Homomorphically Trained Neural Networks

by

Samuel Joseph Patrick McGladery Atkins

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

Department of Electrical and Computer Engineering
University of Toronto

Homomorphically Trained Neural Networks

Samuel Joseph Patrick McGladery Atkins

Master of Applied Science

Department of Electrical and Computer Engineering
University of Toronto

2022

Abstract

This thesis has two main contributions. Firstly, we present *Cryptograd*: an autograd-compatible homomorphically encrypted training and inference library built in Python. Using this library, we successfully train a fully encrypted model using fully encrypted training data on the QSAR Andogen receptor dataset. We obtain an accuracy value of 90% and a total training time of 670s on an Intel Core i7-11700K @ 3.60GHz using 12 cores.

Secondly, we introduce a modified privacy-preserving federated learning scheme. At the core of this scheme is DRMSprop: a decentralized optimizer that utilizes a global gradient vector to improve convergence, stability, and performance. This modified scheme yielded an accuracy increase of 1.7% relative to other optimizers on the census income prediction dataset. Furthermore, it lags behind the optimal insecure model by only 2.3%.

Acknowledgements

First and foremost, I would like to thank my thesis advisor, Glenn Gulak, for being so supportive and understanding. Secondly, I would like to thank Hunter and Shaveer for their insight and guidance. I would also like to thank my parents, Scarlett and Grant, my brothers, the rest of my family, Hannah, and my friends. Your kindness and support has given me courage and strength. Thank you all so much.

Table of Contents

List of Tables

List of Figures

List of Symbols

Constants

β	The model parameters
χ	The input matrix
χ^{ext}	The input matrix with a column of ones prepended
Δ	The scaling factor used to scale h in Section 2.2.1
$\frac{dL}{d\beta}$	The derivative of the loss function with respect to the model parameters
Γ	The example message vector encrypted and decrypted in Section 2.2.2
$\hat{\beta}$	An estimate of the model parameters
κ	The κ function transforms a polynomial by raising its input values to the k^{th} power with respect to the polynomial modulus
λ	The security parameter
μ	The learning rate
ν	The momentum constant
π	The mapping function used to perform the complex conjugate transformation of a message vector
σ	In the context of CKKS encryption, σ is the standard deviation of the noise vectors used during PK and SK key generation. In an ML context, σ represents the sigmoid activation function.
$\sigma(m)$	The evaluation of a polynomial, m, on the odd roots (up to N) of the cyclotomic polynomial
evk	The evaluation key
rtk	The rotation key
A	The matrix formulation of the LWE problem in Section 2.2.2
b	The bias term

c	A ciphertext consisting of two components: c_0 and c_1
c_0	The first component of a ciphertext, c
c_1	The second component of a ciphertext, c
c_r	The rotated ciphertext
$E[G]$	The gradient history vector used by RMSprop, Adadelta, and DRMSprop
G	The per-parameter gradient vector tracked by the Adagrad optimizer
h	The extended and complex conjugate transformation of z (Section 2.2.1)
L	The loss or loss function
N	In the context of the CKKS encryption scheme, N refers to the polynomial modulus degree. In other contexts, N can refer to the length of a vector or the size of a dataset.
p	The projection of $\Delta\ \mathbb{H}$ onto $\sigma(R)$
PK	The public key
q	The polynomial modulus
r	The rounded elements of p using coordinate-wise random rounding
SK	The secret key
v_t	The update vector at time-step t
W	The weights in the parameter vector
y_i^{pred}	A prediction on a single data entry
y_i^{true}	The true label of a single data entry ($\in \{0, 1\}$)
Z	The output of the linear layer in an LR modelling context (Section 2.3.3.2)
z	The input message encoded and decoded in Section 2.2.1
z'	The result of encoding z

Abbreviations

API Application Protocol Interface.

BCE Binary Cross-Entropy.

BDI Big Data Institution.

BLUE Lorica Cybersecurity's encryption library.

CKKS Cheon-Kim-Kim-Song approximate arithmetic fully homomorphic encryption scheme.

CNN Convolutional Neural Network.

DAG Directed Acyclic Graph.

DFT Discrete Fourier Transform.

ETL Extract, Transform, Load.

FHE Fully Homomorphic Encryption.

FL Federated Learning.

FN False Negative.

FP False Positive.

GD Gradient Descent.

HE Homomorphic Encryption.

LR Logistic Regression.

LSE Least Squared Error.

LWE Learning With Errors.

MAE Mean Absolute Error.

ML Machine Learning.

MLP Multi-Layer Perceptron.

MSE Mean Squared Error.

NAG Nesterov's Accelerated Gradient.

NN Neural Network.

PET Privacy Enhancing Technology.

PHE Partially Homomorphic Encryption.

PK Public Key.

RLWE Ring Learning With Errors.

RMSE Root Mean Squared Error.

SGD Stochastic Gradient Descent.

SHE Somewhat Homomorphic Encryption.

SK Secret Key.

SMOTE Synthetic Minority Over-sampling TEchnique.

TN True Negative.

TP True Positive.

Chapter 1

Introduction

1.1 Motivation

The popularity of third-party cloud computing services such as Microsoft Azure, Amazon Web Services, Google Cloud, and IBM Cloud has risen dramatically. These services facilitate unlimited data modelling parallelization opportunities while removing the need to instantiate, manage, and maintain physical servers. Unfortunately, many potential users of these third party computation services are prohibited from using them due to privacy contraints. Specifically, these constraints do not allow private data to be exposed in plaintext to third party vendors.

Homomorphic encryption [1] is a novel encryption technique that allows one to perform an arbitrary sequence of additions and multiplications on encrypted data. This concept has lead to a wide variety of parallelized privacy-preserving modelling and machine learning (ML) opportunities. Using homomorphic encryption, users can maintain data privacy requirements and utilize third party cloud computing tools simultaneously.

The attractive qualities of homomorphic encryption are not free, however. Implementing a scheme for homomorphic encryption and implementing models using this type of encryption is challenging. It requires expertise in mathematics, cryptography, data science, and software engineering. Further, while there has been tremendous strides in the world of encrypted ML inference [2–6], there has been little work in the

realm of encrypted ML training [7–9]. This thesis seeks to implement and provide tools for encrypted model training and ML inference. Further, this thesis focuses on encrypted model optimization, specifically during training. Section 1.2 elaborates on these objectives.

1.2 Thesis Objectives

There are two thesis objectives. Firstly, to aid practical users of homomorphic encryption through *Cryptograd*, a homomorphic encryption training and modelling library in Python that utilizes Autograd, the industry standard backpropagation engine. Cryptograd provides a familiar and flexible application protocol interface (API) for rapid homomorphic encryption model development. The second objective is to investigate and iterate on existing federated learning optimization methods to improve model performance. A novel optimization method called DRMSprop is introduced, yielding improved convergence and performance and incurring minor computational and storage costs relative to existing federated learning optimization methods. Section 1.3 provides an overview and outline of this thesis.

1.3 Thesis Overview

Chapter 2 explains the important cryptographic and mathematical concepts needed to understand the contributions of this thesis. Specifically, fully homomorphic encryption, the CKKS encryption scheme, machine learning at a conceptual level, optimization methods, and fully encrypted ML training are discussed. Chapter 3 presents a new library for encrypted training and inference in Python. This library mimics the Sci-Kit Learn API calls to provide users with a familiar machine learning development interface. Chapter 4 presents a new federated learning homomorphic encryption training scheme. This scheme modifies the existing communication pipeline by incorporating a gradient history vector. The optimization method presented in this

chapter yields improved convergence and better performance when compared to SGD, momentum, NAG, Adagrad, and RMSprop. Finally, Chapter 5 summarizes the contributions and main findings of the thesis. This chapter also presents opportunities for future research.

Chapter 2

Background

2.1 Fully Homomorphic Encryption

2.1.1 Privacy-Preserving Machine Learning

The applications of machine learning (ML) models are proliferating. ML models are prevalent in image classification, speech recognition, anomaly detection, and a plethora of other classification tasks. ML models are particularly effective when used to predict data attributes given large amounts of labelled data. Training is the process of iteratively updating the parameters of a model using a designated training dataset until the model converges. Inference, the complement of training, is the practical usage of a trained model on data the model has not seen before to predict the labels of the unseen data. To cost-effectively utilize the predictive power of ML models, cloud computing is desirable as training and inference on large datasets is computationally expensive.

Although end-to-end encryption exists for data while it is transported to the cloud and stored in the cloud, cloud users still must put faith in the discretion of the cloud service as their data is unencrypted during computation. Companies such as banks and hospitals that have access to large amounts of sensitive labelled data are often bound by confidentiality regulations, such as the Health Insurance Portability and Accountability Act (HIPAA), that prevent them from using third party ML tools. Institutions with access to large amounts of data will be referred to as big data

institutions (BDIs).

In 2009, Craig Gentry utilized lattice-based cryptography to construct the first fully homomorphic encryption (FHE) scheme [10]. Fully homomorphic encryption is named as such because it allows users to perform an arbitrary sequence of additions and multiplications on cipher texts. Using only additions and multiplications, approximations of high-performance ML models are possible. Given the security requirements of cloud-based private data processing, FHE has the potential to be a tremendous breakthrough for BDIs looking to take advantage of the predictive power of cloud-based ML tools in the domain of privacy enhancing technologies (PETs).

2.1.2 Fully Homomorphic Implementations

There are three different types of homomorphic encryption: partially homomorphic encryption (PHE), somewhat homomorphic encryption (SHE), and fully homomorphic encryption (FHE). PHE allows only addition or only multiplication to be performed on ciphertexts an arbitrary number of times. SHE supports addition and multiplication, but the number of operations permitted in an SHE scheme is finite. SHE is sometimes referred to as leveled HE. The most desirable of the three, FHE, allows the performance of both addition and multiplication on ciphertexts a semi-infinite number of times.

Gentry introduced a limited SHE scheme in 2009 [10]. The SHE scheme that he introduced was limited because after each multiplication operation between two ciphertexts, the result would become more noisy. Ultimately, the result would become too noisy and render the decrypted result inaccurate. Gentry elucidated, however, that a modified version of the SHE scheme he proposed is bootstrappable. An encryption scheme is "bootstrappable" if the scheme can homomorphically evaluate its own decryption circuit plus one additional NAND gate. Gentry then illustrated that any SHE scheme that is bootstrappable can be recursively self-embedded to make it fully homomorphic. This discovery laid the foundation for the first FHE scheme.

The security of Gentry's FHE scheme was based on ideal lattices. In implementations of Gentry's lattice-based scheme at that time, secure bootstrapping took up to 30 minutes per operation [1]. An integer-based approach [11] that was simpler and more efficient than the previous lattice-based approach was later introduced. Despite the improvement in efficiency, the uses of homomorphic encryption were severely limited by computational efficiency bottlenecks due to the large number of mathematical operations involved. From 2011 to 2016, many variations of FHE schemes followed Gentry's breakthrough. These encryption schemes include the BGV [12], LTV [13], BFV [14], BLLN [15], TFHE [16], and CKKS [17] schemes.

The CKKS scheme is significantly faster than the other schemes because it utilizes a rescaling procedure. During rescaling, the ciphertext is truncated into a smaller modulus. As a result, the decrypted result is rounded and less accurate but more than sufficiently accurate for machine learning. This thesis focuses on encrypted ML models that utilize the CKKS encryption scheme. This scheme is discussed further in Section 2.2.

2.2 The CKKS Encryption Scheme

CKKS is an FHE scheme that treats encryption noise as error accumulated during computations. This scheme is an "approximate arithmetic" scheme, meaning that some precision loss is incurred during ciphertext operations. Since high precision is not a requirement for ML applications, CKKS is a strong choice for encrypted ML applications. In fact, CKKS has been used to implement logistic regression (LR) models of varying size [18–21]. Further, it has been used to implement encrypted neural network training and inference in various contexts [22–24]. There are several open-source HE development libraries that support the CKKS encryption scheme including HElib [25], Palisade [26], SEAL [27], HEAAN [28], and OpenFHE [29].

Plaintexts in CKKS are complex vectors of length $N/2$, where N is a power of 2. Ciphertexts, on the other hand, are pairs of polynomials with coefficients in $\mathbb{Z}_q$.

There are two phases to transform a plaintext message into a ciphertext: encoding and encryption. Encoding and encryption have complementary transformations, namely decoding and decryption, respectively. Encoding and decoding are explained in Section 2.2.1 and encryption and decryption are explained in Section 2.2.2. Finally, homomorphic addition, multiplication, and rotation are explained in Section 2.2.3.

Prior to discussing the mathematics behind CKKS, notation for various transformations must be defined. The "$\odot$" operator will represent the Hadamard product between two vectors and the multiplication of polynomial coefficients. The lack of an operator delineates scalar multiplication (for example, for $a \in \mathbb{R}, B \in \mathbb{R}^n$, $aB = \{aB_0, aB_1, \ldots, aB_n\}$). The "$\cdot$" operator represents matrix multiplication. Finally, the $\langle x, y \rangle$ operation is the dot product between two vectors (for $x, y \in \mathbb{R}^K$, $\langle x, y \rangle = \sum_i^K x_i \times y_i$).

2.2.1 CKKS Encoding and Decoding

2.2.1.1 Encoding

The objective during the encoding phase is to transform an input message, $z \in \mathbb{C}^{N/2}$, into a polynomial with integer coefficients modulo $(X^N + 1)$: $z' \in \mathbb{Z}[X]/(X^N + 1)$. To accomplish this, we apply the transformations detailed below.

1. Transform $z \in \mathbb{C}^{N/2}$ to $h = \pi^{-1}(z) \in \mathbb{H} = e \in \mathbb{C}^N : e_j = \overline{e_{-j}}$

   - π^{-1} takes the complex conjugates of the values in z and extends z such that $h = \pi^{-1}(z)$ is of length N and $h = \{z_1, z_2, ..., z_{N/2}, \overline{z_{N/2}}, ..., \overline{z_2}, \overline{z_1}\}$

2. Multiply each element in h by some positive scaling factor, $\Delta \in \Re^+$

3. Project Δh onto the basis vectors of $\sigma(R)$: $\beta = (\sigma(1), \sigma(X), ..., \sigma(X^{N-1}))$ to obtain $p = \text{proj}_{\sigma(R)} \Delta h$

   - $\text{proj}_b a = \frac{\langle a,b \rangle}{\langle b,b \rangle} b$
   - $\sigma : R = Z[X]/(X^N + 1) \rightarrow \sigma(R)$

- $\sigma(m)$, for some polynomial m, is $(m(\xi), m(\xi^3), ..., m(\xi^{2N-1}))$ where ξ^i is the i^{th} root of the cyclotomic polynomial, $\Phi_{2N}(X) = X^N + 1$ where N is a power of two.

4. Round the elements of p using "coordinate-wise random rounding" [30] denoted by $\zeta : \mathbb{H} \to \sigma(R)$ to obtain $r = \zeta(p)$

5. Encode r using $\sigma : z' = \sigma^{-1}(r) \in R$

   - To apply σ, we evaluate the polynomial, m, on the odd roots of the cyclotomic polynomial: $(m(\xi), m(\xi^3), ..., m(\xi^{2N-1}))$. To apply σ^{-1}, we must find a polynomial, m, such that $\sigma(m) = (m(\xi), m(\xi^3), ..., m(\xi^{2N-1}))$ is equal to the input, $r = (r_1, ..., r_N)$. This problem is defined in Equation 2.1.

$$\sum_{j}^{N-1} \alpha_j \ (\xi^{2i-1})^j = r_i, i = 1, ..., N \tag{2.1}$$

   - This is a linear equation, $V\alpha = r$ with V being the Vandermonde matrix of $(\xi^{2i-1})_{i=1,...,N}$, α being the coefficients of m we are trying to find, and r being the semi-encoded vector we are trying to encode. Solving for α, we obtain the expression shown in Equation 2.2.

$$\alpha = V^{-1}r \tag{2.2}$$

   σ^{-1} computes α and transforms r into an encoded message vector, m, with α coefficients.

The entire encoding process is summarized by Equation 2.3 (note that z' represents the encoding of the message vector, z).

$$z' = \sigma^{-1}(\zeta(\text{proj}_{\sigma(R)}\Delta\pi^{-1}(z))) \tag{2.3}$$

2.2.1.2 Decoding

To decode an encoded polynomial, m, to a vector, z, we simply apply a subset of the encoding operations in reverse.

1. Apply the σ mapping by evaluating the polynomial, m, on the odd roots of the cyclotomic polynomial: $(m(\xi), m(\xi^3), ..., m(\xi^{2N-1}))$.
2. Divide the result by the scaling factor, Δ.
3. Apply the π transformation by taking the first half of the elements in the resultant vector.

The decoding process is summarized in Equation 2.4.

$$z = \pi(\Delta^{-1}\sigma(m)) \tag{2.4}$$

2.2.2 CKKS Encryption and Decryption

2.2.2.1 Encrypting and Decrypting Vectors

The CKKS encryption scheme is based on the learning with errors (LWE) problem [31]. In CKKS, it is the problem of recovering the secret key given the vector-integer pairs shown in Equation 2.5.

$$(a_i, b_i) = (a_i, \langle a_i, sk \rangle + e_i) \tag{2.5}$$

The a_i term in Equation 2.5 is uniformly sampled from $\mathbb{Z}_q^n$. The secret key is $sk \in \mathbb{Z}_q^n$ and $e_i \in \mathbb{Z}_q$ are small Gaussian noises used to make the problem hard. If e_i is not included in Equation 2.5, then Gaussian elimination could be used to recover the secret key given a_i and b_i.

To encrypt a vector, $\Gamma \in \mathbb{Z}_q^n$, we first randomly generate a secret key, $sk \in \mathbb{Z}_q^n$. The q term in $\mathbb{Z}_q^n$ is called the modulus and n is called the ciphertext polynomial modulus degree. After generating an SK, we generate n $(a_i, \langle a_i, sk \rangle + e_i)$ pairs which leads to the matrix formulation delineated in the Equation 2.6.

$$(A, A \cdot sk \ + e) \tag{2.6}$$

Note that $A \in \mathbb{Z}_q^{n \times n}$, $sk \in \mathbb{Z}_q^n$, and $e \in \mathbb{Z}_q^n$. We select $pk = (-A \cdot sk \ + e, A)$ as our public key (PK). Then, to encrypt a message, $\Gamma \in \mathbb{Z}_q^n$, we apply the transformation detailed in Equation 2.7.

$$\begin{aligned} c &= (c_0, c_1) \\ &= (\Gamma, 0) + pk \\ &= (\Gamma - A \cdot sk \ + e, A) \end{aligned} \tag{2.7}$$

Given a ciphertext, c, it is hard to recover the secret key. In fact, this encryption scheme is quantum resistant [1]. To decrypt a ciphertext pair, (c_0, c_1), we apply the transformation defined in Equation 2.8.

$$\begin{aligned} \tilde{\Gamma} &= c_0 + c_1 \cdot sk \\ &= \Gamma - A \cdot sk + e + A \cdot sk \\ &= \Gamma + e \approx \Gamma \end{aligned} \tag{2.8}$$

CKKS is an approximate encryption scheme. This means that the decrypted result will have some small noise.

When encrypting vectors, the SK is of size $O(n)$ and the PK has size $O(n^2)$. The inefficiency of the size of the PK makes this scheme impractical. Instead, we modify this scheme to be used in the context of integer polynomials. This modified scheme is detailed in the following section.

2.2.2.2 Encrypting and Decrypting Polynomials

The ring learning with errors (RLWE) problem is a variant of the LWE problem, but modified for polynomial rings: $\mathbb{Z}_q[X]/(X^N + 1)$. a, sk, and e are randomly sampled from $\mathbb{Z}_q[X]/(X^N + 1)$ and a public key, $pk = (-a \odot s \ + e, a)$ is selected. To encrypt a message (Γ, 0), we apply the transformation delineated in Equation 2.9.

$$
\begin{aligned}
c &= (c_0, c_1) \\
&= (\Gamma, 0) + pk \\
&= (\Gamma - a \odot sk + e, a)
\end{aligned} \tag{2.9}
$$

Similarly, to decrypt a polynomial, (c_0, c_1), we apply the transformations in Equation 2.10.

$$
c_0 + c_1 \odot sk = \Gamma - a \odot sk + e + a \odot sk = \Gamma + e \approx \Gamma \tag{2.10}
$$

Moving from vectors to polynomials has two major advantages. Firstly, the public key is no longer a matrix of size $O(n^2)$, but a polynomial of size $O(n)$. Secondly, matrix-vector multiplication takes $O(n^2)$ time but polynomial multiplication can be done in $O(n\ \log(n))$ time using the discrete Fourier transform (DFT). These efficiency gains result in a practical encryption scheme that can leverage various homomorphic operations, which are detailed in Section 2.2.3.

2.2.2.3 Encryption Parameters

The security level of the encryption scheme, λ, quantifies the security of the encryption in bits. This parameter is dependent on three parameters: σ, q, and N. To reiterate, σ is the standard deviation of the Gaussian noises used to generate the public and secret keys, q is the polynomial modulus, and N is the polynomial modulus degree. The recommended level of security in [32] is $\lambda = 128$ bits. This level is used to guide parameter selection in this thesis. The authors of [33] provide a lower bound for N in terms of the security parameters, which is detailed in Equation 2.11. Using the inequality in Equation 2.11, we can choose q, N, and σ values to control the security level, possible multiplicative depth, and computational requirements for homomorphic operations.

$$
N > \frac{\lambda + 110}{7.2}\ \log_2(\frac{q}{\sigma}) \tag{2.11}
$$

Typically, σ is set to 3.2, N is selected according to multiplicative depth and security requirements, and q is selected using Equation 2.11 to guarantee a certain security level (λ).

