Privacy-Preserving Machine Learning Training Through Homomorphic Encryption

Matthew Townsend
Network Computer Security
SUNY Polytechnic Institute
Utica NY
townsemc@sunypoly.edu

Keywords – Homomorphic Encryption (HE), Fully Homomorphic Encryption (FHE), Federated Learning (FL), Machine Learning (ML), Cheong Kim Kim Song (CKKS)

Abstract

This paper explores the application of Fully Homomorphic Encryption (FHE) for privacy-preserving training of machine learning models. It investigates FHE as an alternative to traditional privacy techniques like federated learning and differential privacy. The paper details the implementation of a logistic regression model using the TenSEAL library for FHE in Python. While achieving similar accuracy to a non-encrypted model, the FHE model suffers from significantly increased training time and memory usage.

1 Introduction

Privacy Preservation and Machine Learning. These are two of the leading topics in today's technology landscape and are often seen as contradictory. As in the Training of AI/ML, vast amounts of data are collected, from websites, databases, and users. The collection of this user data is why Privacy Preservation has come to the forefront. Many companies are looking to profit from including privacy techniques, now described as ethical AI, and stating they will not use your data on training while giving it to a third party.

The basis for these two topics today needs to be revised. One requires a lot of detailed data to train a perceived good model, while the other seeks to eliminate this level of detail. However, there is a way to bridge that gap between topics. One such method is Fully Homomorphic Encryption (FHE), a new encryption method that keeps the data encrypted while allowing the models to train on the data in full.

During my research of privacy techniques for ML, two concepts took hold: There is no single method solution at the moment, and what is Fully Homomorphic Encryption? This capstone morphed into FHE on Machine Learning. I stayed true to the topic and used it to "secure" the data's training, more in the abstract security of the data than the more direct techniques listed in the following sections.

2 Road-map

For this capstone, I learned about several privacy preservation tools and decided to implement one. This is not production-ready code as the model is rather simplistic and would require the addition of another technique, most likely differential privacy with FHE. The outline for this paper goes as follows

- Techniques for achieving privacy-preservation, Federated Learning, Differential Privacy
- Homomorphic encryption, high-level explanation, development, and chosen scheme
- Implementation of Fully Homomorphic Encryption into an ML model during the training phase.

For the implementation, I limited myself to using Python and Jupyter notebooks for running my code. This will make it easier to troubleshoot my code as the model is in sections allowing me to pinpoint where the errors occur. The model I developed was a logistic regression Neural Network to detect fraudulent transactions.

3 Privacy-Preserving Techniques

There are several different techniques and methods to decrease what can be identified/inferred about individuals and companies based on datasets. Many methods have decreased data quality or trust in others giving their data. These include but are not limited to Federated Learning, Anonymized Learning,

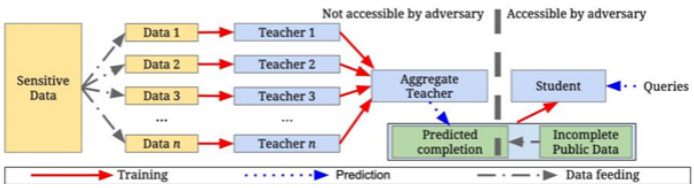
and Differential Privacy. All methods have positives and negatives in implementation including FHE.

3.1 Federated Learning

A technique which relies on users training a model using their own devices, commonly edge devices. Approaches vary for this technology from the Centralized distribution of the computation to the decentralized approach where all devices communicate and share their computation to form a complete model. Finally, heterogeneous FL uses any device that can compute such as IOT, phones, and computers to compute the model on the data accumulated on the devices[2]. In theory, FL is one of the best approaches to keeping data secure but falters in getting devices to join its network. People have a distrust in any of their data being used, why would they download a model to have their data used, it would require extreme simplicity to have people download and run the model.

3.2 Differential Privacy

A technique based on anonymization of the data to generalize the data within the dataset. This can be done through the addition of noise to the dataset which has the drawback of decreasing the accuracy of the model. An example in the paper *Privacy-Preserving Machine Learning* by the Alexandra Institute [2] is seen in Figure 1



https://www.alexandra.dk/wp-content/uploads/2020/10/Alexandra-Instituttet-whitepaper-Privacy-Preserving-Machine-Learning-A-Practical-Guide.pdf - Figure 1

The dataset is divided into subsets that train an individual model, and then all models are aggregated together and used to train the final model. This adds noise to the model prevention of many attacks such as model inversion and membership inference.

