015). https://doi.org/10.1186/1472-6947-15-S5-S3
10. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Proceedings of the 18th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT), Prague, Czech Republic (1999)
11. Popa, R.A., Redfield, C.M., Nickolai Zeldovich, H.B.: Cryptdb: Protecting confidentiality with encrypted query processing. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP), Cascais, Portugal (2011)
12. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. J. ACM **56**(6) (Sep 2009). https://doi.org/10.1145/1568318.1568324, `https://doi.org/10.1145/1568318.1568324`
13. Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms (1978)