B.A.Sc. Thesis

Division of Engineering Science
UNIVERSITY OF TORONTO

Secure Machine Learning with Approximate Arithmetic

by

Vele Tosevski

Supervisor: Glenn Gulak
April 2020

Abstract

Increasingly, companies around the world are collecting and selling customer data for financial gain. Users are willingly giving their unencrypted data to companies in exchange for a service at the expense of their privacy. This thesis explores Fully Homomorphic Encryption with approximate arithmetic, a new form of encryption that allows approximate mathematical computation on encrypted data. It provides a mechanism to homomorphically encrypt data operands on mobile devices and then send them to a third-party for evaluation. We propose a Secure Machine Learning model that performs inference on encrypted operands and returns an encrypted prediction to the user. We propose an approximate activation function and three homomorphic algorithms which, when combined, homomorphically classify a data sample. We show that Secure Machine Learning is practical. Our model performs to 97% accuracy and our homomorphic inference results in only 6% loss. Predictions operate, on average, for 7.7 seconds with average precision on the order of 10^{-4}. We show that companies can perform data analytics on user data for financial gain without compromising their privacy.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Glenn Gulak for his continuous support and faith in my abilities to complete this thesis.

I want to also thank the Department of Engineering Science at the University of Toronto for providing me with a meaningful and challenging education that now, more than ever, I see was definitely worth it.

I want to give a special thanks to Professors Michael Collins and Jason Foster for showing me how to be an engineer, for instilling in me the excitement and depth of what it is to be an engineer.

Lastly, I want to thank my family for their love and endless support through my ups and downs these past four, Engineering Science years. My parents came from Macedonia looking for a better life, giving up their education and careers so that my sister and I could have a better life. This thesis, I hope, puts a smile on their faces and gives them resolution that they made the right decision.

Table of Contents

List of Figures

List of Tables

1 Introduction

Fully Homomorphic Encryption (FHE) is a recent breakthrough in theoretical Computer Science that began with the seminal work by Craig Gentry [1] in 2009. This type of encryption allows sequences of mathematical computations (e.g., addition and multiplication) to be applied to data while the data remains encrypted, which enables many new applications including use in Secure Machine Learning (SML). Extensive study of FHE over the past decade has yielded many different mathematical foundations for this encryption scheme (e.g. CKKS [2]). These schemes vary in capability, some offering exact decryptions while others approximate (floating-point) decryptions. As a result, different tasks use different encryption schemes. Some machine learning (ML) applications require approximate predictions, others may require exact predictions. Those that require approximate predictions usually use a form of ML called inference [3]. Inference relies on the probability of data to infer predictions. Due to this nature, secure ML inference could use a scheme such as CKKS since its computations are approximate. ML inference is also significantly simpler than training due to the reduced number of computations required and therefore will be the focus of this thesis.

This thesis will focus on secure machine learning applications using mobile devices for the enterprise. In particular, from a cell phone, we will homomorphically encrypt a data operand that will be transported to the cloud. The encrypted data operand will then serve as input to an FHE machine learning inference algorithm which generates an encrypted classification output. While in the cloud, the data input, the machine learning model parameters and the inference output will remain encrypted at all times. The encrypted data inference output can be converted back to plaintext using the secret decryption key in a secure environment.

Many different open-source FHE libraries exist (eg. HElib [4], HEAAN [5]). Microsoft SEAL [6] is a native C++ library that performs FHE operations efficiently (by current FHE standards) and employs the CKKS scheme. By using this library, we can focus on finding a better and more computationally efficient SML inference algorithm.

Machine Learning using homomorphically encrypted data is computationally demanding and is not anywhere as computationally efficient as Machine Learning with unencrypted (plaintext) data. Therefore, the research challenge addressed in this thesis is twofold. First, we wish to benchmark FHE encrypt and decrypt speeds on processors found in modern day mobile devices. Second, we wish to create computationally efficient FHE ML algorithms that will have better execution times relative to other existing published FHE ML algorithms as evidenced by careful quantitative benchmarking of representative ML benchmarks using real-world data.

2 Related Works

2.1 Homomorphic Encryption

A problem that faces the world today is lack of privacy. With the rise of social media and global search, users willingly give their private information to large companies who provide services in exchange for their data. Naturally, these services perform computation on a user's private data in order to operate, data that must not be encrypted for the service to work. While in an ideal world this would not be a problem, unfortunately, users cannot know for certain whether their private data is being sold to third-parties or used for reasons other than those pre-warranting its acquirement.

A solution to this trust problem can be to utilize an encryption scheme that allows simple mathematical operations to be performed on encrypted operands and yield correct answers when decrypted, for example

$$x_{encrypted} + x_{encrypted} = (2x)_{encrypted}$$

$$x_{encrypted} * x_{encrypted} = (x^2)_{encrypted}$$

This would allow service providers to perform operations on user data while not compromising user privacy. The mathematical formulation for such a scheme was first proposed [7] by Rivest, Adleman, and Dertouzos in 1978. They proposed the idea of a "privacy homomorphism". A *homomorphism* in algebra is a function that maps from one algebraic space to another while keeping the internal algebraic structure intact. While they did not find an algebraic space that would accomplish that, they did propose the foundation necessary for one to exist.

In the years following, several schemes were proposed in response to this open question. These schemes, however, were part of a class of schemes employing "Partial Homomorphic Encryption" (PHE). PHE schemes are ones which only preserve either the additive structure or multiplicative structure of algebraic mappings but not both. Some examples include the RSA scheme [7] which was homomorphic only in multiplication and the Goldwasser-Micali scheme [8] which was homomorphic in addition. In 1996, Ajtai postulated the existence of lattice-based, trapdoor, one-way functions with applications in cryptography and post-quantum cryptography [9]. This prompted many researchers to search for a cryptographic system that harnessed the power of the lattice problem. Some systems were proposed such as the GGH encryption scheme [10] and the NTRU cryptosystem [11] but were not homomorphic in nature.

In 2009, Gentry proposed a "Fully Homomorphic Encryption" (FHE) scheme based on lattice cryptography [1]. Gentry introduced a "somewhat homomorphic" scheme that included a noise parameter *n* with every ciphertext. When the ciphertext performs multiplication or addition operations, the noise parameter would increase. The novelty in his work was his "bootstrapping" algorithm that he successfully developed, when called, would decrease the noise parameter to less than $\sqrt{n}$ and therefore make his scheme fully homomorphic. However, with single

bootstrapping operations reaching thirty minutes at most, Gentry's proposal proved to be very computationally expensive and therefore impractical for any conceivable use [12].

After Gentry's paper, many optimizations were proposed by various researchers including Brakerski, Fan, and Vercauteren's BFV FHE scheme, which used bootstrapping but was more efficient than Gentry's original proposal [13]. BFV also gave the ability of using a "leveled" FHE (LFHE) scheme, which did not use bootstrapping, effectively making FHE practical for many use cases. In 2016, the CKKS scheme was developed as a LFHE scheme on approximate real number arithmetic [2]. This and the batching technique in CKKS greatly improved FHE and made it even more practical for use. In response to many of these schemes, open-source libraries were created, some of the more famous ones include HELib [4], a low-level, C++ library that implements the BGV [14] scheme with bootstrapping, HeaAn [5], a FHE C++ library that implements the CKKS scheme with bootstrapping [15], and Microsoft SEAL [6], also a C++ library that implements both the CKKS and BFV schemes. With more efficient schemes came more efficient homomorphic mathematical operations (addition and multiplication). As a result, researchers finally possessed the means to homomorphically operate on more complex tasks, some of which included Machine Learning, or in an FHE sense, Secure Machine Learning.

2.2 Machine Learning

In contrast to FHE, Machine Learning has been a field researched since the 1950s. ML harnesses the power of probability and data patterns to predict outcomes to measurable levels of accuracy. For example, datasets, whether they contain images or genetic sequences are transformed into predetermined encodings that when put through a set of functions with weights, weights change in the direction of less error or higher reward, or in other words, models *learn*.

There are three different types of ML – supervised, unsupervised, and reinforcement learning. The learning this paper will focus on is supervised learning, specifically artificial neural networks (ANN). Supervised learning draws its name from models with datasets that have distinct labels. Datasets usually contain "feature" vectors that detail the specific features of an event and correspondingly influence the value of the label. For example, in image recognition networks for models of vehicles, features may include the values of the pixels of an image with a vehicle while the label corresponding would describe the model of the car [16]. Using these feature-label pairs, algorithms can learn the patterns of features that influence these labels and as a result, create predictions of labels given unlabeled feature inputs; this is *inference*.

ANN's are a means for realizing inference. An ANN consists of artificial neurons that take in a signal and output a non-linear value. ANNs contain multiple layers composed of many neurons that when combined, learn and output a prediction. Each neuron contains functions that are adjusted by weights. ANNs learn by adjusting weight values using a computed gradient in the direction of lowest cost. Cost in this case refers to the average sum of squared error between correct labels and incorrectly predicted labels. The lower the cost, the greater the predictive accuracy.

Aside from the mathematical constructs that enable learning, early learning algorithms and machines can be traced back to 1950 when Alan Turing hypothesized a machine that could learn [17]. This same year (1950) also saw the first instance of a neural network machine called SNARC created by Marvin Minsky [18]. In 1958, Rosenblatt created the first perceptron algorithm, a vital part of linear regression and supervised learning today. Cover and Hart created the first "nearest neighbor" algorithm in 1967 that would become fundamental for pattern recognition and supervised learning [19]. The mathematical construct for backpropogation (backprop) or reversed automatic differentiation, a fundamental application to neural networks, was proposed by Linnainmaa in 1976 [20] but not applied to neural networks until Remalhart, Hinton, and Williams published their method in 1986 [21]. Backprop is an algorithm that performs gradient descent efficiently through a "forward-pass" calculating partial derivatives of neuron functions and a "backward-pass" reusing calculated partial derivatives to create a cost gradient.

ML would not be possible without large datasets. A revolutionary dataset used for ML is the MNIST database [22]. MNIST is a database that maps images of handwritten digits to the correct digits represented in the image. Several models using a wide range of ML algorithms such as convolutional neural networks (CNN) [23], deep neural networks (DNN) [24], and linear classifiers [25] have been applied. Another famous database that sparked widespread interest in CNNs was ImageNet [26]. ImageNet is a database containing millions of images with features that mark the objects displayed in each image. Many ML models have been proposed to great effect such as AlexNet [27] which uses two graphics processing units (GPU) to train a CNN within top-5 error (the error attributed to the amount of times the correct label appears in the list of top 5 predictions) of 15.3% and a deep residual learning network [28] that improved the error to within 3.57%.

ML programming frameworks have been created to give researchers ease in creating models. Some such frameworks include TensorFlow [29], a Google, open-source library with multi-language support that provides automatic hardware selection and PyTorch [30], a Python library managed by Facebook that behaves similarly.

Many companies offer services that make use of ANNs, CNNs, DNNs, and various other xNNs but at the expense of user privacy. What if sending GPS coordinates to service providers encrypted can remain encrypted and yet allow the service to function? Similarly, what if there could be an encrypted database of images that can give similar results to AlexNet when used for training? Given the mathematical nature of neural networks and extensive research into its practicality, application and weaknesses, it is easy to see how FHE could play a role.

2.3 Secure Machine Learning

The crossroad between these two fields was investigated shortly after Gentry's 2009 paper. In 2011, Graepal, Lauter, and Naehrig from Microsoft Research demonstrated the possibility of SML on encrypted data [31]. They created an algorithm using LFHE to perform binary classification with gradient descent. However, the early stages of FHE at the time limited the types of activation functions to just a few effective ones and therefore revealed the inefficiency of SML. In 2016, FHE was applied to neural networks. In a paper titled, "CryptoNets", researchers from Microsoft Research applied a LFHE scheme to a neural network operating on the MNIST database [32]. While they achieved a 99% accuracy rate, it took 570 seconds to complete one prediction, most of which can be attributed to slow multiplication operations but still extremely inefficient when compared to the milliseconds it takes to complete predictions over nonencrypted data. Logistic regression was applied to encrypted data in 2018 with a pretty low efficiency but demonstrated a medical use case by training on encrypted genomic sequences to infer an encrypted prediction [33].

The commonality between the above methods has been their inefficiency. With new approximate schemes such as CKKS, precision could be traded for efficiency. "Faster CryptoNets" used approximate activation functions and a fixed-point LFHE scheme to accelerate predictions on the MNIST database [3]. CKKS was used in a financial use case by researchers at IBM to further accelerate prediction on encrypted financial datasets [34]. What is being called the "AlexNet Moment for Homomorphic Encryption", HCNN [35] trained on the encrypted MNIST dataset and achieved a 99% accuracy rate with speeds of 14.105 seconds to classify the entire dataset, a significant improvement over previous examples. The model is a fully homomorphic convolutional network operating on a set of GPUs.

With advancements in the area of SML come new programming libraries that combine FHE and ML effortlessly. A SML version of TensorFlow, TFEncrypted [36], a Python library, provides ease and flexibility with creating NNs and other ML algorithms across various FHE schemes.

Considering all of this research, it is evident that there exist some obstacles to accelerating SML inference, some including expensive activation functions, inefficient computer architecture, and the FHE schemes themselves. Multiple papers have created their own FHE or LFHE schemes as a result.

3 Methods and Results

As mentioned in the introduction, this paper focuses on encrypting a data operand from a mobile device, sending it to the cloud, and having a SML algorithm train and send back an encrypted prediction. Therefore, two project phases were considered: 1) setting up the environment for testing, and 2) creating and implementing a SML algorithm to train and predict encrypted values.

3.1 Phase 1: Setting up the environment

The first aspect was deciding what type of mobile device to use. There are two major operating systems: iOS and Android. iOS uses a native Objective-C environment while Android uses a native Java environment. The challenge, however, was to decide which one to use for the specific task of encrypting and decrypting operands homomorphically. The Android system is significantly more open than the iOS system. In this case, *open* means the ability to call other applications within the same operating system and to use services such as location services, etc. much more easily. This helped make the decision to use Android as an operating system clearer. While homomorphic encryption can be done on both iOS and Android, Android, for testing purposes, was the easiest path towards completing the experiment more efficiently.

The test device for the application was a Samsung Galaxy S9, SM-G960W with a 64bit, 2.8Ghz Quad-Core Kryo 385 Gold and 4GB LPDDR4X RAM running Android OS 9. The mobile device supports location services.

It was decided to use GPS coordinates as the data operand. GPS coordinates are a good representation of user private data. The ultimate goal with FHE is keeping private information private which is why GPS coordinates were the best selection. With GPS coordinates, there can be many applications. Some applications include training an algorithm that detects credit card fraud. GPS coordinates act as one of the features in a credit card fraud dataset. For example, a fraudster uses a stolen credit card number in Mexico to fuel up their car. Visa, the credit card company, notices that the credit card was used 1 minute before that purchase in Ottawa, Canada by the actual credit card holder. The transaction is flagged, and the credit card is locked. Location of payment therefore offers an input operand that allows a risk parameter to be calculated that credit card companies can use to determine if a transaction is fraudulent. In theory, all of the other features used to predict a fraudulent transaction could be homomorphically encrypted such as name, address, etc. of the true credit card holder and taken from multiple companies that cannot share information with one another to create a better learning algorithm for detecting credit card fraud.

The cloud computing platform to be used was also considered. There are several companies that offer cloud computing services such as Amazon, Microsoft, and Google. From this list, Amazon offers the most dynamic, user-friendly and resource abundant service. A full solution can be

created including virtual machines, lambda functions, databases, and API gateways very easily. A lambda function is a generic function that when passed data, performs an operation, returns a result and closes, all without having to establish a set programming language interface between both the caller and the callee. The free tier offers an abundance of resources for students that meet most needs without having to pay a fee. The variety of tutorials both on the Amazon website and Stack Overflow is incredible. As a result, Amazon Web Services (AWS) was chosen as the cloud computing platform for this paper.

To set up the environment for performing and testing a machine learning algorithm with data from a mobile device, the following workflow was realized. A GPS coordinate would be homomorphically encrypted from an Android device. An API gateway would need to be created on an AWS virtual computer that would attach to a lambda function. Once the Android phone calls the API gateway and sends the encrypted GPS coordinates, the API gateway would call a lambda function and pass through the encrypted GPS coordinates. The lambda function would then perform a prediction on the data and send the encrypted result back to the mobile device through the API gateway.

3.1.1 Creating the Application

The first step after deciding on the development environment used was to homomorphically encrypt a data operand on an Android phone. The FHE library chosen was Microsoft SEAL [6]. After reviewing the other libraries mentioned above, SEAL was the most versatile and stable library. It has the most contributors and is being actively maintained by Microsoft Research. Unfortunately, due to time constraints, there was no quantitative testing to show why SEAL was used instead of the other libraries, however, it is planned to be done in the future. SEAL version 3.4 was chosen due to speed improvements.

SEAL is written in C++. While there are wrappers in other languages, there are none in Java. Therefore, it was necessary to create a Java wrapper for the main functions required. Using JNI. Java Native Interface, a wrapper was created for setting parameters, obtaining private and public keys, encrypting and decrypting, all using SEAL 3.4 in Release mode. Release mode was used for production level operational speed. Debug mode is considerably slower. An example of the Set Parameters and Encrypt functions in the wrapper code can be found in Appendix A, Snippet 1. The full wrapper code can be found on GitHub. All of the functions create and return objects according to their function name. For example, Set Parameters sets the parameters of the encryption and returns a Base64 encoding of the parameters for loading into future function requests. The Set Parameters function sets the scheme as CKKS (since our SML algorithm will be using approximate arithmetic), the poly modulus degree as 4096 (the amount of indices in the plaintext array), and the coefficient modulus as the recommended value for the given poly modulus degree. These are recommended parameter values from the official SEAL CKKS examples. Encrypt encrypts a double vector and returns a Base64 encoding of the ciphertext to be used for loading in other functions in the future. Base64 was selected as an encoding for

transferring data seamlessly and without error across different functions and platforms. It is an industry standard encoding that fits this use case scenario.

The application has multiple functions along with the main function to send the GPS coordinates. Figure 8 in Appendix B shows the design of the application. The SEND LOCATION button encrypts the device's GPS coordinates and sends them to AWS via an HTTP POST request in the form of a JSON. The form of the JSON is as follows: { params: string, key: string, latitude: string, longitude: string }. The "key" value in the JSON refers to the private key and is only included for testing purposes. It will not be included in the actual implementation. The three number inputs are for testing purposes. The user can input 3 decimals and then press ENCRYPT or DECRYPT to perform a homomorphic encryption and subsequent decryption locally. The CALCULATE EDPS button calculates the amount of encrypts then immediate decrypts of the inputted decimal values per second. EDPS stands for Encrypts Decrypts Per Second. This allows a means of benchmarking different EDPS speeds by device. The following table shows the EDPS speeds per device.

Table 1: Encrypt Decrypts Per Second (EDPS) Speeds per Device

Device	**Processor**	**RAM**	**Operating System**	**EDPS**
MSI GE62 2QF Apache Pro	Intel Core I7 5700HQ CPU @ 2.7GHz	16.0 GB	Microsoft Windows 10	80.1974
Samsung Galaxy S9 SM-G960W	Quad-Core Kryo 385 Gold @ 2.8GHz	4.0 GB	Android OS 9	15.2462
Samsung Galaxy S8 SM-G950W	Quad-Core Kryo @ 2.35GHz	4.0 GB	Android OS 9	12.1829

The table above demonstrates how hardware affects EDPS speeds when SEAL is used. The first device is a gaming PC with significantly better hardware than the remaining devices which are mobile phones. The EDPS value associated with it is 5x more than the other devices and their EDPS values. This table also shows how a phone with better hardware than another running on the same operating system runs SEAL faster and is therefore more efficient. This result, however, is a little misleading.

While the EDPS speeds are important for benchmarking the performance of SEAL on different devices, they do not affect the ultimate speed of the SML algorithm that will be used. What is important about the EDPS speeds is that they are a significant increase from the first version of the code. The first version of the code had EDPS speeds at 0.3, an order of magnitude below the current speeds. The reason for that was the Context class in SEAL. The Context class includes all parameters and is a vital class for creating other classes such as PrivateKey, Encryptor, Encoder, Decoder, Decrypter, etc. The Context class is similar to Context classes in other open source cryptographic libraries like OpenSSH. In the first version of the application, the wrapper was creating a context variable on every encrypt and decrypt. This delayed the operation substantially. For the current version of the application, the context class is created only once

when Set Parameters is called and remains for the lifetime of the application, increasing EDPS speeds by an order of magnitude.

The GET GPS LOCATION button creates a listener in android that updates the internal coordinate location variables every 5 seconds. The two remaining buttons encrypt and respectfully decrypt the GPS coordinates.

Another important feature for the application is how it saves the keys and parameters for later use. In a most recent version, the application checks whether a file containing the private key, public key, and parameter values exists locally. If it does, it imports the values and if it does not, it creates the values and saves them to a file. The file is kept in the application storage space which cannot be deleted by closing the application or read by other applications without permission. It can only be deleted if the application is deleted from the device. This functionality allows the device to retain its keys and parameters for the duration of the applications existence on the device. The application can therefore encrypt and decrypt data without having to create new keys every time, similar to how data is encrypted and decrypted from devices non-homomorphically. Snippet 2 in Appendix B shows the above functionality. The full application code can be found on GitHub.

3.1.2 Creating the Lambda Function, API Gateway, and Database

The next step in the workflow was creating the server endpoint where the GPS coordinates would be sent. For this, an AWS virtual computer was created. The virtual computer (VC) used was the only one available for Free Tier. The specifications are as follows: Amazon Linux 2 AMI 2.0.20191217.0 operating system with one 64-bit processor and 1.0 GB of RAM.

There were a couple solutions for creating a server. The first solution was writing a server with endpoints matching port numbers of the VC. Some drawbacks of this solution included the constant monitoring of the server to make sure it is running, and the amount of work required to write a server in C++. The ideal language to write a server was decided to be C++ since SEAL is coded in C++. The amount of work and maintenance required was significantly more than the second solution. The second solution was to create a lambda function. Lambda functions are a revolution in serverless technology. With lambda functions, there is no maintenance required and computational resources are scalable as provided by AWS. This allows for faster websites and backend processes. However, there was a barrier standing in the way of using a lambda function – the unsupported C++ runtime. AWS does not support C++ as a language for lambda functions. It does, however, support it in beta mode. This means that AWS has created a C++ Software Development Kit (SDK) but not released it as a production-ready SDK. The SDK had some examples and documentation of how to program with it. A lot of time was spent trying to understand and program simple lambda functions in C++. Nevertheless, after much work, it was decided to use lambda with AWS C++ SDK technology for backend processes.