4 Homomorphic Encryption

The last privacy technique I reviewed, yet I believe it is the most important technique on the market. For this technique, there are two methods to use it, encrypted training, or encrypted testing. The latter, which is by far easier to implement, sklearn a beginner-friendly ML library that can use encrypted testing data with decent results. The first, encrypted training is the path I choose to pursue.

4.1 Traditional Encryption Vs Homomorphic Encryption

In traditional encryption “cryptosystems” - that being the group of functions that form RSA or AES, once you encrypt the data, it is locked in its state. The only way to modify the data is to decrypt the data, it's a strength but also a weakness in modern data systems. For example, figure 2 is a cloud-based computation system. Where the user is in a remote area and needs to send sensitive data to a central server they would do the following.

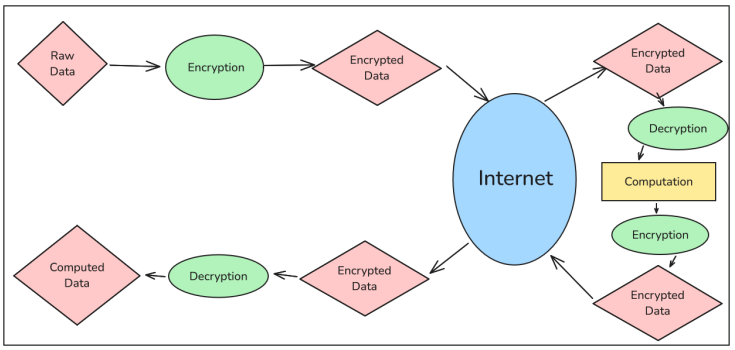


Figure 2

This is the issue with traditional encryption, secure to the point of inconvenience. Require cycles of encryption and decryption just to compute the data. This opens the data to be seen by a third party whether it be an employee of a service provider, or an adversary, by definition it is an attack surface in the machine.

This is where Homomorphic Encryption thrives, in particular for this paper the Fully Homomorphic Encryption (FHE) cryptosystem. The data does not need to be decrypted to allow computation of the data. Most operations, apart from deletion can be done, see Figure 3 for the scenario when FHE is used.

Once the data is encrypted, it does not need to or can be decrypted until the owner uses the key to decrypt the data. This removes the need to have a shred key whether it be public or private, removing that entire attack surface from the cloud computing system for data security. There are both symmetric and asymmetric approaches to HE.

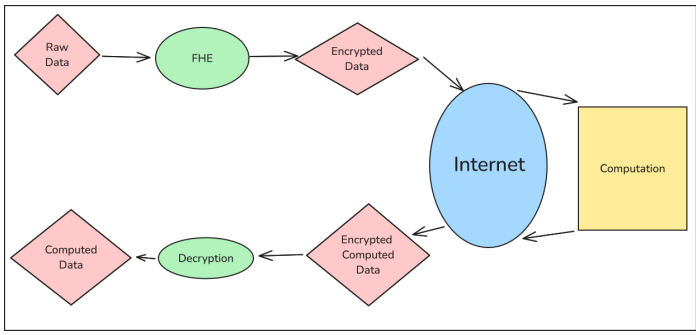
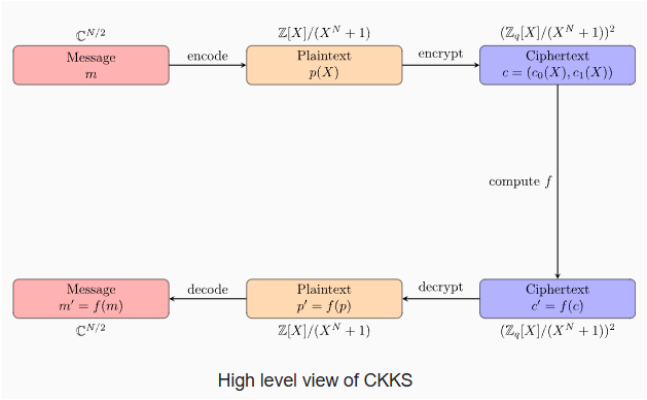


Figure 3

This is just one of many applications which FHE can improve when implemented. These can vary from situations where data confidentiality and integrity are critical such as health care, banking, and more [3]. The next step is being part of larger applications, such as cloud computing as in the previous example, and in this paper as part of a ML/AI system.

4.2 Inter workings

Within the sphere of Homomorphic encryption, there are several types, partial, somewhat, and fully HE. Partial HE allows for one type of operation, add or multiply, infinite times. Somewhat HE enables both operations to be used, but only a limited number of times. Finally, FHE is the combination of both versions, multiple operations, infinite amount of times [4]. This is done through homomorphism, an algebraic concept I will not try to explain here. In short, the data can remain encrypted in all three phases now, at rest, in use, and in transit. Some individuals have decided it is a paradigm shift in information security because of this.