2.2.3 CKKS Operations

HE supports only three homomorphic operations. They are addition, multiplication, and rotation. Division, comparison, and other mathematical functions are not supported directly. Instead, these operations have to be approximated using a combination of addition, multiplication, and rotation. A function that can be computed using only addition, multiplication, and rotation is called an HE-safe function.

2.2.3.1 Addition

Suppose we have messages x and y that are encrypted into two ciphertexts, $c_x = (x_1, x_2)$ and $c_y = (y_1, y_2)$. To add these ciphertexts, we simply add the individual ciphertext components as shown in Equation 2.12.

$$\text{CAdd}(c_x, c_y) = (x_1 + y_1, x_2 + y_2) \tag{2.12}$$

If we decrypt c_{x+y} we obtain the sum of the original messages, x and y.

$$\begin{aligned} \text{Decrypt}(\text{CAdd}(c_x, c_y)) &= x_1 + y_1 + (x_2 + y_2) \odot sk \\ &= x_1 + x_2 \odot sk + y_1 + y_2 \odot sk \\ &= x + y + 2e \end{aligned} \tag{2.13}$$

From Equation 2.13, we can see that if we decrypt the encrypted sum of c_x and c_y, we obtain the result of the plaintext addition of a and b plus some negligible noise, $2e$.

2.2.3.2 Multiplication

Multiplication is more complex than addition. Simply multiplying the two ciphertext components of c_x and c_y will not result in an accurate decryption. Instead,

multiplication is split up into two operations: ciphertext multiplication (CMult) and relinearization (Relin). During ciphertext multiplication, a ciphertext with three elements is computed using the operations defined in Equation 2.14.

$$\begin{aligned} \text{CMult}(c_x, c_y) &= c_x \odot c_y \\ &= (r_1, r_2, r_3) \\ &= (x_1 \odot y_1, x_1 \odot y_2 + y_1 \odot x_2, x_2 \odot y_2) \end{aligned} \tag{2.14}$$

During relinearization, these three components are reduced into two components using an evaluation key and some clever modulus switching operations. The evaluation key that we select is detailed in Equation 2.15.

$$\begin{aligned} \text{evk} &= (\text{evk}_1, \text{evk}_2) \\ &= (-a_0 \odot sk \ + e_0 + p \ sk \odot sk, a_0) (\text{mod } p \cdot q) \end{aligned} \tag{2.15}$$

The p term in Equation 2.15 is a large integer, a_0 is an integer polynomial sampled from $R_{p \cdot q}$, and e_0 is a small random polynomial. The relinearization step is defined in Equation 2.16.

$$\begin{aligned} \text{Relin}(r_1, r_2, r_3, \text{evk}) &= (r_1, r_2) + \lfloor p^{-1} \ r_3 \odot \text{evk} \rceil \\ &= (r_1, r_2) + (\lfloor p^{-1} \ r_3 \odot \text{evk}_1 \rceil, \lfloor p^{-1} \ r_3 \odot \text{evk}_2 \rceil) \end{aligned} \tag{2.16}$$

To reiterate, the "$\odot$" operator in Equation 2.16 multiplies the coefficients of two polynomials. The "$\lfloor x \rceil$" rounding operator rounds the coefficients of the input polynomial, x, to the nearest integer. Now if we decrypt the result of a ciphertext multiplication between c_x and c_y, we obtain a value very close to the result of the plaintext multiplication of x and y (see Equation 2.17).

$$\text{Decrypt}(\text{Relin}(\text{CMult}(c_x, c_y), \text{evk})) \approx x \odot y \tag{2.17}$$

2.2.3.3 Rotation

The rotation operation is also available in HE. To rotate a plaintext vector, $x = \{x_0, x_1, ..., x_N\}$, r units to the left, x becomes $x_r = \{x_r, ..., x_{n-1}, x_0, ..., x_{r-1}\}$. To rotate a ciphertext $c = (c_1, c_2)$ r units to the left in HE, we follow the steps outlined below.

1. Generate a rotation key by applying the $\kappa : f(X) \rightarrow f(X^k)$ transformation, $\text{rtk} = \kappa_{5^r}(sk)$
   - The κ transformation raises the polynomial X input values to the k^{th} power with respect to the polynomial modulus resulting in a polynomial with shifted coefficients.
2. Apply the κ transformation to c to obtain $c' = (\kappa_{5^r}(c_1), \kappa_{5^r}(c_2)) = (c'_1, c'_2)$.
3. The rotated ciphertext, c_r, is $c_r = (0, c'_2) + \lfloor p^{-1}\ c'_1 \odot \text{rtk} \rceil \bmod q$
   - q is the polynomial modulus and p is a large integer used to "switch keys" [34] or "rescale" [35].

The entire rotation process is summarized in Equation 2.18.

$$\text{CRotate}(c, r) = (\lfloor p^{-1}\ \kappa_{5^r}(c_1) \odot \text{rtk} \rceil \bmod q, \kappa_{5^r}(c_2) + \lfloor p^{-1}\ \kappa_{5^r}(c_1) \odot \text{rtk} \rceil \bmod q) \quad (2.18)$$

If c is the encryption of m, then the decryption of c_r matches the rotation of m by r units (see Equation 2.19).

$$\text{Decrypt}(\text{CRotate}(c, r)) = \{m_r, ..., m_{n-1}, m_0, ..., m_{r-1}\} \quad (2.19)$$

2.3 Machine Learning

2.3.1 Types of Machine Learning Problems

The capability of machines to learn patterns from raw data is called machine learning (ML) [36]. A machine "learns" by iteratively updating its internal model

parameters using data samples until the model can predict the labels (or values) of unseen data. Data are observations that can be represented in a variety of ways. For example, data can be represented as numbers, text, or images. More broadly, data can be either structured or unstructured. Structured data are defined by a set of rules and has patterns and structure that make them easily searchable. Unstructured data is everything else. This thesis will focus on structured data.

A tabular dataset is a type of structured data comprised of rows and columns. The rows in the dataset represent unique data entries and the columns represent the features of those data entries. A tabular dataset can be thought of as a matrix containing N rows and D columns where N is the number of data entries and D is the number of features. This thesis examines only tabular dataset contexts.

There are two types of ML problem contexts: unsupervised problems and supervised problems. Supervised learning problems are problems in which some characteristic of some population, such as a feature in a tabular dataset, is targeted for prediction. In this context, labelled data are used to train various statistical models. A labelled data instance is a data entry that has a label that identifies a certain property or characteristic about that particular data entry. For example, a common supervised learning problem is to label emails as "spam" or "not spam". Unsupervised learning, in contrast with supervised learning, deals with problems in which labelled data are not available. In this context, the structure of an unlabelled dataset is characterized to provide insight for the model users. For example, given a dataset describing the spending habits of users at a bank, a common unsupervised learning problem is to identify which of these users demonstrate anomalous behaviour. This thesis focuses on supervised learning problems.

There are two main types of supervised learning problems: classification and regression. Classification is the task of assigning labels to unlabelled data samples. This problem type classifies samples into two or more classes. Regression, on the other hand, is the task of predicting a continuous quantity. Given a dataset describing the

location, size, and owners of houses in an area, an example of a regression problem would be predicting the price of each house. Conversely, a classification problem would be predicting whether the price of the house is above or below $500,000. This thesis is restricted to classification problems.

2.3.2 Practical Machine Learning Considerations

Prior to discussing the various types of machine learning models, a note on class imbalance and tailored performance evaluation is in order. A balanced classification dataset is a dataset in which the number of samples in each class is approximately even. In a binary classification problem, for example, the dataset would be balanced if the number of "positive" samples were approximately equal to the number of "negative" samples. In this context, metrics such as accuracy are useful. More commonly, however, problems are extremely unbalanced.

For instance, in 2018, there were 368.92 billion credit purchases of goods and services worldwide [37]. A very tiny percentage of these transactions were fraudulent. If a model predicted that all 368.92 billion of these credit transactions were legitimate, it would have an accuracy very close to 100%. But in a practical sense, this model would be useless, because the intended purpose of the model would be to separate the fraudulent instances from the legitimate transactions. When a dataset is this imbalanced, if we only look at accuracy, we will have no way of knowing how well the model is performing. It is therefore important to use additional metrics such as precision, F1-score, recall, and the number of false negatives. These additional metrics enrich the classification performance evaluation process by taking into account the number of data samples for each class. These metrics are defined and further discussed in Appendix A.

To increase the performance of a learned model when there is class imbalance, various class re-balancing techniques are used. Down-sampling is a technique to randomly sample a fraction of the majority class so that there are an approximately

equal number of samples across each class. When there is a surplus of available data, this method is sufficient. When data is scarce, however, up-sampling is more useful as it does not drop any data samples. Up-sampling supplements the minority class with more data entries to better balance the number of data entries in each class. These additional data entries are typically re-sampled duplicates of the minority class. The synthetic minority over-sampling technique (SMOTE) [38] is a more complex up-sampling method that synthesizes or creates instances of the minority class that are not exact copies of minority class instances, but are very close. One or more of the methods mentioned above can be used to rectify class imbalances in data and enhance model performance.

2.3.3 Machine Learning Models

2.3.3.1 Data Partitions

The high-level objective of an ML model in a supervised classification context is to be able to predict the labels of data that has not yet been labelled. As mentioned in 2.1.1, training is the process of iteratively updating the parameters of a machine learning model to improve performance, whereas inference is the prediction of labels using an already trained model on unseen new data. Prior to training a model, ML model developers split labelled data into three subsets: a training dataset, a validation dataset, and a testing dataset. Depending on the size of the dataset, the problem context, and the training techniques implemented, the dataset partitions can vary substantially in relative size. Figure 2.1 illustrates typical ranges for training, validation, and testing dataset splitting. The training dataset is typically the largest data partition accounting for 50%-80% of the total dataset. The validation and testing datasets are often approximately equal accounting for 20%-50% of the total dataset. Certain data splitting strategies lead to stronger generalizations. The Kennard-Stone algorithm and SPXY systematic sampling method are two examples of algorithms that attempt to maximize the generalizability of models through effective data split-

ting [39].

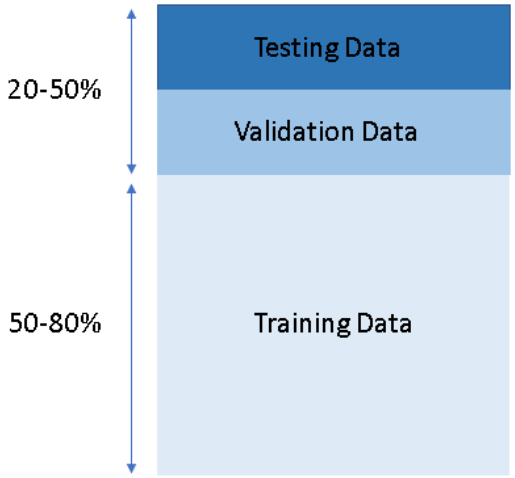

Figure 2.1: Typical training, validation, and testing data splits

Every ML model has an internal state that is represented by a finite number of parameters. These parameters are typically initialized to zero or to a small non-zero value based on the dimensions of the dataset. ML "training" is the process of iteratively updating the state, or parameters, of an ML model using the training dataset. During training, the model undergoes a process called backpropagation. Backpropagation is the process of propagating the loss of a model backwards during training and updating the model parameters accordingly. There are a handful of additional parameters called hyperparameters that control various aspects of the model and training process. The hyperparameters are optimized by evaluating the performance of the model on the validation dataset using various hyperparameter combinations. Once the model has been tuned, it can be compared to other tuned models using the testing dataset. The generalizability of a model is a model's ability to generate accurate predictions using data it has not encountered before. A model that performs well on the training data but poorly on the testing data is a model that generalizes poorly.

The training problem can be expressed in simple mathematical terms. Training

data consists of input data and output data. The input data is a single or multi-column matrix where each column represents an independent data feature. The output data is a singular column of labels corresponding to each of those input data elements. Given N training data inputs, $x_i = \{x_{i0}, x_{i1}, ..., x_{iD}\} \in \Re^D$, and N corresponding data labels, $y_i \in \{0, 1\}$, the training problem is finding a function, f, that minimizes the error between the model's predictions and the data labels. Several different parameterizations of f have been postulated and labelled as models. Examples of a few of these models are discussed in the subsequent section.

2.3.3.2 Logistic Regression

An LR model predicts on each element in the input data using a parameter vector, β. The parameter vector contains weights $\{w_0, w_1, ..., w_D\}$ and a bias term, b:

$$\beta = \{b, W\} = \{b, w_0, w_1, ..., w_D\} \in \Re^{D+1}$$

Given an input matrix, χ:

$$\chi = \begin{bmatrix} x_{00} & x_{01} & \dots & x_{0D} \\ x_{10} & x_{11} & \dots & x_{1D} \\ & & \dots & \\ x_{N0} & x_{N1} & \dots & x_{ND} \end{bmatrix} \in \Re^{N \times D}$$

We prepend a column of ones to obtain a slightly larger matrix:

$$\chi^{ext} = \begin{bmatrix} 1 & x_{00} & x_{01} & \dots & x_{0D} \\ 1 & x_{10} & x_{11} & \dots & x_{1D} \\ & & & \dots & \\ 1 & x_{N0} & x_{N1} & \dots & x_{ND} \end{bmatrix} \in \Re^{N \times (D+1)}$$

Then, to compute a prediction, we first take the matrix multiplication of X^{ext} and β:

$$Z = \chi^{ext} \cdot \beta \in \Re^N \tag{2.20}$$

Z is then passed through the sigmoid function, a special type of function called an activation function. The sigmoid activation function, the gradient of the sigmoid function, and various sigmoid function approximations are discussed at length in Section 2.3.4.

$$Y^{hat} = \sigma(Z) \in \Re^N \tag{2.21}$$

The matrix multiplication plus activation function data transformation above is also used by other model types including artificial neural networks.

To train an LR model, an optimization method called gradient descent is commonly used. The different optimization methods are discussed at length in Section 2.4. To utilize gradient descent, a loss function must be defined. A loss function accepts a model's prediction and label, and outputs a continuous value. This value quantifies the quality of a prediction relative to its corresponding label. Loss functions can be chosen to optimize the performance of the model relative to the various classification performance metrics discussed in Section A. An industry standard loss function for binary classification is the binary cross-entropy (BCE) loss function defined in (2.22).

$$BCE = -\frac{1}{N}\sum_{i}^{N}(y_i^{true} \cdot \log(y_i^{pred}) + (1 - y_i^{true}) \cdot \log(1 - y_i^{pred})) \tag{2.22}$$

For a binary classification problem, the model's prediction is a continuous value between 0 and 1. The BCE loss function heavily penalizes "confident" incorrect predictions (predictions that are close to 0 when the label is 1, and close to 1 when the label is 0) logarithmically.

During gradient descent, the gradient of the loss function with respect to the model parameters is computed. For an LR model utilizing the BCE loss function, the model parameters are the β vector, $L = BCE$, and the gradient of the loss function with respect to this vector is delineated below.

$$\frac{dL}{d\beta} = (Y^{hat} - Y^{true}) \cdot \chi^{ext} \in \Re^{D+1} \tag{2.23}$$

Instead of using the entire dataset to compute one big parameter update, the training dataset is split into batches. Then, for each batch, the predictions (Y^{pred}) and gradient with respect to the model parameters ($\frac{dL}{d\beta}$) are computed. The parameters of the model are then updated using the gradient descent update formula in (2.24).

$$\beta = \beta - \mu \cdot \frac{dL}{d\beta} \tag{2.24}$$

μ is a special parameter called the learning rate. This parameter is a continuous scalar value (typically ranging from 0.001 to 0.01) that scales the magnitude of the parameter updates. A smaller learning rate corresponds with slow parameter updates and a larger learning rate corresponds with fast parameter updates. The parameters are updated repeatedly until the entire dataset has been iterated over. When this occurs, one epoch is complete. Depending on the complexity of the model and the size of the dataset, the training process can take a handful of epochs or thousands of epochs. Once the training process is complete, the parameter vector, β, will yield predictions that minimize the loss function in a way consistent with the separability of the data.

2.3.3.3 Neural Networks

Artificial neural networks, also known as neural networks, or multi-layer perceptrons (MLPs), like LR models, are used to classify data and perform other data driven tasks. MLPs are extremely powerful because, according according to the universal approximation theorem, so long as the activation function used is sigmoid-like and the output data is continuous, an MLP with at least one hidden layer can generate an approximation of the output data to any degree of precision [40]. This theorem does not specify the number of parameters, also called nodes in this context, needed to generate this precision nor the complexity of the parameter optimization process. However, this theorem provides a guarantee that for each separable labelled dataset, there exists a single-layer MLP capable of approximating the relationship prevalent

in that dataset to a high degree of precision.

MLPs, like LR models, define a function, f, which accepts an input dataset, X, that minimizes the loss between $f(X)$ and an output dataset of labels, Y. The form of MLPs can vary based on the complexity of the dataset, the distributions of the independent variables, and the desired throughput.

Consider a one-layer MLP, for example. Suppose we have an input dataset, $\chi \in \Re^{N \times D}$, and an output dataset, $Y \in \Re^{N}$. Now suppose that we have two parameter matrices, $W_0 \in \Re^{D \times 2D}$ and $W_1 \in \Re^{2D \times 1}$, and two bias vectors, $b_0 \in \Re^{2D}$ and $b_1 \in \Re^{1}$. The equation used to compute the output of the first layer in an MLP is defined in (2.25)

$$L_1 = \sigma(\chi \cdot W_0 + b_0) \in \Re^{N \times 2D} \tag{2.25}$$

The input data, χ, has been transformed from a matrix of size $\Re^{N \times D}$ to a matrix of size $\Re^{N \times 2D}$. The output layer transformation transforms L_1 into a prediction vector by performing an additional matrix multiply and sigmoid activation function. The output layer formula for a one-layer MLP is shown in (2.26).

$$Y^{pred} = \sigma(L_1 \cdot W_1 + b_1) \in \Re^{N} \tag{2.26}$$

This prediction vector is then passed into a loss function along with the label vector, Y^{true}. Then, the gradient with respect to each of the model parameters ($\frac{dL}{dW_0}$, $\frac{dL}{db_0}$, $\frac{dL}{W_1}$, and $\frac{dL}{b_1}$) is taken and the model parameters are updated using an optimization method (gradient descent, for example).

The model above has one hidden layer as there is only one intermediary matrix-multiply plus sigmoid operation before the output layer. LR models, for example, have only output layers. The size of the one hidden layer in this example is $2D$ (the number of columns in the parameter matrix, W_0). Model developers can, in general, vary the size of the hidden layers, the number of hidden layers, and the activation and loss functions used, as well as the initial values of the parameter matrices.

This thesis focuses on MLPs and LR models. Convolutional neural networks (CNNs), unlike MLPs or LR models, operate on unstructured data and are prevalent in a plethora of image recognition and segmentation problem contexts. Extending our analysis to CNNs in the future is a natural continuation of this thesis and has been identified as such in Section 5.2.

2.3.4 Approximating Layers for Homomorphic Encryption

As discussed in Section 2.1, the only mathematical operations that HE schemes support are addition and multiplication. The fully-connected layer type discussed in Section 2.3.3.2 is a matrix multiply between the input data and a weight matrix. A matrix multiply consists of additions and multiplications and is, therefore, an operation that can be implemented in HE. Consider the sigmoid activation function detailed in (2.27).

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.27}$$

This function contains an exponential function and a division. Since this function cannot be composed of only additions, multiplications, and rotations, it is not HE-safe. Thus, to utilize this activation function (and its gradient during backpropagation), we can compute approximations using polynomials, as polynomials are HE-safe.

2.3.4.1 Approximating the Sigmoid Activation Function

The sigmoid activation function is illustrated in Figure 2.2. Prior to discussing HE-safe approximations of this function, we must understand what makes this function an attractive choice as an activation function. Firstly, it is non-linear. To satisfy the universal approximation theorem [40], this property must be true. In other words, an activation function must exist. Secondly, this activation function confines large negative values to zero and large positive values to 1. Since the model is expecting predictions in the 0-1 range, this property is especially attractive. Finally, the function

is monotonically increasing. This property is more challenging to satisfy with an approximation (especially if the approximation is polynomial in nature).

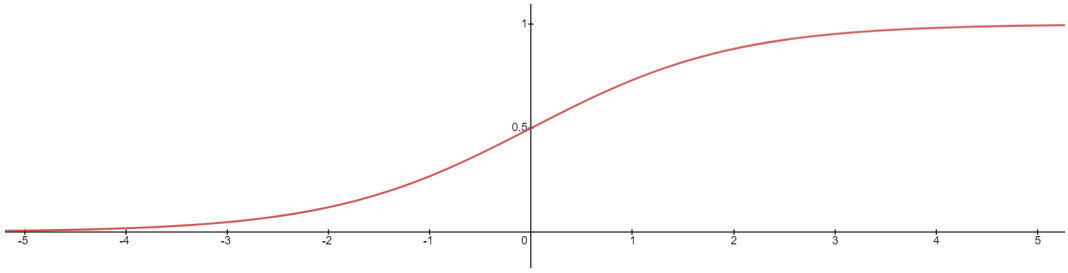

Figure 2.2: The sigmoid activation function

To approximate the sigmoid activation function, we will use an n-th degree polynomial approximation that minimizes the least squared error (LSE) delineated in (2.28).

$$LSE = \sum_{i}^{N} (y_i^{hat} - y_i^{true})^2 \tag{2.28}$$

The degree of the polynomial that we use to approximate the sigmoid function is called the approximation degree. A 3rd-degree least squares polynomial approximation, for example, computes the coefficients a_0, a_1, a_2, a_3 in (2.29) that minimize the LSE loss in (2.28).

$$y = a_0 + a_1\ x + a_2\ x^2 + a_3\ x^3 \tag{2.29}$$

Polynomial approximations of various degrees are contrasted with the sigmoid function in Figure 2.3. We can see from this illustration that higher-degree polynomial approximations fit the sigmoid function more closely. The trade-off is that higher-degree approximations incur greater homomorphic multiplicative depth. Based on the desired range of the outputs and the expected range of the inputs, we can design sigmoid approximations on a case-by-case basis by controlling the density of the points used to compute the LSE. For example, if it is important that the approximation be very accurate between -10 and 10, but accuracy beyond this range is not important, we can generate a dense region of points in the -10 to 10 range and sparsely populate the rest of the domain.