After choosing the server technology, it was necessary to attach an API gateway to the lambda function, in other words, a controller that launches the lambda function in response to an incoming HTTP POST request. AWS has a simple way of doing that. In the AWS Graphic User Interface (GUI), an API gateway was created and URL attached to the gateway for network requests. For testing purposes, a firewall, API key, and authorization settings were not added which allowed any incoming connection. A simple POST request template was created and the lambda function was attached to the gateway. The POST request was designed to receive *application/json* data from incoming requests and call the lambda function with the data as its input.

Since the application used SEAL version 3.4, it was necessary to use SEAL version 3.4 in the lambda function. After setting up the environment, the only remaining question was how can it be tested? What kind of lambda function can be created to test its functionality? The result was to create a lambda function that parses the received, encrypted coordinates, creates an entry in a database for each coordinate pair, decrypts the encrypted coordinates, and returns the decrypted coordinates to the testing application to be verified.

A DynamoDB (DDB) database was created. DDB is an AWS hosted NoSQL database. It is a fully distributed database designed for failure. The structure is not like any other database since the columns in its tables are not well defined. DDB stores mainly primary keys/columns that serve as unique identifiers for each row which is all it does (unless you designate secondary keys/columns for sorting). The names of the remaining columns are created when data is inserted. Since data is inserted as a JSON, the keys in the JSON datagram are parsed as columns in the tables. Values in the tables are also very arbitrary with no defined lengths or datatypes. Values can be strings, numbers, lists, JSONs, and some other general datatypes. Strings can be of any length but no larger than 400KB.

For the test application, four tables were created. Each table shares the same primary key – a unique number (in this case, the time of the row entry since epoch in milliseconds) that connects the tables together. The four tables are for SEAL parameters, secret keys, latitudes, and longitudes, all with Base64 encoded string values (the reason why it is not one table is because the values from the mobile application together would be larger than 400KB per row). Considering the tables as one, each row entry contains the data sent by the application (in production, secret keys should not be a part of tables but for testing purposes and for mainly attributing the key to the encrypted value, they were added).

The lambda function therefore creates a connection to the database every time it is called. The data from the mobile application is first parsed, checked, and then prepared for entry into the database. Using a batch insert, four inserts are performed at once to four different tables with the same primary key. The GPS coordinates are then decrypted and returned to be sent back to the application for verification. The lambda function code is available on GitHub. Appendix C contains some relevant code snippets that may aid in the above explanation.

In conclusion, the testing environment was created on both the client and server side and tested. The server successfully adds values to the database and the client successfully verifies the data it sends to the server. The environment is ready to start running a SML algorithm.

3.1.3 Creating the Map Demo

To provide a proof of concept and truly show the power of Fully Homomorphic Encryption in our everyday lives, a demo was created to showcase how encrypted location tracking would work. With the testing environment complete, it was simple to design a workflow and implement. The workflow was as follows: 1), retrieve encrypted location coordinates from cloud, 2) decrypt the coordinates on the mobile device, 3) launch a map with the coordinates as markers showing where the mobile device holder went while doing the experiment.

To get the coordinates from the cloud, another lambda function was created that would query the four tables, put the data together, and send it back to the data owner. The lambda function was coded in Javascript + NodeJS. There was no need this time to code it in C++ since none of the functionality required libraries only written in C++. It was necessary to create a fifth table to hold timestamps and device IDs. Since our implementation featured the ability for many devices to use the application at once, each application required a distinct ID to distinguish between each one. The device ID is created by the Android application at startup and is kept on the device until the application is deleted. This ensures that the device owner can launch and use the application however they please without any interruptions. The device ID follows the RFC 4122 [37] standard. This standard describes a namespace for UUID's which industry follows.

The lambda function performs a query and gathers the data. However, the problem with sending back the data was that the data, in total, exceeded 6 MB in size. Lambda functions can only retrieve and return data up to 6 MB in size. Our data, of about 50 or so GPS coordinate pairs, was upwards of 20 MB. The solution was to save the data in the cloud and return a link to download the data from the cloud. AWS has a service called S3. S3 provides storage containers for data called "buckets". Buckets can contain any data one needs. In our case, we used S3 to contain the data from the lambda function. The lambda function uploads the location coordinates data to S3 as a JSON. It then assigns a 60 second valid link that the client could use to download the data. It then returns the link to the client. The client gets the link and performs another request. The client downloads the data as a JSON and the client now has the coordinates in encrypted form on their device.

Using the SEAL Java Wrapper described in section 3.1.1, the list of coordinates is decrypted into a list of unencrypted coordinates. The list is then passed to a Map activity that transforms the coordinates into markers. The Map library chosen was Google Maps since it is native to Android, owned by Google, and because there are a vast number of tutorials and demos that simplified the implementation process. The Android map code can be found in the onMap function on Github. Parts of the onMap function are shown in snippet 3, Appendix B. Appendix

B, Figure 9 shows the final result of the demo. It shows a mobile device user with the Homomorphic Encryption app walking from Carbonic Coffee in Toronto, ON, Canada to the University of Toronto and subsequently around the campus. The experiment was conducted by having the user press SEND LOCATION every couple of seconds during their walk. The coordinates were stored in the cloud encrypted and retrieved by the user at the end of the experiment, still in encrypted form, by pressing the LAUNCH MAP button. The LAUNCH MAP button seen in Figure 10, Appendix B performs the workflow described in this section.

Using the device ids as keys, we also added the possibility of downloading two sets of data onto one device. Since the secret key is included in the returned data, the two sets of data can be decrypted on the same device into two different sets of decrypted coordinates. The two different devices' locations can therefore be displayed on one map with different colored markers.

This presents a remarkable advance in privacy. Now, it is practical for devices to send their encrypted GPS coordinates or other sensitive information to an unsecure cloud where they will remain encrypted with the ability for evaluation to be performed on them. One of these applications could be sending encrypted credit card information with transactions to a cloud to determine whether the transaction is fraudulent or not. The next section details the second half of this thesis, a secure machine learning algorithm for detecting fraudulent credit card transactions.

3.2 Phase 2: Creating and Implementing the SML Inference Algorithm

3.2.1 Creating the Secure Machine Learning Inference Algorithm

3.2.1.a Selecting the Dataset

The first step in performing any machine learning process is to train on a dataset. In our case, since we chose to predict fraudulent credit card transactions, we needed to find a credit card transaction dataset. Luckily, there is a dataset on Kaggle of the kind. The Credit Card Fraud Detection dataset [38] is a collection of European transactions performed in September, 2013. There are 492 fraudulent transactions out of 284,807 total transactions making the dataset heavily imbalanced. The data features are confidential and therefore a set of 28 features extracted from a Principal Component Analysis (PCA). A PCA is an analysis tool that converts a dataset into principal components, axes that are orthogonal to each other with respect to variance [39]. This reduces the dataset to independent features. This ensures that the features in our dataset have no meaningful classification (are confidential) and can only be distinguished by their orthogonality. There are also "time since first transaction" and "transaction amount" features which are nonconfidential. The labels for the dataset are a simple 1 or 0 with a positive classification being 1 and a negative classification being 0.

3.2.1.b Preprocessing the Dataset

The main research problem after selecting the dataset was how to classify fraudulent transactions based on a heavily imbalanced dataset? With imbalanced datasets, a machine learning model can easily overfit on the data that are in abundance since most iterations (or gradient descent steps) will be based on batches of data that may include only one positive datapoint or none at all. This skews the final gradient towards, in our case, negative classifications. Therefore, machine learning algorithms based on imbalanced datasets need to use their data more wisely.

After some research, we noticed that Google provides a TensorFlow tutorial on classifying with imbalanced datasets [40]. The tutorial uses the same dataset that we are using as the basis for its model. The tutorial goes through some data preprocessing to try and balance the dataset. The main technique that it uses and the one that gives the highest precision is oversampling the dataset. Oversampling the dataset means replicating rows within the dataset multiple times over to try and balance the dataset. In our case, the tutorial oversamples the positive samples so that the likelihood of them appearing in each batch increases and therefore the prediction precision increases. Although the ideal case would be to have more unique positive samples, the reality of credit card fraud is that rates are low but damage could be high. This means that although the amount of fraudulent transactions is a fraction of 1% of the total volume of transactions, the damage that individual fraudulent transactions can do can be massive.

We chose to follow the tutorial's advice and oversample our data. We chose not to include the time column since it was not explained by the dataset owner what the data means. We normalized the data and clipped it between -1 and 1. Clipping means making every datapoint

outside the clipping interval equal to its nearest bound. For example, the number 2 would take on the value of 1 and the number -10 would take on the value of -1. This was justified since the data points after normalization took on a mean of 0 and unit variance. Because of this, most of the data was already within the [-1, 1] interval and thus clipping did not negatively impact our model's strength. After splitting our dataset into training, validation, and test, and after oversampling, our train dataset grew to 363,942 data points from 182,276 data points, exactly double the size of initial negative samples, 181,966, making the ratio of positive to negative samples 1:1, and thus the mean of each batch approximately 0.5. The dataset was balanced, and training could begin.

3.2.1.c Defining the Model

There are two avenues for secure machine learning training. One could perform encrypted training or plaintext training.

Encrypted training is training on encrypted data. This means that all weight matrices and bias vectors along with the data are encrypted during the training of a model. This results in encrypted weight matrices and bias vectors that can be used to perform predictions on encrypted data, yielding encrypted predictions that an end-user can decrypt. A positive aspect of encrypted training is that every part of training is encrypted and therefore highly secure. A negative aspect is time. It takes a lot of time to perform encrypted training because of the time it takes SEAL to perform fundamental ciphertext-ciphertext operations.

Plaintext training is training on unencrypted data and inference on encrypted data. The model trains on unencrypted data and updates unencrypted weight matrices and bias vectors. The weight matrices and bias vectors are encoded into plaintexts which are then used for addition, multiplication, and rotation with ciphertext test data to yield encrypted predictions. SEAL rotation is similar to left-shift and right-shift bit operations in digital systems, except, in this case, a cyclic rotation, that could be both to the right or left, of the ciphertext members [2].

Simply, a model can be trained on unencrypted data using ML libraries such as TensorFlow, parameters can be extracted and used in an FHE inference step to create encrypted predictions. A positive aspect of this approach is time. It is much faster to train on unencrypted data than encrypted data. A negative aspect is the need for unencrypted data to train on. For example, a hospital with confidential information would need to find a way to send an external ML company unencrypted data that can be used for training. They could do a PCA like our dataset. Because the goal of this thesis is to find a way of making faster, approximate predictions and not faster training, we decided to take the second avenue. We decided to use TensorFlow to train our unencrypted data and extract the weight matrices and bias vectors to use for encrypted classification.

The artificial neural network (ANN) model we used is slightly different from the one in the tutorial. Figure 1 below shows our model.

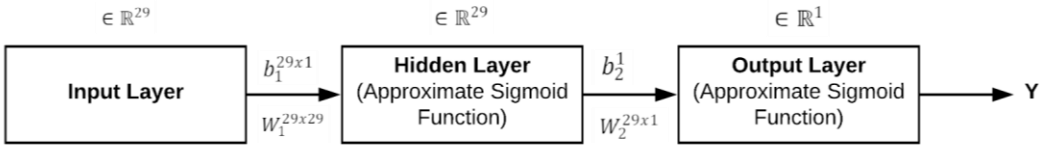


Figure 1: Artificial Neural Network (ANN) Model

This model contains one hidden layer of 29 units. 29 units ensures a square 29x29 weight matrix. We had to use 29 to aid in the implementation of our FHE inference step algorithms discussed in section 3.2.4.a. The 29 units feed-forward through an activation function. In the tutorial model, the Rectified Linear Unit (ReLU) function is used for the hidden layer. Here, we have chosen to use a Sigmoid function. The following is the Sigmoid function:

$$\sigma(x) = \frac{1}{1+e^{-x}} \quad (1)$$

Because we are replicating this model in SEAL for encrypted classification, we chose to use an approximate, polynomial sigmoid function. FHE evaluation consists of three fundamental mathematical operations: addition, multiplication, and rotation. It is therefore impossible to do pure division or pure exponentiation. Since the Sigmoid function contains division and exponentiation, FHE cannot purely implement it. However, there exist polynomial approximations that can approximate the Sigmoid to within an interval.

The model then feeds into an output layer that contains 1 unit. The output layer neuron performs the same approximate Sigmoid function evaluation and outputs a positive or negative class prediction. In plaintext training, the prediction is unencrypted, later, in the inference process, it is encrypted.

3.2.2 The Approximate Activation Function

Research into approximations of the Sigmoid function is vast with many applications. One reason for finding good approximations of the Sigmoid function is efficiency. Because the sigmoid function is an infinite polynomial, it is hard for digital systems to compute outputs efficiently [41]. The problem lies with the "exp" function within the Sigmoid. Many papers have outlined different ways of approximating the Sigmoid to help boost performance ([41], [42], [43], [44]).

One approximation could be piecewise. A piecewise approximation consists of sets of intervals over which different functions operate and overall give an approximate Sigmoid output [41]. Piecewise approximations, while efficient, are impossible to implement on encrypted values. The piecewise approach includes if/else checks to determine in which interval the operand lies.

This means that the program must know the value of the operand to compute its output which goes against the very nature of encrypted computation.

Another approach is to use predetermined tables of the Sigmoid function [44]. Predetermined tables offer the ability of looking up values based on operands. However, again, the algorithm must know what "x" is to look-up it's value. In the same way, we cannot use this approach as it only works on unencrypted inference.

The approach we decided to take was using a polynomial approximation of the Sigmoid function. Since SEAL is able to perform polynomial calculations, this approach gained momentum. There exist a class of polynomials called Chebyshev polynomials. Chebyshev polynomials have the following definition [45]:

$$T_n(cos(x)) = cos\ (nx) \tag{2}$$

$$\begin{cases} T_0(x) = 1 \\ T_1(x) = x \\ T_{n+1} = 2xT_n(x) - T_{n-1}(x) \end{cases} \tag{3}$$

Since the Sigmoid is trigonometric in nature and Riemann integrable [43], it can be approximated by a class of Chebyshev polynomials. A paper published in 2012 by Miroslav Viček [43] provides an algorithm for calculating the coefficients of a Chebyshev polynomial series that fits the Sigmoid function to within 2% error. Using this intuition, we found that Python's scientific toolkit, NumPy [46], has an automatic Chebyshev polynomial fitting class. This class fits a Chebyshev polynomial to an inputted function. Inputting the Sigmoid function, we extracted coefficients of a degree-15 Chebyshev polynomial over an interval of [-100, 100]. The following graph shows the extracted polynomial:

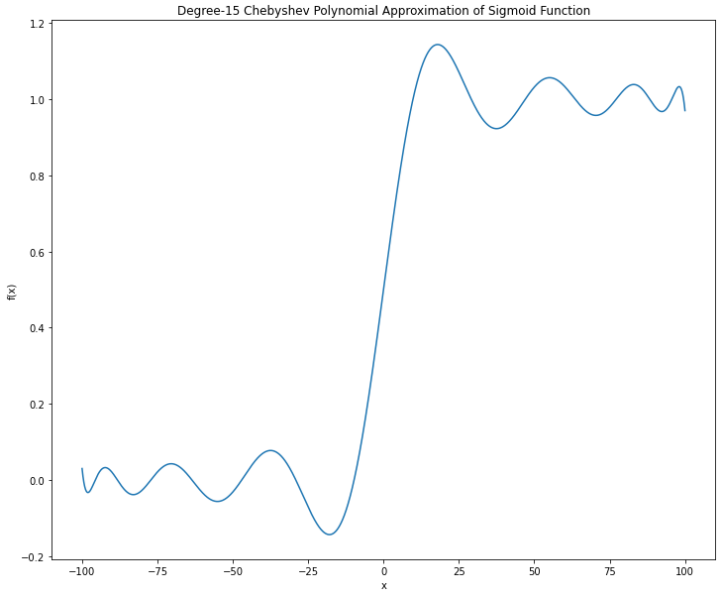


Figure 2: Degree-15 Chebyshev Polynomial Approximation of Sigmoid Function

The polynomial was chosen to be degree-15 as it was the closest to the Sigmoid function without exponentially increasing the degree. A degree-15 polynomial also offers a compromise between accuracy and computational efficiency. The polynomial that we used was the same one as above but rescaled to make all of the values fall within [0, 1] (shifted up by 0.15 and squeezed by a factor of 0.76). The coefficients with degrees less than 10^{-20} were also made zero due to their great precision that SEAL would also just encode as 0. The final equation was as follows:

$$f(x) = 0.494 + 0.18x - 4.536 * 10^{-2}x^3 + 5.489 * 10^{-5}x^5 - 3.378 * 10^{-7}x^7 + 1.141 * 10^{-9}x^9 - 2.145 * 10^{-12}x^{11} + 2.104 * 10^{-15}x^{13} - 8.395 * 10^{-19}x^{15} \quad (4)$$

3.2.3 Training and Results

After preprocessing the dataset and defining the approximate polynomial, it was time to start training our ANN model shown in Figure 1, section 3.2.1.c. We followed the same procedure as the tutorial. To further optimize our model, we added an Adam optimizer which optimizes the learning rate at each step [47]. We also incorporated Early Stopping to make sure the model stops training before it overfits [48]. We used the Binary Cross Entropy function as our loss function. Note that this function contains operators such as "log" that have no pure equivalent in SEAL but since this is only used during training and not classification, it can be used in our case. We set the number of epochs to 1000 and batch size proportional to 20 steps per gradient descent step (length of dataset / 20).

The training was performed on Google Colaboratory [49], a free machine learning platform that runs python notebooks. The model took about 30 seconds to train and stopped early at epoch 67/1000. The code used a Python 3 runtime and ran on a standard CPU with 12.72 GB of RAM. The python notebook file can be found on Github and screenshots in Appendix D with all results.

After training, we observed the following results:

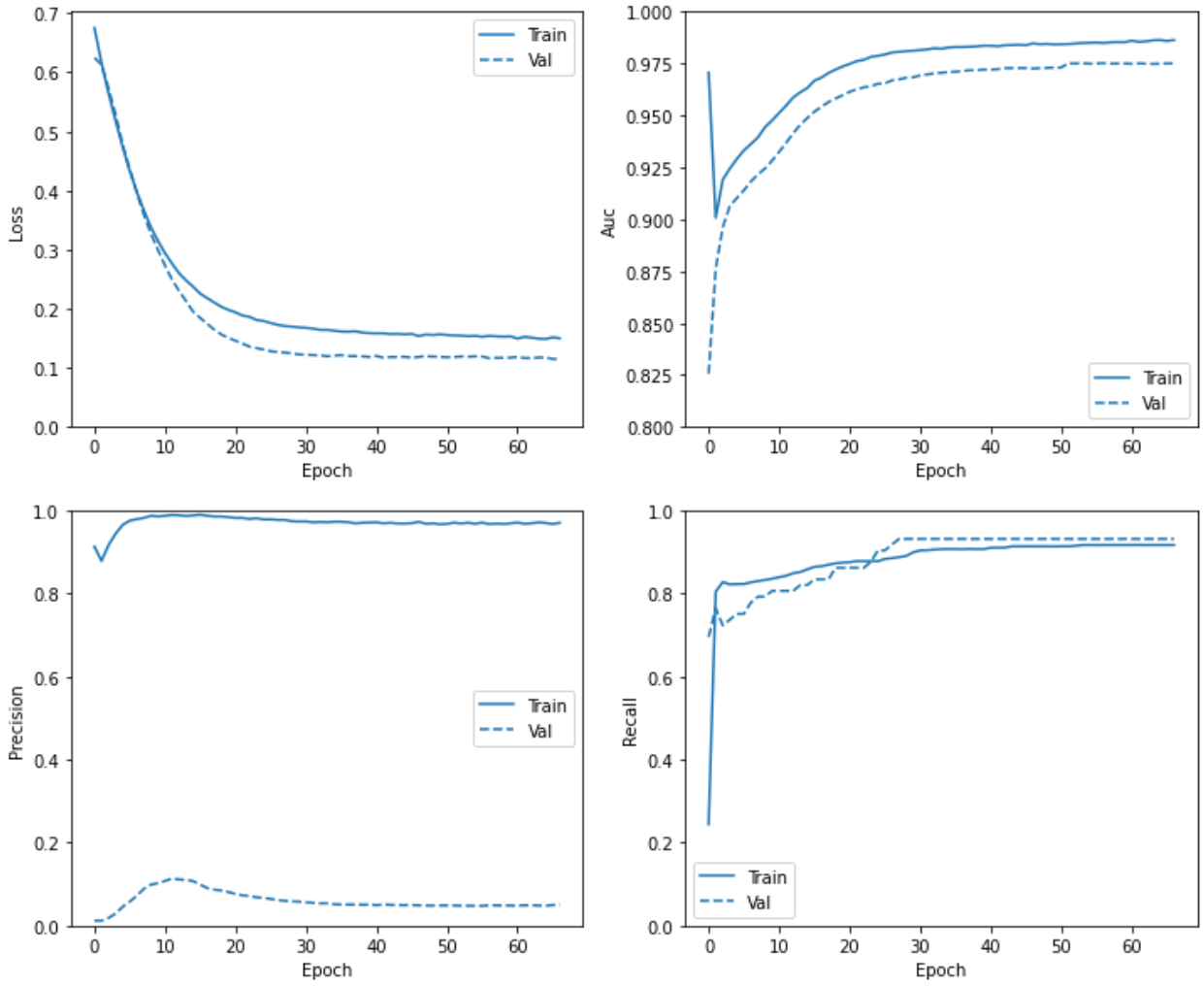


Figure 3: Loss, AUC, Precision, and Recall Plots of Training

The observed results were promising. As seen in Figure 3, our training loss fell, our precision and recall increased to almost 1. Our validation set precision, however, was not promising but below, we will see the reason behind that. The following table shows the test set results:

Table 2: Test Results of ANN

Metric	Value
Loss	0.1165
True Positives	99
False Positives	1705
True Negatives	55146
False Negatives	12
Accuracy	0.97
Precision	0.055
Recall	0.89
AUC	0.98

To better understand the above results, the following is a short summary:

Legitimate Transactions Detected (True Negatives): 55146

Legitimate Transactions Incorrectly Detected (False Positives): 1705

Fraudulent Transactions Missed (False Negatives): 12

Fraudulent Transactions Detected (True Positives): 99

A true positive and false positive is a prediction of 1 that is true if it matches the label and false if it does not. A true negative and false negative is a prediction of 0 that follows the same logic. Accuracy is defined as the amount of correct predictions over the total amount of predictions. Precision is the percentage of correctly classified fraudulent transactions over all transactions classified as fraudulent (true positives / (true positives + false positives)). Recall is the percentage of transactions correctly classified as fraudulent over all actual fraudulent transactions (true positives / (true positives + false negatives)). AUC is the area under the curve, the curve being false positives versus true positives. The AUC is the probability that a classifier will rank a positive sample higher than a negative sample [40].