High level view of CKKS

https://blog.openmined.org/ckks-explained-part-1-simple-encoding-and-decoding/ Figure4

Figure 4, displays the overview of the encryption process by CKKS, a recent scheme which will be seen again in this paper. The one notable difference is that the message or what you are trying to encrypt is not plain text, instead, it is an encoded version of your message which is then encrypted. This also means that you must decode your results. Most implementations auto-encode the data. What is different with CKKS is that the plaintext is a polynomial, as they offer better efficiency while staying secure [5]. This is also why CKKS is still an option with ML, it is far faster than other schemes, especially when constant encryption and decryption cycles are required.

4.3 Generations

Despite its short lifespan, it has undergone four generations of schema, from 2009 through today. The generations began once they achieved complete encryption using lattice-based or quantum-safe cryptography[6][7]. The schemes iterate upon each other, taking the benefits of the previous while slowly advancing. The next generations need to take on the cryptosystem Achilles heal, its need for resources.

4.4 Implementation and Standardization

There are several implementations of HE from institutions, private companies, and startups. The biggest being HElib by IBM, SEAL by Microsoft, and OPENFHE made by a conglomeration of institutions and others. Recent implementations have come from the startup area as well, such as Zama AI, with their concrete ML framework for using sklearn functions with HE, and the TFHE implementation[8]. For this paper, I choose the TENSEAL implementation[9], a Python wrapper for the SEAL library. This worked great for being restricted to Python and has CKKS support. One improvement I would make in the future is to examine not only popular libraries but ones under recent development.

There is an attempt at standardization of HE development with the same conglomeration that made OPENFHE, aptly called the homomorphic encryption organization[10]. In trying to keep this expanding sphere together and organized, some recent developments include debating on hardware accelerators. This is needed in this industry as it has excited the initial stages of development and is slowly reaching mainstream use by corporations, and soon hobbyists which will only enhance the speed of development.

4.5 Issues

One issue that occurs within all HE especially in fourth-generation schemes is the accumulation of noise. Each operation results in noise being added to the data. Once the noise reaches a

threshold, the data becomes distorted and unusable. Due to this many schemes use a method called bootstrapping[6]. It systematically decrypts the model to revert any noise accumulation and re-encrypts itself.

The next most glaring issue is that FHE is a resource-intensive technology to implement. As seen later in this paper the amount of resources needed to train on encrypted data is far greater and longer to complete. Unfortunately, there are no easy fixes to this problem in the current day. One estimate released by IBM stated that the compute penalty was 50:1 and 20:1 memory penalty in total[11].

5 Implementation

As previously listed in Section 1, the restrictions are to use Python for the entirety of this project. This is not difficult as there are native FHE libraries in Python along with wrappers for C-based libraries, such as TenSEAL. The second reason is that Python is the most popular language for data analysis and by extension ML/AI.

5.1 Dataset training

The dataset I used is a rather simplistic fraud detection set as it has a small number of features but nearly 40,000 rows for good variation when I shuffle the data. I received this dataset through a class lab, and one of the best-performing datasets I had and took up the least space encrypted. I did perform preprocessing on the data such as dropping the feature of "paymentMethod" as it is a string value. Figure 5 is a code snippet for this process. After this
section I grouped the data and standardized the set.

```
data = pd.read_csv("payment_fraud.csv")
# drop some features
data = data.drop(columns=["paymentMethod"])
```

Figure 5

For training, I used a train test split(TTS) function to split the dataset into separate train and test sets. In particular, I used the sk-learn implementation of TTS as it simplified my code and for some reason improved the result of my implementation of TTS.

```
def train_test_split(x, y):
    # shuffle the data
    sklearn.utils.shuffle(x, y)
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
    return x_train, x_test, y_train, y_test

x_train, x_test, y_train, y_test = Credit_data()
```

Figure 6

In Figue 6, we see the function definition for Train_Test_Split, which uses the sklearn TTS. I also added a shuffle to ensure variation in the data and not just the first section or the same data.

5.2 Non-Encrypted Model

For both models, Non-encrypted and encrypted, I used pytorch to construct them, as the standard sk-learn library does not support the type 'ckks-vector' produced by TenSEAL encryption. I decided to build both to have a more equal comparison. I defined a Logistic Regression torch Neural Network. In short, it predicts the data set, based on what it learns from the training set. In the forward method, I applied the activation function sigmoid, as it ensures the output is in the range of 0 to 1. So for example if the output is 0.93 (93%) it would classify it as likely fraud if 1 is fraud, likewise if 1 is legitimate. This will come into the fold later. The definition of the model is a class so you can call it for the model.