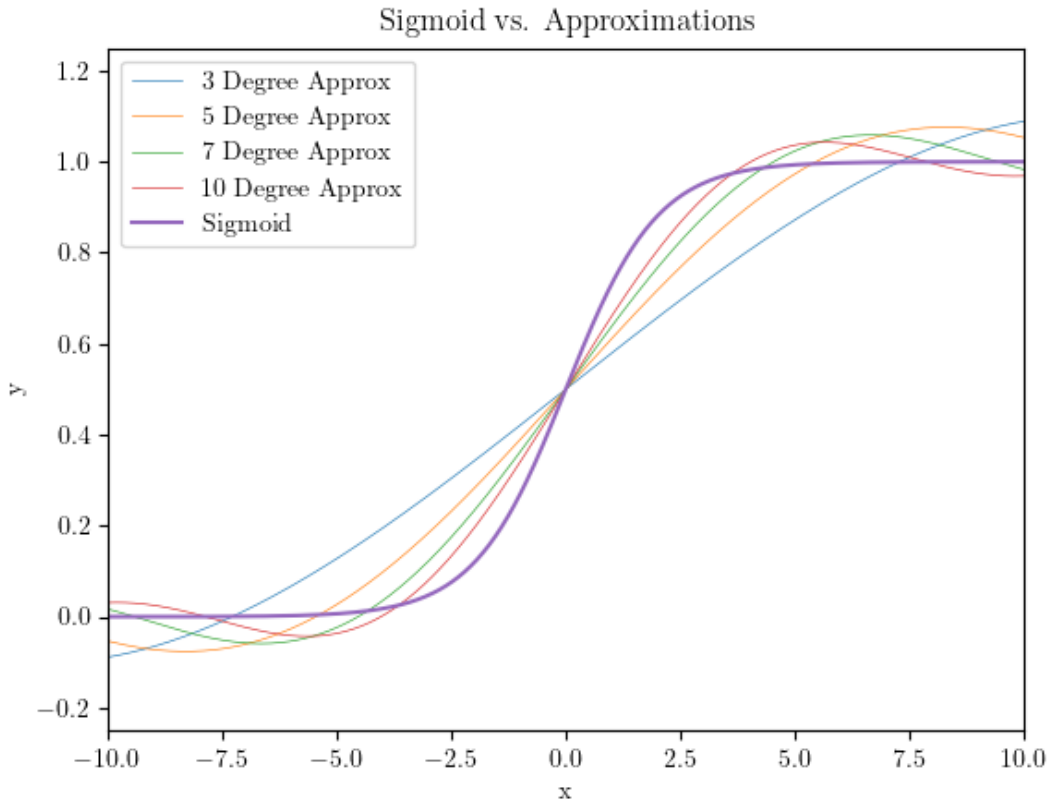

Figure 2.3: Approximating the sigmoid activation function with a least squares approximation

2.3.4.2 Approximating the Gradient of the Sigmoid Function

During backpropagation the gradient of the sigmoid function delineated in (2.30) and illustrated in Figure 2.4 is computed.

$$\frac{d}{dx}\sigma(x) = \frac{e^{-x}}{(1+e^{-x})^2} \tag{2.30}$$

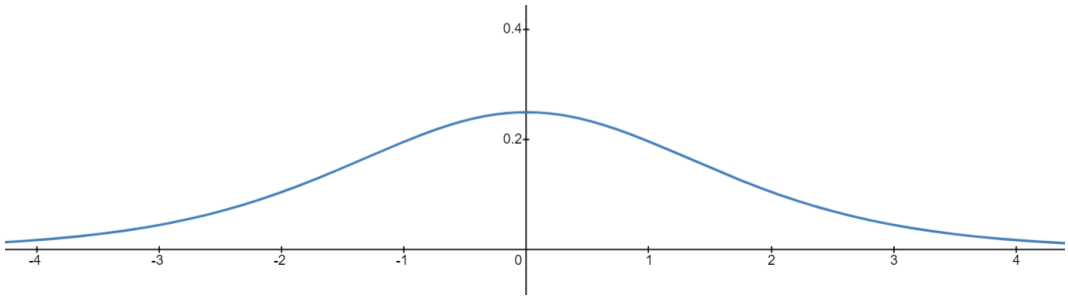

Figure 2.4: The gradient of the sigmoid function

The exponential function and a division operation are used in (2.30) so the gradient of the sigmoid function is also not HE-safe. The desirable qualities of this function are that it truncates large positive and negative values to zero. Secondly, it monotonically increases and decreases towards and away from zero, respectively. Keeping these qualities in mind, we can approximate this function with a least-squares polynomial approximation. A comparison of various polynomial approximations is displayed in Figure 2.5. We can, once again, tailor the approximation based on the problem context by controlling the density of the points used to compute the LSE for various domain ranges.

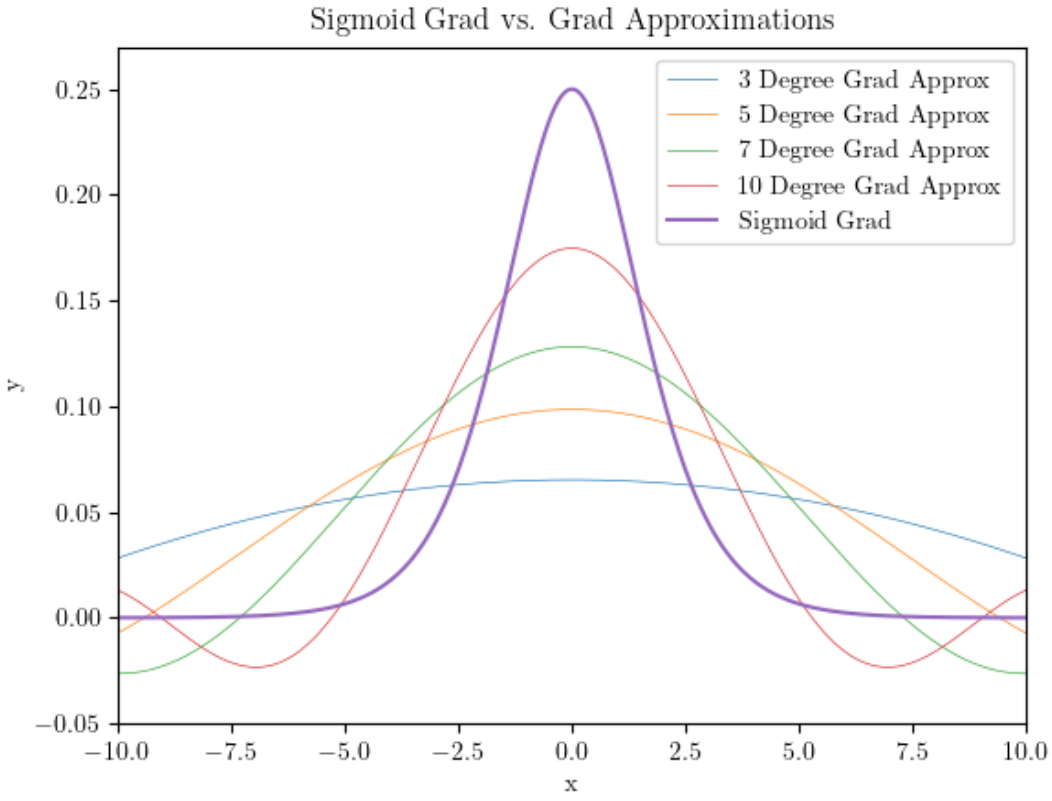

Figure 2.5: Approximating the gradient of the sigmoid activation function with a least squares approximation

2.4 Optimization Methods

Suppose we have an objective function, $f(\beta)$. Optimization is the process of finding the values of the input parameters, β, that minimize (or maximize) the value of the objective function, $f(\beta)$. This problem is expressed mathematically in Equation 2.31.

$$\hat{\beta} = \arg\min_{\beta} f(\beta) \tag{2.31}$$

An ML model is a series of mathematical transformations applied to the input data to produce some output that minimizes some loss function. During training, the parameters of the model are iteratively updated using some optimization method in the hopes of convergence to some absolute minimum loss value. Whether a model converges to an optimal set of parameters depends largely on the nature of the optimization space.

2.4.1 Optimization Space Characteristics

2.4.1.1 Convexity and Local Extremum

A multi-variate convex function is a function that is twice differentiable and has a positive semidefinite Hessian matrix. In two dimensions, the second derivative of a convex function is positive for all possible input values ($f''(x) > 0 \; \forall \; x \in \Re$). A convex function has only one extremum (a point where $f'(x) = 0$). Non-convex functions can have multiple extrema. A local minimum is a value at which the derivative of the loss function with respect to the parameters is zero, but the loss value is not the minimum of the entire loss function. The global minimum, in contrast to the local minimum, is the absolute minimum value of the loss function. Figure 2.6 illustrates a non-convex function. The minimum value on the right delineated by the red dotted line is a local minimum, while the green dotted line delineates a global minimum.

A saddle point is a point on the loss function in which the derivative of the loss function is zero, but it is not a local extremum (minimum or maximum). Consider the graph of $z = x^2 - y^2$ shown in Figure 2.7. The central point at $(x, y, z) = (0, 0, 0)$ is a

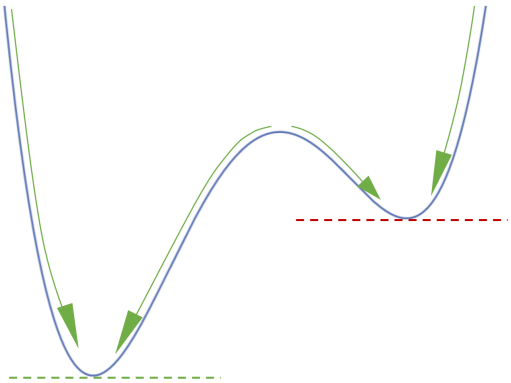

Figure 2.6: A non-convex function with local and global minima

saddle point. Local minima and saddle points can be particularly problematic during optimization as optimization methods can get "stuck". There are a wide variety of optimization methods that attempt to rectify this issue.

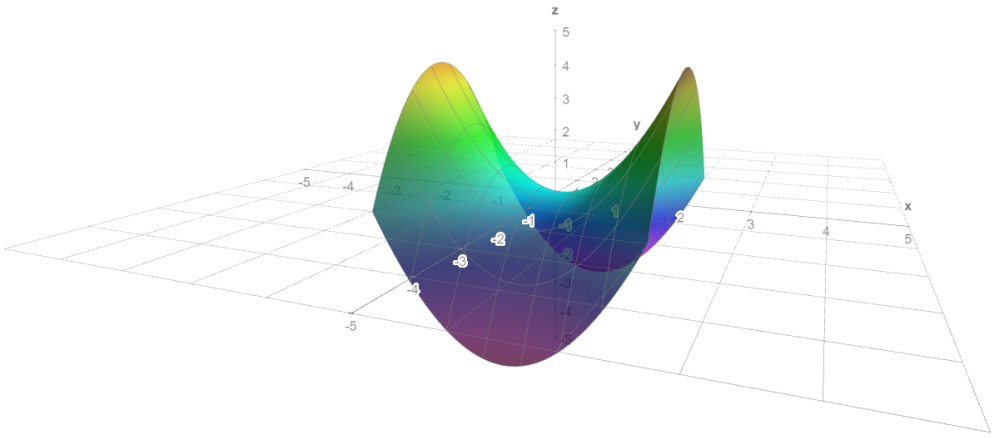

Figure 2.7: A visualization of a saddle point on the function $z = x^2 - y^2$

2.4.1.2 ML Model Optimization

Gradient descent is considered the grandfather of all ML optimization methods. Momentum [41], Nesterov's accelerated gradient (NAG) [42], Adagrad [43], Adadelta [44], and RMSprop [45] use methods that iterate on gradient descent to obtain faster and more reliable convergence for various problem contexts. Each of these optimization methods is discussed at length in the subsequent sections.

2.4.2 Gradient Descent

Gradient descent (GD) utilizes the gradient of the loss function with respect to the model parameters by updating the model parameters in the direction of this gradient. In this way, the model descends down the loss curve until it reaches a minima. Equation 2.32 defines the parameter updates for the GD optimization method.

$$\beta = \beta - \mu \frac{dL}{d\beta} \tag{2.32}$$

Note that β is the parameter vector, μ is a pre-defined constant called the learning rate, and $\frac{dL}{d\beta}$ is the gradient of the loss function with respect to the model parameters. This optimization method is easy to implement, understand, and compute. Furthermore, it can operate on input data of varying sizes. Stochastic gradient descent (SGD) is a modification of GD that utilizes only one data point per weight update [46]. Mini-batch gradient descent utilizes small batches of data [47].

GD methods (GD, SGD, and mini-batch GD) tend to get stuck in local minima and saddle points. Furthermore, GD methods require careful selection of a good learning rate. A learning rate that is too slow leads to slow convergence and increases the chance that the model gets trapped in local minima. Conversely, learning rates that are too fast can cause the loss of the model to oscillate and diverge. Adaptations of the gradient descent algorithm attempt to rectify these issues by adding terms to the parameter update equation. These gradient descent adaptations are discussed in the subsequent sections.

2.4.3 Momentum

Momentum is a GD optimization method that dampens oscillatory behaviour and accelerates convergence [41] by incorporating past parameter updates into the current update vector. Equations 2.33 and 2.34 delineate the parameter updates when utilizing the momentum optimization technique.

$$\beta_t = \nu\beta_{t-1} + \mu\ \Delta_\theta J(\theta) \tag{2.33}$$

$$\theta = \theta - \beta_t \tag{2.34}$$

ν is the momentum constant which ranges from 0.8 to 1. The $\nu\beta_{t-1}$ term impacts the parameter updates by increasing the magnitude of subsequent parameter updates occurring in the same direction and decreasing parameter updates in opposing directions. In this way, oscillation occurring from parameter updates in opposing directions are reduced. Further, parameter updates occurring in the same direction accelerate the velocity of the parameter updates resulting in faster convergence.

2.4.4 Nesterov's Accelerated Gradient

Nesterov's accelerated gradient (NAG) iterates on the concept of momentum by predicting the next position of the parameters after applying momentum [42]. Instead of computing the gradient with respect to the current state of the parameters, it computes the gradient with respect to an approximation of the next state of parameters. The formulae for the NAG optimization method are defined in Equations 2.35 and 2.36.

$$\beta_t = \nu\beta_{t-1} + \mu\ \Delta_\theta J(\theta - \nu\beta_{t-1}) \tag{2.35}$$

$$\theta = \theta - \nu_t \tag{2.36}$$

By predicting the next parameter update and then making a correction, NAG results in more accurate parameter updates, less oscillation, and faster convergence.

2.4.5 Adagrad

Adagrad is a first order optimization method that utilizes a per-parameter dynamic learning rate [43]. Each function parameter has a designated learning rate that is incrementally updated each iteration. Adagrad keeps track of a per-parameter gradient

vector, G. This vector contains the sum of the squared gradients for each parameter (see Equations 2.37 and 2.38).

$$G = [G_0, G_1, ..., G_D] \tag{2.37}$$

$$G_i = \sum_{j=0}^{t} g_{ij}^2 \tag{2.38}$$

g_{ij} is the gradient of the loss function with respect to parameter i at time-step j. t is the current time-step. The parameter update vector is defined in Equation 2.39.

$$v_t = \frac{\mu}{\sqrt{G + e}} \cdot g_t \tag{2.39}$$

μ is the learning rate constant, e is a small non-zero vector on the order of 1×10^{-8} added to G to prevent division by zero, and g_t is the gradient of the loss function with respect to the parameters at the current time-step, t. Using the update vector, v_t, the full Adagrad parameter update is shown in Equation 2.40.

$$\beta = \beta - v_t \tag{2.40}$$

A huge benefit of Adagrad is it removes the need to tune the learning rate [48]. Furthermore, since Adagrad performs larger updates for infrequent data and smaller updates for frequent data, it performs exceptionally well on sparse data. Word embedding data, for example, is very sparse and Adagrad has shown significant performance improvements in this arena relative to other optimization methods [49].

An issue with Adagrad is the per-parameter learning rate diminishes rapidly. The denominator tracks the squared gradient values over time. G grows each iteration as the squared gradient is always positive. Eventually, the per-parameter learning rate becomes too small and the model stops learning. Adadelta and RMSprop rectify this issue by adjusting the contributions of past and future per-parameter gradients.

2.4.6 Adadelta & RMSprop

Adadelta [44] and RMSprop [45] both attempt to rectify the diminishing learning rate problem affecting Adagrad by discounting past parameter gradients. Mathemat-

ically, the parameter updates for Adadelta and RMSprop are the same as Adagrad except for one key difference: G_i is a decaying moving average of squared parameter gradients instead of the total sum of all past squared parameter gradients.

The RMSprop parameter updates are detailed in Equations 2.41, 2.42, 2.43, and 2.44, below. The Adadelta parameter updates have been omitted because they are nearly identical to the RMSprop parameter updates.

$$E[G] = \{E[G]_0, E[G]_1, ..., E[G]_D\} \tag{2.41}$$

$$E[G]_{it} = 0.9\ E[G]_{i(t-1)} + 0.1\ g_{it}^2 \tag{2.42}$$

$$v_t = \frac{\mu}{\sqrt{E[G] + e}} \cdot g_t \tag{2.43}$$

$$\beta = \beta - v_t \tag{2.44}$$

g_{it} is the gradient of the loss function with respect to parameter i at time-step t and $E[G]_{it}$ is the value of $E[G]_i$ at time-step t.

By discounting the history of the average of the parameter updates (as shown in Equation 2.42), RMSprop and Adadelta are able to take advantage of the benefits of Adagrad (rapid convergence and strong performance, especially on sparse data) while simultaneously avoiding the diminishing learning rate problem.

Although RMSprop and Adadelta are more novel than GD, SGD, momentum, NAG, and Adagrad, the performance of an optimization method is dependent on the dynamics of the objective function. Determining which optimization method to use is a challenging and exciting problem relevant to FHE ML training.

2.5 Training FHE Models

There are a handful of contexts in which FHE training is applicable. Generally speaking, there are three main FHE ML model training categories: unencrypted private training, fully encrypted public training, and federated learning. Depending on the need for privacy, server availability, budget considerations, collaboration oppor-

tunities, and the way in which data is being gathered, an FHE user may prefer one over another. These training categories are discussed in the subsequent sections.

2.5.1 Unencrypted Private Training

Unencrypted private training is useful for clients who have a private server available for training or a batch of low-risk training data that does not need to be secret, but who like to perform inference securely on a public server. In this context, the client is able to develop an HE-safe ML model in plaintext with full knowledge of the input data and training data outputs. Then, after a model has been optimized for the task, it can be encrypted and deployed onto a public server for encrypted inference. The private training and encryption process is illustrated in Figure 2.8.

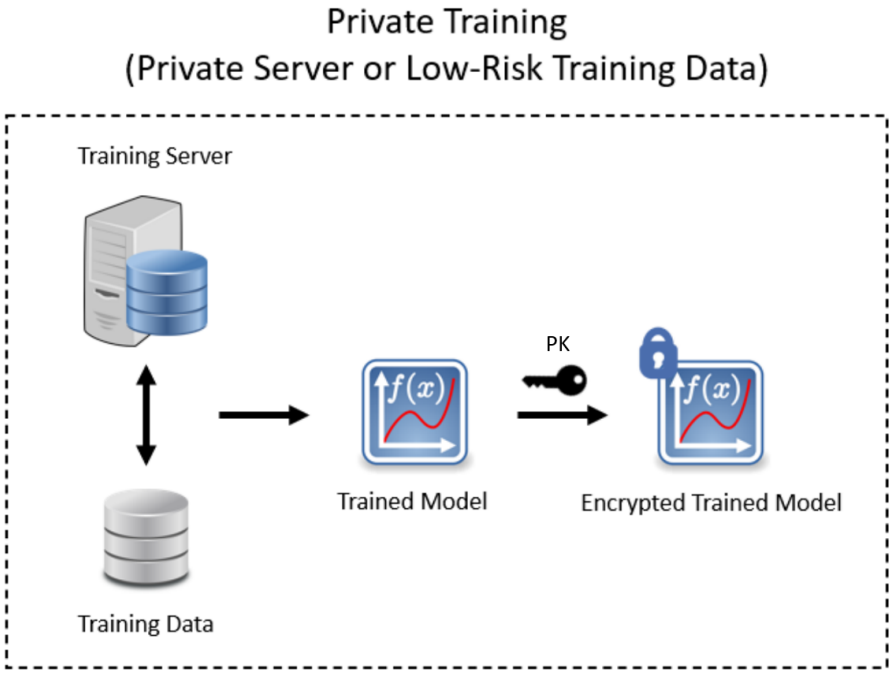

Figure 2.8: Private model training and encryption for future deployment. Note that this entire process takes place on a private server, not on the cloud. After training the model privately, the model can be encrypted and then uploaded to the cloud for inference.

For clients who would like to take advantage of cloud-based tools for rapid computation of model outputs, this process is particularly attractive for two reasons.