Our test loss fell dramatically to around 0.11 and thus achieved a test accuracy of about **97%**. While accuracy is important, we were more interested in the other metrics. Recall improved to about 0.89 and the AUC improved to 0.98. The precision was rather low but is alright. The reason why this is not a problem is because it is better to have non-fraudulent transactions incorrectly classified rather than fraudulent transactions missed. This is why recall is more important than precision. Recall shows us that almost all of the fraudulent transactions were caught and correctly classified as fraudulent. Precision, however, shows us that 1,705 people will be frustrated by being falsely accused of fraud which, arguably, is better than losing millions as a result of fraud. The tradeoff is simple: lower user happiness for dramatically less financial loss. However, user happiness is not impacted that much considering nearly 55,146 transactions were correctly classified as non-fraudulent. Negatively impacted users could also be turned into their own dataset where a more expensive model with higher predictive power could be run to

help re-classify them as non-fraudulent. The results therefore proved our model a success and a viable contender for fraud detection.

After confirming our results, the weight matrices and bias vectors were extracted from the model and saved to CSV files. The parameters were now ready to be imported into SEAL and be used for encrypted inference.

3.2.4 FHE Inference Using SEAL

3.2.4.a Choosing the Encryption Parameters and Algorithms

A Visual Studio 2019 [50] project was created to implement the encrypted classification. With any SEAL implementation, the first step is to choose the encryption parameters. With the parameters, one can go ahead and extract all necessary encryption keys and begin evaluation. Choosing the right parameters depends on the types of fundamental operations used, which effectively means the types of algorithms used. The main choice involves a poly modulus degree and coefficient modulus degree. Note that "ciphertext" and "plaintext" will be abbreviated by "Ctxt" and "Ptxt" for the remainder of this thesis.

As described in section 3.1.1, the poly modulus degree is the number of indices in a Ptxt array. The poly modulus degree determines the maximum, total, bit size amount of the coefficient modulus primes. The coefficient modulus is a list of prime numbers, with specific sizes in bits, that corresponds to the coefficients of the Ptxt polynomial. For example, the sum of the sizes in bits of the prime numbers in the coefficient modulus must be less than 411 for a poly modulus degree of 16,384 to be considered quantum safe. Similarly, the sum of the sizes in bits of the prime numbers in the coefficient modulus must be less than 827 for a poly modulus degree of 32,768 to be quantum safe. Quantum safe refers to how safe an encryption is if a theoretical quantum computer were to mount an attack. The higher the total size in bits of the moduli in the coefficient modulus array and, therefore, the higher the poly modulus degree, the more time it takes to run FHE operations with SEAL.

Another important parameter is the scale. The scale determines the level of precision SEAL applies to encoded values. The scale is a large power of two, usually on the order of greater than 2^{30}. The larger the scale, the greater the precision when decoding. The scale must be equal to the central primes' sizes in the coefficient modulus. The first and the last prime in the coefficient modulus are not used in evaluation. They are used for scaling and precision. This means that in a coefficient modulus of length 13, only 11 primes are used for evaluation and their bit sizes must be equal to the bit size of the scale.

The effective length of the coefficient modulus limits the multiplicative depth of the whole FHE process. Multiplicative depth corresponds to the length of the largest chain of multiplications in the process. The longer the chain, the greater the depth. A large multiplicative depth must be avoided since every multiplication removes a prime from the coefficient modulus of the Ctxt

being evaluated. For example, two Ctxts, A and B, have coefficient modulus values of [60, 40, 60]. The resulting multiplication between the two has a coefficient modulus of [60, 60]. Once the length of the coefficient modulus of a Ctxt decreases to 2, the Ctxt cannot be further operated on. In addition, with every multiplication, Ptxt or Ctxt, the resulting Ctxt's scale is the multiplication of the two operands' scales. For example, a Ptxt with scale 2^{40} multiplied by a Ctxt of scale 2^{40} results in a Ctxt with scale 2^{80}. If the "rescale" operation is not applied to the Ctxt, its scale remains and the remaining noise budget decreases. As soon as the noise budget reaches 0, SEAL raises an exception and stops the program. Therefore, we must be careful in our choice of poly modulus degree, coefficient modulus, and scale.

The first step in deciding the values of these parameters was to calculate the multiplicative depth of our forward pass. To calculate the multiplicative depth, we must analyze how much depth each individual algorithm requires to operate. The following figure details the FHE inference step with its corresponding algorithms.

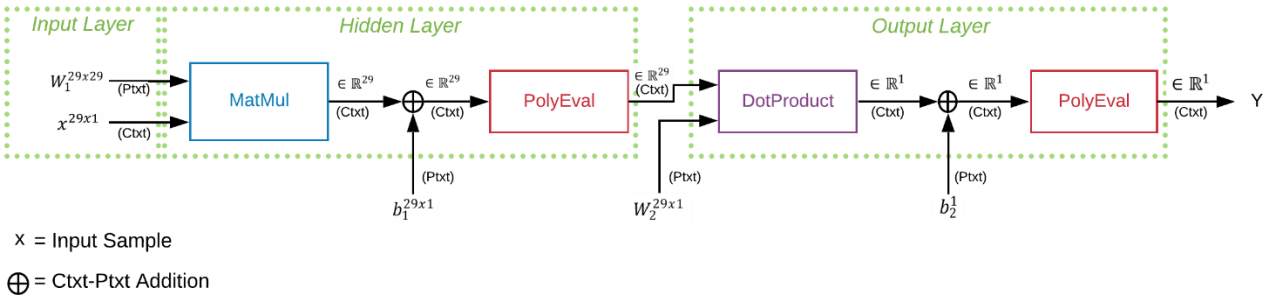


Figure 4: FHE Inference Step

The classification process reduces to three different algorithms: 1) matrix multiplication, 2) polynomial evaluation, and 3), dot product. "W" and "b" in Figure 4 refer to the extracted weights and biases. Y is the encrypted prediction.

MatMul

The matrix multiplication step is a FHE algorithm described by Halevi and Shoup [51]. The algorithm is a way of performing matrix multiplication of a square Ptxt matrix and one dimensional Ctxt. Since our weight matrix is square (29x29), we can apply this algorithm. This algorithm transforms the Ptxt matrix into *d* vectors of size *d* (*d* being the dimension of the Ptxt matrix). The vectors consist of the various diagonals of the matrix. The vectors are then linearly transformed by multiplying each one by the Ctxt, rotated. The following diagram was created by Microsoft Research.

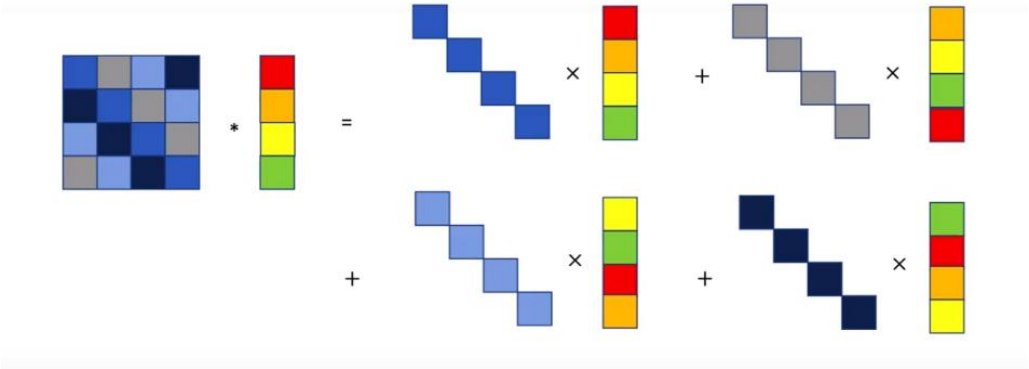


Figure 5: Matrix Multiplication of Ptxt and Ctxt [52]

The Ptxt matrix is the blue matrix and the Ctxt is the colourful vector. This diagram shows how the matrix diagonals are created and how the Ctxt gets rotated and multiplied at each turn.

This algorithm is O(d), meaning its performance is linear in dimension. It is optimized this way to reduce the amount of Ctxt-Ctxt multiplications. The following table, again from Microsoft Research, shows the performance of the various, fundamental SEAL operations.

Table 3: SEAL Fundamental Operation Performance [52]

Fundamental Operation	Complexity	Runtime (ms)
Ctxt Rotation (Generic/Single)	N log N	809 / 161
Ctxt – Ctxt Multiplication (including Relin and Rescale)	N log N	202
Ctxt – Ptxt Multiplication	N	3.7
Ctxt – Ctxt Addition	N	1.2

As seen in the table above, it is best to avoid as many Ctxt-Ctxt multiplications as possible. It is unknown on what system these results were gathered and with what parameters (most likely with a poly modulus degree of 16384), but the proportions serve as a loose performance benchmark. The table above mentions “Relin” and “Rescale” operations. Relinearization must occur after every Ctxt-Ctxt multiplication to reduce the size of the resulting Ctxt. If not performed, the noise budget decreases and eventually reaches 0, rendering the Ctxt useless [6]. Rescaling the Ctxt must also occur to reduce the resulting Ctxt’s scale back to normal [2]. There are also two types of rotations, generic and single. A generic rotation gives the possibility of rotating the ciphertext elements both to the right and to the left. It takes longer because the specific Galois keys (which must be created to perform any rotation) are much larger, accounting for both positive and negative directions. Single rotation gives the possibility of rotation in only a single direction. This means the Galois keys are smaller in size and therefore, rotation is more efficient [2]. In our case, we only use positive rotations (to the right) and therefore, single rotations.

The algorithm increases the multiplicative depth by only one which is good.

PolyEval

A polynomial evaluation algorithm was necessary to evaluate the result of the approximate Sigmoid function we created that was used during training. The PolyEval algorithm needed to be efficient and avoid many Ctxt-Ctxt multiplications. Because our function is a polynomial, this means that in order to get a function of degree 15, we would need to multiply our Ctxt 15 times in a worst-case scenario. This increases the multiplicative depth by 15 which is extreme and impossible. Luckily, there exist algorithms that decrease the multiplication depth from O(n) to O(log(n)). One such algorithm is the tree algorithm [52]. The tree algorithm first calculates the powers of the polynomial in a tree-like way, using previously computed values to compute later values. Once the powers are calculated, they are multiplied with their respective Ptxt coefficients and summed to complete the evaluation. The following figure from Microsoft Research shows the computation of the powers.

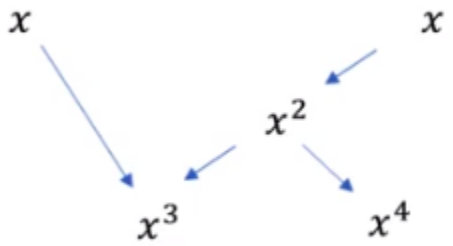


Figure 6: Tree-like Computation of the Powers of a Polynomial [52]

For a degree-15 polynomial, this algorithm increases the depth by 5. When performed twice, like in our model, it increases the depth by 10. For a degree-7 polynomial, it increases the depth by 4, and when performed twice, 8. As we can see, the depth is drastically reduced from O(n) to O(log(n)). The efficiency is better considering less Ctxt-Ctxt multiplications occur.

DotProductPlain

The next algorithm required was a dot product algorithm. The dot product algorithm was also borrowed from Microsoft Research [53]. The dot product operation between two vectors is an elementwise multiplication between two vectors and sum across resulting vector elements. While SEAL can perform a Ctxt-Ptxt multiplication, which is elementwise in nature, it cannot perform a sum over vector elements. The naïve way to perform a sum across vector elements is to take the sum of the resulting multiplied Ctxt over n rotations. The resulting Ctxt would have the dot-product value in its first index after decoding. This requires, in our case, 29 rotations which would drastically diminish the efficiency and increase the running time of our algorithm. The algorithm provided by Microsoft Research dynamically adds and rotates to decrease the complexity to O(log(n)) instead of O(n) where "n" is the length of the vector. The following figure shows how the algorithm works.

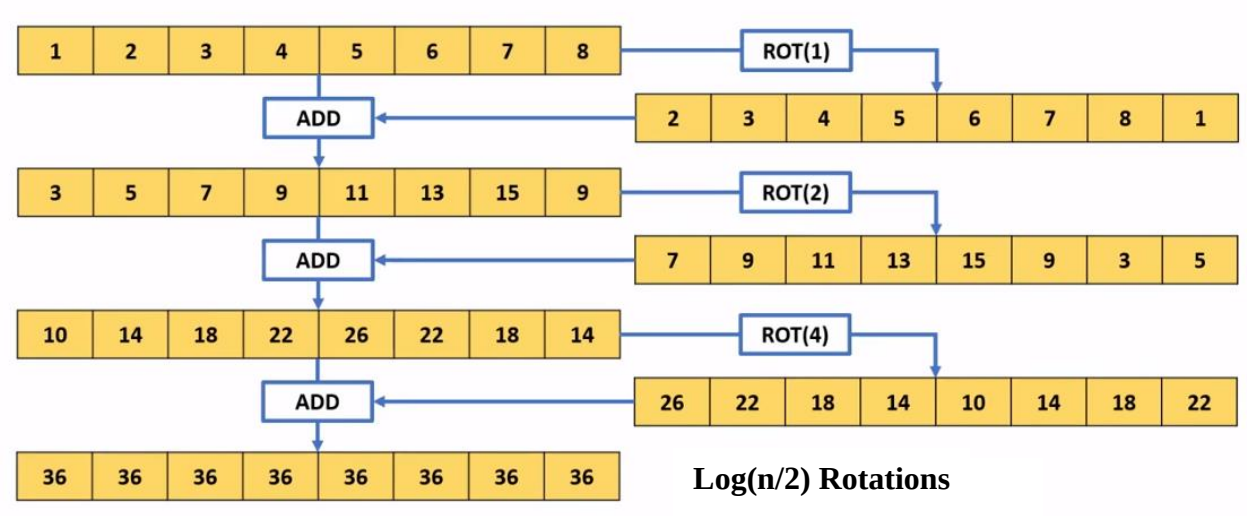


Figure 7: DotProductPlain Algorithm [53]

The algorithm is a series of n/2 rotations and additions. The ROT algorithm above is a rotation to the right, in the positive direction. Once the multiplication has completed, the Ctxt gets rotated by the next integer in the n/2 series and added to itself. After n/2 rotations, the result of the dot product lies in every position from indexes 0 to n-1 in the Ctxt. The algorithm increases the depth by 1 because of the initial multiplication.

After we defined all the algorithms, it was time to select the encryption parameters. Since the highest amount of precision in the polynomial was 10^{-19}, the scale had to be set to 2^{60}. Because the multiplicative depth totaled 13, the coefficient modulus had to have a length of 15 with the central 13 elements equal to the bit size of the scale, 60. The first and last elements had to be equal to 40 since less than that would decrease the precision exponentially. The sum of the elements of the coefficient modulus was therefore $2(40) + 13(60) = 860$. Since 860 > 411, the poly modulus degree had to be set to 32,768, the highest value. A value of 860 is also not post-quantum safe. 60 is also the highest power that the coefficient modulus can have as an element and that the scale can equal. The parameters were at their maximum values. This meant that the algorithm would be much less secure and efficient.

The optimization that we performed on these parameters was due to the following realization. We did not necessarily need a polynomial to evaluate our inputs at degrees of precision in the 10^{-19} range. The point of this thesis is to use approximation as means of efficiency to improve the practicality of FHE SML solutions. We decreased the scale from 2^{60} to 2^{40} which prompted SEAL to encode any polynomial coefficients less than 10^{-8} as 0, effectively making our degree-15 polynomial a degree-7 polynomial. Any coefficient after degree-7 would be encoded as 0. This decreased the depth of the PolyEval function from 5 to 4, and therefore the size of the coefficient modulus from 15 to 13. The poly modulus degree had to remain the same, but the sum of the elements of the coefficient modulus decreased from 860 to 520, therefore, ensuring the algorithm was post-quantum safe.

With the drastic decrease in parameters, our algorithm's efficiency improved drastically. After running our algorithm, we noticed a decrease in runtime for all sub-algorithms. The MatMul algorithm runtime decreased from 22 seconds to 11 seconds. The first PolyEval function runtime decreased from 5 seconds to 2 seconds. The runtime of one prediction decreased from 30 seconds to 13 seconds. The full algorithm code can be found in Appendix E and on Github as a Visual Studio 2019 solution. The following table is a summary of the final parameters.

Table 4: Final SEAL Parameters

Parameter	Value
Poly Modulus Degree	32,768
Coefficient Modulus	[40, 40, … , 40] size = 13
Scale	2^{40}

3.2.4.b Final Results and Analysis

To perform a test of the results, the dataset's 492 positive samples were concatenated with 492 random negative samples to give a total test dataset of size 984. This would prove the algorithm's ability to accurately predict both positive and negative samples. The same metrics as in section 3.2.3 were used to evaluate the model's performance. Inference was also conducted using the Ptxt Python model on the test dataset and extracted to compare the Ptxt model's predictions with the FHE model's predictions. The algorithm was run on an MSI GE62 Apache Pro laptop with an i7-5700HQ CPU @ 2.70 GHz. The amount of installed RAM was 16.0 GB. The laptop was running Windows 10, version 10.0.17763, build 17763. We compiled the code using Visual Studio 2019's default compiler: C/C++ Optimizing Compiler Version 19.22.27905 for x86. We used Visual Studio 2019's standard maximum speed optimization and built the program in x64 release mode using SEAL 3.4 in release as well. The following table is a summary of the results. The output of our algorithm is split across Appendix F and Appendix G where the final results including some zero prediction examples are shown in Appendix F and some positive predictions are shown in Appendix G.

Table 5: Final FHE Inference Results

Metric	Value
Total time to complete run of full dataset (hours)	4.4
Average time to complete one prediction (seconds)	16
Final Loss	0.06
Legitimate Transactions Detected (True Negatives)	445
Legitimate Transactions Incorrectly Detected (False Positives)	47
Fraudulent Transactions Missed (False Negatives)	32
Fraudulent Transactions Detected (True Positives)	460
Precision	0.91
Recall	0.93
Average SEAL Ptxt vs. Ctxt Calculation Error	$5.3 * 10^{-6}$
Average NumPy vs. SEAL Calculation Error	0.16

Our analysis begins with the amount of time it took to complete 984 predictions, or in other words, the efficiency of our model. 4 hours and 24 minutes is reasonable when one accounts that one prediction takes an average of 16 seconds to complete. By analyzing the output of our algorithm in Appendices F and G, we notice that the MatMul function is the least efficient of the other functions. 10 out of the average 16 seconds it takes to complete one prediction pertain to the MatMul function. The reason is because it runs in O(n). Since there are 29 rotations, additions, and Ptxt-Ctxt multiplications, using Table 3, we see that the MatMul function should run for about 5 seconds. Assuming a poly modulus degree of 16384 was used to obtain the results of Table 3, we can naively extrapolate these 5 seconds to 10 seconds since our poly modulus degree is double showing that the runtime was as expected. The PolyEval function also runs relative to the benchmarks set in Table 3. Considering there are seven Ctxt-Ctxt multiplications, eight additions, and seven Ptxt-Ctxt multiplications, the function should run for about 1.4 seconds and, therefore, 2.8 seconds in our case which is what the results confirm. The DotProductPlain function contains six rotations and additions as well as one Ptxt-Ctxt multiplication. This means that the function should run for about 976 milliseconds and, therefore, about 1.9 seconds in our case. The results show that our function is actually running for about half of that time. A reason for this could be that our naïve assumption earlier that twice the poly modulus degree means twice the runtime is false but nevertheless, it runs more efficiently than expected. Therefore, the model is performing to the best of its abilities.

The combination of this inference model and the SEAL parameters we used would be impractical for a 300 transactions/second input, the average, daily, upper limit of Canada's largest payment processor, Moneris. This is true if a company inputs all its transactions in real time to use our model as a means of flagging fraudulent transactions in the real time. It is possible, though, that a company would not want to run our model in the real time. They could relax their current fraudulent transaction flagging system and use our model as a final determinant of whether a transaction that was marked as fraudulent, is fraudulent. There are

many use-case scenarios. It seems reasonable that an encrypted transaction can be classified in 16 seconds and therefore, our conclusion is that our model performs efficiently.

To conclude that our model is practical, we must analyze its predictive capacity. The final loss was very low. 0.06 is an incredible result considering all the approximations we introduced in our model. Note that the loss is a good measure of the predictive power of our model. Since it is the square loss between the model's prediction and its real label, it also describes our model's level of confidence. The closer the loss is to 0, the more predictions there are close to 1 or 0, and, therefore, the more confident our model. The conclusion is therefore that this setup and model performs to a high degree of confidence.

The other metrics further strengthen our model's practicality. The model correctly predicted 460 fraudulent transactions out of 507 predicted positives, making the precision 91%. It also correctly predicted 460 fraudulent transactions out of the 492 actual, fraudulent transactions, making the recall 93%. Recall is the most important metric as outlined in section 3.2.3. This means that only 32 fraudulent transactions were missed out of 492, a good result. Our model performs well within the expectations that we set in section 3.2.3. Almost all of the fraudulent transactions were caught and only 47 people out of 982 would complain about being accused of fraud.

The model's approximate behavior is also very precise. The average SEAL Plain vs. Cipher Calculation error, which is the average error between manually calculating the result and SEAL calculating the encrypted result, is on the order of 10^{-6}. This shows that our model is precise and perhaps a bit too precise. Later, we show how we can use this to make our model more efficient. The average NumPy vs. SEAL Calculation error is also low. This metric measures the average difference in our predictions using NumPy and encrypted predictions using SEAL. An average difference of 0.16 would only make a difference in final metrics if 0.16 causes the prediction to flip from positive to negative. However, our results do not show this behaviour. The model is, therefore, very close to a Ptxt model which is desired.

Putting all of these results together, taking into account the high level of performance and practical efficiency, this model was found to be practical for use with encrypted data. This strengthens the increased practicality claim of FHE researchers. We have shown that it is practical to perform inference on encrypted data using the latest research in fully homomorphic encryption.