After defining the model, you then define the training for the model to undergo. The training is an iterative process and is based on the epochs defined. For each epoch in this model, the parameters are updated using gradient descent to minimize loss. The loss is calculated using binary cross entropy, which measures the difference between the prediction and the label.

At the end of the model, we are left with the evaluation of the overall scores not just per epoch or prediction. In the evaluation, I measured the average time per epoch, the overall accuracy of the model by comparing the predictions to the label and determining which are correct, and lastly the memory usage for the block.

5.3 Encrypted Model

This version of the model requires significantly more code to complete as it requires self-definition of the passes, equations, and loss. To begin with, the initialization takes a model as input, extracts its weights and biases, and stores them into separate

lists. Then initializes two variables for gradients and one as count for iterations.

5.3.1 Encrypted Operations

The first operation defined is the forward pass which takes the encrypted data as input. It combines the weights with the input and then adds the bias to the result. Lastly, it uses the approximated sigmoid function on the results(Figure 7). The approximate sigmoid is defined later as a polynomial approximation to conserve resources as a standard sigmoid would be costly. This polynomial approximation is one reason for the scheme chosen in the next section.

```
@staticmethod
def sigmoid(enc_x):
    # this is a degree 3 polynomial approximation of the sigmoid function
    return enc_x.polyval([0.5, 0.197, 0, -0.004])
```

Figure 7

Backward pass takes in encrypted x, labels, and results as input. The main operations are to Calculate the difference between the results to the labels and to calculate the new gradients based on the difference. Lastly, it iterates the count variable for the next pass. The aptly named update parameters, update the parameters which are the bias and weight according to the gradients determined in the backward pass. Then it resets the gradients for the next pass-through.

Lastly, the final three functions are plain_accuracy, encrypt, and decrypt. The plain accuracy function converts the weight and bias into torch tensors w and b, runs the two tensors through test data, and returns the output which is used to calculate the correct results. Encrypt and decrypt use the built-in TenSEAL functions encrypt() and decrypt(). It applies these functions to the weight and bias so we can evaluate the model after it has finished, and I can use standard functions again, these processes are seen in Figure 8.

```
def encrypt(self, context):
#Encrypts the weights and bias
    self.weight = ts.ckks_vector(context, self.weight)
    self.bias = ts.ckks_vector(context, self.bias)

def decrypt(self):
#Decrypts the weights and bias
    self.weight = self.weight.decrypt()
    self.bias = self.bias.decrypt()
```

Figure 8

5.4 Performing Encryption – CKKS

The scheme I chose to implement was the CKKS to encrypt the training dataset. This is because of its ability to perform approximations, which I used with the sigmoid function. The second reason is that it is common for implementations to use it for their models.
The implementation of the scheme was concise and could be done with one line, although I separated the polynomial modulus and coefficients to ensure they were correct. The next step is to generate the keys and then perform the encryption which I have timed. Figure 9 displays this process.

```
poly_mod_degree = 8192
# the bit-length of the modulus chain
coeff_mod_bit_sizes = [40, 21, 21, 21, 21, 21, 21, 40]
# create TenSEALContext
enc_training = ts.context(ts.SCHEME_TYPE.CKKS, poly_mod_degree, -1, coeff_mod_bit_sizes)
# generate keys
enc_training.global_scale = 2 ** 21
enc_training.generate_galois_keys()

t_start = time()
enc_x_train = [ts.ckks_vector(enc_training, x.tolist()) for x in x_train]
enc_y_train = [ts.ckks_vector(enc_training, y.tolist()) for y in y_train]
t_end = time()
```

Figure 9

5.5 Running the Encrypted Model

In this separate box, I create an instance of the ELR class, and initialize the plain_accuracy as accuracy. The training is not too different than before but calling the function defined, forward pass, backward pass, update params, etc. Then for results I get the average time per epoch, accuracy, difference between non-encrypted and encrypted, and memory usage

```
for epoch in range(EPOCHS):
    ELR.encrypt(enc_training)

    t_start = time()
    for enc_x, enc_y in zip(enc_x_train, enc_y_train):
        # forward pass
        enc_out = ELR.forward(enc_x)
        # backward pass
        ELR.backward(enc_x, enc_out, enc_y)
    ELR.update_parameters()
    t_end = time()
    times.append(t_end - t_start)
    # decrypt the model and calculate the accuracy
    ELR.decrypt()
    EN_accuracy = ELR.plain_accuracy(x_test, y_test)
    print(f"Accuracy at epoch #{epoch + 1} is {EN_accuracy:.4f}")
```

Figure 10

Here is the training loop in Figure 10, notice when it calls the encrypt and decrypt functions. It re-encrypts the data at the start of the loop, trains then decrypts after to pull the metrics out of the data. So FHE is being completed, this is not simulated encryption training.