Firstly, since the model developers are provided with full access to the intermediary inputs and outputs during training, the trained model will be optimal (for HE-safe components). This is not the case for fully secure training (discussed in Section 2.5.2). Secondly, inference is completely secure as the input and output data entering and exiting the public server are encrypted according to the user's privacy constraints. This training scenario is not always applicable, however. If neither low-risk data nor a private server is available, then another privacy-preserving solution must be used. This thesis implements an encrypted Autograd ML library to assist model developers while implementing HE-safe models in plaintext (Section 3).

2.5.2 Fully Encrypted Public Training

In this context, models are trained entirely on encrypted input and output training data. This scenario is applicable for clients who need near total privacy guarantees during the training process because neither a private training server nor a batch of low-risk training data are available. Note that this training process is not completely secret, since the client still needs to know the shape of the input data, which model types are applicable for the problem context, the hyperparameters that will likely yield convergence, and some way of verifying that the model has converged after training so that it can be deployed for inference.

The fully-encrypted training process is illustrated in Figure 2.9. This figure complements Figure 2.8, which illustrates the private training process. From Figure 2.9, we can see that the model and the data are encrypted before training occurs (as indicated by the cycle logo in the public server box). In private training, training is done using plaintext data on a private server. Fully encrypted training maintains complete data privacy during computation (training). After the encrypted model has been trained on the encrypted data, it can be used for encrypted inference.

There are a handful of issues with this approach. Firstly, since the input and output data are entirely encrypted during training, the backpropagation and optimization

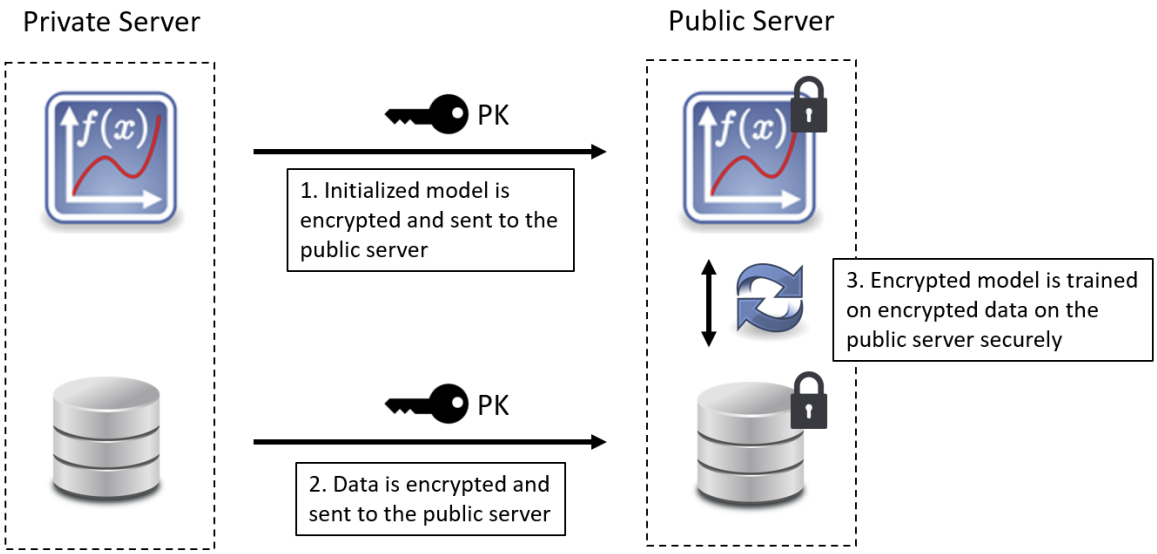

Figure 2.9: Fully-encrypted public model training. Step 1 is to initialize a model, encrypt it using the public key, and send it to the public server. Step 2 is to encrypt the data and send it to the public server. Step 3 is to train the encrypted model on the encrypted data securely on the public server.

processes must also be done in HE. Adagrad, Adadelta, RMSprop, and other high-performance optimizers cannot be used, because they rely on square root and division operations which are not supported in HE. Furthermore, loss functions that do not have an HE-safe parameter gradient cannot be used either. Tuning the loss function to optimize some user-defined metric may not be possible given the functional constraints imposed by HE. Secondly, the re-scaling and bootstrapping operations required to refresh the multiplicative depth during training impose a significant computational overhead. For example, fully training a simple logistic regression model consisting of 201 model parameters on encrypted financial data (containing 422,108 samples) using an efficient parallelized bootstrapping scheme can take nearly 18 hours to complete [20]. Finally, there is no guarantee that the hypothesized model architecture, hyperparameters, and optimization strategy will yield convergence. It is quite possible that, after the model has completed its training loop, it may fail to converge or may even have diverged. The client will not know whether a model has failed to converge until the training process is complete which, as mentioned previously, can

take a significant amount of time. The time needed to confirm whether or not a model is working properly makes the model tuning process challenging.

Despite its drawbacks, fully encrypted training may be an attractive choice in some contexts. If training time is not an issue and the learning problem is well understood, it is possible to achieve convergence and deploy an effective model trained entirely on encrypted data.

2.5.3 Federated Learning

Federated learning (FL) is an alternate approach to privacy-preserving ML. It allows multiple clients to work together securely to train and deploy a decentralized encrypted model. In an FL scheme, there is a decentralized public aggregator server and N client servers. The public aggregator server tracks the state of the model by housing the global encrypted model parameters. The aggregator server has access to only the public key. The client servers, on the other hand, have access to both the public and private keys. Note that we assume that the clients are honest-but-curious [50], meaning that the clients will follow the FL protocol but will use transmissions to extract information about other clients. A visualization of how the keys, clients, and aggregator are configured is illustrated in Figure 2.10.

Note that there is only one set of public and private keys. Each client holds the exact same public and private keys. The aggregator server holds only the public key. In multi-key FL schemes, each client holds part of the private key. In this way, aggregator information can be decrypted only if all N clients consent. In this way, multi-key FL defends against $k < N - 1$ dishonest participants [51]. This thesis focuses on single key FL, but notes that there are many benefits to multi-key FL schemes.

Prior to training, the keys and aggregator model must be initialized. To initialize the keys, one of the clients defines private and public keys for FHE according to some security threshold. These public and private keys are distributed to the rest of the

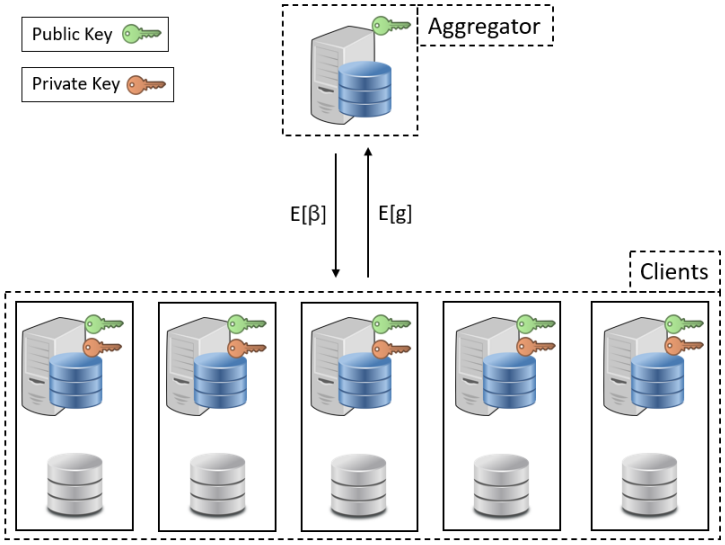

Figure 2.10: A visualization of an FHE FL scheme. Note that in this example, there are 5 clients. FL schemes can support an arbitrary number of clients. Also note that the communication between the aggregator server and the clients is done individually. This means that the aggregator sends each client the encrypted parameter vector and each client sends the aggregator back an encrypted gradient vector.

clients using a secure key sharing algorithm on a private communication channel. The aggregator server is provided with the public key. To initialize the aggregator model, a model is trained in plaintext on a local private server. Then, the initial parameters of the model are encrypted using the public key and housed on the aggregator server.

Once the keys and aggregator server are initialized, the clients can iteratively update the global parameters during training. The FL training process is delineated below and illustrated in Figure 2.11. In this context, the aggregator server only needs to store the encrypted model parameters, $E[\beta]$.

1. The aggregator encrypts and sends each client the encrypted model parameters
    - Depending on the scheme, the aggregator server can store and transmit additional information to improve model convergence (see Chapter 4).
2. The client decrypts the model parameters using the secret key.
3. The client computes and encrypts a gradient vector using the model parameters

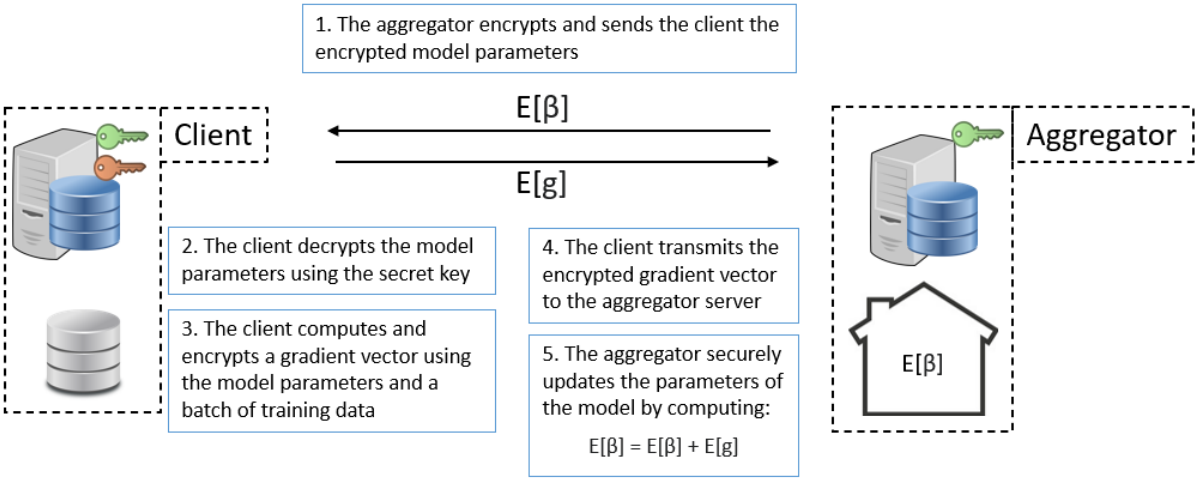

Figure 2.11: The FL training process at a high-level. Note that only the communication between one of the N clients and the aggregator is shown in this Figure. This process is repeated for each of the N clients over and over again, depending on the number of training epochs, or depending on when new data is received. The house icon underneath the server icon in the aggregator box specifies the storage requirements for the aggregator server. The aggregator only needs to store the encrypted model parameters.

and a batch of training data.

4. The client transmits the encrypted gradient vector to the aggregator server.

   - Note that this type of FL scheme is vulnerable to an attack from $k < N-1$ honest-but-curious conspirators. Multi-key FHE FL schemes prevent this type of attack by mandating client consensus when decrypting aggregator transmissions [51].

5. The aggregator securely updates the parameters of the model by computing $E[\beta] = E[\beta] + E[g]$.

The applications of FL are vast and growing ([52], [50], [53], [54]). Parties that have multiple sources of input data can take advantage of FL by maintaining a singular global model state and using the inputs from each data source to iteratively update the global state securely. Furthermore, FL can enable collaboration among multiple data

owners even under rigid privacy constraints. This thesis implements a decentralized Adam optimizer for encrypted federated learning using FHE (Chapter 4).

2.6 Summary

This Section has outlined the fundamental concepts needed to interpret the rest of this thesis. As mentioned in the introduction, this thesis has two main contributions: an encrypted Autograd library for secure machine learning applications and a distributed federated learning scheme for decentralized secure ML. These two contributions utilize CKKS, an approximate arithmetic FHE scheme, and focus on structured data in a supervised learning problem context. The next Section discusses the first thesis contribution, Cryptograd: an encrypted Autograd library for secure machine learning.

Chapter 3

Cryptograd: an Autograd-Based FHE Training Library

3.1 Introduction

This chapter will focus on the first two HE training contexts: private plaintext training for HE-safe models (Section 2.5.1) and fully encrypted training (Section 2.5.2). In a private plaintext training context, HE model developers search for a model that best optimizes performance on data new to the model using a plaintext training dataset. This scenario is different from non-FHE training contexts because the models in question must be HE-safe. Being HE-safe means that the trained model must be able to be approximated with HE-safe components, encrypted, and hosted on a public server that may be insecure. During training, however, the model can be trained on plaintext.

In a fully encrypted context, the model components must be approximated with HE-safe components *prior* to training. Because the model is no longer being trained with plaintext data, layers such as the sigmoid activation function have to be approximated with HE-safe functions because the backpropagation loop must also work using encrypted data.

When working with plaintext data, model development teams utilize large-scale model development software platforms such as Dataiku [55]. These software platforms enable model developers to combine various data transformations, resulting

in customized, efficient, and high-performing models. These software platforms utilize an automatic differentiation optimization framework called autograd. Autograd computes gradient vectors, which are used during model training, automatically. Autograd is an essential component of these model development software platforms. Without it, users have to compute the gradient of their loss functions by hand and then manually integrate this gradient into parameter updates. Furthermore, Dataiku allows users to seamlessly separate training from inference. Once a model has been trained, it can be deployed onto a server with the click of a button and its inference performance can be analyzed over time.

When training models privately and then deploying them for encrypted inference, or when using encrypted training data to train a model entirely in HE, model developers cannot utilize Dataiku, which is built for plaintext models and plaintext data using functions incompatible with HE. To create tools that facilitate model development in HE, autograd for encrypted learning is needed. To date, there has been no research into the usage of autograd for HE model development. This Section of the thesis implements *Cryptograd*: an FHE training library built on top of a customized implementation of autograd.

3.2 Background

3.2.1 Symbolic Differentiation vs. Automatic Differentiation

Symbolic differentiation is the process of taking the derivative of a function manually using fundamental calculus principles. Numerical differentiation is the process of computing the derivative of a function empirically by iterating on the definition of the derivative (shown in Equation 3.1). Automatic differentiation (autograd) is a technique used to automatically compute the true derivative of a function. Autograd is of particular interest to the field of ML as it allows ML software libraries to differentiate ML models of arbitrary configurations precisely.

$$f'(x) \equiv \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{3.1}$$

Autograd models a mathematical function as a directed acyclic graph (DAG). Each node in the graph represents an operation. When taking the derivative of this function, the graph is traversed in reverse topological order. The algorithm for topological sorting is shown in Algorithm 1.

Algorithm 1 Topological Sorting Algorithm

```
1:  function TOPOSORT(v_n)
2:      c ← {}                  ▷ A dictionary housing the number of children for each node
3:      s ← [v_n]                                  ▷ A stack used to iterate over the nodes
4:      while s do
5:          n ← s.pop()
6:          if n in c then
7:              c[n] = c[n] + 1
8:          else
9:              c[n] = 1
10:             s.extend(n.parents)
        x = [v_n]                                  ▷ A list of nodes with no children
11:     while c do
12:         n ← x.pop()
13:         yield n                                ▷ Yields the topologically sorted nodes
14:         for p in n.parents do
15:             if c[p] == 1 then
16:                 x.append(p)
17:             else
18:                 c[p] = c[p] − 1
```

After obtaining a topological ordering of the nodes, the derivatives of the individual operations are computed and synthesized. The algorithm for backpropagation is shown in Algorithm 2. This function accepts a topologically sorted list of nodes and an index, k, that defines with respect to which input node the derivative should be taken. In this way, we can compute the derivative of an arbitrary function with respect to an arbitrary input argument automatically. Implementing autograd for encrypted ML models is one of the fundamental contributions of this thesis. Section 3.2.2 shows an example of autograd being used practically.

Algorithm 2 Backpropagation Using Autograd

```
1: T ← [t_1, t_2, ..., t_N]                    ▷ T is a topologically sorted list of nodes
2: function GRAD(T, k)
3:     dL_N ← 1
4:     for i = N − 1, ..., k do
5:         c ← v_i.children
6:         dL_i = 0
7:         for j in c do
8:             dv_{j|i} = dv_j / dv_i           ▷ The derivative of v_j with respect to v_i
9:             dL_i = dL_i + dL_j dv_{j|i}
10:    return dL_k
```

3.2.2 Autograd in Action

This Section uses a practical example to demonstrate how autograd works. Equation 3.2 delineates E, a function of two variables: A and B.

$$E(A, B) = A + B\ A^2 \tag{3.2}$$

Suppose we want to take the derivative of this function with respect to A. By treating B as a constant, we can treat the function, E, as a polynomial. The derivative of a polynomial, $f = a_0 + a_1 x + ... + a_n x^n$ is $\frac{df}{dx} = a_1 + ... + n\ a_n x^{n-1}$. We can use this definition to obtain the derivative of E (shown in Equation 3.3).

$$\frac{dE}{dA} = 1 + 2\ A\ B \tag{3.3}$$

As mentioned in Section 3.2.1, taking the derivative of a function by hand using fundamental calculus principles (as we have done above) is called symbolic differentiation. We wish to compute the derivative of the above function (and of any function) automatically.

To compute the derivative of E using autograd, we first must define a DAG that represents the mathematical operations applied in E. This graph is displayed in Figure 3.1. The input nodes, A and B, are at the front (left-most) part of the graph. These two nodes are multiplied together to create $C = A\ B$. Then, C is multiplied with A yielding $D = C\ A = A\ B\ A = B\ A^2$. Finally, D is added to A resulting in

$E = A + D = A + B\ A^2$. We can see that this graph models $E(A, B)$ above. These equations are re-defined in Equations 3.4, 3.5, and 3.6.

$$C = A \cdot B \tag{3.4}$$

$$D = C \cdot A \tag{3.5}$$

$$E = A + D \tag{3.6}$$

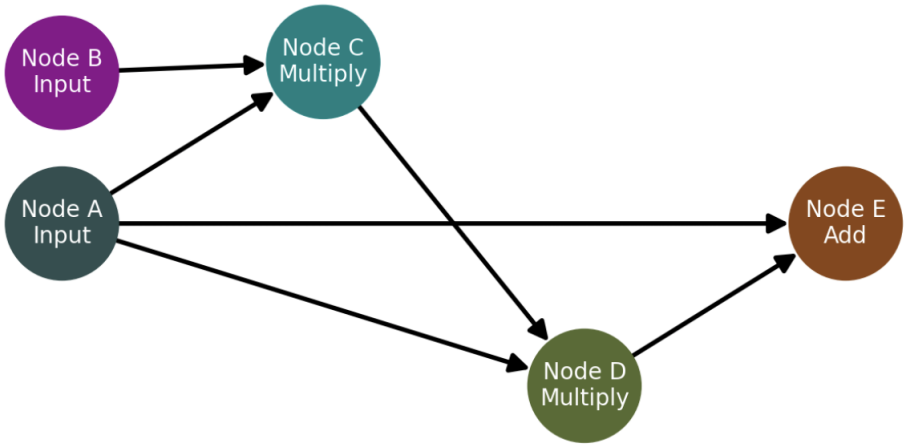

Figure 3.1: A graph representing the basic mathematical operations in $E(A, B)$ (Equation 3.2)

To take the derivative of $E(A, B)$ using autograd, we first obtain a topological sorting of the nodes $\{A, B, C, D, E\}$ using Algorithm 1. These nodes are already topologically sorted as $\{A, B, C, D, E\}$ satisfies the definition of a topologically sorted list of nodes.

After obtaining a valid topological sorting of the nodes, we traverse the topologically sorted list in reverse order. By following the Algorithm 2, we set the derivative of the $i = 5^{th}$ node (node E) to 1: $dL_N = dL_5 = \frac{dE}{dE} = 1$. Then, we proceed to the next node at $i = 4$: node D. To take the derivative of E with respect to D, we iterate over each of D's children. For each child, we compute the derivative of the

child with respect to D, multiply it by dL_j (where j is the index of the child), and add it to $dL_4 = \frac{dE}{dD}$ which has been initialized to 0. D only has one child, E, and so $dL_4 = \frac{dE}{dD}\ dL_5 = \frac{d(A+D)}{dD}\ dL_5 = 1 \times 1 = 1$. If we repeat this process for all of the nodes in the graph, we obtain the expressions for the derivatives of E with respect to each node defined in Equations 3.7, 3.8, 3.9, 3.10, and 3.11.

$$dL_5 = \frac{dE}{dE} = 1 \tag{3.7}$$

$$dL_4 = \frac{dE}{dD} = 1 \tag{3.8}$$

$$dL_3 = \frac{dE}{dC} = A \tag{3.9}$$

$$dL_2 = \frac{dE}{dB} = A^2 \tag{3.10}$$

$$dL_1 = \frac{dE}{dA} = A\ B + 1 + C = A\ B + 1 + A\ B = 1 + 2\ A\ B \tag{3.11}$$

We can see from Equation 3.10 that the derivative of $E(A, B)$ with respect to B is A^2, and from Equation 3.11 that the derivative with respect to A is $1 + 2\ A\ B$. These derivatives which were obtained using autograd are equivalent to the correct symbolic derivatives of $E(A, B)$. This automatic differentiation process can be extended to an arbitrary function that utilizes any combination of primitive operations.

3.3 Cryptograd Implementation

Cryptograd was developed using the Python programming language. There are three main components of Cryptograd:

1. A customized implementation of automatic differentiation in Python that can operate on encrypted input arguments.

2. An encryption library called BLUE that implements CKKS.

3. An encrypted ML training and inference library in Python that utilizes the autograd implementation for parameter updates and the BLUE encryption library for encrypted parameter updates and inference.

The implementation of each of these components will be discussed at length in the subsequent sections.

3.3.1 Encrypted Automatic Differentiation Implementation

If we pass a NumPy array to a Python function, we will have no knowledge of the mathematical transformations that results in the output. Thus, we will not be able to define a graph of operations and we will not be able to take the gradient of that function. If, however, we box the input NumPy array into a container object, we can see which functions are invoked on the input container objects and in what order.