For future work, we plan to optimize our secure machine learning algorithm further. Some areas of improvement could include decreasing the degree of our approximate polynomial. By decreasing the degree, we decrease the multiplicative depth of our algorithm. Decreasing the depth leads to decreasing the poly modulus degree. If we can decrease our poly modulus degree, we can dramatically increase the efficiency of our algorithm. To decrease our poly modulus degree by half and keep the same scale, we would need to have a polynomial of maximum order-3. However, this would not be post-quantum safe since the sum of the elements in the coefficient modulus would have to be 438, which is greater than 411. If we change the scale to

2^{30}, we can decrease the poly modulus degree by half and keep the polynomial degree-7. Keeping it degree-7 would keep the prediction interval the same as our results above. Decreasing the polynomial degree would decrease our interval and cause predictions outside of [0. 1] which is not preferred. Decreasing the scale would increase the average SEAL Ptxt vs. Ctxt Calculation error by a couple of orders of magnitude but since our initial order of magnitude was very low, on the order of 10^{-6}, we can still increase it to improve efficiency. Since the sum of the coefficient modulus elements would be 390, it would be post-quantum safe.

We changed the parameters to the following summarized table values and obtained the following optimized results. The output of our algorithm is split across Appendix H and Appendix I where the final results including some zero prediction examples are shown in Appendix H and some positive predictions are shown in Appendix I.

Table 6: Optimized SEAL Parameters

Parameter	Value
Poly Modulus Degree	16,384
Coefficient Modulus	[30, 30, … , 30] size = 13
Scale	2^{30}

Table 7: Optimized FHE Inference Results

Metric	Value
Total time to complete run of full dataset (hours)	2.1
Average time to complete one prediction (seconds)	7.7
Final Loss	0.06
Legitimate Transactions Detected (True Negatives)	445
Legitimate Transactions Incorrectly Detected (False Positives)	47
Fraudulent Transactions Missed (False Negatives)	32
Fraudulent Transactions Detected (True Positives)	460
Precision	0.91
Recall	0.93
Average SEAL Ptxt vs. Ctxt Calculation Error	$7.8 * 10^{-4}$
Average NumPy vs. SEAL Calculation Error	0.16

The results show that the model's predictive capabilities remained the same. The main differences were precision and efficiency.

The average SEAL Ptxt vs. Ctxt Calculation error increased by two orders of magnitude. While this is a decrease in precision, it is a good decrease in precision since the precision is still rather high. In a task that predicts ones and zeros, precision does not need to be on the order of 10^{-6}. With the other results being exactly the same, we can conclude that our decrease in precision did not negatively affect our model's predictive capacity. It did, however, increase our efficiency.

The total time to complete a run of the full dataset shrunk by a factor of 2.1. The average time to complete a prediction did as well. The change in parameters positively impacted our model's efficiency. With a decrease in poly modulus degree, there is a resulting decrease in the length of CKKS encoded vectors. This means that operations should, theoretically, take less time to perform. Here, we see that this is what actually happens. While secure machine learning inference using fully homomorphic encryption is still much less efficient than Ptxt inference (since it takes less than a second to classify the whole dataset in NumPy), we can say, even more so now, that secure machine learning using fully homomorphic encryption is practical.

Some further improvements could be to look for better Ptxt-Ctxt matrix multiplication algorithms since that is where most of the classification time elapses. Once could also try and decrease the encryption parameters further to achieve better efficiency, but it would still not decrease our runtime by at least an order of magnitude (since runtime is effectively halved by halving the parameter values). It appears that further research needs to be done to find better FHE schemes that compute rotations and Ctxt-Ctxt multiplications faster. No matter how hard one tries to improve the runtime of a model using the CKKS scheme, rotation and Ctxt-Ctxt multiplication algorithms must improve on their complexity boundaries as set in Table 3. Our runtime is, therefore, bounded by these fundamental complexities. As for precision, it can be said that FHE using CKKS, although approximate, can still achieve a great level of precision.

4 Conclusion

Secure Machine Learning is an ever-growing field. This thesis has demonstrated that it is a field worth researching. Our results have shown that SML is not only practical but can achieve a high degree of correctness similar to that of plaintext Machine Learning. Its practicality is strengthened by proving that Fully Homomorphic Encryption can run on modern mobile devices efficiently. Mobile devices contain an immense wealth of a person's personal information, for example, the places they go, the people they communicate with, and the credit cards they use for their purchasing habits. Today, unfortunately, people give their personal data willingly to big corporations in exchange for services that enable corporations to use their data for financial gain. FHE offers a win-win scenario. This thesis has shown that people can encrypt their information homomorphically and send it to a third-party without offering the possibility of decrypting their data (GPS coordinates sent to a cloud). In other words, privacy is retained. It has shown that third-party services can use a person's encrypted personal information to perform data analytics while still attaining financial or other personal gain (SML inference algorithm predicting fraudulent credit card transactions).

This thesis has also demonstrated the power of approximate arithmetic on improving the efficiency of FHE, ML inference. Using a specially crafted Chebyshev polynomial approximation of the Sigmoid function, we constructed a powerful machine learning model that, while linear, performed to an accuracy of 97%. Using Microsoft SEAL [6], CKKS [2], and three optimized algorithms (MatMul, PolyEval, and DotProductPlain), we showed that it is possible to classify encrypted transactions as fraudulent with a final loss of 6% and more so, that efficient, approximate logistic regression inference on encrypted data is possible and practical.

The next step should be training on encrypted data. While this thesis showed the practicality of inference, efficient training on encrypted data is the ultimate goal. With training on encrypted data, secure machine learning as a service (SMLAAS) can be realized. A dataset can be sent from a mobile device encrypted, a model can be trained on the encrypted data, and, as shown in this thesis, inference could be performed efficiently to send an encrypted prediction back to a user telling them whether they should eat ice cream or chocolate cake today. In a world becoming less private and closer to each individual, this is not only a research challenge, but a necessity.

References

[1] C. Gentry, “A fully homomorphic encryption scheme,” PhD, Stanford University, Stanford, CA, USA, 2009.

[2] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic Encryption for Arithmetic of Approximate Numbers,” ePrint IACR, 421, 2016. Accessed: Jan. 24, 2020. [Online]. Available: http://eprint.iacr.org/2016/421.

[3] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei, “Faster CryptoNets: Leveraging Sparsity for Real-World Encrypted Inference,” *arXiv:1811.09953 [cs]*, Nov. 2018, Accessed: Jan. 25, 2020. [Online]. Available: http://arxiv.org/abs/1811.09953.

[4] S. Halevi and V. Shoup, “Algorithms in HElib,” in *Advances in Cryptology – CRYPTO 2014*, vol. 8616, J. A. Garay and R. Gennaro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 554–571.

[5] S. Hong, “HEAAN: Homomorphic Encryption for Arithmetic of Approximate Numbers,” Jan. 21, 2020. https://github.com/snucrypto/HEAAN (accessed Jan. 24, 2020).

[6] “Microsoft SEAL: Fast and Easy-to-Use Homomorphic Encryption Library,” *Microsoft Research*. https://www.microsoft.com/en-us/research/project/microsoft-seal/ (accessed Jan. 24, 2020).

[7] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[8] S. Goldwasser and S. Micali, “Probabilistic encryption & how to play mental poker keeping secret all partial information,” in *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, San Francisco, California, USA, May 1982, pp. 365–377, doi: 10.1145/800070.802212.

[9] M. Ajtai, “Generating hard instances of lattice problems,” in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 99–108.

[10] O. Goldreich, S. Goldwasser, and S. Halevi, “Public-key cryptosystems from lattice reduction problems,” in *Advances in Cryptology — CRYPTO ’97*, Berlin, Heidelberg, 1997, pp. 112–131, doi: 10.1007/BFb0052231.

[11] J. Hoffstein, J. Pipher, and J. H. Silverman, “NTRU: A ring-based public key cryptosystem,” in *International Algorithmic Number Theory Symposium*, 1998, pp. 267–288.

[12] C. Gentry and S. Halevi, “Implementing Gentry’s Fully-Homomorphic Encryption Scheme,” ePrint IACR, 520, 2010. Accessed: Jan. 24, 2020. [Online]. Available: https://eprint.iacr.org/2010/520.

[13] J. Fan and F. Vercauteren, “Somewhat Practical Fully Homomorphic Encryption,” ePrint IACR, 144, 2012. Accessed: Jan. 24, 2020. [Online]. Available: https://eprint.iacr.org/2012/144.

[14] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “Fully Homomorphic Encryption without Bootstrapping,” ePrint IACR, 277, 2011. Accessed: Jan. 24, 2020. [Online]. Available: https://eprint.iacr.org/2011/277.

[15] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, “Bootstrapping for Approximate Homomorphic Encryption,” ePrint IACR, 153, 2018. Accessed: Jan. 24, 2020. [Online]. Available: https://eprint.iacr.org/2018/153.

[16] H. J. Lee, “Neural Network Approach to Identify Model of Vehicles,” in *Advances in Neural Networks - ISNN 2006*, Berlin, Heidelberg, 2006, pp. 66–72, doi: 10.1007/11760191_10.

[17] A. M. Turing, “Computing Machinery and Intelligence,” *Mind*, vol. LIX, no. 236, pp. 433–460, Oct. 1950, doi: 10.1093/mind/LIX.236.433.

[18] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, International. pg. 16 - 17: Pearson US Imports & PHIPEs, 2002.

[19] T. Cover and P. Hart, “Nearest neighbor pattern classification,” *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, Jan. 1967, doi: 10.1109/TIT.1967.1053964.
[20] S. Linnainmaa, “Taylor expansion of the accumulated rounding error,” *BIT*, vol. 16, no. 2, pp. 146–160, Jun. 1976, doi: 10.1007/BF01931367.
[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986, doi: 10.1038/323533a0.
[22] Y. LeCun, C. Cortes, and C. J. Burges, “The MNIST database of handwritten digits, 2009,” *URL http://yann. lecun. com/exdb/mnist*, 2009.
[23] В. В. Романюк, “Training Data Expansion and Boosting of Convolutional Neural Networks for Reducing the MNIST Dataset Error Rate,” *Research Bulletin of the National Technical University of Ukraine “Kyiv Politechnic Institute,”* vol. 0, no. 6, pp. 29–34, 2016, doi: 10.20535/1810-0546.2016.6.84115.
[24] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, “Deep, Big, Simple Neural Nets for Handwritten Digit Recognition,” *Neural Computation*, vol. 22, no. 12, pp. 3207–3220, Sep. 2010, doi: 10.1162/NECO_a_00052.
[25] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998, doi: 10.1109/5.726791.
[26] J. Deng, W. Dong, R. Socher, L.-J. Li, Kai Li, and Li Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2009, pp. 248–255, doi: 10.1109/CVPR.2009.5206848.
[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.
[28] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 770–778, doi: 10.1109/CVPR.2016.90.
[29] M. Abadi *et al.*, “TensorFlow: A system for large-scale machine learning,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283, Accessed: Jan. 25, 2020. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.
[30] A. Paszke *et al.*, “PyTorch: An Imperative Style, High-Performance Deep Learning Library,” in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.
[31] T. Graepel, K. Lauter, and M. Naehrig, “ML Confidential: Machine Learning on Encrypted Data,” Dec. 2012, Accessed: Jan. 26, 2020. [Online]. Available: https://www.microsoft.com/en-us/research/publication/ml-confidential-machine-learning-on-encrypted-data-2/.
[32] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, “CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy,” Feb. 2016, Accessed: Jan. 26, 2020. [Online]. Available: https://www.microsoft.com/en-us/research/publication/cryptonets-applying-neural-networks-to-encrypted-data-with-high-throughput-and-accuracy/.
[33] H. Chen *et al.*, “Logistic regression over encrypted data from fully homomorphic encryption,” *BMC Medical Genomics*, vol. 11, no. 4, p. 81, Oct. 2018, doi: 10.1186/s12920-018-0397-z.
[34] O. Masters *et al.*, “Towards a Homomorphic Machine Learning Big Data Pipeline for the Financial Services Sector,” ePrint IACR, 1113, 2019. Accessed: Jan. 26, 2020. [Online]. Available: http://eprint.iacr.org/2019/1113.

[35] A. A. Badawi *et al.*, "The AlexNet Moment for Homomorphic Encryption: HCNN, the First Homomorphic CNN on Encrypted Data with GPUs," *arXiv:1811.00778 [cs]*, Feb. 2019, Accessed: Jan. 26, 2020. [Online]. Available: http://arxiv.org/abs/1811.00778.
[36] "TFEncrypted: A Framework for Machine Learning on Encrypted Data," Jan. 26, 2020. https://github.com/tf-encrypted/tf-encrypted (accessed Jan. 26, 2020).
[37] P. Leach and M. Mealling, "RFC 4122 Standard." https://www.ietf.org/rfc/rfc4122.txt (accessed Apr. 01, 2020).
[38] Machine Learning Group - ULB, "Credit Card Fraud Detection." https://kaggle.com/mlg-ulb/creditcardfraud (accessed Apr. 02, 2020).
[39] K. P. F.R.S, "LIII. On lines and planes of closest fit to systems of points in space," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, no. 11, pp. 559–572, Nov. 1901, doi: 10.1080/14786440109462720.
[40] Google, "Classification on imbalanced data | TensorFlow Core," *TensorFlow*. https://www.tensorflow.org/tutorials/structured_data/imbalanced_data (accessed Apr. 02, 2020).
[41] O. Çetin, D. of E. and E. E. Bozok University, F. Temurtaş, D. of E. and E. E. Bozok University, Ş. Gülgönül, and A. TURKSAT Satellite Communication and Cable TV AS, "An application of multilayer neural network on hepatitis disease diagnosis using approximations of sigmoid activation function," *Dicle Tıp Dergisi; Cilt 42, Sayı 2 (2015): Dicle Tıp Dergisi / Dicle Medical Journal*, 2015, Accessed: Apr. 04, 2020. [Online]. Available: http://agris.fao.org/agris-search/search.do?recordID=TR2016017424.
[42] K. Basterretxea, J. M. Tarela, and I. del Campo, "Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons," Jan. 2004, doi: 10.1049/ip-cds:20030607.
[43] M. Vlček, "CHEBYSHEV POLYNOMIAL APPROXIMATION FOR ACTIVATION SIGMOID FUNCTION," *NNW*, vol. 22, no. 4, pp. 387–393, 2012, doi: 10.14311/NNW.2012.22.023.
[44] J. Schlessman, "Approximation of the sigmoid function and its derivative using a minimax approach," Lehigh, 2002.
[45] E. W. Weisstein, "Chebyshev Polynomial of the First Kind." https://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html (accessed Apr. 04, 2020).
[46] The SciPy community, "NumPy — NumPy 1.19." https://numpy.org/ (accessed Apr. 04, 2020).
[47] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980 [cs]*, Jan. 2017, Accessed: Apr. 04, 2020. [Online]. Available: http://arxiv.org/abs/1412.6980.
[48] T. Zhang and B. Yu, "Boosting with early stopping: Convergence and consistency," *Ann. Statist.*, vol. 33, no. 4, pp. 1538–1579, Aug. 2005, doi: 10.1214/009053605000000255.
[49] "Google Colaboratory." https://colab.research.google.com/notebooks/welcome.ipynb (accessed Apr. 04, 2020).
[50] Microsoft, "Visual Studio 2019," *Visual Studio*. https://visualstudio.microsoft.com/vs/ (accessed Apr. 05, 2020).
[51] S. Halevi and V. Shoup, "Faster Homomorphic Linear Transformations in HElib," 244, 2018. Accessed: Apr. 05, 2020. [Online]. Available: http://eprint.iacr.org/2018/244.
[52] Hao Chen, "Techniques in PPML," *Microsoft Research*, Dec. 02, 2019. https://www.microsoft.com/en-us/research/video/private-ai-bootcamp-techniques-in-ppml/ (accessed Apr. 05, 2020).
[53] Kim Laine, "Microsoft SEAL," *Microsoft Research*, Dec. 02, 2019. https://www.microsoft.com/en-us/research/video/microsoft-seal/ (accessed Apr. 06, 2020).

Appendices

Appendix A: SEAL Java Wrapper Code

```
extern "C"
JNIEXPORT jstring JNICALL
Java_com_example_testhomomorphicencryptionapp_MainActivity_setParameters
(JNIEnv *env, jobject obj)
{
   EncryptionParameters parms(scheme_type::CKKS);
   //size_t poly_modulus_degree = 8192;
    size_t poly_modulus_degree = 4096;
   parms.set_poly_modulus_degree(poly_modulus_degree);
   parms.set_coeff_modulus(CoeffModulus::Create(
      poly_modulus_degree, { 30, 24, 24, 30 }));

    jstring result = env->NewStringUTF(encodeSealToBase64(parms).c_str());

   context = SEALContext::Create(parms);

   return result;
}

extern "C"
JNIEXPORT jstring JNICALL
Java_com_example_testhomomorphicencryptionapp_MainActivity_encryptDoubleArray
(JNIEnv *env, jobject obj, jstring parmsJBA, jstring publicKey, jdoubleArray data)
{
   string rawByteStrKey = base64_decode(jstring2string(env, publicKey));

   jboolean isCopyData;
   jdouble* rawjDoublesData = env->GetDoubleArrayElements(data, &isCopyData);
   double* rawDoublesData = (double *)rawjDoublesData;
   int rawSizeData = env->GetArrayLength(data);

   vector<double> datav(rawDoublesData, rawDoublesData+rawSizeData);

   stringstream inKey(rawByteStrKey);

   PublicKey publicKeyIn;
   stringstream out;
    try {
      publicKeyIn.load(context, inKey);
      Encryptor encryptor(context, publicKeyIn);
      CKKSEncoder encoder(context);

      Plaintext plain;
      double scale = pow(2.0, 40);
      encoder.encode(datav, scale, plain);

      Ciphertext encrypted;
        encryptor.encrypt(plain, encrypted);

      encrypted.save(out);
   }
   catch (std::exception& e) {
      cerr << e.what() << endl;
```

```
    }
    const string tmp = out.str();
    const string resStr = base64_encode(tmp);

    jstring result = env->NewStringUTF(resStr.c_str());

    return result;

}
```

Snippet 1: Set Parameters and Encrypt function from SEAL Java Wrapper

Appendix B: Android Application Design and Code

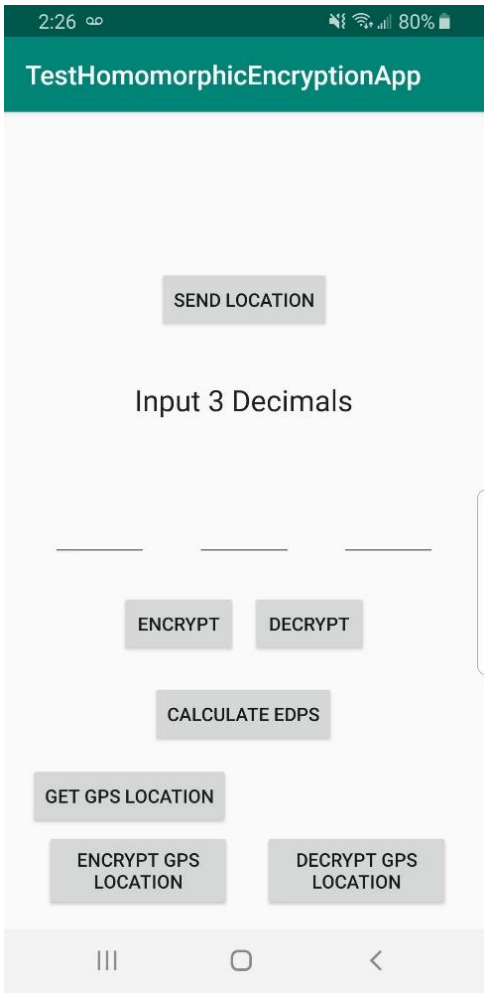


Figure 8: Design of Android Application

```
static {
    System.loadLibrary("SealJavaWrapper");
}

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    this.dec0View = findViewById(R.id.inputDec0);
    this.dec1View = findViewById(R.id.inputDec1);
    this.dec2View = findViewById(R.id.inputDec2);

    this.EDPSRead = findViewById(R.id.EDPSRead);

    this.longRead = findViewById(R.id.longRead);
    this.latRead = findViewById(R.id.latRead);

    this.locationManager =
(LocationManager)getSystemService(Context.LOCATION_SERVICE);

    SEALConfig tmp = loadSEALConfig("SEALConfig.txt");
    if (tmp == null) {
        this.config = new SEALConfig();
        this.config.setParams(this.setParameters());
        this.config.setPrivateKey(this.getPrivateKey(this.config.getParams()));
        this.config.setPublicKey(this.getPublicKey(this.config.getParams(),
this.config.getPrivateKey()));
        this.saveSEALConfig(this.config, "SEALConfig.txt");
    } else {
        String junk = this.setParameters();
        this.config = tmp;
    }

}

public void saveSEALConfig(SEALConfig SEALConfig, String fileName){
    ObjectOutput out = null;
    try {
        out = new ObjectOutputStream(new FileOutputStream(new
File(getFilesDir(),"")+File.separator+fileName));
        out.writeObject(SEALConfig);
        out.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public SEALConfig loadSEALConfig(String fileName){
    SEALConfig config = new SEALConfig();

    ObjectInputStream input;
    try {
        input = new ObjectInputStream(new FileInputStream(new File(new
File(getFilesDir(),"")+File.separator+fileName)));
        config = (SEALConfig) input.readObject();
        Log.v("serialization","Config Uid="+SEALConfig.getSerialVersionUID());
        input.close();
    } catch (StreamCorruptedException e) {
        e.printStackTrace();
```

```
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }

    if (config.isEmpty()) {
        return null;
    }
    return config;
}
```

Snippet 2: Saving and loading SEAL configuration (keys and parameters)