6 Results

My results from running these two models were better than I expected, yes it was heavy and slow, but the accuracy was nearly identical. The results for the non-encrypted model were an average of 0 seconds per epoch and 0.741 accuracy. For the encrypted model I and 48 seconds per epoch and 0.736 accuracy. A 48:1 time penalty and a 0.005 accuracy difference in favor of non-encrypted.

The memory used was also starkly different between the two models. The non-encrypted model used 345.7 MB, compared to the 2304.9 MB required to run the Encrypted model. Around 6.8 times as much memory is needed to run the encrypted model. The process to run encryption used 2288.16 MB and 12 seconds to complete, with parameter tuning to fit the need. I had previous versions take minutes to run and full system memory.

The size difference comes from the size of the data it is processing. The size drastically increases, reaching 392586908 bytes compared to 14,336 bytes. Figure 11 is pulled from the notebook containing the results of the block dedicated to the size of the datasets.

```
############# Data summary #############
Size of x_train: 14336 bytes
Size of enc_x_train: 392529200 bytes
Size of y_train: 3584 bytes
Size of enc_y_train: 392569378 bytes
Memory usage: 2288.16 MB
########################################
```

Figure 11

7 Conclusion

I set out to research privacy techniques that apply to Machine Learning, I achieved this. It came in the way of research for Federated Learning and Differential Privacy. Finally, Fully Homomorphic Encryption then just a technique I went with, opened an entire field of cryptography which I am interested in joining. By applying it to a ML model I have seen the relatively small, but greatly expanding group of people that are a part of this scene. When I chose this topic I planned to start my graduate degree and focus on solely AI/ML classes, while I intend to follow this path, I found a different area of computer science I am interested in and will take classes if they are available and possibly use in some form for my master's thesis in a few years.

7.1 Problems and Issues that arose

The primary problem I had was getting the implementation to complete its training cycle. After 3 different models and 2 libraries, pytorch finally came through. I tried to work with a simple sklearn model, thought I could reuse some projects, but all failed with errors of ‘ckksvector’ type not supported. I was able to perform simulated encrypted training, in which the data is encrypted up to and after the point of training when it is decrypted. This error removed any chance of using or trying sklearn, I did not learn until my model with pytorch was running that the company Zama AI had sklearn friendly encryption. This problem compounded as I began to experiment with the modulus and coefficients which led me to having memory problems. I would crash a google colab notebook trying to run the encryption. Next would be the time as everything would say it takes a long time so I would wait when I first began this project for 20-30 minutes thinking it was taking this long when it just wasn't reading the data in.

Not until after the model was running did the next problem arise, this paper, how do I explain FHE to an incoming freshmen. I took the route of explaining standard encryption and went to the point where the math behind it was needed. This is wht I did not mention homomorphism as an algebraic concept. Likewise I had to then try to explain my model in a way for them which have little to no knowledge of coding as NCS freshmen.

7.2 Improvements in review

After reviewing my work I believe another improvement I could have made was going all in on just FHE or ML, not combining both. More resources and knowledge are trying to get databases and infrastructure to use FHE then they are working on the ML angle. This is an emerging field meeting an inferno, the code is messy, deep knowledge of the topics is required, but it is possible.

Works Cited

[1] https://www.v7labs.com/blog/federated-learning-guide

[2] Aslanyan, Z.,& Vasilikos, P. (2020). *Privacy-Preserving Machine Learning*[White Paper]. Alexandra Institute.

https://www.alexandra.dk/wp-content/uploads/2020/10/Alexandra-Instituttet-whitepaper-Privacy-Preserving-Machine-Learning-A-Practical-Guide.pdf

[3]https://digitalprivacy.ieee.org/publications/topics/homomorphic-encryption-use-cases

[4]https://www.splunk.com/en_us/blog/learn/homomorphic-encryption.html

[5]https://blog.openmined.org/ckks-explained-part-1-simple-encoding-and-decoding/

[6] https://www.geeksforgeeks.org/homomorphic-encryption/#

[7]https://people.csail.mit.edu/vinodv/FHE/FHE-refs.html

[8]https://www.zama.ai/

[9]https://github.com/OpenMined/TenSEAL

[10]https://homomorphicencryption.org/

[11]https://arstechnica.com/gadgets/2020/07/ibm-completes-successful-field-trials-on-fully-homomorphic-encryption/