This is the strategy employed by the makers of autograd [56] and Google's JAX [57]. The gradient component of the Cryptograd library also employs this strategy. The input arguments to a function are boxed in "Node" object containers. When a function is called with these Node wrapped objects that contain NumPy arrays (or BLUE encrypted data objects), the operations in the function can easily be traced and a directed graph can be created. The gradient can then be computed using Algorithms 1 and 2.

Consider Listing 3.1. By wrapping the input argument, "x", in a Node object yielding "x_wrapped", we can see all of the operations performed on "x_wrapped" when "my_polynomial" is called by implementing tracing functionality in the implementation of the Node class. We can use this idea to design a function that accepts a function as an input and computes and returns the gradient of that function with respect to a desired input argument.

```
1
2  import cryptograd.gradient.Node
3  import numpy as np
4
5  x = np.array([2, 4, 1])
6  x_wrapped = Node(x)
7
8  def my_polynomial(x):
9    return 1 + 2 * x + x^2
10
```

```
11 my_polynomial(x_wrapped)
```

Listing 3.1: An illustrative example of function tracing and argument boxing

Cryptograd's autograd implementation utilizes the process defined below to compute function gradients:

1. Wrap the arguments of the input function in Node object containers.
    - The input is a function that can have any number of parameters.
    - The way in which these input parameters are combined and transformed is unknown to the gradient function. In other words, the function is a "black-box" to the gradient function.
2. Call the input function using the wrapped input arguments.
3. Create a directed graph of Nodes by recording which operations are invoked and in what order
    - Each Node in the graph has the following properties:
        - A string delineating the name of the Node
        - 0, 1, or 2 parents (representing the inputs to the function being applied at the Node)
        - $N \geq 0$ children
        - A forward function
        - A backward function (computed using a dictionary storing all of the possible primitive operations and their derivatives)
4. Using Algorithm 1, compute a topological ordering of the Nodes in the directed graph
5. Using Algorithm 2, traverse the graph in reverse topological order and compute the gradient of the function with respect to the desired input argument

- The resulting gradient function can accept NumPy array arguments or BLUE encrypted data objects

The above process allows us to take the N^{th} derivative of an arbitrary function with an arbitrary number of parameters with respect to any of the input arguments. Furthermore, Cryptograd's autograd implementation allows users to pass BLUE "cipher array" objects to the returned gradient function pointer, enabling encrypted gradient computation and a cohesive developer environment.

3.3.2 Encryption Library

The ML and gradient components of Cryptograd are built on top of the BLUE encryption library. BLUE was developed by Lorica, a Toronto-based encrypted ML development company focused on secure cloud computing for BDIs [58]. This library was built and optimized for C++ and exposed to Python using PyBind11 [59]. This library utilizes the CKKS encryption scheme previously discussed.

BLUE provides users with the ability to encode, encrypt, decrypt, and decode vectors. Furthermore, users can manually generate the secret and public keys as well as other encryption parameters such as the polynomial modulus degree, the encoding method, and the relinearization keys needed to run encrypted ML. Ciphertext addition, multiplication, and rotation are also supported by BLUE. Bootstrapping, however, is not available in this thesis. As a computational alternative, whenever the multiplicative depth of ciphertext operations has been exhausted, the ciphertexts are re-encrypted to replenish the depth.

3.3.3 Encrypted ML Training and Inference Library

After completing the implementation of the gradient function and exposing the existing encryption functionality in the BLUE library to Python, a machine learning training library that utilizes these tools was designed and implemented. This

library has many user-friendly features that enable rapid encrypted model development, including automatic differentiation, model and layer templating, and model performance reporting. Listing 3.2 shows an example Python snippet of how a user can utilize Cryptograd to train and test a plaintext LR model for encrypted inference.

```
1
2  # Importing the Cryptograd library components
3  from cryptograd.ML.layers.HE.shared import sigmoid_approx
4  from cryptograd.ML.layers.rescaling_layer import RescalingLayer
5  from cryptograd.ML.layers.vectorized_linear_layer import
       VectorizedLinearLayer
6  from cryptograd.ML.model import Model
7  from cryptograd.ML.layers.HE.HE_sigmoid_BCE_layer import (
8      HESigBCELayer,
9  )
10
11 # Defining the LR model
12 class LR(Model):
13     def __init__(self, **kwargs):
14         super().__init__(**kwargs)
15
16     def init_layers(self, X: Node):
17         self.layers.append(
18             VectorizedLinearLayer(input_dim=X.shape[1], output_dim
       =1, lr=self.lr)
19         )
20         self.layers.append(RescalingLayer(scalar=0.01))
21         self.layers.append(HESigBCELayer(degree=7))
22
23     def predict_proba(self, X: Node) -> Node:
24         for i in range(len(self.layers) - 1):
25             X = self.layers[i].forward(X, None)
26         return sigmoid_approx(X, degree=7)
27
28 # Defining the training parameters
29 training_params = {
30     "lr": 0.001,
31     "batch_size": 64,
32     "n_epochs": 5,
33 }
34
35 # Instantiating the model
36 lr = LR(**training_params)
37
38 # Fitting the model to the training data
39 lr.fit(Node(X_train), Node(y_train))
40
41 # Inferring on the testing data
42 y_test_preds = lr.predict(Node(X_test))
```

Listing 3.2: A Python code snippet illustrating the usage of Cryptograd

As shown in Listing 3.2, the ML training and inference library requires that users inherit from a master "Model" object template when defining a new model (in this case, the "LRModel" class). The "Model" class implements the "fit", "predict", "predict_proba", "__init__", "forward", and "backward" functions. The user must define the abstract "init_layers" function by appending HE-safe layer types (that they can define themselves by inheriting the "Layer" prototype or can import from a series of already implemented layers) to "self.layers".

In the "fit" function, the "Model" class automatically fits the parameters of the model to the training data by iterating over "self.layers" at each training iteration. The gradient function discussed in the previous section is used to automatically compute parameter updates. These parameters can be encrypted selectively by passing in an encryption flag to the various implemented layers. The functions defined above were modelled after the SciKit-Learn [60] software library to emit a familiar and simple model development experience.

3.4 Setup

3.4.1 Hardware Environment

The hardware setup defined in this section is used for all of the experiments in Chapters 3 and 4. The operating system is Ubuntu 20.04.04 LTS. The system utilizes 128 GB of RAM and an 11th Gen Intel Core i7-11700K @ 3.60GHz with 16 cores. 12 cores were used for experimentation.

3.4.2 Dataset

The input dataset for the experiments detailed below is the QSAR androgen receptor binary classification dataset [61]. It has 1024 features, 1686 data entries, and two classes. The output classes are labels indicating whether the data molecules are androgen binder (positive) or non-binder (negative). The classification problem is not balanced as there are 1488 non-binder/negative samples and only 198 binder/-

positive samples. This imbalance was not rectified. This dataset was selected as it only has categorical features which makes fully encrypted training more manageable given computational constraints. The input data was split into a training dataset and testing dataset. The training dataset has 80% of the original data and the testing dataset has the remaining 20% of the original data.

3.5 Experiments

Prior to beginning experimentation, we first outline some hypotheses to test and some expectations to verify. Firstly, we expect the addition of autograd to incur some non-zero timing degradation. We will measure this timing degradation and compare it to the time that it takes to run the other system components. Secondly, we require that the introduction of autograd into the training process does not change the nature of the model parameter updates. We expect the model parameters to be identical to the model parameters that would have been obtained had symbolic differentiation been used. To verify this, we will compare the distribution of model outputs at each iteration. Finally, we will characterize the performance of the Cryptograd library by evaluating a model trained on encrypted data in terms of model performance, training time, and inference time. To summarize, using the dataset detailed in Section 3.4.2, we will perform the following three experiments:

1. Timing Experiment

    - We determine the time that it takes to train models using symbolic differentiation versus models using autograd.
    - We use this result to quantify the timing cost incurred when utilizing Cryptograd.

2. Empirical Equivalence to Symbolic Differentiation Experiment

- We examine the distribution of model parameters for models using symbolic differentiation versus the distribution of model parameters for models using Cryptograd.
- This sub-experiment is included to verify that the Cryptograd implementation is empirically identical to symbolic differentiation.

3. Cryptograd Empirical Performance and Timing Analysis Experiment
   - We analyze the precision, recall, accuracy, and a detailed timing analysis of the Cryptograd system in a fully encrypted training environment to demonstrate the practicality of the Cryptograd system and characterize its performance.

3.5.1 Timing Experiment

To quantify the cost of using autograd during training, we consider the total training time for a model utilizing symbolic differentiation and a model utilizing autograd. All settings in the symbolic differentiation model and in the autograd model are identical except for one detail: the symbolic differentiation model utilizes a hard-coded parameter derivative and the autograd model computes this derivative automatically. For $N = 500$ iterations, we execute the process outlined in Listing 3.3. This experiment yields a list of floats of autograd timing results and a list of floats of symbolic model timing results.

```
1
2  def time_models(N):
3      # Initialize the timing result return arrays
4      autograd_deltas = []
5      symbolic_deltas = []
6
7      for i in range(N):
8          # Re-initializing autograd and symbolic models at each
9          # iteration
10         autograd_model = AutogradModel(**training_params)
11         symbolic_model = SymbolicModel(**training_params)
12
13         # Quantifying the training time for the autograd model
```

```
        autograd_start = time.time()
        autograd_model.fit(X_train, y_train)
        autograd_deltas.append(time.time() - autograd_start)

        # Quantifying the training time for the symbolic model
        symbolic_start = time.time()
        symbolic_model.fit(X_train, y_train)
        symbolic_model.append(time.time() - symbolic_start)

    # Return the timing result arrays
    return autograd_deltas, symbolic_deltas
```

Listing 3.3: Autograd timing experiment setup

We use the following model architecture for the Timing experiment and the Empirical Equivalence to Symbolic Differentiation experiment:

1. Vectorized linear layer

   - Let $X \in \Re^{B,D+1}$ symbolize the input data augmented with a column of ones for the bias term, $W \in \Re^{D+1}$ represent the parameters of the model, and $z_1 \in \Re^{B}$ represent the output of the vectorized linear layer.
   - The vectorized linear layer linearly transforms the data in the following way: $z_1 = X\ W$

2. A 3^{rd} degree approximation of the sigmoid layer

   - Since model performance was not a priority for the Timing experiment and Empirical Equivalence to Symbolic Differentiation experiment, we selected a simple third degree approximation of the sigmoid function.
   - If z_2 is the output of this layer, then $z_2 = 0.14 + 0.72\ (0.5 + 0.168\ z_1 + 3.506 \times 10^{-18}\ z_1^2 - 1.47 \times 10^{-3}\ z_1^3)$.
   - This approximation maintains reasonable accuracy on the $-8 < x < 8$ range (see Figure 3.2).

The hyperparameters used for these experiments are detailed in Table 3.1.

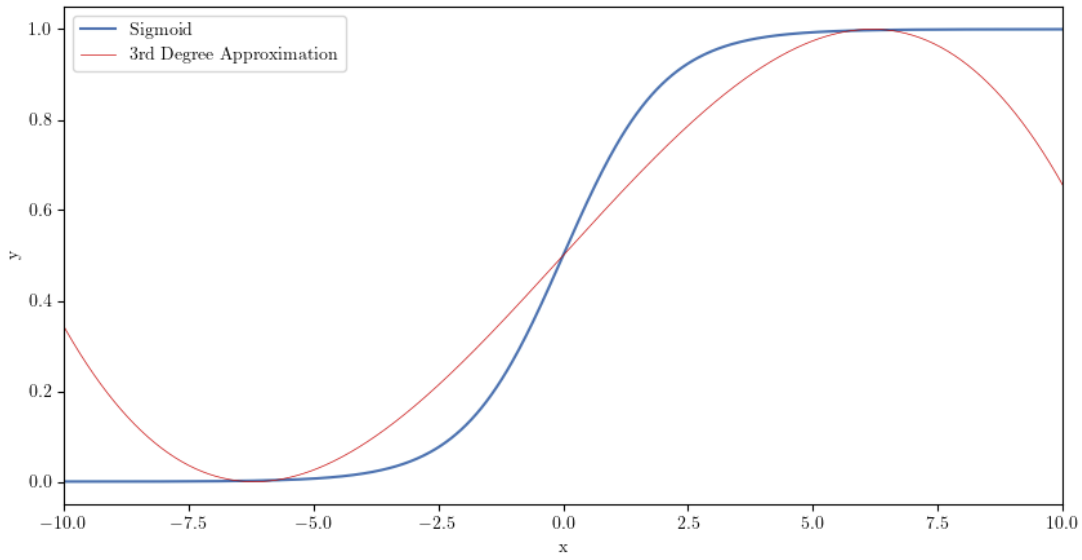

Figure 3.2: An illustration of the 3^{rd} degree sigmoid approximation used for the Timing experiment and the Empirical Equivalence to Symbolic Differentiation experiment. The red line is the true sigmoid function and the blue line is the third degree approximation.

Table 3.1: Hyperparameter settings for the Androgen LR experiment

Hyperparameter	**Value**
LR	0.001
Batch Size	64
Number of Epochs	5

3.5.2 Empirical Equivalence to Symbolic Differentiation

After each training loop, we record the 1025 parameters used (1024 for the weights and 1 for the bias). Then, to compare models, we take the root mean squared error (RMSE) between their model parameters. We selected this error metric because it is a reliable error metric used to compare the similarity of vectors containing floating point values. To verify that the autograd model and the symbolic differentiation model are empirically equivalent after training, we generate two distributions:

1. A distribution of RMSEs between the autograd model and a symbolic differentiation model (which we will call “S1”, or “Symbolic 1”)

2. A distribution of RMSEs between Symbolic 1 and another model utilizing symbolic differentiation ("S2" or "Symbolic 2")

The RMSE distribution between the S1 and S2 parameters serves as a control. The S1 and S2 models are truly empirically equivalent and differ only in the noise introduced during parameter initialization. If we did not create this control and instead compared the parameters of the symbolic model with the autograd model directly, we would have no frame of reference for how low the RMSE should be. If the autograd vs. S1 RMSE distribution is statistically equivalent to the S1 vs. S2 RMSE distribution, we can be certain that the symbolic differentiation model and the autograd model are empirically equivalent.

For $N = 500$ iterations, we execute the process outlined in Listing 3.4. This experiment yields two lists of RMSE floating point values representing the RMSE distributions between S1 and S2, and S1 and the autograd model.

```
1
2  def verify_model_equivalence(N):
3      # Initialize the RMSE return arrays
4      s1_s2_RMSEs = []
5      autograd_RMSEs = []
6
7      for i in range(N):
8          # Re-initializing autograd and symbolic models at each
9          # iteration
10         autograd_model = AutogradModel(**training_params)
11         s1 = SymbolicModel(**training_params)
12         s2 = SymbolicModel(**training_params)
13
14         # Training the s1, s2, and autograd models
15         autograd_model.fit(X_train, y_train)
16         s1.fit(X_train, y_train)
17         s2.fit(X_train, y_train)
18
19         # Generating the RMSE values for s1 vs. s2 and s1 vs.
       autograd
20         s1_s2_RMSEs.append(RMSE(s1.params, s2.params))
21         autograd_RMSEs.append(RMSE(s1.params, autograd.params))
22
23     # Return the RMSE arrays
24     return s1_s2_RMSEs, autograd_RMSEs
```

Listing 3.4: Autograd empirical equivalence experiment

As mentioned in the previous sub-section, the Empirical Equivalence to Symbolic Differentiation and Timing experiment utilize the same model architecture and hyperparameter settings (detailed above).

3.5.3 Cryptograd Empirical Performance and Timing Analysis

The precision, recall, accuracy, time to train, and time to infer for a Cryptograd model trained entirely using encrypted data will be evaluated. As mentioned in Section 3.1, when training using encrypted data, we must generate HE-safe approximations for the activation functions and the gradients of the activation functions. Further, we must design a robust model that is able to converge. The model that yielded the strongest results is detailed below:

1. A vectorized linear layer.

    - Let $X \in \Re^{B,D+1}$ symbolize the input data augmented with a column of ones for the bias term, $W \in \Re^{D+1}$ represent the parameters of the model, and $z_1 \in \Re^{B}$ represent the output of the vectorized linear layer.
    - The vectorized linear layer linearly transforms the data in the following way: $z_1 = X\ W$

2. A re-scaling layer: $c = 0.01$.

    - The re-scaling layer re-scales the output of the previous layer by multiplying the output by a constant, c. If z_2 represents the output from the second layer, then $z_2 = c\ z_1$.

3. A 7^{th} degree approximation of the sigmoid layer.

    - As mentioned in the background section, the sigmoid function is not computable in HE. Thus, we must generate an approximation. The approximation used for this layer is shown Figure 3.3. If z_3 is the output of this

layer, then $z_3 = 0.92\ (0.5\ +\ 0.153\ z_2 + 5.12 \times 10^{-18}\ z_2^2 - 0.00218\ z_2^3 - 1.2 \times 10^{-19}\ z_2^4 + 1.42 \times 10^{-5}\ z_2^5 + 4.95 \times 10^{-22}\ z_2^6 - 3.122 \times 10^{-8}\ z_2^7) + 0.05$

- This approximation was tuned using least squares and was selected to minimize the difference between the approximation and the sigmoid function in the range $-10 < x < 10$.
- A high-degree of precision is required for this approximation. Otherwise, the model can exhibit sporadic learning behaviour, produce large outputs (positive or negative), and potentially diverge.

4. The binary cross entropy loss function.

   - By using the binary cross entropy loss function, we do not need to take the gradient of the sigmoid approximation because the parameter updates simplify to the expressions delineated in Equations 2.23 and 2.24 (see Section 2.3.3.2).

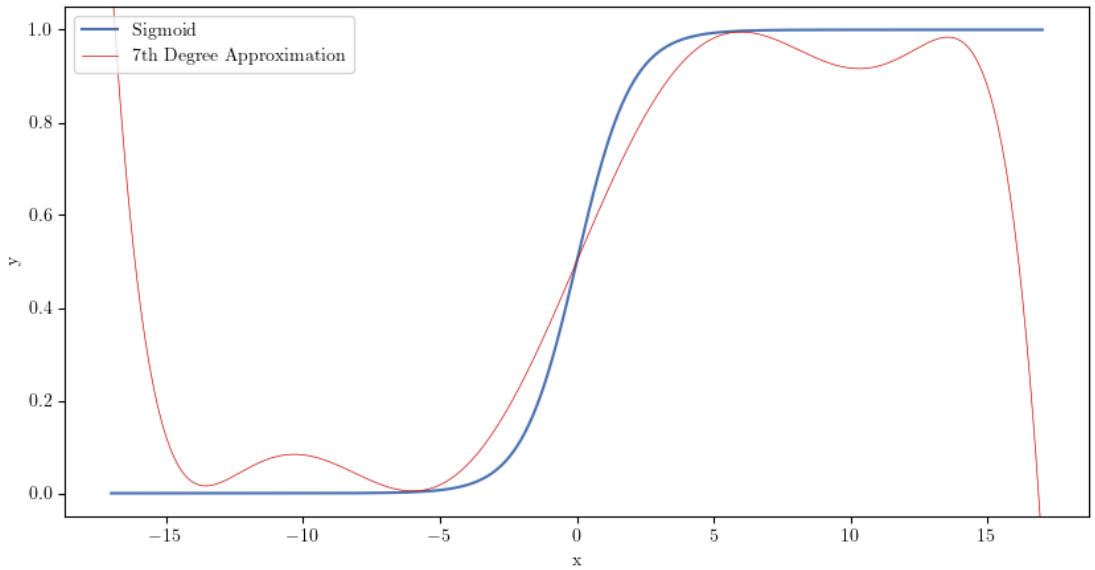

Figure 3.3: An illustration of the 7^{th} degree sigmoid approximation used for the Empirical Equivalence to Symbolic Differentiation experiment and the Timing experiment.

Finally, we must define encryption parameters that we will use to encrypt the data and model parameters. In a real-world training context, bootstrapping is required to replenish the multiplicative depth. Bootstrapping requires a polynomial modulus degree (q in Section 2.2.3) greater than or equal to 32,768. A degree this large imposes significant computational overhead. Since we do not have bootstrapping available, we will use a smaller polynomial modulus degree of 16,384. This polynomial modulus degree provides a 128-bit security guarantee [62] without imposing severe computational costs.

3.6 Results and Discussion

3.6.1 Timing Experiment

To reiterate, this experiment compares the time that it takes to train a model using Cryptograd with autograd enabled relative to training a model using symbolic differentiation. Both of these models are trained in plaintext. Table 3.2 delineates the average training time for the symbolic differentiation model and the autograd model. Table 3.2 also contains the t-score and the p-value obtained when conducting a two-tailed independent t-test between the symbolic differentiation and autograd timing distributions. The symbolic differentiation and autograd timing distributions are illustrated in Figure 3.4.

From Table 3.2 and Figure 3.4, we can see that the time to train the Cryptograd model with autograd enabled is statistically greater ($p < 10^{-100}$) than the time that it took to train the model trained using symbolic differentiation. The difference amounts to approximately 20ms, or 10%.