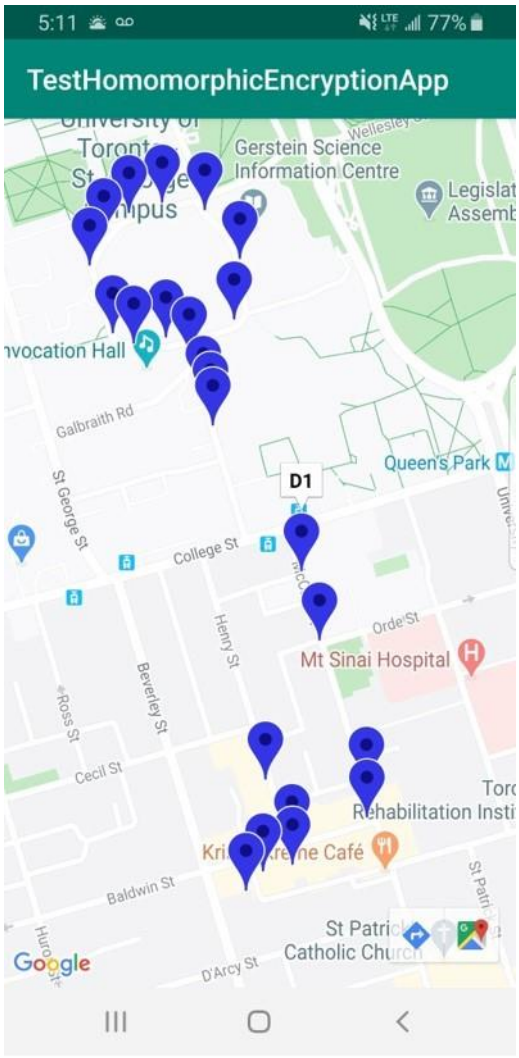


Figure 9: Decrypted GPS Coordinates on Map Depicting User's Walk

```
public void onMap(View view){
    JsonObjectRequest request = new JsonObjectRequest(
        Request.Method.GET,
        reqUrl2,
        new JSONObject(),
        response -> {
            try {
                String reqUrl3 = response.getString("url");
                JsonObjectRequest jsonObjectRequest = new JsonObjectRequest
                    (
                        Request.Method.GET,
                        reqUrl3,
                        new JSONObject(),
                        response1 -> {
                            //System.out.println(response1.toString());
                            PathsLocationData paths = null;
                            try {
                                paths = mapper.readValue(response1.toString(),
PathsLocationData.class);
                            } catch (IOException e) {
                                e.printStackTrace();
                            }
                            if (paths != null) {
                                System.out.println("Got the paths");

                                // Decrypt path 1
                                ArrayList<LatLng> path1 = new ArrayList<>();
                                int i;
                                for (i = 0; i < paths.path1.path.size(); i++){
                                    double lat =
decryptDoubleArray(this.config.getParams(),
                                            paths.path1.key,
                                            paths.path1.path.get(i).latitude
                                    )[0];
                                    double lng =
decryptDoubleArray(this.config.getParams(),
                                                paths.path1.key,
                                                paths.path1.path.get(i).longitude
                                    )[0];
                                    LatLng temp = new LatLng(lat, lng);
                                    path1.add(temp);
                                }

                                // Decrypt path 2
                                ArrayList<LatLng> path2 = new ArrayList<>();
                                for (i = 0; i < paths.path2.path.size(); i++){
                                    double lat =
decryptDoubleArray(this.config.getParams(),
                                                paths.path2.key,
                                                paths.path2.path.get(i).latitude
                                    )[0];
                                    double lng =
```

```
decryptDoubleArray(this.config.getParams(),
                                            paths.path2.key,
                                            paths.path2.path.get(i).longitude
                                    )[0];
                                    LatLng temp = new LatLng(lat, lng);
                                    path2.add(temp);
                                }

                                System.out.println("Done");
                                Intent myIntent = new Intent(MainActivity.this,
MapsMarkerActivity.class);
                                myIntent.putExtra("path1", path1);
                                myIntent.putExtra("path2", path2);
                                MainActivity.this.startActivity(myIntent);
                            }
                        },
                        error -> {
                            System.out.println(error.toString());
                        }
                    );
                int socketTimeout = 30000;//30 seconds - change to what you want
                RetryPolicy policy = new DefaultRetryPolicy(socketTimeout,
DefaultRetryPolicy.DEFAULT_MAX_RETRIES, DefaultRetryPolicy.DEFAULT_BACKOFF_MULT);
                jsonObjectRequest.setRetryPolicy(policy);

                // Access the RequestQueue through your singleton class.

MySingleton.getInstance(MainActivity.this.getApplicationContext()).addToRequestQueue(
jsonObjectRequest);
            } catch ( Exception e ) {
                e.printStackTrace();
            }
        },
        error -> {
            // Handle error
            EDPSRead.setText("Resp Err: " + error.getMessage());
        }
    );
    int socketTimeout = 30000;//30 seconds - change to what you want
    RetryPolicy policy = new DefaultRetryPolicy(socketTimeout,
DefaultRetryPolicy.DEFAULT_MAX_RETRIES, DefaultRetryPolicy.DEFAULT_BACKOFF_MULT);
    request.setRetryPolicy(policy);
    MySingleton.getInstance(this.getApplicationContext()).addToRequestQueue(request);
}
```

Snippet 3: onMap function that retrieves encrypted GPS coordinates, decrypts, and displays as markers on Map

Figure 10: Application UI with Launch Map Functionality

Appendix C: AWS Function Design and Code

```
AWS_LOGSTREAM_DEBUG(TAG, "received payload: " << request.payload);

   JsonValue json(request.payload);
   if (!json.WasParseSuccessful()) {
       return invocation_response::failure("Failed to parse input JSON", "Invalid
JSON");
   }

   auto v = json.View();

   if (!v.ValueExists("params") || !v.ValueExists("key") || !v.ValueExists("latit
ude") || !v.ValueExists("longitude") || !v.GetObject("params").IsString() || !v.G
etObject("key").IsString() || !v.GetObject("latitude").IsString() || !v.GetObject
("longitude").IsString()) {
       return invocation_response::failure("Missing input value params, key, or d
ata", "InvalidJSON");
   }

   auto paramsEnc = v.GetString("params");
   auto keyEnc = v.GetString("key");
   auto latEnc = v.GetString("latitude");
   auto longEnc = v.GetString("longitude");
```

Snippet 4: Parsing and checking the values of the POST request

```
// add id to tables
   prParams.AddItem("id", avId);
   prKey.AddItem("id", avId);
   prLat.AddItem("id", avId);
   prLong.AddItem("id", avId);

   // add param, key, lat, long values to respective tables

   Aws::DynamoDB::Model::BatchWriteItemRequest bwir;

   Aws::DynamoDB::Model::AttributeValue avParams;
   avParams.SetS(paramsEnc);
   prParams.AddItem("params", avParams);
   Aws::DynamoDB::Model::WriteRequest wrParams;
   wrParams.SetPutRequest(prParams);
   vector<WriteRequest> tableParamsWRVect{wrParams};
   bwir.AddRequestItems(tableParams, tableParamsWRVect);
```

Snippet 4: Example creation of database write requests and addition to batch write request

```
Aws::Utils::Outcome dbRes = client.BatchWriteItem(bwir);
   if (!dbRes.IsSuccess())
   {
       std::cout << dbRes.GetError().GetMessage() << std::endl;
       return invocation_response::failure("DB input did not succeed.", "InvalidQ
uery");
   }
   std::cout << "Done!" << std::endl;
```

Snippet 5: Example execution of a database batch write request and possible error handling

Appendix D: Logistic Regression, Credit Card Fraud, Neural Network Python Notebook With Results


```
from __future__ import absolute_import, division, print_function, unicode_literals


try:
  # %tensorflow_version only exists in Colab.
  %tensorflow_version 2.x
except Exception:
  pass


import tensorflow as tf
from tensorflow import keras

import os
import tempfile

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

import math

import sklearn
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

Import and Preprocess datset

```
mpl.rcParams['figure.figsize'] = (12, 10)
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']


file = tf.keras.utils
raw_df = pd.read_csv('https://storage.googleapis.com/download.tensorflow.org/data/creditcard.
raw_df.head()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533

```
raw_df[['Time', 'V1', 'V2', 'V3', 'V4', 'V5', 'V26', 'V27', 'V28', 'Amount', 'Class']].descri
```

	Time	V1	V2	V3	V4	
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e
mean	94813.859575	3.919560e-15	5.688174e-16	-8.769071e-15	2.782312e-15	-1.552563e
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e

```
neg, pos = np.bincount(raw_df['Class'])
total = neg + pos
print('Examples:\n    Total: {}\n    Positive: {} ({:.2f}% of total)\n'.format(
    total, pos, 100 * pos / total))
```

```
Examples:
    Total: 284807
    Positive: 492 (0.17% of total)
```

```
cleaned_df = raw_df.copy()

# You don't want the `Time` column.
cleaned_df.pop('Time')

# The `Amount` column covers a huge range. Convert to log-space.
eps=0.001 # 0 => 0.1¢
cleaned_df['Log Amount'] = np.log(cleaned_df.pop('Amount')+eps)


# Use a utility from sklearn to split and shuffle our dataset.
train_df, test_df = train_test_split(cleaned_df, test_size=0.2)
train_df, val_df = train_test_split(train_df, test_size=0.2)

# Form np arrays of labels and features.
train_labels = np.array(train_df.pop('Class'))
bool_train_labels = train_labels != 0
val_labels = np.array(val_df.pop('Class'))
test_labels = np.array(test_df.pop('Class'))

train_features = np.array(train_df)
val_features = np.array(val_df)
test_features = np.array(test_df)
```

```
# Normalize and clip the data

scaler = StandardScaler()
train_features = scaler.fit_transform(train_features)

val_features = scaler.transform(val_features)
test_features = scaler.transform(test_features)

clip = 1

train_features = np.clip(train_features, -clip, clip)
val_features = np.clip(val_features, -clip, clip)
test_features = np.clip(test_features, -clip, clip)

np.savetxt('test_features.csv', test_features[:1000], delimiter=',')
np.savetxt('test_labels.csv', test_labels[:1000], delimiter=',')

print('Training labels shape:', train_labels.shape)
print('Validation labels shape:', val_labels.shape)
print('Test labels shape:', test_labels.shape)

print('Training features shape:', train_features.shape)
print('Validation features shape:', val_features.shape)
print('Test features shape:', test_features.shape)
```

```
Training labels shape: (182276,)
Validation labels shape: (45569,)
Test labels shape: (56962,)
Training features shape: (182276, 29)
Validation features shape: (45569, 29)
Test features shape: (56962, 29)
```

```
# Attempt to create Chebyshev polynomial approximation algorithm

# # Tools

# import operator as op
# from functools import reduce

# def ncr(n, r):
#     r = min(r, n-r)
#     numer = reduce(op.mul, range(n, n-r, -1), 1)
#     denom = reduce(op.mul, range(1, r+1), 1)
#     return numer / denom


# # Given

# p = 16
# q = 4
```

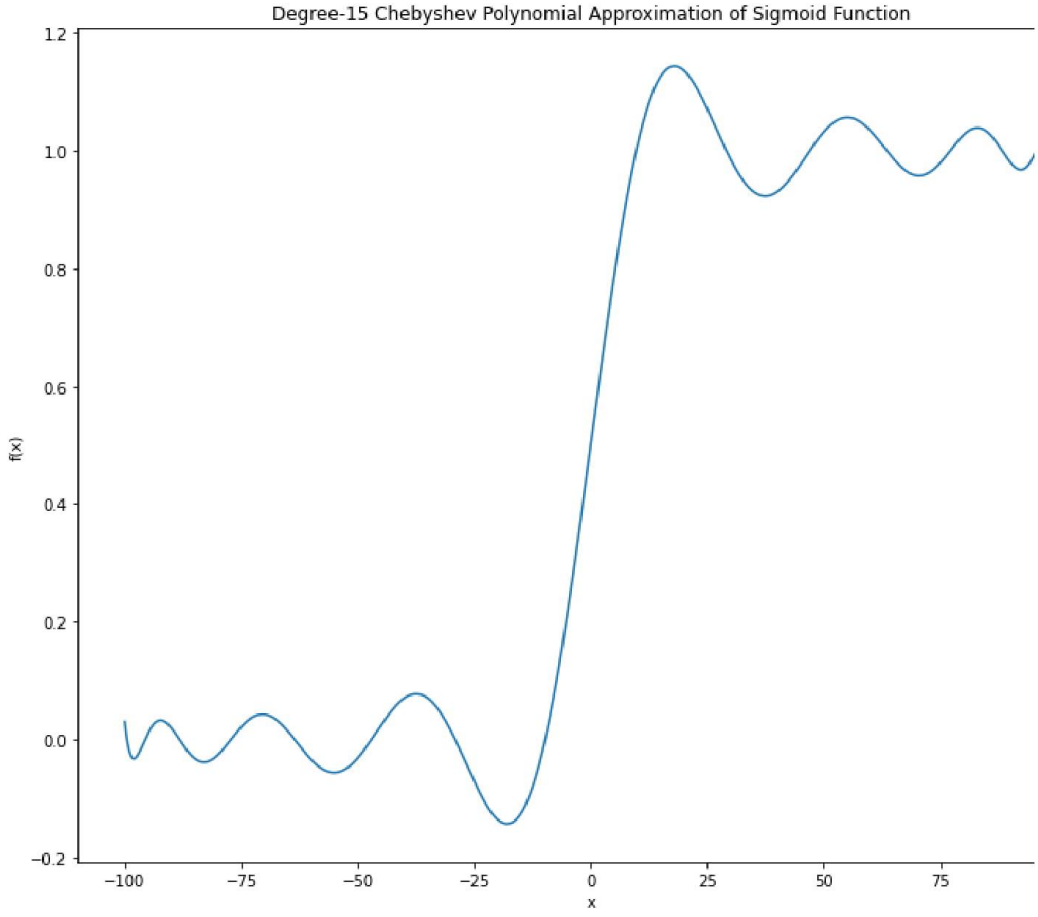
```
# N = p + q + 1

# alpha = np.zeros(N+1, dtype=float)
# a = np.zeros(N+1, dtype=float)

# # Initialization

# alpha[N] = ((-1)**p) * (2**(-2*(p+q))) * N/2 * ncr(p+q, q)
# alpha[N-1] = 2 * (q-p) * alpha[N]

# for mew in range(N-1, 2, -1):
#   alpha[mew-1] = 2/(N+1-mew) * ( (q-p)*alpha[mew] - (N+1+mew)/2*alpha[mew+1] )

# for mew in range(N, 1, -1):
#   a[mew] = alpha[mew]/mew

# a[0] = 1 - np.sum(a)

# sig_approx = np.polynomial.Chebyshev(a)

# x, y = sig_approx.linspace(1000, [-1, 1])

# plt.plot(x, y)
# plt.show()

# print(sig_approx)
```

Extract approximate function Chebyshev polynomial coefficients / save to disk

```
# Extract approximate function Chebyshev polynomial coefficients / save to disk

def sigmoid(x):
  return 1/(1 + np.exp(-x))

approx = np.polynomial.chebyshev.Chebyshev.interpolate(sigmoid, 15, [-100, 100])
coef = np.polynomial.chebyshev.cheb2poly(approx.coef)
for i in range(16):
  if i == 0:
    coef[i] = 0.76*(coef[i]+0.15)*(0.04**i)
  else:
    coef[i] = 0.76*coef[i]*(0.04**i)
np.savetxt('coef.csv', coef, delimiter=',')
print(coef)
x, y = approx.linspace(1000000, [-100, 100])
plt.title("Degree-15 Chebyshev Polynomial Approximation of Sigmoid Function")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.plot(x, y)
plt.show()

poly = ""
```

```
for i in range(len(coef)):
  if i == 0:
    poly += str(coef[i]) + " "
  else:
    poly += "+ (" + str(coef[i]) + ")x" + "^(" + str(i) + ") "

print(poly)
```

```
[ 4.94000000e-01  1.80385159e-01  1.09015019e-17 -4.53564039e-03
 -2.95926839e-20  5.48868668e-05 -1.54555892e-21 -3.37775893e-07
  1.56159513e-23  1.14101140e-09 -5.82665205e-26 -2.14479855e-12
  9.44042206e-29  2.10396440e-15 -5.56640625e-32 -8.39487958e-19]
```




```
0.4939999999999998 + (0.18038515864766605)x^(1) + (1.0901501923399337e-17)x^(2) + (-0.00
```

```
# Helper functions

def plot_metrics(history):
  metrics = ['loss', 'auc', 'precision', 'recall']
```

```
    metrics = ['loss', 'auc', 'precision', 'recall']
    for n, metric in enumerate(metrics):
      name = metric.replace("_"," ").capitalize()
      plt.subplot(2,2,n+1)
      plt.plot(history.epoch,  history.history[metric], color=colors[0], label='Train')
      plt.plot(history.epoch, history.history['val_'+metric],
               color=colors[0], linestyle="--", label='Val')
      plt.xlabel('Epoch')
      plt.ylabel(name)
      if metric == 'loss':
        plt.ylim([0, plt.ylim()[1]])
      elif metric == 'auc':
        plt.ylim([0.8,1])
      else:
        plt.ylim([0,1])

      plt.legend()

  def plot_cm(labels, predictions, p=0.5):
    cm = confusion_matrix(labels, predictions > p)
    plt.figure(figsize=(5,5))
    sns.heatmap(cm, annot=True, fmt="d")
    plt.title('Confusion matrix @{:.2f}'.format(p))
    plt.ylabel('Actual label')
    plt.xlabel('Predicted label')

    print('Legitimate Transactions Detected (True Negatives): ', cm[0][0])
    print('Legitimate Transactions Incorrectly Detected (False Positives): ', cm[0][1])
    print('Fraudulent Transactions Missed (False Negatives): ', cm[1][0])
    print('Fraudulent Transactions Detected (True Positives): ', cm[1][1])
    print('Total Fraudulent Transactions: ', np.sum(cm[1]))

  def plot_roc(name, labels, predictions, **kwargs):
    fp, tp, _ = sklearn.metrics.roc_curve(labels, predictions)

    plt.plot(100*fp, 100*tp, label=name, linewidth=2, **kwargs)
    plt.xlabel('False positives [%]')
    plt.ylabel('True positives [%]')
    plt.xlim([-0.5,20])
    plt.ylim([80,100.5])
    plt.grid(True)
    ax = plt.gca()
    ax.set_aspect('equal')
```

Define Approximate Function

```
  def test2(x):
    return tf.constant(0.494, dtype=tf.float32) + tf.add_n([
                    0.180385159*tf.pow(x, 1),
                    1.09015019*10**(-17)*tf.pow(x, 2),
                    -4.53564039*10**(-3)*tf.pow(x, 3),
```

```
                    0*-2.95926839*10**(-20)*tf.pow(x, 4),
                    5.48868668*10**(-5)*tf.pow(x, 5),
                    0*-1.54555892*10**(-21)*tf.pow(x, 6),
                    -3.37775893*10**(-7)*tf.pow(x, 7),
                    0*1.56159513*10**(-23)*tf.pow(x, 8),
                    1.14101140*10**(-9)*tf.pow(x, 9),
                    0*-5.82665205*10**(-26)*tf.pow(x, 10),
                    -2.14479855*10**(-12)*tf.pow(x, 11),
                    0*9.44042206*10**(-29)*tf.pow(x, 12),
                    2.10396440*10**(-15)*tf.pow(x, 13),
                    0*-5.56640625*10**(-32)*tf.pow(x, 14),
                    -8.39487958*10**(-19)*tf.pow(x, 15)
  ])
```

Define Model

```
METRICS = [
      keras.metrics.TruePositives(name='tp'),
      keras.metrics.FalsePositives(name='fp'),
      keras.metrics.TrueNegatives(name='tn'),
      keras.metrics.FalseNegatives(name='fn'),
      keras.metrics.BinaryAccuracy(name='accuracy'),
      keras.metrics.Precision(name='precision'),
      keras.metrics.Recall(name='recall'),
      keras.metrics.AUC(name='auc'),
]

def make_model(metrics = METRICS, output_bias=None):
  if output_bias is not None:
    output_bias = tf.keras.initializers.Constant(output_bias)
  model = keras.Sequential([
      keras.layers.Dense(
          29, activation=test2,#tf_relu_approx,
          input_shape=(train_features.shape[-1],)),
      # keras.layers.Dropout(0.5),
      keras.layers.Dense(1, activation=test2,
                         bias_initializer=output_bias),
  ])

  model.compile(
      optimizer=keras.optimizers.Adam(lr=1e-3),
      loss=keras.losses.BinaryCrossentropy(),
      metrics=metrics)

  return model


EPOCHS = 100
BATCH_SIZE = 2048

early_stopping = tf.keras.callbacks.EarlyStopping(
```

```
        monitor='val_auc',
        verbose=1,
        patience=10,
        mode='max',
        restore_best_weights=True)

    model = make_model()
    model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_8 (Dense)              (None, 29)                870
_________________________________________________________________
dense_9 (Dense)              (None, 1)                 30
=================================================================
Total params: 900
Trainable params: 900
Non-trainable params: 0
_________________________________________________________________
```

```
    model.predict(train_features[:10])
```

```
array([[0.30068403],
       [0.31374204],
       [0.28836524],
       [0.29494905],
       [0.28450078],
       [0.28452685],
       [0.30861765],
       [0.3060624 ],
       [0.30793655],
       [0.31983685]], dtype=float32)
```

```
    results = model.evaluate(train_features, train_labels, batch_size=BATCH_SIZE, verbose=0)
    print("Loss: {:0.4f}".format(results[0]))
```

```
Loss: 0.3565
```

```
    initial_bias = np.log([pos/neg])
    initial_bias
```

```
array([-6.35935934])
```

```
    model = make_model(output_bias = initial_bias)
    model.predict(train_features[:10])
```

```
array([[0.08857045],
       [0.08620724],
       [0.07861236],
       [0.07924572],
       [0.08581206],
       [0.09104541],
       [0.0834038 ],
       [0.08642712],
       [0.08921149],
       [0.08311865]], dtype=float32)
```

```
results = model.evaluate(train_features, train_labels, batch_size=BATCH_SIZE, verbose=0)
print("Loss: {:0.4f}".format(results[0]))
```

```
Loss: 0.0938
```

```
initial_weights = os.path.join(tempfile.mkdtemp(),'initial_weights')
model.save_weights(initial_weights)
```

Oversample

```
pos_features = train_features[bool_train_labels]
neg_features = train_features[~bool_train_labels]

pos_labels = train_labels[bool_train_labels]
neg_labels = train_labels[~bool_train_labels]
```

```
print(pos_features.shape)
print(neg_features.shape)

ids = np.arange(len(pos_features))
choices = np.random.choice(ids, len(neg_features))

res_pos_features = pos_features[choices]
res_pos_labels = pos_labels[choices]

res_pos_features.shape
```

```
(309, 29)
(181967, 29)
(181967, 29)
```

```
resampled_features = np.concatenate([res_pos_features, neg_features], axis=0)
resampled_labels = np.concatenate([res_pos_labels, neg_labels], axis=0)

order = np.arange(len(resampled_labels))
np.random.shuffle(order)
resampled_features = resampled_features[order]
resampled_labels = resampled_labels[order]
```

```
resampled_features.shape
```

```
(363934, 29)
```

```
BUFFER_SIZE = 100000

def make_ds(features, labels):
  ds = tf.data.Dataset.from_tensor_slices((features, labels))#.cache()
  ds = ds.shuffle(BUFFER_SIZE).repeat()
  return ds

pos_ds = make_ds(pos_features, pos_labels)
neg_ds = make_ds(neg_features, neg_labels)


for features, label in pos_ds.take(1):
  print("Features:\n", features.numpy())
  print()
  print("Label: ", label.numpy())
```

```
Features:
 [-1.          1.         -1.          1.         -1.         -1.
 -1.          1.         -1.         -1.          1.         -1.
 -0.35797325 -1.         -1.         -1.         -1.         -1.
 -0.61809421  1.          1.         -0.26273372 -0.81012227 -1.
  1.          1.          1.          1.         -1.        ]

Label:  1
```

```
resampled_ds = tf.data.experimental.sample_from_datasets([pos_ds, neg_ds], weights=[0.5, 0.5]
resampled_ds = resampled_ds.batch(BATCH_SIZE).prefetch(2)


for features, label in resampled_ds.take(1):
  print(label.numpy().mean())
```

```
0.49072265625
```

```
resampled_steps_per_epoch = np.ceil(2.0*neg/BATCH_SIZE)
resampled_steps_per_epoch
```

```
278.0
```

```
val_ds = tf.data.Dataset.from_tensor_slices((val_features, val_labels)).cache()
val_ds = val_ds.batch(BATCH_SIZE).prefetch(2)
```

Train

```
resampled_model = make_model()
resampled_model.load_weights(initial_weights)
```

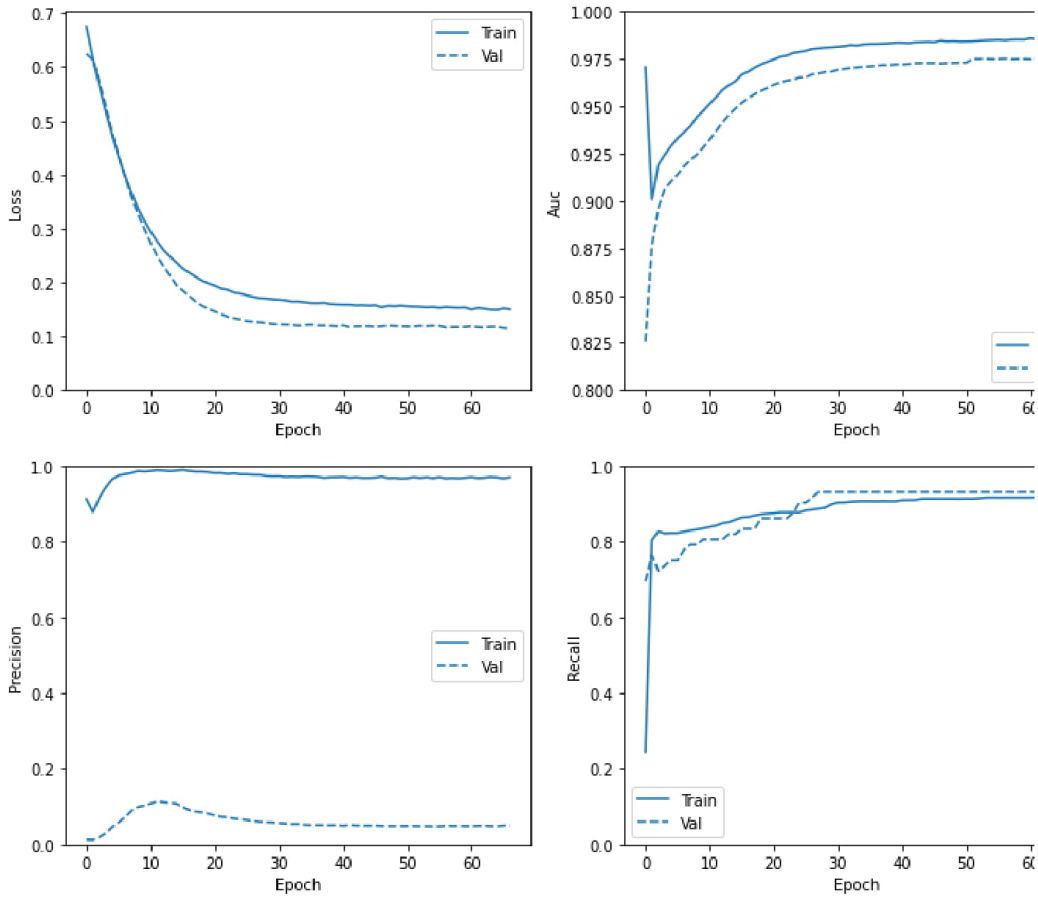
```
# Reset the bias to zero, since this dataset is balanced.
output_layer = resampled_model.layers[-1]
output_layer.bias.assign([0])

resampled_history = resampled_model.fit(
    resampled_ds,
    # These are not real epochs
    steps_per_epoch = 20,
    epochs=10*EPOCHS,
    callbacks = [early_stopping],
    validation_data=(val_ds))
```

```
Epoch 1/1000
20/20 [==============================] - 1s 66ms/step - loss: 0.6751 - tp: 5178.0000 - f
Epoch 2/1000
20/20 [==============================] - 1s 25ms/step - loss: 0.6156 - tp: 16446.0000 -
Epoch 3/1000
20/20 [==============================] - 0s 23ms/step - loss: 0.5643 - tp: 17157.0000 -
Epoch 4/1000
20/20 [==============================] - 0s 22ms/step - loss: 0.5177 - tp: 16921.0000 -
Epoch 5/1000
20/20 [==============================] - 0s 21ms/step - loss: 0.4738 - tp: 16766.0000 -
Epoch 6/1000
20/20 [==============================] - 0s 22ms/step - loss: 0.4332 - tp: 16808.0000 -
Epoch 7/1000
20/20 [==============================] - 1s 25ms/step - loss: 0.3978 - tp: 16897.0000 -
Epoch 8/1000
20/20 [==============================] - 0s 24ms/step - loss: 0.3668 - tp: 16940.0000 -
Epoch 9/1000
20/20 [==============================] - 1s 32ms/step - loss: 0.3380 - tp: 17085.0000 -
Epoch 10/1000
20/20 [==============================] - 0s 23ms/step - loss: 0.3149 - tp: 17197.0000 -
Epoch 11/1000
20/20 [==============================] - 1s 31ms/step - loss: 0.2942 - tp: 17181.0000 -
Epoch 12/1000
20/20 [==============================] - 1s 28ms/step - loss: 0.2764 - tp: 17294.0000 -
Epoch 13/1000
20/20 [==============================] - 1s 28ms/step - loss: 0.2600 - tp: 17399.0000 -
Epoch 14/1000
20/20 [==============================] - 0s 20ms/step - loss: 0.2477 - tp: 17466.0000 -
Epoch 15/1000
20/20 [==============================] - 1s 30ms/step - loss: 0.2372 - tp: 17446.0000 -
Epoch 16/1000
20/20 [==============================] - 1s 28ms/step - loss: 0.2254 - tp: 17853.0000 -
Epoch 17/1000
20/20 [==============================] - 0s 20ms/step - loss: 0.2178 - tp: 17737.0000 -
Epoch 18/1000
20/20 [==============================] - 1s 31ms/step - loss: 0.2104 - tp: 17891.0000 -
Epoch 19/1000
20/20 [==============================] - 1s 26ms/step - loss: 0.2032 - tp: 17871.0000 -
Epoch 20/1000
20/20 [==============================] - 1s 30ms/step - loss: 0.1980 - tp: 17989.0000 -
Epoch 21/1000
20/20 [==============================] - 0s 25ms/step - loss: 0.1938 - tp: 18111.0000 -
Epoch 22/1000
20/20 [==============================] - 0s 25ms/step - loss: 0.1885 - tp: 18009.0000 -
Epoch 23/1000
20/20 [==============================] - 0s 24ms/step - loss: 0.1861 - tp: 18070.0000 -
Epoch 24/1000
20/20 [==============================] - 1s 28ms/step - loss: 0.1809 - tp: 18013.0000 -
Epoch 25/1000
20/20 [==============================] - 0s 24ms/step - loss: 0.1791 - tp: 18084.0000 -
Epoch 26/1000
20/20 [==============================] - 1s 25ms/step - loss: 0.1757 - tp: 18002.0000 -
Epoch 27/1000
20/20 [==============================] - 1s 28ms/step - loss: 0.1728 - tp: 18147.0000 -
Epoch 28/1000
20/20 [==============================] - 1s 25ms/step - loss: 0.1705 - tp: 18122.0000 -
Epoch 29/1000
```

```
20/20 [==============================] - 0s 22ms/step - loss: 0.1698 - tp: 18253.0000 -
Epoch 30/1000
20/20 [==============================] - 1s 27ms/step - loss: 0.1684 - tp: 18324.0000 -
Epoch 31/1000
20/20 [==============================] - 0s 22ms/step - loss: 0.1676 - tp: 18576.0000 -
Epoch 32/1000
20/20 [==============================] - 0s 22ms/step - loss: 0.1664 - tp: 18562.0000 -
Epoch 33/1000
20/20 [==============================] - 0s 24ms/step - loss: 0.1642 - tp: 18536.0000 -
Epoch 34/1000
20/20 [==============================] - 0s 23ms/step - loss: 0.1642 - tp: 18469.0000 -
Epoch 35/1000
20/20 [==============================] - 1s 28ms/step - loss: 0.1629 - tp: 18723.0000 -
Epoch 36/1000
20/20 [==============================] - 1s 30ms/step - loss: 0.1614 - tp: 18673.0000 -
Epoch 37/1000
20/20 [==============================] - 1s 29ms/step - loss: 0.1612 - tp: 18599.0000 -
Epoch 38/1000
20/20 [==============================] - 1s 31ms/step - loss: 0.1618 - tp: 18790.0000 -
Epoch 39/1000
20/20 [==============================] - 1s 26ms/step - loss: 0.1596 - tp: 18521.0000 -
Epoch 40/1000
20/20 [==============================] - 1s 28ms/step - loss: 0.1587 - tp: 18522.0000 -
Epoch 41/1000
20/20 [==============================] - 1s 27ms/step - loss: 0.1584 - tp: 18763.0000 -
Epoch 42/1000
20/20 [==============================] - 1s 29ms/step - loss: 0.1584 - tp: 18393.0000 -
Epoch 43/1000
20/20 [==============================] - 1s 28ms/step - loss: 0.1571 - tp: 18551.0000 -
Epoch 44/1000
20/20 [==============================] - 1s 26ms/step - loss: 0.1575 - tp: 18743.0000 -
Epoch 45/1000
20/20 [==============================] - 0s 24ms/step - loss: 0.1567 - tp: 18602.0000 -
Epoch 46/1000
20/20 [==============================] - 1s 26ms/step - loss: 0.1574 - tp: 18665.0000 -
Epoch 47/1000
20/20 [==============================] - 0s 23ms/step - loss: 0.1537 - tp: 18633.0000 -
Epoch 48/1000
20/20 [==============================] - 0s 24ms/step - loss: 0.1563 - tp: 18734.0000 -
Epoch 49/1000
20/20 [==============================] - 0s 21ms/step - loss: 0.1556 - tp: 18794.0000 -
Epoch 50/1000
20/20 [==============================] - 1s 28ms/step - loss: 0.1566 - tp: 18679.0000 -
Epoch 51/1000
20/20 [==============================] - 0s 20ms/step - loss: 0.1555 - tp: 18729.0000 -
Epoch 52/1000
20/20 [==============================] - 0s 25ms/step - loss: 0.1548 - tp: 18773.0000 -
Epoch 53/1000
20/20 [==============================] - 0s 20ms/step - loss: 0.1545 - tp: 18873.0000 -
Epoch 54/1000
20/20 [==============================] - 1s 29ms/step - loss: 0.1535 - tp: 18736.0000 -
Epoch 55/1000
20/20 [==============================] - 1s 29ms/step - loss: 0.1541 - tp: 18937.0000 -
Epoch 56/1000
20/20 [==============================] - 0s 23ms/step - loss: 0.1526 - tp: 18772.0000 -
Epoch 57/1000
20/20 [==============================] - 0s 21ms/step - loss: 0.1540 - tp: 18604.0000 -
Epoch 58/1000
```

```
20/20 [==============================] - 0s 24ms/step - loss: 0.1532 - tp: 18683.0000 -
Epoch 59/1000
20/20 [==============================] - 0s 21ms/step - loss: 0.1528 - tp: 18631.0000 -
Epoch 60/1000
20/20 [==============================] - 1s 25ms/step - loss: 0.1532 - tp: 18921.0000 -
Epoch 61/1000
20/20 [==============================] - 0s 20ms/step - loss: 0.1496 - tp: 18646.0000 -
Epoch 62/1000
20/20 [==============================] - 0s 21ms/step - loss: 0.1524 - tp: 18758.0000 -
Epoch 63/1000
20/20 [==============================] - 1s 25ms/step - loss: 0.1510 - tp: 18603.0000 -
Epoch 64/1000
20/20 [==============================] - 0s 23ms/step - loss: 0.1493 - tp: 18725.0000 -
Epoch 65/1000
20/20 [==============================] - 1s 31ms/step - loss: 0.1487 - tp: 18705.0000 -
Epoch 66/1000
20/20 [==============================] - 1s 28ms/step - loss: 0.1516 - tp: 18797.0000 -
Epoch 67/1000
19/20 [===========================>..] - ETA: 0s - loss: 0.1501 - tp: 17713.0000 - fp: 5
20/20 [==============================] - 1s 29ms/step - loss: 0.1499 - tp: 18666.0000 -
Epoch 00067: early stopping
```

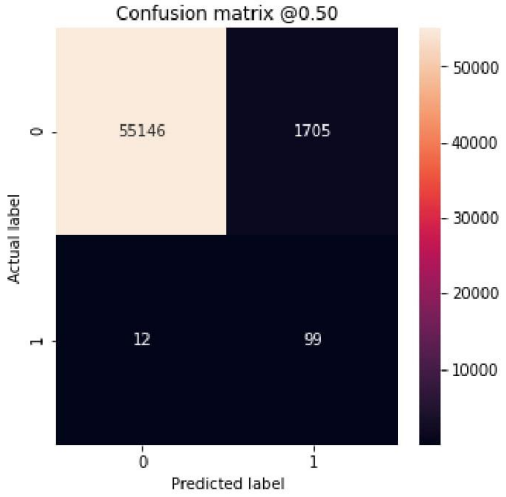
```
plot_metrics(resampled_history)
```




```
train_predictions_resampled = resampled_model.predict(train_features, batch_size=BATCH_SIZE)
test_predictions_resampled = resampled_model.predict(test_features, batch_size=BATCH_SIZE)


resampled_results = resampled_model.evaluate(test_features, test_labels,
                                             batch_size=BATCH_SIZE, verbose=0)
for name, value in zip(resampled_model.metrics_names, resampled_results):
  print(name, ': ', value)
print()

plot_cm(test_labels, test_predictions_resampled)

weights1 = resampled_model.layers[0].get_weights()[0]
biases1 = resampled_model.layers[0].get_weights()[1]
weights2 = resampled_model.layers[1].get_weights()[0]
biases2 = resampled_model.layers[1].get_weights()[1]

# with np.printoptions(precision=16, suppress=True):
```

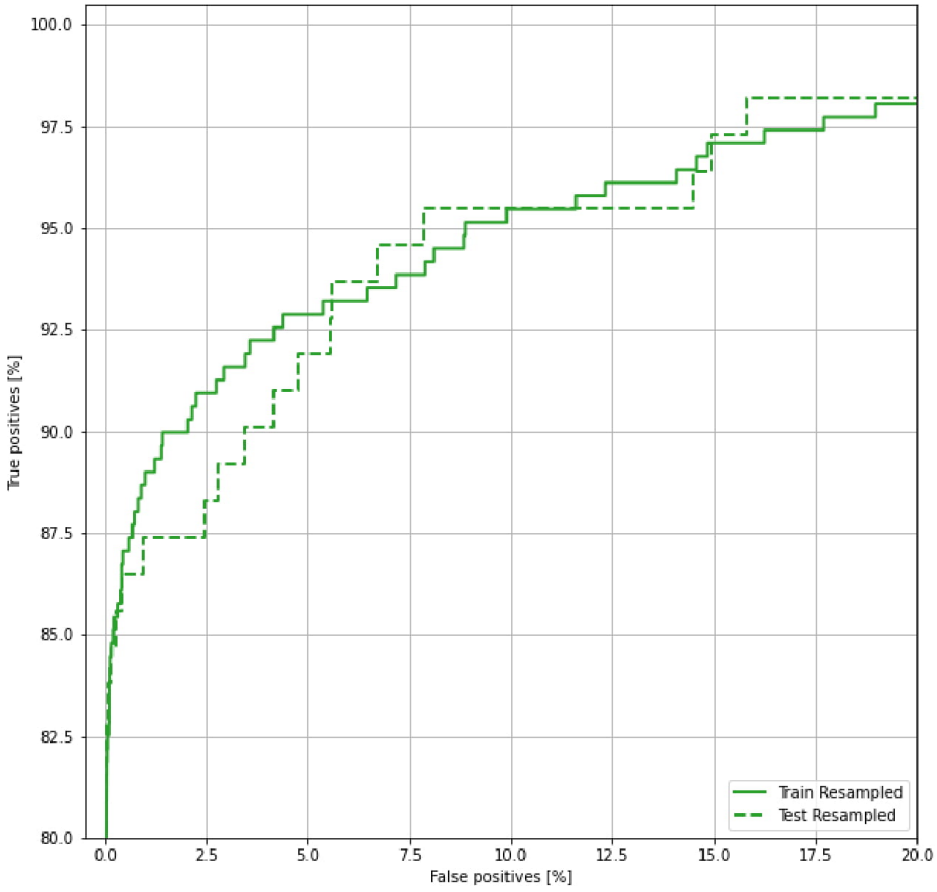
```
#     print(weights1)

# print(weights1)
# print(biases1)
# print(weights2)
# print(biases2)
```

```
loss :  0.11650435626506805
tp :  99.0
fp :  1705.0
tn :  55146.0
fn :  12.0
accuracy :  0.9698570966720581
precision :  0.05487804859876633
recall :  0.8918918967247009
auc :  0.9844763875007629

Legitimate Transactions Detected (True Negatives):  55146
Legitimate Transactions Incorrectly Detected (False Positives):  1705
Fraudulent Transactions Missed (False Negatives):  12
Fraudulent Transactions Detected (True Positives):  99
Total Fraudulent Transactions:  111
```




```
plot_roc("Train Resampled", train_labels, train_predictions_resampled,  color=colors[2])
plot_roc("Test Resampled", test_labels, test_predictions_resampled,  color=colors[2], linesty
plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7f48ef72d5c0>
```



Save Model and Weights/Biases to CSV

```
!pip install h5py
```

```
Requirement already satisfied: h5py in /usr/local/lib/python3.6/dist-packages (2.10.0)
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (from h5py)
Requirement already satisfied: numpy>=1.7 in /usr/local/lib/python3.6/dist-packages (fro
```

```
# serialize model to JSON
model_json = resampled_model.to_json()
with open("model2.json", "w") as json_file:
  json_file.write(model_json)
# serialize weights to HDF5
resampled_model.save_weights("model2.h5")
print("Saved model to disk")
```

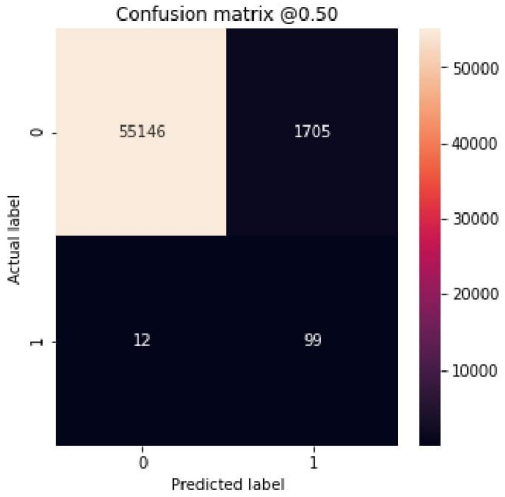
```
Saved model to disk
```

```
from keras.models import model_from_json

# load json and create model
json_file = open('model2.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
# loaded_model = model_from_json(loaded_model_json, custom_objects={'test2': test2})
loaded_model = make_model()
# load weights into new model
loaded_model.load_weights("model2.h5")
print("Loaded model from disk")
```

```
Loaded model from disk
```

```
# Test load

train_predictions_resampled = loaded_model.predict(train_features, batch_size=BATCH_SIZE)
test_predictions_resampled = loaded_model.predict(test_features, batch_size=BATCH_SIZE)

resampled_results = loaded_model.evaluate(test_features, test_labels,
                                          batch_size=BATCH_SIZE, verbose=0)
for name, value in zip(loaded_model.metrics_names, resampled_results):
  print(name, ': ', value)
print()

plot_cm(test_labels, test_predictions_resampled)
```

```
loss :  0.11650435626506805
tp :  198.0
fp :  3410.0
tn :  110292.0
fn :  24.0
accuracy :  0.9698570966720581
precision :  0.05487804859876633
recall :  0.8918918967247009
auc :  0.9844763875007629

Legitimate Transactions Detected (True Negatives):  55146
Legitimate Transactions Incorrectly Detected (False Positives):  1705
Fraudulent Transactions Missed (False Negatives):  12
Fraudulent Transactions Detected (True Positives):  99
Total Fraudulent Transactions:  111
```




```
# Retrieve Weights

weights1 = loaded_model.layers[0].get_weights()[0]
biases1 = loaded_model.layers[0].get_weights()[1]
weights2 = loaded_model.layers[1].get_weights()[0]
biases2 = loaded_model.layers[1].get_weights()[1]

np.savetxt('weights1.csv', weights1, delimiter=',')
np.savetxt('biases1.csv', biases1, delimiter=',')
np.savetxt('weights2.csv', weights2, delimiter=',')
np.savetxt('biases2.csv', biases2, delimiter=',')
```