The time that it takes to compute the gradients of the input functions is a one-time cost. This cost will be the same for encrypted models because the gradients of the input functions will be computed before the model is encrypted. This is why the time to train was recorded using plaintext data. If a fully-encrypted training loop

Table 3.2: Symbolic differentiation vs. autograd timing results

Metric	**Reported Value**
Symbolic differentiation average training time (s)	0.12 ± 0.01
Autograd model average training time (s)	0.14 ± 0.01
Two-tailed independent t-test t-score	-31.387
Two-tailed independent t-test p-value	$< 1 \times 10^{-100}$

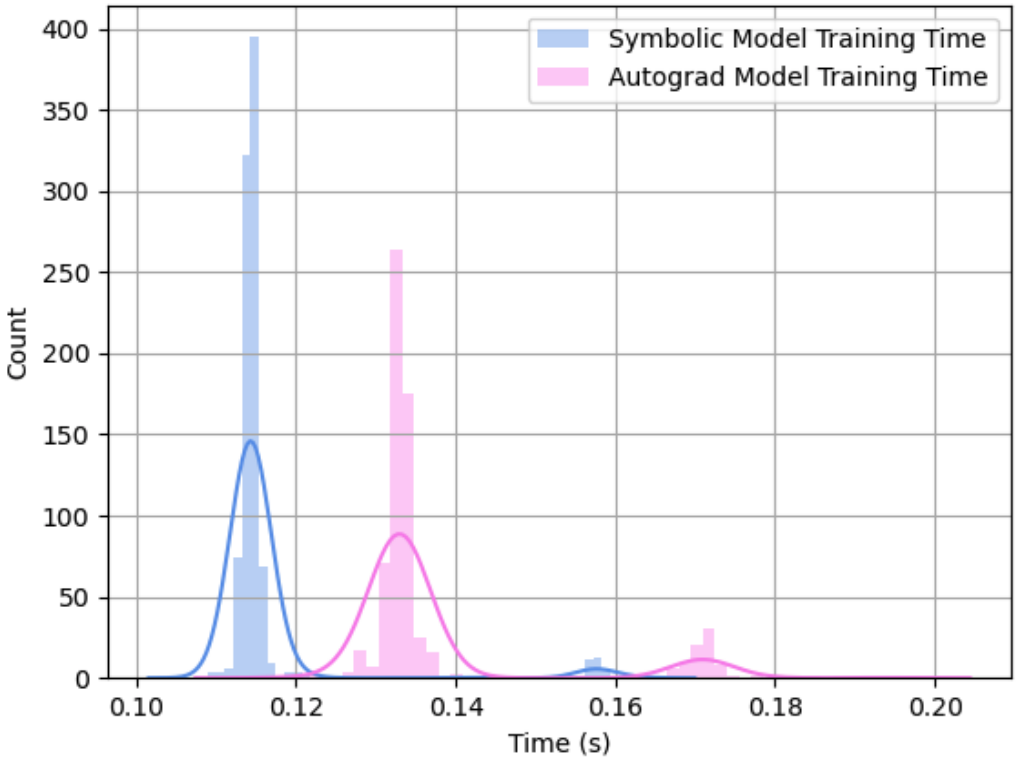

Figure 3.4: The timing distributions for the symbolic differentiation and autograd models when trained using the QSAR androgen data set

was used, it would have been computationally infeasible to generate a distribution of training times for both autograd and symbolic differentiation. Therefore, the additional 20ms incurred from using autograd in plaintext will translate to the same additional 20ms when training a fully-encrypted model using autograd. In Section

3.6.3, we determined that it took approximately 670s to train a fully-encrypted model using fully-encrypted data. The additional 20ms incurred using autograd is neglible compared to the 670s that it takes to train a fully-encrypted model.

3.6.2 Empirical Equivalence to Symbolic Differentiation

The Empirical Equivalence to Symbolic Differentiation experiment is a comparison of the parameter distributions between a Cryptogrtad model with autograd enabled and a symbolic differentiation model. The purpose of this experiment is to verify that the parameter distributions obtained after training using these two models are statistically equivalent. Table 3.3 below shows the average RMSE between S1 and S2 and the average RMSE between S1 and the autograd model. For an explanation of what S1, S2, and the autograd model are referring to, see the Empirical Equivalence to Symbolic Differentiation experiment explanation discussion in Section 3.5. The S1 vs. S2 RMSE distribution and the S1 vs. autograd RMSE distribution are shown Figure 3.5.

Table 3.3: The average RMSE values between S1 and S2 and the average RMSE values between S1 and the Cryptograd model with autograd enabled.

Metric	**Reported Value**
Average RMSE between trained S1 and S2 parameters	0.062 ± 0.001
Average RMSE between trained S1 and autograd parameters	0.065 ± 0.001
Two-tailed independent t-test t-score	1.3431
Two-tailed independent t-test p-value	0.18

From Table 3.3 and Figure 3.5, we can see that the distribution of RMSE values between S1 and the autograd model is statistically equivalent ($p = 0.18$) to the

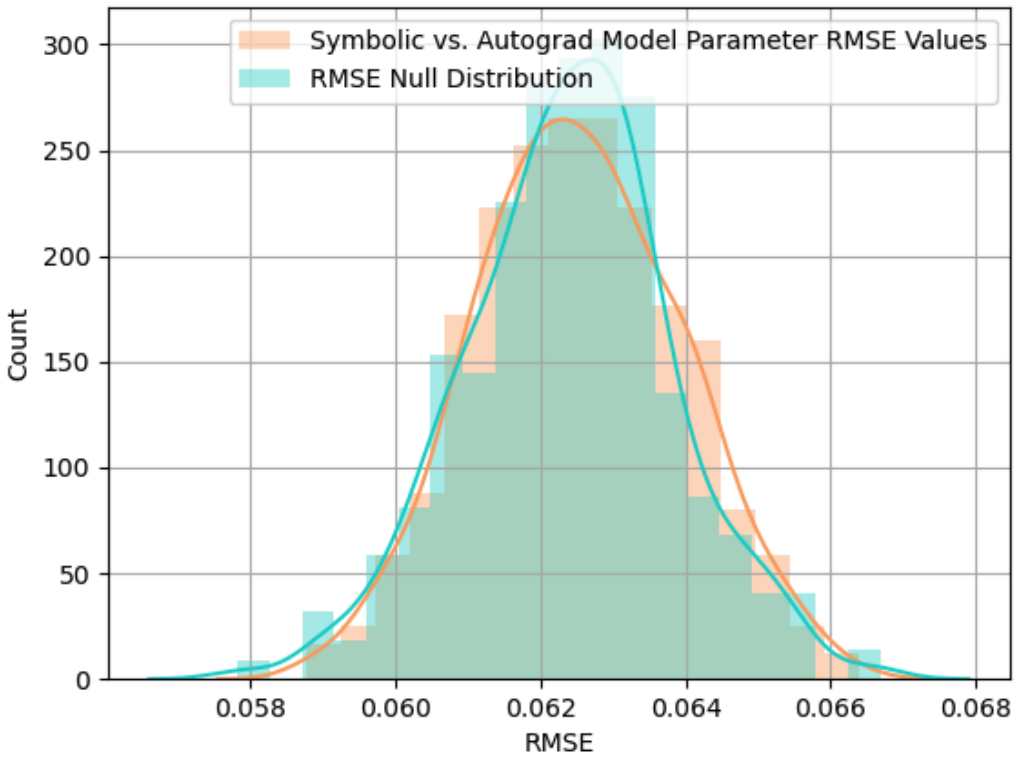

Figure 3.5: A visualization of the control parameter RMSE distribution (S1 vs. S2 parameter RMSE) versus the autograd vs. symbolic differentiation parameter RMSE distribution.

distribution of RMSE values between S1 and S2 (the null distribution shown). This equivalence implies that the Cryptograd model with autograd enabled is producing the same model parameters as the symbolic differentiation model. Thus, we can conclude that the Cryptograd model with autograd enabled is behaving as expected.

3.6.3 Cryptograd Empirical Performance and Timing Analysis

Figure 3.6 shows the training loss curve for the LR model with the parameters defined in Table 3.1. From this figure, we can see that the model exhibits some oscillation, but is able to converge to a global minimum. After training, the model was able to obtain an accuracy value of 90%. Tables 3.4 and 3.5 delineate the class-specific performance results on the testing dataset and a timing analysis of the implemented

model, respectively. Table 3.4 indicates that the model was much more likely to predict class 0 than class 1. Given the class imbalance prevalent in the dataset, this bias makes sense. By tuning the loss function and re-balancing the dataset, we can encourage more class 1 predictions, if class 1 recall is a priority.

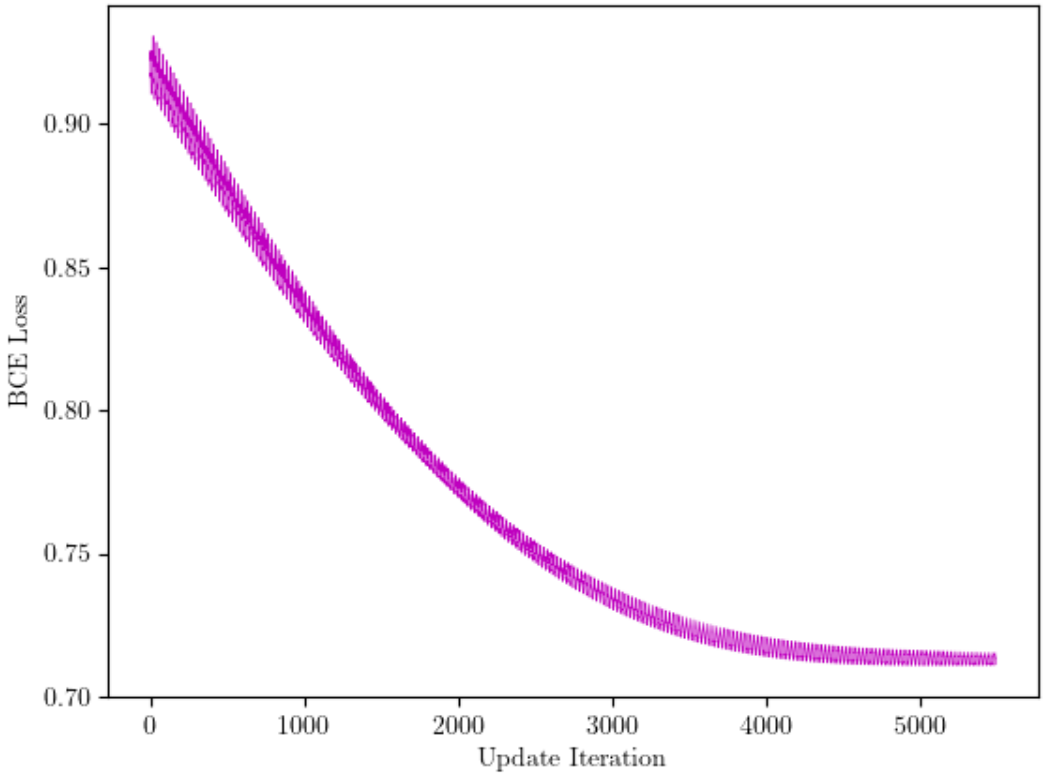

Figure 3.6: The training loss curve for a fully encrypted model training on the Androgen dataset.

Table 3.4: Class-specific performance metrics.

Class	Precision	Recall
0	0.89	1.00
1	1.00	0.21

Table 3.5: Timing analysis for training and inference.

Metric	Time (s)
Time to Encrypt 1 Batch	33.7 ± 0.2s
Time to Train 1 Epoch	67.2 ± 0.1s
Total Time to Train	670 ± 2s
Inference Time	31 ± 1s

With respect to timing, the model takes a relatively long time (670s) to train.

The Cryptograd library took about 2s to train an equivalent plaintext model for the exact same problem context. Thus, encrypting and training a model on the Androgen dataset is about 300x slower than the equivalent training process in plaintext. This slowdown is acceptable as training is performed only periodically. Data packing optimizations, a lower-degree activation function, and parallelization can be explored in future work to speed up the time to train and infer leading to a training time orders of magnitude faster.

3.7 Summary

In this Section, a training library for encrypted model development was implemented. In Section 3.6.1 we determined that the cost of utilizing the implemented autograd functionality in a training loop is approximately 20ms. Relative to the time to train a fully encrypted model (670 $\pm$ 2s), this performance degradation is small. In Section 3.6.2, we verified that the parameter updates when utilizing autograd are equivalent to the parameter updates obtained when using symbolic differentiation. Finally, in Section 3.6.3, we demonstrated the practicality of Cryptograd in a real-world setting by training a fully encrypted model on fully encrypted data using the Androgen dataset, obtaining an accuracy of 90% and a total training time of 670s using 12 of the 16 cores on an 11th Gen Intel Core i7-11700K @ 3.60GHz.

Currently, implementing a fully encrypted training loop that can be deployed to a server is very challenging. It requires expertise in a variety of fields including cryptography, optimization, and software engineering. The library presented in this chapter bridges the gaps among cryptographers, mathematicians, and model developers by providing an intuitive encrypted model development framework written in Python. This library lowers the barrier to entry that homomorphic encryption imposes on potential users and will holistically enable the broad usage of homomorphic encryption in a privacy-preserving ML setting.

Chapter 4

Decentralized FL for Secure Training and Inference

4.1 Introduction

This chapter focuses on FHE training in a federated learning (FL) context (discussed in Section 2.5.3). Specifically, a handful of modern optimization methods (SGD, momentum, NAG, Adagrad, RMSprop) are implemented in an encrypted FL training context, as well as a novel optimization method called DRMSprop (decentralized RMSprop), which stores the global gradient history on the public server. DRMSprop results in more accurate parameter updates which, in turn, lead to stronger performance and better generalizability (Section 4.5).

4.1.1 Motivation

As discussed in Section 2.5.3, in an FL training context, there is an aggregator server and N clients. The N clients have access to a unique subset of the total dataset. The decentralized aggregator server aggregates the parameter update vectors supplied by each of the N clients. In this way, multiple participants are able to utilize more informed models, as the aggregator has access to input data from a plethora of sources. This decentralized encrypted model can be deployed for encrypted inference making encrypted FL an attractive method for collaborative training. (See Figure 2.11 in Section 2.5.3 for a visualization of the FL training process. Note that this

figure shows only the communication between the aggregator and one of the clients. In an FL scheme, there are many clients. The process shown in Figure 2.11 is happening in parallel with many clients simultaneously).

Figures 4.1 and 4.2 compare training and testing loss curves for an individual client to the aggregator server. This preliminary result was obtained using the binary classification income prediction dataset detailed in Section 4.4.1. The training and testing losses in these Figures are the loss of the model on the entirety of the training and testing datasets at each iteration. These losses would not normally be computed in an actual FL training loop, but were computed to better understand the performance of the scheme.

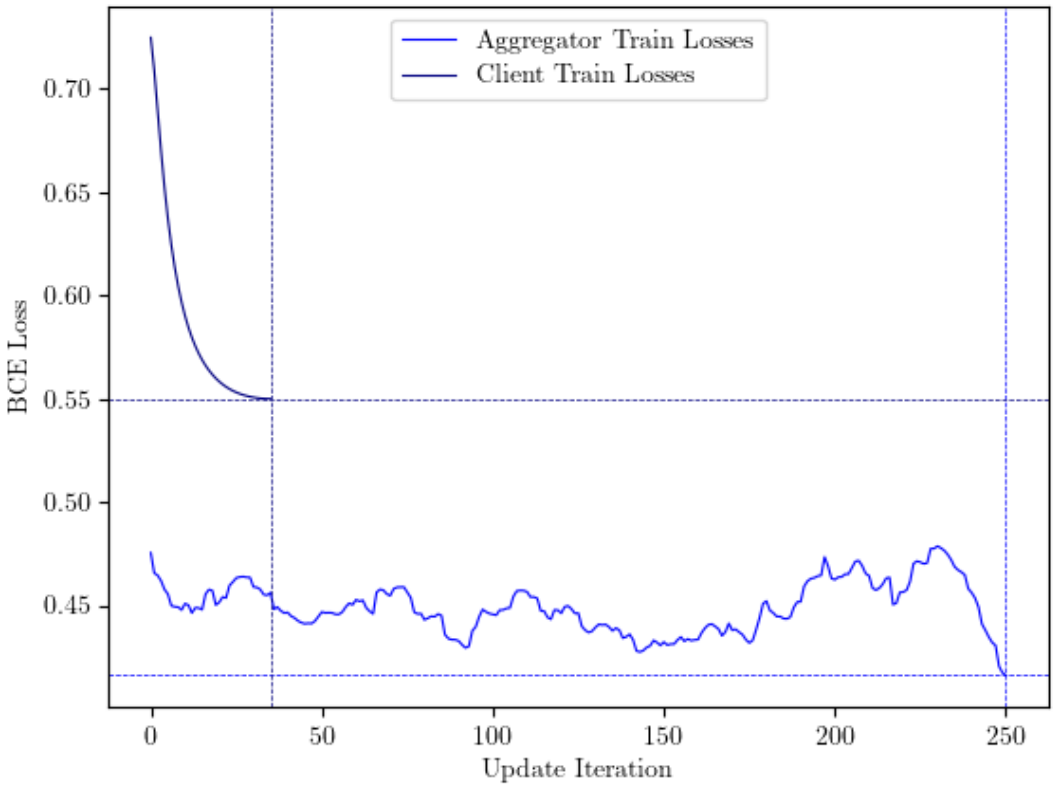

Figure 4.1: A loss curve comparing the client training loss to the aggregator training loss.

Notice that the aggregator is able to obtain much better performance (lower training and testing losses) than the average client, which is to be expected since the

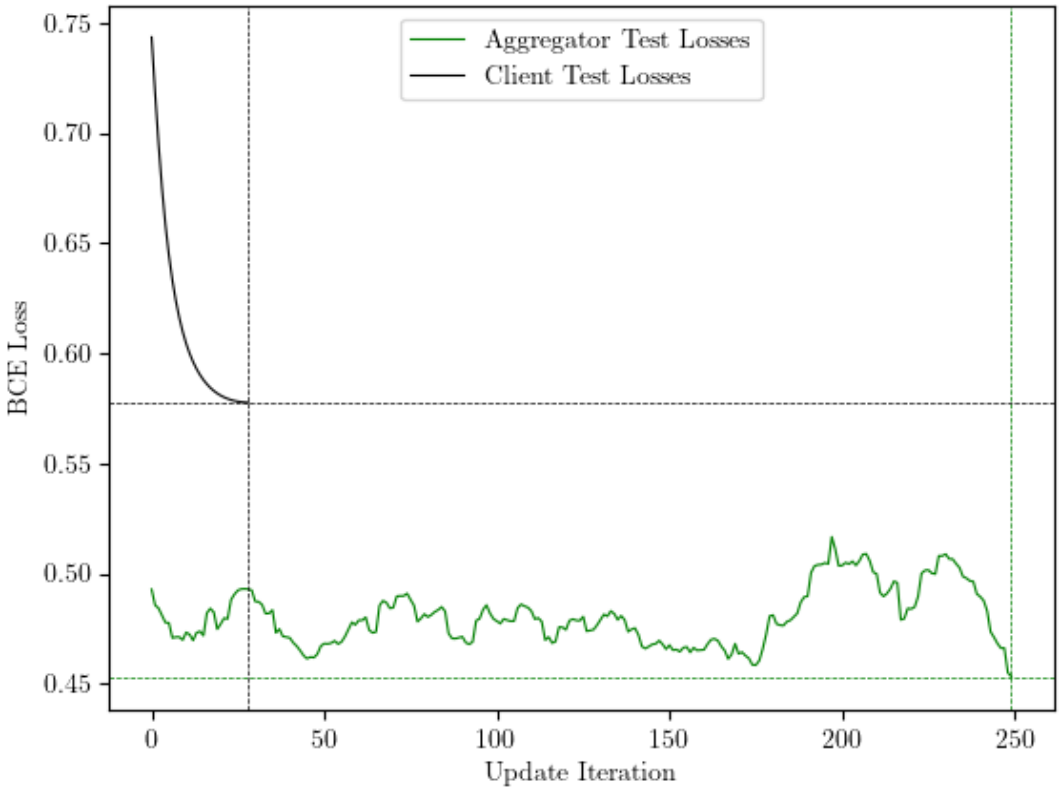

Figure 4.2: A loss curve comparing the client testing loss to the aggregator testing loss.

aggregator has access to training data from all $N = 500$ clients, whereas each client has access to only a small fraction of the training data. By taking advantage of an aggregator server, individual clients are able to cooperatively elevate model performance while maintaining data privacy.

Note that the benefits that individual clients experience from receiving aggregator model data grow as the size of the training data that each individual client is exposed to shrinks, as shown in Figure 4.3. Figure 4.3 shows the testing loss curves for clients utilizing training data percentages ranging from 0.1% to 1.0%. Model generalizability and performance drop off sharply when clients do not have access to sufficient training data. In contexts where individual client nodes need to perform secure and accurate inferences, but do not have access to sufficient plaintext training data on a private server, the benefits of an aggregator server are significant.

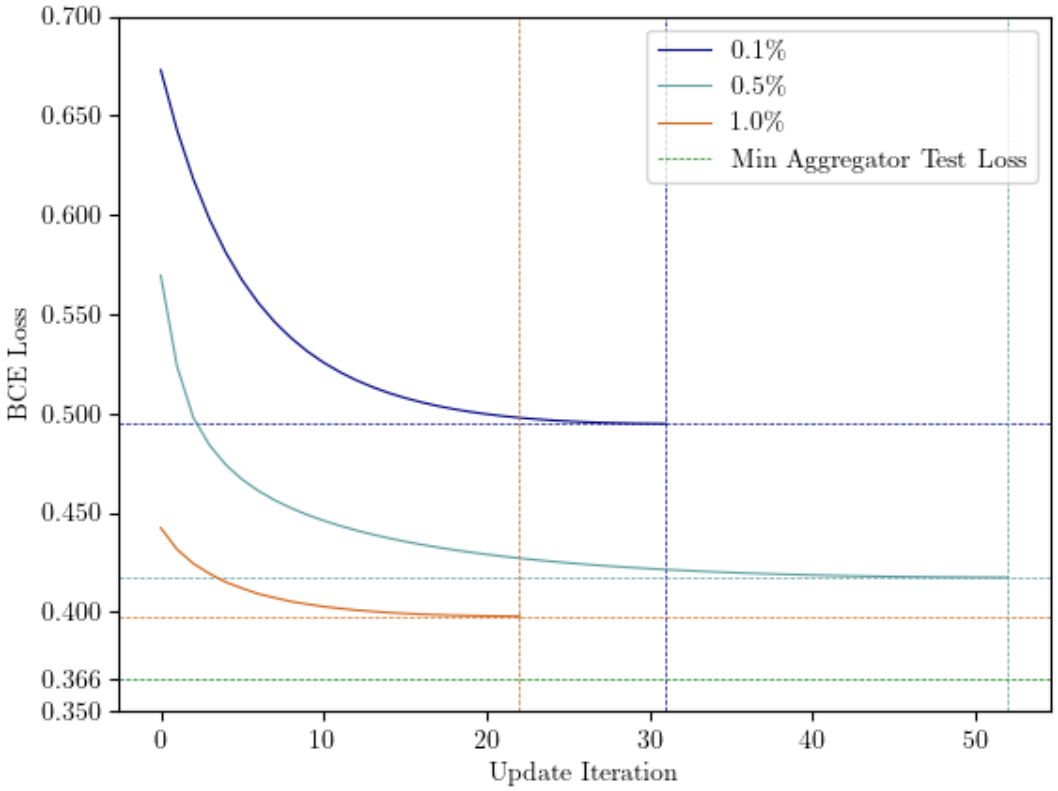

Figure 4.3: Testing loss curves illustrating the impact of varying the amount of training data clients can access.

4.1.2 Literature Review

Much research has been done to optimize pre-trained models encrypted for FHE inference [2–6]. Some research has been done on encrypted training, but mostly in a centralized setting [7–9]. Papers that propose the use of FHE to guarantee privacy in a decentralized FL setting focus on security. For example, [50]'s FL system protects this gradient information using FHE. [53] presents general FHE FL architectures and [52] suggests a novel encoding strategy for more efficient gradient computations. Finally, [54] combines HE with differential privacy (DP) to prevent data leakage among honest-but-curious clients in an FL training context. While these papers ensure absolute privacy among clients, even in the presence of malicious actors, there has been little research in the arena of decentralized model optimization.

For example, [50] and [54] use stochastic gradient descent (SGD), [53] suggests com-

puting generic gradient vectors, and [52] computes a quantized batch of GD gradient vectors. By using other more modern optimization methods such as Nesterov's accelerated gradient (NAG), Adagrad, and RMSprop, we can improve the performance of decentralized FL systems.

4.2 DRMSprop

In a typical FL scheme, the client uses the global model parameters supplied by the aggregator in conjunction with a batch of training data to compute a gradient update using SGD. Each client could also take advantage of other optimization methods (momentum, NAG, Adagrad, RMSprop) by storing local gradient update information at each iteration. For example, a client could use momentum by storing and incorporating the previous local gradient update into subsequent gradient updates. Note that these optimization methods can track the gradient history computed by each individual client locally, but not the gradient history originating from other clients, because clients do not share gradient information. Thus, we will refer to optimization methods that store gradient information stemming only from local data as local optimization methods.

DRMSprop incorporates global gradient information into local parameter updates by storing an encrypted global gradient history vector on the aggregator server. Including global gradient information into local parameter updates requires some modifications to the FL training process, as detailed below and in Figure 4.4.

1. The aggregator encrypts and sends each client the encrypted model parameters, $E[\beta]$, *and* an encrypted global gradient history vector, $E[G]$.

2. The client decrypts $E[\beta]$ and $E[G]$ using the private key.

3. The client computes and encrypts a gradient vector, g, as well as an updated global gradient history vector, G_{t+1} using the RMSprop update equations (detailed in Equations 2.41 and 2.42).

4. The client transmits $E[g]$ and $E[G_{t+1}]$ to the aggregator server.

5. The aggregator securely updates the parameters of the model by computing $E[\beta] = E[\beta] + E[g]$ and replaces $E[G]$ with $E[G_{t+1}]$.

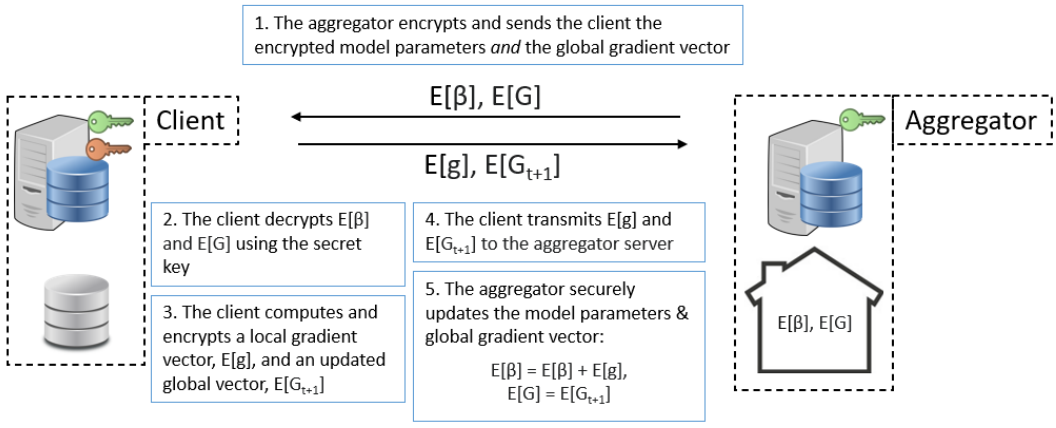

Figure 4.4: The modified DRMSprop FL scheme. The green key is the PK and the red key is the SK. Notice that each client has access to both the PK and the SK, but the aggregator has access to only the PK. In this figure, only the communication between one client and the aggregator is shown, but this process occurs multiple times with all N clients. The original scheme is shown in Figure 2.11. Notice that, in the original scheme, there is no $E[G]$ term.

Note that the main difference between DRMSprop (detailed in Figure 4.4) and the original FL scheme (Figure 2.11) is the DRMSprop scheme requires storage and transportation of the gradient vector, $E[G]$. This vector contains the global RMSprop gradient information (see Section 2.4.6). It is updated by each client using equation 2.42 when that particular client computes the gradient.

4.3 FL Scheme Security

In this section, the FL scheme described above will be evaluated according to its robustness against three types of attacks. Firstly, attacks from malicious actors that intercept information during transmission. Secondly, attacks from parties that gain access to the aggregator server and are privy to all computation performed on this server. Finally, attacks from other malicious clients that have privileged access to the

system. Note that the above attacks are all information attacks. The key assumption in the FL scheme defined above is that the participants are honest-but-curious. This means that the participants are actively trying to recover sensitive information, but are honestly corresponding with the system.

This system is robust against an attacker that gains access to transmission information. All gradient and model parameter information is encrypted using homomorphic encryption prior to transmission. Since we assume that the attacker has not gained access to the secret key, the FL system defined above is robust to this type of an attack. Homomorphic encryption also prevents attackers from extracting information from the aggregator server during computation. The aggregation of the model parameters is done without decrypting any of the gradient or parameter information.

With respect to malicious client attacks, however, this scheme has two vulnerabilities. Firstly, without the use of multi-key encryption or additional limitations on aggregator model queries, a client can query the aggregator for the model parameters at each time-step. Then, the malicious client actor can take the difference between the model parameter states to recover gradient information from other clients. [50] proves that input data can be recovered from gradient information, and so malicious clients can recover input data from other clients using this type of an attack. The use of multi-key encryption prevents this type of an attack by modifying the FL scheme above to incorporate a consensus system during decryption [51].

The second type of malicious client attack is called a membership inference attack (MIA). After the model has been trained, a malicious client can determine if a certain data element was included in the training data. It has been shown that FL is vulnerable to these types of attacks, however, as the number of clients in the FL scheme grows, the effectiveness of the MIA attack is diminished [63].

4.4 Experimental Setup

In Section 4.5, we compare the performance of local optimization methods in an FL context. Specifically, we contrast the aggregator training loss curves, testing loss curves, and accuracy curves for models utilizing SGD, local momentum, local NAG, local Adagrad, and local RMSprop. Then, we compare the performance of DRMSprop with local Adagrad and local RMSprop using equal plots to compare the local optimizers. Finally, we compare the performance of DRMSprop with a centralized insecure model trained using Sci-Kit Learn. Our goal when choosing an optimization method is to minimize the difference between the performance of the decentralized model encrypted using FHE and the centralized insecure model. The number of clients, N, is 500 for all experiments. We assume that each client has the same amounts of training data available. Furthermore, we assume that each client is using the exact same optimization method and is cooperating (computing $E[G]$ and g from Figures 2.11 and 4.4 honestly).

The dataset used for these experiments, rationale for selecting this dataset, and the ETL processing pipeline used to transform this data are detailed in Section 4.4.1. The primary HE model architecture, HE layer approximations, and centralized insecure benchmark model are explained in Section 4.4.2. The hardware setup for this section matches the hardware setup used to evaluate the Cryptograd library in Chapter 3 (Section 3.4.1), namely, an Intel Core i7-11700K @ 3.60GHz utilizing 12 of 16 available cores. Further, the Cryptograd library (Chapter 3) was used to implement, train, and evaluate all of the optimizers and models used in this section.

4.4.1 Dataset

The dataset used for the experiments in this section is the Census-Income (KDD) Data Set [64] hosted on the UCI ML dataset database. This binary classification dataset has 299,285 records and 40 attributes describing American citizens, as well as

a label column that indicates whether or not a data entry had a personal net income greater than $50,000. The dataset contains categorical and integer values.

After importing the data and removing NaN values, there were 187,141 class 0 data entries and 12,382 class 1 data entries. To rebalance the data, SMOTE [38] was used. We chose to use SMOTE instead of down-sampling the data because FL is generally more suitable for big data contexts. The columns selected for the analysis, each column's data type, and a short column description are included in Table 4.1. These features were chosen through trial and error to obtain an optimization space that was challenging enough to contrast the performance of the various optimization methods tested in the subsequent section. Features that had a strong linear correlation to the target variable were removed until the optimization problem became sufficiently difficult.

Each categorical feature was dummy (one-hot) encoded to prevent implicit relationships from forming in the data. Furthermore, the "age" category was parsed into 4 age ranges (1-25, 25-50, 50-75, 75+) and then dummy encoded. This parsing strategy was chosen to prevent data bloating and sparsity. Given the size of the dataset, none of the other selected columns was particularly sparse. After accounting for not a number (NaN) values, rebalancing, parsing various categorical features, and one-hot encoding, the remaining dataset had 384,478 rows and 285 columns.

This dataset was chosen to simulate a real-world FL context. The census income dataset is vast, so it is useful to simulate an FL context in which there are tremendous amounts of training data distributed among many clients. Furthermore, the census income dataset has many features, allowing us to make the optimization space more challenging by removing features that linearly correlate with the predictor variable. By making the optimization space more challenging, we can better appreciate the performance advantages and disadvantages of the various optimization methods discussed in Section 4.5.

After parsing the data, we split it into a training dataset, a testing dataset, and

Table 4.1: The data dictionary for the personal income dataset.

Feature	**Data Type**	**Description**
Age	Integer	The person's age.
Class	Categorical	The person's employment class.
Education	Categorical	The level of education that this person has received.
Enrollment	Boolean	Whether this person is currently enrolled in any educational institution.
Marital Status	Categorical	The person's marital status.
Industry Code	Categorical	The industry the person works in.
Occupation Code	Categorical	The person's general occuptation.
Sex	Categorical	The person's sex.
Unionized	Boolean	Whether the person's place of work has unionized or not.
State	Categorical	The person's state of residence.
Father's Birth	Categorical	This person's father's place of birth.
Mother's Birth	Categorical	This person's mother's place of birth.
Self Birth	Categorical	This person's place of birth.
Citizenship	Categorical	This person's citizenship status.

an initialization dataset. The testing dataset contains 20% of the total dataset, the training dataset contains 19.5% of the total dataset, and the initialization dataset contains 0.05% of the total dataset. As mentioned in Section 4.4, we used $N = 500$ clients for each experiment. Thus, we split the training dataset into 500 equal partitions. Each client is assigned one of these datasets. During training, each client utilizes its assigned dataset as well as the most up to date aggregator parameters, g and $E[G]$ (if utilizing the DRMSprop scheme) to compute gradient updates.

4.4.2 Models

An LR model with a sigmoid approximation layer was used to model the input data. The sigmoid approximation layer was tuned according to the range of the outputs of the model after training. The range of model outputs varied significantly depending on which optimization method was used so the sigmoid approximation layer had to be tuned for each optimization method individually. Other than the sigmoid approximation layer variation, the model architecture was held constant throughout the experiments to ensure a fair comparison of the different optimization methods. We could have implemented an NN to improve the performance of the decentralized HE model, however, the purpose of these experiments is to demonstrate the performance of various optimization methods, not model architectures.

The HE-safe LR model discussed above was compared with a centralized insecure fully-connected NN. The NN was implemented using PyTorch [65]. It has one hidden layer and one output layer. The hidden layer has $D \times 2 = 570$ units. These layers utilize perfect sigmoid activation functions and a fully centralized training loop.

4.5 Results and Discussion

4.5.1 Local Optimizer Comparison

Figures 4.5 and 4.6 illustrate the testing accuracy and training loss curves of the aggregator model when using SGD, local momentum, local NAG, local Adagrad, and local RMSprop in an FL context. Note that the horizontal and vertical dotted lines indicate minimum loss values and maximum accuracy values. Note that the training loss curve in Figure 4.6 illustrates the loss of the aggregator on the total training dataset at each iteration.

From Figure 4.5, we can see that local Adagrad and local RMSprop perform the strongest. Interestingly, each optimizer's training loss begins to increase (with SGD being the exception) within the first few update iterations. This training loss could be

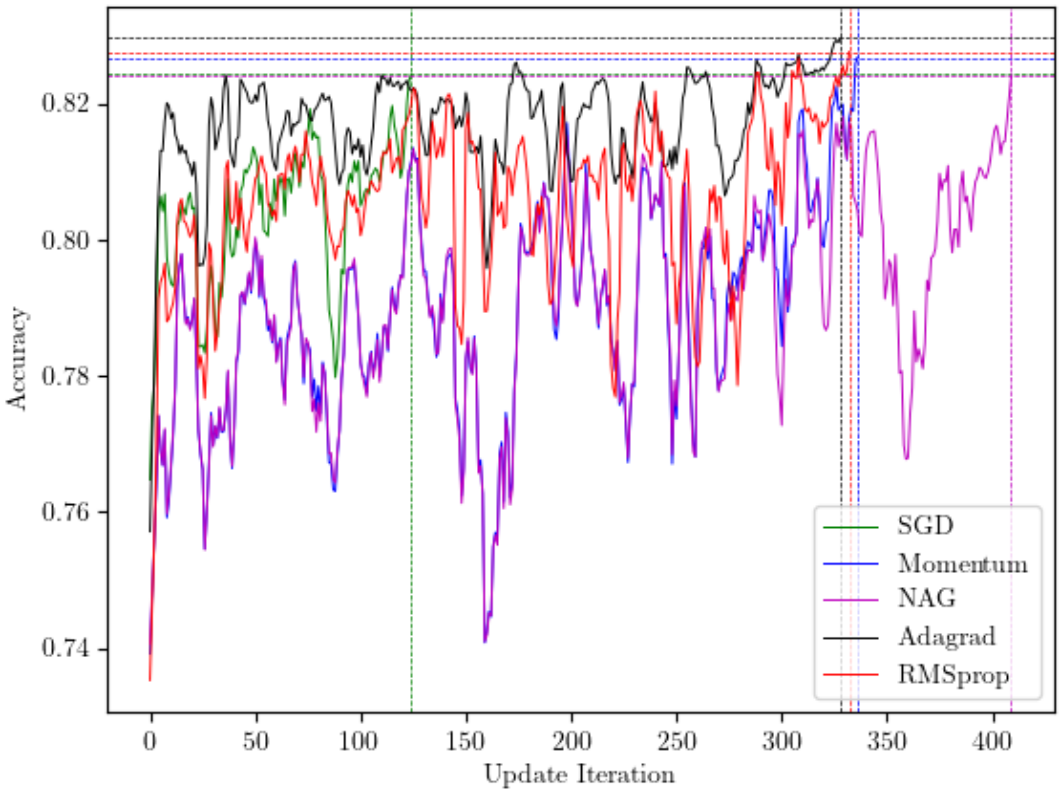

Figure 4.5: A comparison of the accuracy curves of the local optimizers (SGD, local momentum, local NAG, local Adagrad, and local RMSprop). The horizontal and vertical dotted lines indicate where each optimizer reached it's maximum testing accuracy value.

due to over-fitting to local data, since each optimizer has access to a small percentage of the total training data for training purposes. Since SGD does not have a momentum term, it is possible that the SGD optimizer is getting stuck in some sort of local minimum, causing it to follow a different path from the other optimizers, all of which utilize a momentum term.

None of the local optimizers above modify the way in which information is exchanged between the aggregator and its clients. Encrypted gradient information is passed to the aggregator server and the aggregator server returns encrypted model parameters (as in Section 2.5.3). If implementing a modified FL scheme (like the one presented in 4.2) is not possible, the results above indicate that the local Adagrad and local RMSprop optimizers yield the strongest performance on the holdout testing

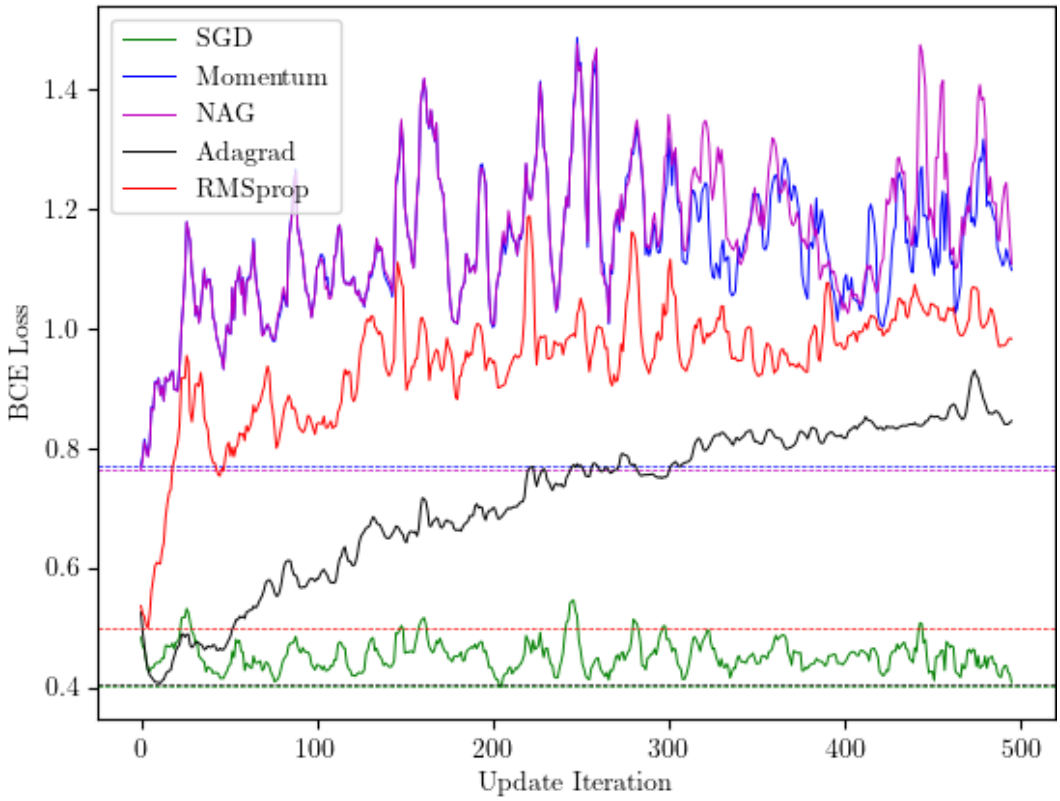

Figure 4.6: A comparison of the training losses of the local optimizers (SGD, local momentum, local NAG, local Adagrad, and local RMSprop). The horizontal lines in this figure indicate the minimum training loss values obtained by each optimizer.

dataset, despite their peculiar training curves.

4.5.2 DRMSprop Performance

Figures 4.7 and 4.8 illustrate the testing accuracy and training loss curves for the local Adagrad, local RMSprop, and DRMSprop optimizers in an FL context. Two things stand out from these Figures. Firstly, the maximum accuracy obtained by DRMSprop (Figure 4.7) is noticeably higher than the accuracy obtained using the other optimization methods. Secondly, the DRMSprop training curve (Figure 4.8) monotonically approaches a minimum value smoothly and efficiently. The other local optimizers behave more sporadically and are non-monotonic.