Appendix E: FHE SEAL Inference Code

```
#include "examples.h"

using namespace std;
using namespace seal;

class Model
{
public:
    int degree;
    double scale;
    shared_ptr<SEALContext> ctx;
    PublicKey publicKey;
    SecretKey secretKey;
    RelinKeys relinKeys;
    GaloisKeys galoisKeys;

    vector<vector<double>> weights1;
    vector<double> biases1;
    vector<double> weights2;
    vector<double> biases2;

    vector<double> coef;

    vector<vector<double>> test_features;
    vector<double> test_labels;
    vector<double> npRes;

    vector<Ciphertext> x;

public:

    Model()
    {
        cout << "BEGIN INIT" << endl;

        // Set up params
        EncryptionParameters parms(scheme_type::CKKS);
        degree = 7;

        cout << "Polynomial Degree = " << degree << endl;
```

```
        ifstream fsParam("params.seal", ios::binary);
        if (fsParam.good()) {
            parms.load(fsParam);
        }
        else {
            int depth = ceil(log2(degree));
            cout << "depth = " << depth << endl;
            vector<int> moduli(13, 40);
            moduli[0] = 40;
            moduli[moduli.size() - 1] = 40;
            size_t poly_modulus_degree = 16384*2;
            parms.set_poly_modulus_degree(poly_modulus_degree);
            parms.set_coeff_modulus(CoeffModulus::Create(
                poly_modulus_degree, moduli));
            ofstream fs("params.seal", ios::binary);
            parms.save(fs);
            fs.close();
        }
        fsParam.close();

        // Set up scale
        scale = pow(2.0, 40);

        // Set up context
        ctx = SEALContext::Create(parms);

        print_parameters(ctx);
        cout << endl;

        cout << ctx->parameters_set() << endl;

        cout << "Importing/Generating keys..." << endl;

        KeyGenerator keygen(ctx);

        ifstream fsPriv("priv.key", ios::binary);
        ifstream fsPub("pub.key", ios::binary);
        ifstream fsRelin("relin.key", ios::binary);
        ifstream fsGal("gal.key", ios::binary);
        if (fsPriv.good() && fsPub.good() && fsRelin.good() && fsGal.good()) {
            secretKey.load(ctx, fsPriv);
            publicKey.load(ctx, fsPub);
            relinKeys.load(ctx, fsRelin);
            galoisKeys.load(ctx, fsGal);
```

```
        }
        else {
            // Create Keys
            KeyGenerator keygen(ctx);
            publicKey = keygen.public_key();
            secretKey = keygen.secret_key();
            relinKeys = keygen.relin_keys();
            vector<int> steps;
            for (int i = 1; i < 30; i++) {
                steps.push_back(i);
            }
            steps.push_back(32);

            galoisKeys = keygen.galois_keys(steps);

            ofstream fsPrivO("priv.key", ios::binary);
            ofstream fsPubO("pub.key", ios::binary);
            ofstream fsRelinO("relin.key", ios::binary);
            ofstream fsGalO("gal.key", ios::binary);
            secretKey.save(fsPrivO);
            publicKey.save(fsPubO);
            relinKeys.save(fsRelinO);
            galoisKeys.save(fsGalO);
            fsPrivO.close();
            fsPubO.close();
            fsRelinO.close();
            fsGalO.close();
        }
        fsPriv.close();
        fsPub.close();
        fsRelin.close();
        fsGal.close();

        cout << "DONE" << endl;

        string filename = "D:\\Documents\\University Files\\Year 4\\Thesis\\dev\\
KerasMLModel\\model1t0.04clip1\\realCoefficients\\weights1.csv";
        importMatrix(filename, ',' , weights1);

        filename = "D:\\Documents\\University Files\\Year 4\\Thesis\\dev\\KerasML
Model\\model1t0.04clip1\\realCoefficients\\biases1.csv";
        importVector(filename, ',', biases1);
```

```
        filename = "D:\\Documents\\University Files\\Year 4\\Thesis\\dev\\KerasML
Model\\model1t0.04clip1\\realCoefficients\\weights2.csv";
        importVector(filename, ',', weights2);

        filename = "D:\\Documents\\University Files\\Year 4\\Thesis\\dev\\KerasML
Model\\model1t0.04clip1\\realCoefficients\\biases2.csv";
        importVector(filename, ',', biases2);

        filename = "D:\\Documents\\University Files\\Year 4\\Thesis\\dev\\KerasML
Model\\model1t0.04clip1\\realCoefficients\\coef.csv";
        importVector(filename, ',', coef);

        filename = "D:\\Documents\\University Files\\Year 4\\Thesis\\dev\\KerasML
Model\\model1t0.04clip1\\realCoefficients\\test_features.csv";
        importMatrix(filename, ',', test_features);

        filename = "D:\\Documents\\University Files\\Year 4\\Thesis\\dev\\KerasML
Model\\model1t0.04clip1\\realCoefficients\\test_labels.csv";
        importVector(filename, ',', test_labels);

        filename = "D:\\Documents\\University Files\\Year 4\\Thesis\\dev\\KerasML
Model\\model1t0.04clip1\\realCoefficients\\plainRes.csv";
        importVector(filename, ',', npRes);

        cout << "DONE INIT" << endl;
    }

    void importMatrix(const std::string& filename, char sep, vector<vector<double
>>& output)
    {
        std::ifstream src(filename);

        if (!src)
        {
            std::cerr << "\aError opening file.\n\n";
            exit(EXIT_FAILURE);
        }
        string buffer;
        while(getline(src, buffer))
        {
            size_t strpos = 0;
            size_t endpos = buffer.find(sep);
            vector<double> row;
            while (endpos < buffer.length())
```

```
            {
                string numberStr = buffer.substr(strpos, endpos - strpos);
                double number = stod(numberStr);
                row.push_back(number);
                strpos = endpos + 1;
                endpos = buffer.find(sep, strpos);
            }
            string numberStr = buffer.substr(strpos);
            double number = stod(numberStr);
            row.push_back(number);
            output.push_back(row);
        }
    }

    void importVector(const std::string& filename, char sep, vector<double>& outp
ut)
    {
        std::ifstream src(filename);

        if (!src)
        {
            std::cerr << "\aError opening file.\n\n";
            exit(EXIT_FAILURE);
        }
        string buffer;
        while (getline(src, buffer))
        {
            string numberStr = buffer.substr(0);
            double number = stod(numberStr);
            output.push_back(number);
        }
    }

    void compute_all_powers(const Ciphertext &ctx, int degree, Evaluator &evaluat
or, RelinKeys &relin_keys, vector<Ciphertext> &powers) {

        powers.resize(degree + 1);
        powers[1] = ctx;

        vector<int> levels(degree + 1, 0);
        levels[1] = 0;
        levels[0] = 0;

        for (int i = 2; i <= degree; i++) {
```

```
            // compute x^i
            int minlevel = i;
            int cand = -1;
            for (int j = 1; j <= i / 2; j++) {
                int k = i - j;
                //
                int newlevel = max(levels[j], levels[k]) + 1;
                if (newlevel < minlevel) {
                    cand = j;
                    minlevel = newlevel;
                }
            }
            levels[i] = minlevel;
            // use cand
            if (cand < 0) throw runtime_error("error");
            //cout << "levels " << i << " = " << levels[i] << endl;
            // cand <= i - cand by definition
            Ciphertext temp = powers[cand];
            evaluator.mod_switch_to_inplace(temp, powers[i - cand].parms_id());

            evaluator.multiply(temp, powers[i - cand], powers[i]);
            evaluator.relinearize_inplace(powers[i], relin_keys);
            evaluator.rescale_to_next_inplace(powers[i]);
        }
        return;
    }

    void dotProductPlain(Plaintext &ptxt, Ciphertext &input, int dim,
        Evaluator &evaluator, int slotsCount, Ciphertext &destination) {

        cout << "Beginning dot product evaluation..." << endl;

        auto t1 = std::chrono::high_resolution_clock::now();

        // Perform plain-cipher multiplication
        evaluator.mod_switch_to_inplace(ptxt, input.parms_id());
        evaluator.multiply_plain(input, ptxt, destination);
        evaluator.rescale_to_next_inplace(destination);
        destination.scale() = scale;

        // Rotate and add destination vector to get dot product
        Ciphertext temp;
        for (size_t i = 1; i <= 32; i<<=1) {
            evaluator.rotate_vector(destination, i, galoisKeys, temp);
```

```
            if (i == 0) {
                destination = temp;
            }
            else {
                evaluator.add_inplace(destination, temp);
            }
        }

        auto t2 = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(t2
- t1).count();
        cout << "Time it took to run dotProductPlain (microSec): " + to_string(du
ration) << endl;

        return;
    }

    void matmul(vector<Plaintext> ptxt_diag, Ciphertext ct_v, Ciphertext &enc_res
ult, int dim, Evaluator &evaluator) {

        cout << "Beginning matmul Operation..." << endl;

        auto t1 = std::chrono::high_resolution_clock::now();

        // Perform the multiplication
        Ciphertext temp;
        for (int i = 0; i < dim; i++) {
            // rotate
            evaluator.rotate_vector(ct_v, i, galoisKeys, temp);
            // multiply
            evaluator.multiply_plain_inplace(temp, ptxt_diag[i]);
            evaluator.rescale_to_next_inplace(temp);
            temp.scale() = scale;
            if (i == 0) {
                enc_result = temp;
            }
            else {
                evaluator.add_inplace(enc_result, temp);
            }
        }

        auto t2 = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>(t2
- t1).count();
```

```
        cout << "Time it took to run matmul (microSec): " + to_string(duration) <
< endl;

        return;
    }

    void encodeMatrixIntoDiag(vector<Plaintext> &ptxt_diag, CKKSEncoder &encoder,
 int dim, vector<vector<double>> M, double scale) {
        // Encode the diagonals
        for (int i = 0; i < dim; i++) {
            vector<double> diag(dim);
            for (int j = 0; j < dim; j++) {
                diag[j] = M[j][(j + i) % dim];
            }
            encoder.encode(diag, scale, ptxt_diag[i]);
        }
        return;
    }

    void polyeval_tree(int degree, Ciphertext &enc_result, Ciphertext &ctx, CKKSE
ncoder &encoder, Encryptor &encryptor,
        vector<double> &coeffs, Evaluator &evaluator, Decryptor &decryptor) {

        cout << "Beginning polyeval Operation..." << endl;

        chrono::high_resolution_clock::time_point time_start, time_end;
        chrono::microseconds time_diff;

        vector<Plaintext> plain_coeffs(degree + 1);

        //cout << "Poly = ";
        for (size_t i = 0; i < degree + 1; i++) {
            // coeffs[i] = (double)rand() / RAND_MAX;
            encoder.encode(coeffs[i], scale, plain_coeffs[i]);
            /*vector<double> tmp;
            encoder.decode(plain_coeffs[i], tmp);
            cout << "Real = " << coeffs[i] << ", " << endl;
            cout << "Decoded = " << tmp[0] << ", " << endl;*/
        }
        //cout << endl;
        //cout << "encryption done " << endl;

        // compute all powers
        vector<Ciphertext> powers(degree + 1);
```

```
        time_start = chrono::high_resolution_clock::now();

        compute_all_powers(ctx, degree, evaluator, relinKeys, powers);
        //cout << "All powers computed " << endl;

        // result =a[0]
        encryptor.encrypt(plain_coeffs[0], enc_result);

        //for (int i = 1; i <= degree; i++){
        //  decryptor.decrypt(powers[i], plain_result);
        //  encoder.decode(plain_result, result);
        //  cout << "power  = " << result[0] << endl;
        //}

        // result += a[i]*x[i]
        for (int i = 1; i <= degree; i++) {
            // Even coeff (except 0 and 2) are zero, continue to next to avoid tr
ansparent ctx
            if (plain_coeffs[i].is_zero()) {
                continue;
            }
            // Continue with algo
            //cout << i << "-th sum started" << endl;
            Ciphertext temp;
            evaluator.mod_switch_to_inplace(plain_coeffs[i], powers[i].parms_id()
);
            evaluator.multiply_plain(powers[i], plain_coeffs[i], temp);
            evaluator.rescale_to_next_inplace(temp);
            //cout << "got here " << endl;
            evaluator.mod_switch_to_inplace(enc_result, temp.parms_id());
            enc_result.scale() = scale;
            temp.scale() = scale;
            evaluator.add_inplace(enc_result, temp);
            //cout << i << "-th sum done" << endl;
        }
        time_end = chrono::high_resolution_clock::now();
        time_diff = chrono::duration_cast<chrono::microseconds>(time_end - time_s
tart);
        cout << "Done Polyeval [" << time_diff.count() << " microseconds]" << end
l;
    }
};
```

```
void testModel() {
    Model mdl;

    CKKSEncoder encoder(mdl.ctx);
    Encryptor encryptor(mdl.ctx, mdl.publicKey);
    Decryptor decryptor(mdl.ctx, mdl.secretKey);
    Evaluator evaluator(mdl.ctx);

    int dim = 29;
    int degree = mdl.degree;

    Plaintext plain_result;
    vector<double> result;
    vector<double> predictions(mdl.test_labels.size(), -1);

    double loss = 0;
    int fp = 0;
    int tp = 0;
    int fn = 0;
    int tn = 0;
    double avgCalcErr = 0;
    double avgActErr = 0;

    // encode into diagonals
    vector<Plaintext> ptxt_diag(dim);
    mdl.encodeMatrixIntoDiag(ptxt_diag, encoder, dim, mdl.weights1, mdl.scale);

    auto t1 = std::chrono::high_resolution_clock::now();

    for (int s = 0; s < mdl.test_features.size(); s++) {

        cout << "------------------------- Starting s = " << s << " prediction -
------------------------" << endl;

        vector<double> v = mdl.test_features[s];

        // Plaintext computation
        vector<double> resReal(dim, 0);
        for (int i = 0; i < mdl.weights1.size(); i++) {
            for (int j = 0; j < dim; j++) {
                resReal[i] += mdl.weights1[i][j] * v[j];
            }
            resReal[i] += mdl.biases1[i];
        }
```

```
        // repeat v throughout plaintext
        Plaintext ptxt_vec;
        vector<double> vrep(encoder.slot_count());
        for (int i = 0; i < vrep.size(); i++) vrep[i] = v[i % v.size()];
        encoder.encode(vrep, mdl.scale, ptxt_vec);

        // encrypt v
        Ciphertext ctv;
        encryptor.encrypt(ptxt_vec, ctv);

        cout << "Current depth = " << ctv.coeff_mod_count() << endl;

        // Perform matmul of ctv and diag plaintext vector
        Ciphertext enc_result;
        mdl.matmul(ptxt_diag, ctv, enc_result, dim, evaluator);

        cout << "Current depth = " << enc_result.coeff_mod_count() << endl;

        // Add bias 1 to answer
        Plaintext biases1_plain;
        encoder.encode(mdl.biases1, mdl.scale, biases1_plain);
        evaluator.mod_switch_to_inplace(biases1_plain, enc_result.parms_id());
        evaluator.add_plain_inplace(enc_result, biases1_plain);

        // Verify result
        //decryptor.decrypt(enc_result, plain_result);
        //encoder.decode(plain_result, result);

        //for (int i = 0; i < dim; i++) {
        //  cout << "actual: " << result[i] << ", expected: " << resReal[i] << en
dl;
        //}

        // Go through first kernel
        Ciphertext result_ctxt;
        mdl.polyeval_tree(degree, result_ctxt, enc_result, encoder, encryptor, md
l.coef, evaluator, decryptor);

        cout << "Current depth = " << result_ctxt.coeff_mod_count() << endl;

        // Verify result
        vector<double> expected_result(dim, 0);
        for (int k = 0; k < dim; k++) {
```

```
            double expected_result_sing = mdl.coef[degree];
            for (int i = degree - 1; i >= 0; i--) {
                expected_result_sing *= resReal[k];
                expected_result_sing += mdl.coef[i];
            }
            expected_result[k] = expected_result_sing;
            //cout << "Expected Result[" << k << "] = " << expected_result[k] <<
endl;
        }

        //decryptor.decrypt(result_ctxt, plain_result);
        //encoder.decode(plain_result, result);

        /*for (int i = 0; i < dim; i++) {
            cout << "Actual : " << result[i] << ", Expected : " << expected_resul
t[i] << ", diff : " << abs(result[i] - expected_result[i]) << endl;
        }*/

        // Go through second layer dot product

        double dotProduct = 0;
        for (int i = 0; i < dim; i++) {
            dotProduct += expected_result[i] * mdl.weights2[i];
        }
        dotProduct += mdl.biases2[0];

        Plaintext weights2_plain;
        encoder.encode(mdl.weights2, mdl.scale, weights2_plain);
        Ciphertext destination;
        mdl.dotProductPlain(weights2_plain, result_ctxt, dim, evaluator, encoder.
slot_count(), destination);

        cout << "Current depth = " << destination.coeff_mod_count() << endl;

        // Add bias 2
        Plaintext biases2_plain;
        encoder.encode(mdl.biases2, mdl.scale, biases2_plain);
        evaluator.mod_switch_to_inplace(biases2_plain, destination.parms_id());
        evaluator.add_plain_inplace(destination, biases2_plain);

        // Verify result
        /*decryptor.decrypt(destination, plain_result);
        encoder.decode(plain_result, result);
```

```
        cout << "Actual : " << result[0] << ", Expected : " << dotProduct << ", d
iff : " << abs(result[0] - dotProduct) << endl;*/

        double finalClass = mdl.coef[degree];
        for (int i = degree - 1; i >= 0; i--) {
            finalClass *= dotProduct;
            finalClass += mdl.coef[i];
        }

        // Go through last kernel
        Ciphertext finalResult;
        mdl.polyeval_tree(degree, finalResult, destination, encoder, encryptor, m
dl.coef, evaluator, decryptor);

        cout << "Current depth = " << finalResult.coeff_mod_count() << endl;

        // Verify result
        decryptor.decrypt(finalResult, plain_result);
        encoder.decode(plain_result, result);

        //cout << "Actual final class = " << result[0] << ", Expected final class
 = " << finalClass << endl;

        predictions[s] = result[0];

        double diff = abs(predictions[s] - mdl.test_labels[s]);
        loss += pow(diff, 2);
        avgCalcErr += abs(predictions[s] - finalClass);

        bool isFraud;
        if (predictions[s] < 0.5) isFraud = false;
        if (predictions[s] >= 0.5) isFraud = true;

        if (mdl.test_labels[s] == 0 && !isFraud) {
            tn += 1;
        }
        if (mdl.test_labels[s] == 0 && isFraud) {
            fp += 1;
        }
        if (mdl.test_labels[s] == 1 && !isFraud) {
            fn += 1;
        }
        if (mdl.test_labels[s] == 1 && isFraud) {
            tp += 1;
```

```
        }

        avgActErr += abs(mdl.npRes[s] - predictions[s]);

        cout << "Ciphertext Prediction = " << predictions[s] << endl;
        cout << "Plaintext Prediction = " << finalClass << endl;
        cout << "Numpy Prediction = " << mdl.npRes[s] << endl;
        cout << "Real Label = " << mdl.test_labels[s] << endl;
        cout << "Difference between Ciphertext Prediction and Plaintext Predictio
n = " << abs(predictions[s] - finalClass) << endl;
        cout << "Difference between Ciphertext Prediction and Numpy Prediction =
" << abs(predictions[s] - mdl.npRes[s]) << endl;
        cout << "Difference between Ciphertext Prediction and Real Label = " << d
iff << endl;
        cout << "Loss = " << loss << endl;
    }

    auto t2 = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1
).count();
    cout << endl;
    cout << "-------------------------------------------------------------------
---------------------------" << endl;
    cout << endl;
    cout << "Time it took to run 100 predictions (Seconds): " + to_string(duratio
n/1000000) << endl;
    cout << "Final Loss = " << loss / mdl.test_labels.size() << endl;
    cout << "Legitimate Transactions Detected (True Negatives) = " << tn << endl;
    cout << "Legitimate Transactions Incorrectly Detected (False Positives) = " <
< fp << endl;
    cout << "Fraudulent Transactions Missed (False Negatives) = " << fn << endl;
    cout << "Fraudulent Transactions Detected (True Positives) = " << tp << endl;
    cout << "Precision = " << tp / (tp + fp + 0.00001) << endl;
    cout << "Recall = " << tp / (tp + fn + 0.00001) << endl;
    cout << "Average SEAL Plain to Cipher Calculation Error = " << avgCalcErr / m
dl.test_labels.size() << endl;
    cout << "Average Numpy to SEAL Calculation Error = " << avgActErr / mdl.test_
labels.size() << endl;
    cout << endl;
    cout << "-------------------------------------------------------------------
---------------------------" << endl;
    cout << endl;

}
```

Appendix F: FHE SEAL Inference Results (Negative Predictions)

```
Difference between Ciphertext Prediction and Numpy Prediction = 0.255212
Difference between Ciphertext Prediction and Real Label = 0.280979
Loss = 60.2573
------------------------- Starting s = 972 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 11937135
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2238769 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 726165
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [960170 microseconds]
Current depth = 2
Ciphertext Prediction = 0.412217
Plaintext Prediction = 0.412204
Numpy Prediction = 0.01442
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 1.34913e-05
Difference between Ciphertext Prediction and Numpy Prediction = 0.397797
Difference between Ciphertext Prediction and Real Label = 0.412217
Loss = 60.4272
------------------------- Starting s = 973 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 11786688
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2295578 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 754892
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [994708 microseconds]
Current depth = 2
Ciphertext Prediction = 0.0304762
Plaintext Prediction = 0.0304738
```

```
Numpy Prediction = 0.0417877
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 2.32601e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.0113115
Difference between Ciphertext Prediction and Real Label = 0.0304762
Loss = 60.4281
------------------------- Starting s = 974 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 11937786
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2285506 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 801270
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [1022330 microseconds]
Current depth = 2
Ciphertext Prediction = 0.152143
Plaintext Prediction = 0.152138
Numpy Prediction = 0.0245971
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 4.80988e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.127546
Difference between Ciphertext Prediction and Real Label = 0.152143
Loss = 60.4513
------------------------- Starting s = 975 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 12234406
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2448172 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 783316
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [1043678 microseconds]
Current depth = 2
Ciphertext Prediction = 0.0458292
Plaintext Prediction = 0.0458263
```

```
Numpy Prediction = 0.0238633
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 2.944e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.0219659
Difference between Ciphertext Prediction and Real Label = 0.0458292
Loss = 60.4534
------------------------- Starting s = 976 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 12745549
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2387700 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 769474
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [981776 microseconds]
Current depth = 2
Ciphertext Prediction = 0.0329463
Plaintext Prediction = 0.0329457
Numpy Prediction = 0.0139195
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 5.82517e-07
Difference between Ciphertext Prediction and Numpy Prediction = 0.0190268
Difference between Ciphertext Prediction and Real Label = 0.0329463
Loss = 60.4545
------------------------- Starting s = 977 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 12057811
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2324733 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 757932
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [1048266 microseconds]
Current depth = 2
Ciphertext Prediction = 0.074457
Plaintext Prediction = 0.074455
```

```
Numpy Prediction = 0.359738
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 1.94336e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.285281
Difference between Ciphertext Prediction and Real Label = 0.074457
Loss = 60.46
------------------------- Starting s = 978 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 11851790
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2264234 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 748921
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [971265 microseconds]
Current depth = 2
Ciphertext Prediction = 0.14813
Plaintext Prediction = 0.148126
Numpy Prediction = 0.100885
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 4.53555e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.0472448
Difference between Ciphertext Prediction and Real Label = 0.14813
Loss = 60.4819
------------------------- Starting s = 979 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 11960749
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2316217 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 769973
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [1014733 microseconds]
Current depth = 2
Ciphertext Prediction = 0.0695718
Plaintext Prediction = 0.06957
```

```
Numpy Prediction = 0.0105759
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 1.73069e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.0589958
Difference between Ciphertext Prediction and Real Label = 0.0695718
Loss = 60.4868
------------------------- Starting s = 980 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 12141134
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2315057 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 733033
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [972151 microseconds]
Current depth = 2
Ciphertext Prediction = 0.0680089
Plaintext Prediction = 0.0680045
Numpy Prediction = 0.0410553
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 4.41141e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.0269536
Difference between Ciphertext Prediction and Real Label = 0.0680089
Loss = 60.4914
------------------------- Starting s = 981 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 12162907
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2324935 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 746459
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [1009735 microseconds]
Current depth = 2
Ciphertext Prediction = 0.0143096
Plaintext Prediction = 0.0143093
```

```
Numpy Prediction = 0.0547593
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 3.4105e-07
Difference between Ciphertext Prediction and Numpy Prediction = 0.0404496
Difference between Ciphertext Prediction and Real Label = 0.0143096
Loss = 60.4916
------------------------- Starting s = 982 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 12065746
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2371164 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 800326
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [993646 microseconds]
Current depth = 2
Ciphertext Prediction = 0.0793696
Plaintext Prediction = 0.0793647
Numpy Prediction = 0.00713859
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 4.81647e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.072231
Difference between Ciphertext Prediction and Real Label = 0.0793696
Loss = 60.4979
------------------------- Starting s = 983 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 12024840
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2221388 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 706785
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [954010 microseconds]
Current depth = 2
Ciphertext Prediction = 0.022526
Plaintext Prediction = 0.0225256
```

```
Numpy Prediction = 0.269934
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 4.06172e-07
Difference between Ciphertext Prediction and Numpy Prediction = 0.247408
Difference between Ciphertext Prediction and Real Label = 0.022526
Loss = 60.4984

--------------------------------------------------------------------------------
--------------

Time it took to run 984 predictions (Seconds): 15849
Final Loss = 0.0614821
Legitimate Transactions Detected (True Negatives) = 445
Legitimate Transactions Incorrectly Detected (False Positives) = 47
Fraudulent Transactions Missed (False Negatives) = 32
Fraudulent Transactions Detected (True Positives) = 460
Precision = 0.907298
Recall = 0.934959
Average SEAL Plain to Cipher Calculation Error = 5.32709e-06
Average Numpy to SEAL Calculation Error = 0.163014

--------------------------------------------------------------------------------
--------------

+-----------------------------------------------------------+
| The following examples should be executed while reading |
| comments in associated files in native/examples/.       |
+-----------------------------------------------------------+
| Examples                   | Source Files                 |
+----------------------------+------------------------------+
| 1. BFV Basics              | 1_bfv_basics.cpp             |
| 2. Encoders                | 2_encoders.cpp               |
| 3. Levels                  | 3_levels.cpp                 |
| 4. CKKS Basics             | 4_ckks_basics.cpp            |
| 5. Rotation                | 5_rotation.cpp               |
| 6. Performance Test        | 6_performance.cpp            |
| 7. FHELR Model             | model.cpp                    |
+----------------------------+------------------------------+
[   3355 MB] Total allocation from the memory pool

> Run example (1 ~ 7) or exit (0):
```

Appendix G: FHE SEAL Inference Results (Positive Predictions)

```
Plaintext Prediction = 0.97565
Numpy Prediction = 0.980555
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 3.75463e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.00490866
Difference between Ciphertext Prediction and Real Label = 0.0243539
Loss = 21.978
-------------------------- Starting s = 457 prediction --------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 10490179
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [1951319 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 662982
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [851386 microseconds]
Current depth = 2
Ciphertext Prediction = 0.817324
Plaintext Prediction = 0.817307
Numpy Prediction = 0.797462
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 1.71486e-05
Difference between Ciphertext Prediction and Numpy Prediction = 0.0198624
Difference between Ciphertext Prediction and Real Label = 0.182676
Loss = 22.0114
-------------------------- Starting s = 458 prediction --------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 10386931
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [1951575 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 653026
Current depth = 6
Beginning polyeval Operation...
```

```
Done Polyeval [851575 microseconds]
Current depth = 2
Ciphertext Prediction = 0.978414
Plaintext Prediction = 0.978416
Numpy Prediction = 0.975507
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 1.98863e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.00290757
Difference between Ciphertext Prediction and Real Label = 0.0215858
Loss = 22.0118
------------------------- Starting s = 459 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 10449486
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2206964 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 653517
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [900136 microseconds]
Current depth = 2
Ciphertext Prediction = 0.965485
Plaintext Prediction = 0.965479
Numpy Prediction = 0.450994
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 5.78103e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.514491
Difference between Ciphertext Prediction and Real Label = 0.0345152
Loss = 22.013
------------------------- Starting s = 460 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 10472966
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [1970683 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 671808
Current depth = 6
Beginning polyeval Operation...
```

```
Done Polyeval [870022 microseconds]
Current depth = 2
Ciphertext Prediction = 0.977909
Plaintext Prediction = 0.977911
Numpy Prediction = 0.979473
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 2.46536e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.00156397
Difference between Ciphertext Prediction and Real Label = 0.0220912
Loss = 22.0135
------------------------- Starting s = 461 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 10753729
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [1952283 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 660891
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [846599 microseconds]
Current depth = 2
Ciphertext Prediction = 0.796765
Plaintext Prediction = 0.796748
Numpy Prediction = 0.978176
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 1.70308e-05
Difference between Ciphertext Prediction and Numpy Prediction = 0.181412
Difference between Ciphertext Prediction and Real Label = 0.203235
Loss = 22.0548
------------------------- Starting s = 462 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 10320807
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [1978803 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 662504
Current depth = 6
Beginning polyeval Operation...
```

```
Done Polyeval [848724 microseconds]
Current depth = 2
Ciphertext Prediction = 0.956341
Plaintext Prediction = 0.956349
Numpy Prediction = 0.980663
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 8.125e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.0243217
Difference between Ciphertext Prediction and Real Label = 0.0436591
Loss = 22.0567
------------------------- Starting s = 463 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 10477603
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [1965638 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 730272
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [897746 microseconds]
Current depth = 2
Ciphertext Prediction = 0.743395
Plaintext Prediction = 0.743377
Numpy Prediction = 0.835145
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 1.73704e-05
Difference between Ciphertext Prediction and Numpy Prediction = 0.0917503
Difference between Ciphertext Prediction and Real Label = 0.256605
Loss = 22.1226
------------------------- Starting s = 464 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 11972778
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [1961633 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 658415
Current depth = 6
Beginning polyeval Operation...
```

```
Done Polyeval [853379 microseconds]
Current depth = 2
Ciphertext Prediction = 0.951036
Plaintext Prediction = 0.951027
Numpy Prediction = 0.972658
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 8.4448e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.0216227
Difference between Ciphertext Prediction and Real Label = 0.0489643
Loss = 22.125
------------------------- Starting s = 465 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 10454077
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [1990539 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 674760
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [865568 microseconds]
Current depth = 2
Ciphertext Prediction = 0.971467
Plaintext Prediction = 0.971463
Numpy Prediction = 0.974267
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 4.26612e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.00279956
Difference between Ciphertext Prediction and Real Label = 0.0285328
Loss = 22.1258
------------------------- Starting s = 466 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 10538722
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2124188 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 713913
Current depth = 6
Beginning polyeval Operation...
```

```
Done Polyeval [891349 microseconds]
Current depth = 2
Ciphertext Prediction = 0.962988
Plaintext Prediction = 0.962982
Numpy Prediction = 0.977086
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 6.25975e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.0140984
Difference between Ciphertext Prediction and Real Label = 0.0370121
Loss = 22.1271
------------------------- Starting s = 467 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 11188230
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2125788 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 722577
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [861695 microseconds]
Current depth = 2
Ciphertext Prediction = 0.621329
Plaintext Prediction = 0.621312
Numpy Prediction = 0.978914
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 1.76326e-05
Difference between Ciphertext Prediction and Numpy Prediction = 0.357585
Difference between Ciphertext Prediction and Real Label = 0.378671
Loss = 22.2705
------------------------- Starting s = 468 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 11459081
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2213933 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 737423
Current depth = 6
Beginning polyeval Operation...
```

```
Done Polyeval [909802 microseconds]
Current depth = 2
Ciphertext Prediction = 0.979042
Plaintext Prediction = 0.979043
Numpy Prediction = 0.980457
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 7.74899e-07
Difference between Ciphertext Prediction and Numpy Prediction = 0.0014151
Difference between Ciphertext Prediction and Real Label = 0.0209582
Loss = 22.271
------------------------- Starting s = 469 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 10920188
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [2136804 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 656652
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [866386 microseconds]
Current depth = 2
Ciphertext Prediction = 0.24442
Plaintext Prediction = 0.244412
Numpy Prediction = 0.980616
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 7.82752e-06
Difference between Ciphertext Prediction and Numpy Prediction = 0.736196
Difference between Ciphertext Prediction and Real Label = 0.75558
Loss = 22.8419
------------------------- Starting s = 470 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 11423971
Current depth = 11
Beginning polyeval Operation...
```

Appendix H: Optimized FHE SEAL Inference Results (Negative Predictions)

```
Difference between Ciphertext Prediction and Numpy Prediction = 0.262946
Difference between Ciphertext Prediction and Real Label = 0.288713
Loss = 60.2801
------------------------- Starting s = 972 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5063212
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [994251 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 314507
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [405161 microseconds]
Current depth = 2
Ciphertext Prediction = 0.411784
Plaintext Prediction = 0.412204
Numpy Prediction = 0.01442
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 0.000419555
Difference between Ciphertext Prediction and Numpy Prediction = 0.397364
Difference between Ciphertext Prediction and Real Label = 0.411784
Loss = 60.4497
------------------------- Starting s = 973 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5072803
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [983855 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 326281
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [440421 microseconds]
Current depth = 2
Ciphertext Prediction = 0.0319602
Plaintext Prediction = 0.0304738
```

```
Numpy Prediction = 0.0417877
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 0.00148635
Difference between Ciphertext Prediction and Numpy Prediction = 0.00982746
Difference between Ciphertext Prediction and Real Label = 0.0319602
Loss = 60.4507
------------------------- Starting s = 974 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5085640
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [954132 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 320712
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [407252 microseconds]
Current depth = 2
Ciphertext Prediction = 0.15086
Plaintext Prediction = 0.152138
Numpy Prediction = 0.0245971
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 0.00127862
Difference between Ciphertext Prediction and Numpy Prediction = 0.126262
Difference between Ciphertext Prediction and Real Label = 0.15086
Loss = 60.4735
------------------------- Starting s = 975 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5117073
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [986691 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 320654
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [409943 microseconds]
Current depth = 2
Ciphertext Prediction = 0.0477566
Plaintext Prediction = 0.0458263
```

```
Numpy Prediction = 0.0238633
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 0.00193028
Difference between Ciphertext Prediction and Numpy Prediction = 0.0238933
Difference between Ciphertext Prediction and Real Label = 0.0477566
Loss = 60.4757
------------------------- Starting s = 976 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5122862
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [994109 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 318934
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [404475 microseconds]
Current depth = 2
Ciphertext Prediction = 0.0322312
Plaintext Prediction = 0.0329457
Numpy Prediction = 0.0139195
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 0.000714503
Difference between Ciphertext Prediction and Numpy Prediction = 0.0183117
Difference between Ciphertext Prediction and Real Label = 0.0322312
Loss = 60.4768
------------------------- Starting s = 977 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5085198
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [945324 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 312984
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [415680 microseconds]
Current depth = 2
Ciphertext Prediction = 0.073328
Plaintext Prediction = 0.074455
```

```
Numpy Prediction = 0.359738
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 0.001127
Difference between Ciphertext Prediction and Numpy Prediction = 0.28641
Difference between Ciphertext Prediction and Real Label = 0.073328
Loss = 60.4822
------------------------- Starting s = 978 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5124530
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [967458 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 316878
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [406439 microseconds]
Current depth = 2
Ciphertext Prediction = 0.146856
Plaintext Prediction = 0.148126
Numpy Prediction = 0.100885
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 0.00126947
Difference between Ciphertext Prediction and Numpy Prediction = 0.0459708
Difference between Ciphertext Prediction and Real Label = 0.146856
Loss = 60.5037
------------------------- Starting s = 979 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5154945
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [967471 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 320734
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [426423 microseconds]
Current depth = 2
Ciphertext Prediction = 0.068469
Plaintext Prediction = 0.06957
```

```
Numpy Prediction = 0.0105759
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 0.00110101
Difference between Ciphertext Prediction and Numpy Prediction = 0.0578931
Difference between Ciphertext Prediction and Real Label = 0.068469
Loss = 60.5084
------------------------- Starting s = 980 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5105538
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [948246 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 313213
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [407054 microseconds]
Current depth = 2
Ciphertext Prediction = 0.0704509
Plaintext Prediction = 0.0680045
Numpy Prediction = 0.0410553
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 0.00244634
Difference between Ciphertext Prediction and Numpy Prediction = 0.0293955
Difference between Ciphertext Prediction and Real Label = 0.0704509
Loss = 60.5134
------------------------- Starting s = 981 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5073367
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [987909 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 339154
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [411153 microseconds]
Current depth = 2
Ciphertext Prediction = 0.0140308
Plaintext Prediction = 0.0143093
```

```
Numpy Prediction = 0.0547593
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 0.00027848
Difference between Ciphertext Prediction and Numpy Prediction = 0.0407285
Difference between Ciphertext Prediction and Real Label = 0.0140308
Loss = 60.5136
------------------------- Starting s = 982 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5121034
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [995379 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 317620
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [411315 microseconds]
Current depth = 2
Ciphertext Prediction = 0.0820576
Plaintext Prediction = 0.0793647
Numpy Prediction = 0.00713859
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 0.00269287
Difference between Ciphertext Prediction and Numpy Prediction = 0.074919
Difference between Ciphertext Prediction and Real Label = 0.0820576
Loss = 60.5203
------------------------- Starting s = 983 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5136697
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [970777 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 318584
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [419861 microseconds]
Current depth = 2
Ciphertext Prediction = 0.0219819
Plaintext Prediction = 0.0225256
```

```
Numpy Prediction = 0.269934
Real Label = 0
Difference between Ciphertext Prediction and Plaintext Prediction = 0.000543675
Difference between Ciphertext Prediction and Numpy Prediction = 0.247952
Difference between Ciphertext Prediction and Real Label = 0.0219819
Loss = 60.5208

--------------------------------------------------------------------------------
--------------

Time it took to run 100 predictions (Seconds): 7571
Final Loss = 0.0615049
Legitimate Transactions Detected (True Negatives) = 445
Legitimate Transactions Incorrectly Detected (False Positives) = 47
Fraudulent Transactions Missed (False Negatives) = 32
Fraudulent Transactions Detected (True Positives) = 460
Precision = 0.907298
Recall = 0.934959
Average SEAL Plain to Cipher Calculation Error = 0.000782457
Average Numpy to SEAL Calculation Error = 0.163019

--------------------------------------------------------------------------------
--------------

+-----------------------------------------------------------+
| The following examples should be executed while reading |
| comments in associated files in native/examples/.       |
+-----------------------------------------------------------+
| Examples                  | Source Files                  |
+---------------------------+-------------------------------+
| 1. BFV Basics             | 1_bfv_basics.cpp              |
| 2. Encoders               | 2_encoders.cpp                |
| 3. Levels                 | 3_levels.cpp                  |
| 4. CKKS Basics            | 4_ckks_basics.cpp             |
| 5. Rotation               | 5_rotation.cpp                |
| 6. Performance Test       | 6_performance.cpp             |
| 7. FHELR Model            | model.cpp                     |
+---------------------------+-------------------------------+
[   1678 MB] Total allocation from the memory pool

> Run example (1 ~ 7) or exit (0):
```

Appendix I: Optimized FHE SEAL Inference Results (Positive Predictions)

```
Numpy Prediction = 0.973551
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 0.000946946
Difference between Ciphertext Prediction and Numpy Prediction = 0.24051
Difference between Ciphertext Prediction and Real Label = 0.266959
Loss = 2.77208
------------------------- Starting s = 43 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5237477
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [983085 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 323063
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [433345 microseconds]
Current depth = 2
Ciphertext Prediction = 0.977789
Plaintext Prediction = 0.978207
Numpy Prediction = 0.98052
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 0.000417779
Difference between Ciphertext Prediction and Numpy Prediction = 0.00273096
Difference between Ciphertext Prediction and Real Label = 0.0222112
Loss = 2.77257
------------------------- Starting s = 44 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5032583
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [948304 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 316662
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [412550 microseconds]
```

```
Current depth = 2
Ciphertext Prediction = 0.950619
Plaintext Prediction = 0.951905
Numpy Prediction = 0.969632
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 0.00128537
Difference between Ciphertext Prediction and Numpy Prediction = 0.0190128
Difference between Ciphertext Prediction and Real Label = 0.0493805
Loss = 2.77501
------------------------- Starting s = 45 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5025814
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [959194 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 318182
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [407643 microseconds]
Current depth = 2
Ciphertext Prediction = 0.977834
Plaintext Prediction = 0.978256
Numpy Prediction = 0.980742
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 0.000421502
Difference between Ciphertext Prediction and Numpy Prediction = 0.002908
Difference between Ciphertext Prediction and Real Label = 0.0221658
Loss = 2.7755
------------------------- Starting s = 46 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5032629
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [962647 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 316690
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [407310 microseconds]
```

```
Current depth = 2
Ciphertext Prediction = 0.95352
Plaintext Prediction = 0.952838
Numpy Prediction = 0.97941
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 0.000682102
Difference between Ciphertext Prediction and Numpy Prediction = 0.02589
Difference between Ciphertext Prediction and Real Label = 0.0464802
Loss = 2.77766
------------------------- Starting s = 47 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 4998205
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [968160 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 320658
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [406243 microseconds]
Current depth = 2
Ciphertext Prediction = 0.968507
Plaintext Prediction = 0.968097
Numpy Prediction = 0.877267
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 0.000409836
Difference between Ciphertext Prediction and Numpy Prediction = 0.0912401
Difference between Ciphertext Prediction and Real Label = 0.0314928
Loss = 2.77865
------------------------- Starting s = 48 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 4997882
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [951259 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 313002
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [408145 microseconds]
```

```
Current depth = 2
Ciphertext Prediction = 0.741784
Plaintext Prediction = 0.740804
Numpy Prediction = 0.977615
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 0.000980812
Difference between Ciphertext Prediction and Numpy Prediction = 0.23583
Difference between Ciphertext Prediction and Real Label = 0.258216
Loss = 2.84533
------------------------- Starting s = 49 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5017547
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [948401 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 328645
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [413369 microseconds]
Current depth = 2
Ciphertext Prediction = 0.919578
Plaintext Prediction = 0.918591
Numpy Prediction = 0.980597
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 0.000986769
Difference between Ciphertext Prediction and Numpy Prediction = 0.0610196
Difference between Ciphertext Prediction and Real Label = 0.0804224
Loss = 2.85179
------------------------- Starting s = 50 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5011029
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [960521 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 319244
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [407526 microseconds]
```

```
Current depth = 2
Ciphertext Prediction = 0.939788
Plaintext Prediction = 0.939004
Numpy Prediction = 0.978038
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 0.000784423
Difference between Ciphertext Prediction and Numpy Prediction = 0.0382499
Difference between Ciphertext Prediction and Real Label = 0.060212
Loss = 2.85542
------------------------- Starting s = 51 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5030317
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [985535 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 343704
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [464028 microseconds]
Current depth = 2
Ciphertext Prediction = 0.948949
Plaintext Prediction = 0.948177
Numpy Prediction = 0.979877
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 0.000771221
Difference between Ciphertext Prediction and Numpy Prediction = 0.030928
Difference between Ciphertext Prediction and Real Label = 0.0510514
Loss = 2.85803
------------------------- Starting s = 52 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5300359
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [1078126 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 321127
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [536530 microseconds]
```

```
Current depth = 2
Ciphertext Prediction = 0.977635
Plaintext Prediction = 0.978071
Numpy Prediction = 0.980893
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 0.000436118
Difference between Ciphertext Prediction and Numpy Prediction = 0.00325873
Difference between Ciphertext Prediction and Real Label = 0.0223654
Loss = 2.85853
------------------------- Starting s = 53 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5195367
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [941393 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 312127
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [406063 microseconds]
Current depth = 2
Ciphertext Prediction = 0.9746
Plaintext Prediction = 0.975238
Numpy Prediction = 0.978038
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 0.000637286
Difference between Ciphertext Prediction and Numpy Prediction = 0.0034376
Difference between Ciphertext Prediction and Real Label = 0.0253997
Loss = 2.85917
------------------------- Starting s = 54 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
Time it took to run matmul (microSec): 5010287
Current depth = 11
Beginning polyeval Operation...
Done Polyeval [944009 microseconds]
Current depth = 7
Beginning dot product evaluation...
Time it took to run dotProductPlain (microSec): 316009
Current depth = 6
Beginning polyeval Operation...
Done Polyeval [420156 microseconds]
```

```
Current depth = 2
Ciphertext Prediction = 0.976289
Plaintext Prediction = 0.976834
Numpy Prediction = 0.979786
Real Label = 1
Difference between Ciphertext Prediction and Plaintext Prediction = 0.000545721
Difference between Ciphertext Prediction and Numpy Prediction = 0.00349793
Difference between Ciphertext Prediction and Real Label = 0.0237115
Loss = 2.85973
------------------------- Starting s = 55 prediction -------------------------
Current depth = 12
Beginning matmul Operation...
```