These results indicate that the DRMSprop optimizer does a better job of optimizing the performance of the centralized model. Furthermore, since the loss values obtained

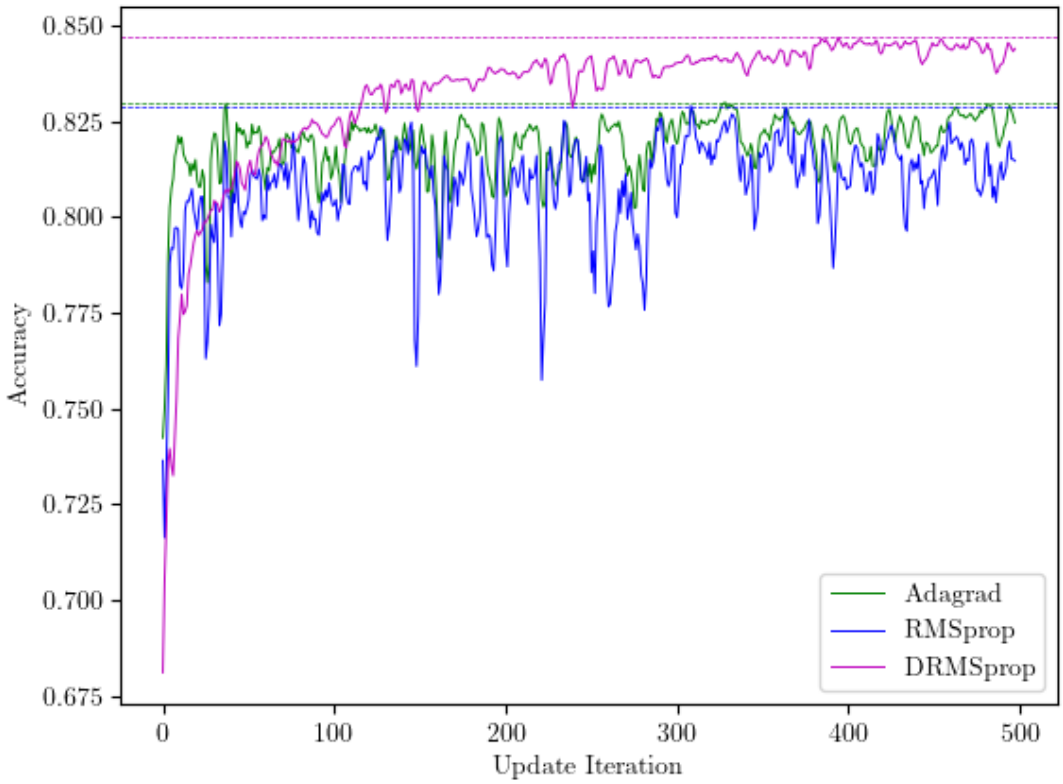

Figure 4.7: A comparison of the accuracy curves of local Adagrad, local RMSprop, and the contribution of this thesis, DRMSprop.

by DRMSprop are smooth relative to the other optimizers, convergence will be more consistent when using DRMSprop.

Table 4.2 delineates the maximum testing accuracy values obtained by each of the optimizers, and Figure 4.9 compares the testing accuracy curve of the HE-safe model utilizing the DRMSprop optimizer with the maximum testing accuracy obtained by the centralized plaintext NN model. From Table 4.2 and Figure 4.9, we can see that the difference between the centralized plaintext NN model and the HE-model utilizing DRMSprop amounts to about 3%. The other optimizers lag behind DRMSprop by about 2%, with RMSprop being the strongest local optimizer.

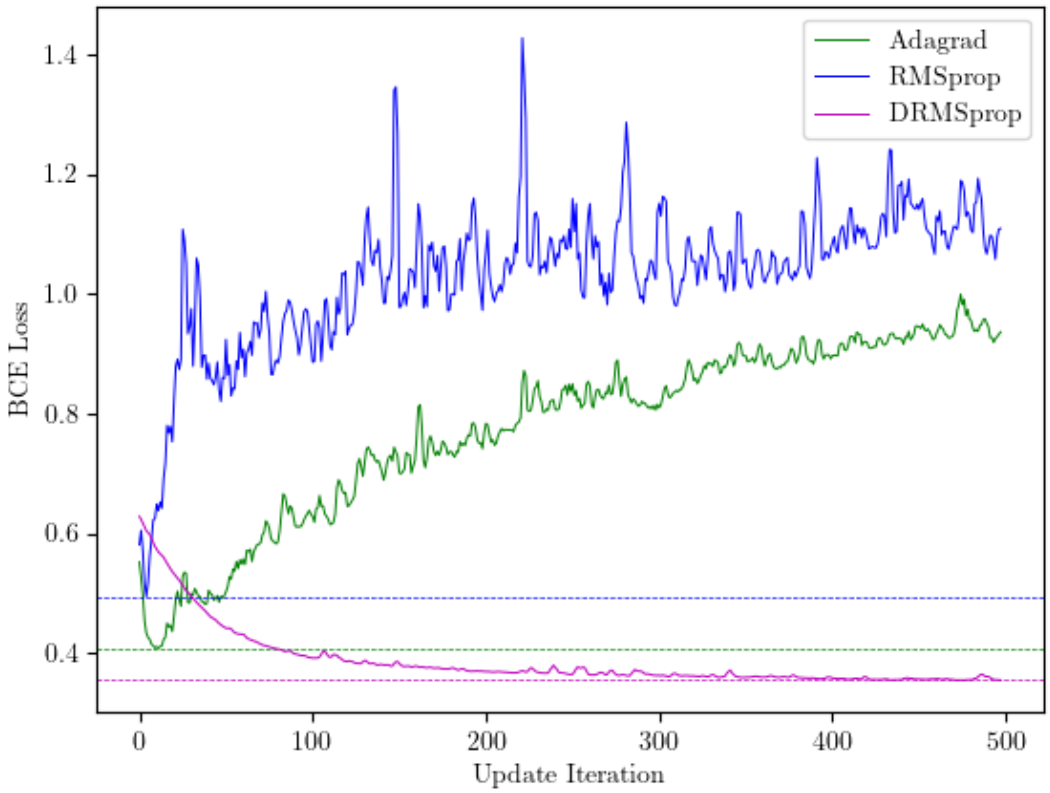

Figure 4.8: A comparison of the training loss curves of local Adagrad, local RMSprop, and the contribution of this thesis, DRMSprop.

Table 4.2: The maximum aggregator testing accuracies for each optimizer for this dataset.

Optimizer/Model	**Maximum Accuracy Value**
SGD	82.4%
Local Momentum	82.6%
Local NAG	82.4%
Local Adagrad	83.0%
Local RMSprop	82.8%
DRMSprop	84.7%
Centralized NN	87.0%

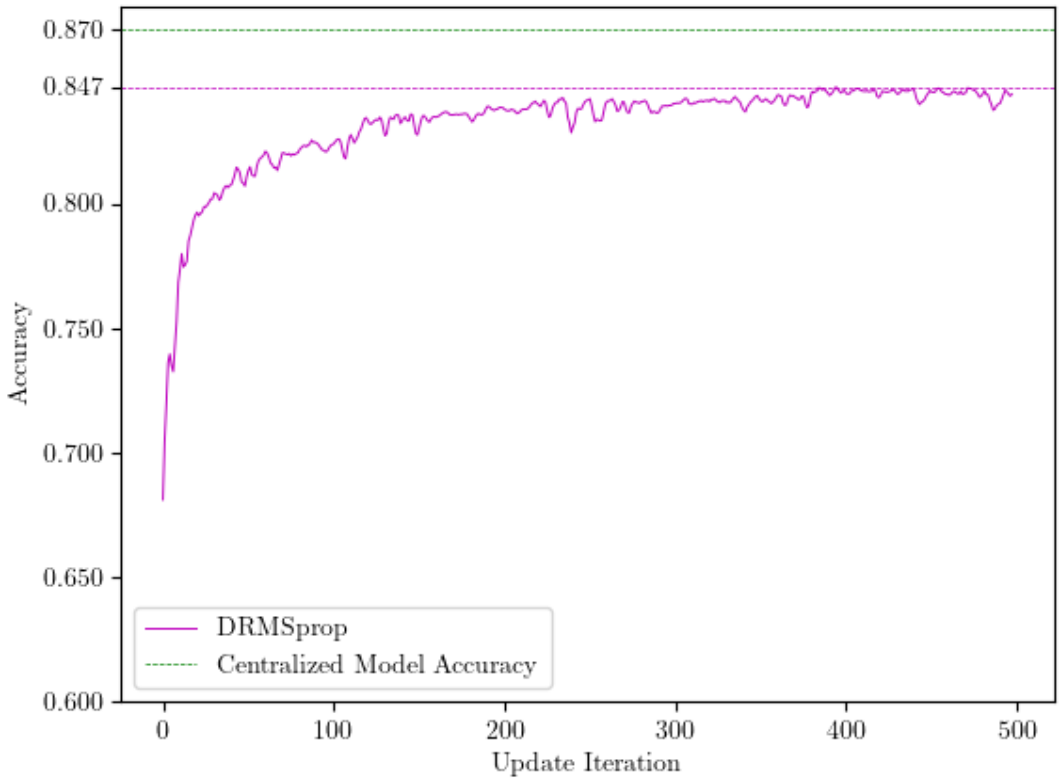

Figure 4.9: The accuracy curve of the DRMSprop model versus the optimal accuracy obtained by the centralized plaintext NN model.

4.5.3 Time and Space Storage and Communication Requirements

Each of the local optimizers require the aggregator to store the encrypted model parameters. For an LR model with D dimensionality, this amounts to a single ciphertext. For a multi-layer NN with k layers, assuming one ciphertext per layer is required, there will be k ciphertexts stored on the aggregator server ($O(k)$ ciphertext storage). DRMSprop requires the aggregator server to store k ciphertexts for the model parameters and k ciphertexts for the most up-to-date gradient history vector, G ($O(2k) = O(k)$ ciphertext storage).

With respect to communication, the local optimizers transmit only the encrypted gradient updates to the aggregator server and receives the most up to date encrypted model parameters. The updated model parameters are computed on the aggregator server securely using homomorphic addition. DRMSprop, on the other hand, trans-

mits both the encrypted gradient updates as well as the encrypted gradient history vector. It receives both the most up to date encrypted model parameters and the encrypted gradient history vector from the aggregator. Model parameter updates are also done securely on the aggregator server using homomorphic addition. In short, DRMSprop transmits and receives an additional k ciphertexts at each iteration to facilitate the usage of the gradient history vector. For the example experiment, since we were utilizing an LR model, only 1 additional ciphertext per client was transported to and from the aggregator server during each training iteration.

4.6 Summary

The DRMSprop optimizer obtained a testing accuracy of 84.7%. This is a 1.7% improvement in the performance of the local Adagrad optimizer which yielded a testing accuracy of 83.0%. Based on the performance of the centralized model, this privacy-preserving system is only 2.3% less accurate than a centralized solution, since the centralized NN was able to obtain an accuracy of 87.0%.

With respect to time and space requirements, utilizing the DRMSprop optimizer requires an additional k (where k is the number of fully-connected model layers) ciphertexts to be transmitted to, stored on, and received from the aggregator server at each iteration. For an LR model, one additional ciphertext per iteration is not significant. Perhaps in situations where multi-layer NNs are being used and the training process is run for many epochs, these additional space and transmission requirements may have a larger impact on the memory and networking limitations of the system. If a modified FL scheme is not possible, then RMSprop and Adagrad are strong local optimizer choices obtaining accuracy values of 82.8% and 83.0%, respectively. To summarize, careful optimizer selection leads to significant stability, convergence, and performance increases in federated learning, as illustrated by the results in this chapter.

Chapter 5

Contributions, Conclusions, and Future Work

5.1 Contributions and Conclusions

The contributions of this thesis are twofold. Firstly, an encrypted training and inference library written in Python for intuitive model training and deployment was developed and presented. In Chapter 3, we demonstrated that the timing cost of utilizing Autograd, while not zero, is negligible (20ms). We also demonstrated that the parameters obtained when using the Cryptograd library are empirically equivalent to parameters obtained using hard-coded derivatives. Finally, we implemented a fully-encrypted model using this library, which obtained an accuracy of 90% on the Androgen dataset and had a total training time of 670s on an Intel Core i7-11700K @ 3.60GHz with 16 cores. This library is a proof-of-concept demonstrating that a scalable and intuitive encrypted training and inference library built using Autograd is possible.

The second contribution of this thesis is a modified federated learning scheme that utilizes DRMSprop: a decentralized optimizer that utilizes a shared gradient history vector. This scheme requires minor additional communication and storage relative to other optimization methods, but yields a 1.7% increase in accuracy on a large income prediction dataset relative to the other local optimizers for $N = 500$. Accuracy is reduced by 2.3% over the optimal centralized model. The contributions made in

Chapter 4 will lead to better federated learning training convergence, performance, and reliability.

5.2 Future Work

5.2.1 Convolutional Layers

In Chapter 3, we focused on fully connected layers and structured data. A logical extension would be to implement automatic differentiation functionality for convolutional layers. Not only would autodiff functionality for convolutional layers unlock a new set of ML tasks, it would also make it easier to benchmark the performance of the model, since the MNIST image recognition dataset is the standard used for benchmarking ML frameworks and algorithms [2, 4, 5].

5.2.2 Automatic Sigmoid Approximations for Inference

Currently, implementing a custom sigmoid approximation using the Cryptograd library requires the definition of a layer type with the desired approximation degree and hard coded parameters. These parameters are based on the range of the model outputs after training. If the model outputs range from -10 to 10, for example, then it is necessary to generate an approximation that is accurate over this range. It would be very useful to have functionality built into the library to do this automatically. It would be interesting to investigate different ways to automatically generate this approximation by treating the approximation as a part of the optimization process.

5.2.3 Cryptograd Optimization Opportunities

The Cryptograd library is written in Python and built on top of the C++ BLUE encryption library. The BLUE encryption library is high-performance and has been optimized in many ways. Python, however, is one of the slowest programming languages. Given that the main training loop is written in Python, the performance of the library is much slower than it could be. Furthermore, there were only a few

software parallelization considerations and no hardware acceleration considerations utilized in the present work. By looking into these optimization opportunities, the performance of the library could be significantly improved.

Bibliography

[1] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," *LNCS*, vol. 6632, pp. 129–148, May 2011. DOI: 10.1007/978-3-642-20465-4_9.

[2] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," Proceedings of Machine Learning Research, vol. 48, M. F. Balcan and K. Q. Weinberger, Eds., pp. 201–210, 2016. [Online]. Available: https://proceedings.mlr.press/v48/gilad-bachrach16.html.

[3] A. Sanyal, M. J. Kusner, A. Gascón, and V. Kanade, "Tapas: Tricks to accelerate (encrypted) prediction as a service," *arXiv*, 2018. DOI: 10.48550/ARXIV.1806.03461. [Online]. Available: https://arxiv.org/abs/1806.03461.

[4] E. Hesamifard, H. Takabi, and M. Ghasemi, "Deep neural networks classification over encrypted data," *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, CODASPY '19, pp. 97–108, 2019. DOI: 10.1145/3292006.3300044. [Online]. Available: https://doi.org/10.1145/3292006.3300044.

[5] G. Lloret-Talavera *et al.*, "Enabling homomorphically encrypted inference for large DNN models," *IEEE Transactions on Computers*, vol. 71, no. 5, pp. 1145–1155, 2022. DOI: 10.1109/tc.2021.3076123. [Online]. Available: https://doi.org/10.1109%2Ftc.2021.3076123.

[6] S. Meftah, B. H. M. Tan, K. M. M. Aung, L. Yuxiao, L. Jie, and B. Veeravalli, "Towards high performance homomorphic encryption for inference tasks on cpu: An mpi approach," *Future Generation Computer Systems*, vol. 134, pp. 13–21, 2022, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2022.03.033. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X22001145.

[7] K. Nandakumar, N. Ratha, S. Pankanti, and S. Halevi, "Towards deep neural network training on encrypted data," pp. 40–48, 2019. DOI: 10.1109/CVPRW.2019.00011.

[8] F. Bergamaschi, S. Halevi, T. T. Halevi, and H. Hunt, "Homomorphic training of 30,000 logistic regression models," *Cryptology ePrint Archive, Paper 2019/425*, 2019, https://eprint.iacr.org/2019/425. DOI: 10.1007/978-3-030-21568-2_29. [Online]. Available: https://eprint.iacr.org/2019/425.

[9] S. Carpov, N. Gama, M. Georgieva, and J. R. Troncoso-Pastoriza, "Privacy-preserving semi-parallel logistic regression training with fully homomorphic encryption," 2019, https://eprint.iacr.org/2019/101. [Online]. Available: https://eprint.iacr.org/2019/101.

[10] C. Gentry, "Fully homomorphic encryption using ideal lattices," *Proceedings of the Annual ACM Symposium on Theory of Computing*, vol. 9, pp. 169–178, Jan. 2009. DOI: 10.1145/1536414.1536440.

[11] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Advances in Cryptology – EUROCRYPT 2010*, H. Gilbert, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 24–43, ISBN: 978-3-642-13190-5.

[12] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12, Cambridge, Massachusetts: Association for Computing Machinery, 2012, pp. 309–325, ISBN: 9781450311151. DOI: 10.1145/2090236.2090262. [Online]. Available: https://doi.org/10.1145/2090236.2090262.

[13] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '12, New York, New York, USA: Association for Computing Machinery, 2012, pp. 1219–1234, ISBN: 9781450312455. DOI: 10.1145/2213977.2214086. [Online]. Available: https://doi.org/10.1145/2213977.2214086.

[14] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," *Proceedings of Advances in Cryptology-Crypto*, vol. 7417, Aug. 2012. DOI: 10.1007/978-3-642-32009-5_50.

[15] J. Bos, K. Lauter, J. Loftus, and M. Naehrig, "Improved security for a ring-based fully homomorphic encryption scheme," *Proceedings of Cryptography and Coding - 14th IMA International Conference*, vol. 8308, 45—64, Dec. 2013. DOI: 10.1007/978-3-642-45239-0_4.

[16] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, pp. 34–91, Apr. 2019. DOI: 10.1007/s00145-019-09319-x.

[17] J. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," *Advances in Cryptology - ASIACRYPT*, vol. 10624, pp. 409–437, Nov. 2017. DOI: 10.1007/978-3-319-70694-8_15.

[18] M. Kim, Y. Song, S. Wang, Y. Xia, and X. Jiang, "Secure logistic regression based on homomorphic encryption: Design and evaluation," *JMIR Medical Informatics*, vol. 6, pp. 23–31, 2018.

[19] A. Kim, Y. Song, M. Kim, K. Lee, and J. Cheon, "Logistic regression model training based on the approximate homomorphic encryption," *BMC Medical Genomics*, vol. 11, pp. 23–31, Oct. 2018. DOI: 10.1186/s12920-018-0401-7.

[20] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic regression on homomorphic encrypted data at scale," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 9466–9471, 2019. DOI: 10.1609/aaai.v33i01.33019466. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/5000.

[21] K. Han, S. Hong, J. H. Cheon, and D. Park, "Efficient logistic regression on large encrypted data," *IACR Cryptol. ePrint Arch.*, vol. 2018, pp. 662–693, 2018.

[22] E. Lee *et al.*, "Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions," 2021, https://eprint.iacr.org/2021/1688. [Online]. Available: https://eprint.iacr.org/2021/1688.

[23] K. Mihara, R. Yamaguchi, M. Mitsuishi, and Y. Maruyama, "Neural network training with homomorphic encryption," *ArXiv*, vol. abs/2012.13552, 2020.

[24] J.-W. Lee *et al.*, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *IEEE Access*, vol. 10, pp. 30 039–30 054, 2022. DOI: 10.1109/ACCESS.2022.3159694.

[25] S. Halevi and V. Shoup, *HElib*, 2022. [Online]. Available: https://github.com/homenc/HElib (visited on 06/10/2022).

[26] D. Cousins *et al.*, *Palisade*, 2022. [Online]. Available: https://gitlab.com/palisade/palisade-release (visited on 06/10/2022).

[27] *Microsoft SEAL (release 4.0)*, https://github.com/Microsoft/SEAL, Microsoft Research, Redmond, WA., Mar. 2022. (visited on 06/10/2022).

[28] J. H. Cheon, A. Kim, M. Kim, and Y. Song, *Palisade*. [Online]. Available: https://github.com/snucrypto/HEAAN.

[29] A. A. Badawi *et al.*, *OpenFHE: Open-source fully homomorphic encryption library*, Cryptology ePrint Archive, Paper 2022/915, https://eprint.iacr.org/2022/915, 2022. [Online]. Available: https://eprint.iacr.org/2022/915.

[30] V. Lyubashevsky, C. Peikert, and O. Regev, "A toolkit for ring-lwe cryptography," in *Advances in Cryptology – EUROCRYPT 2013*, T. Johansson and P. Q. Nguyen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 35–54, ISBN: 978-3-642-38348-9.

[31] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *J. ACM*, vol. 56, no. 6, pp. 1–40, 2009, ISSN: 0004-5411. DOI: 10.1145/1568318.1568324. [Online]. Available: https://doi.org/10.1145/1568318.1568324.

[32] "Data protection in virtual environments," *Defense Advanced Research Projects Agency*, 2022. [Online]. Available: https://www.darpa.mil/program/data-protection-in-virtual-environments#:~:text=The%20Data%20Protection%20in%20Virtual,across%20DoD%20and%20commercial%20applications. (visited on 06/20/2022).

[33] R. Lindner and C. Peikert, "Better key sizes (and attacks) for lwe-based encryption," *Cryptology ePrint Archive, Paper 2010/613*, 2010, https://eprint.iacr.org/2010/613. [Online]. Available: https://eprint.iacr.org/2010/613.

[34] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *Advances in Cryptology – EUROCRYPT 2018*, J. B. Nielsen and V. Rijmen, Eds., Cham: Springer International Publishing, 2018, pp. 360–384, ISBN: 978-3-319-78381-9.

[35] J. H. Cheon, A. Kim, and D. Yhee, "Multi-dimensional packing for heaan for approximate matrix arithmetics," *IACR Cryptol. ePrint Arch.*, vol. 2018, pp. 1245–1273, 2018.

[36] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[37] "Global card brands in 2019 - purchase volume." (2019), [Online]. Available: https://nilsonreport.com/research_featured_chart.php (visited on 06/23/2022).

[38] K. W. Bowyer, N. V. Chawla, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique," *CoRR*, vol. abs/1106.1813, 2011. arXiv: 1106.1813. [Online]. Available: http://arxiv.org/abs/1106.1813.

[39] Y. Xu and R. Goodacre, "On splitting training and validation set: A comparative study of cross-validation, bootstrap and systematic sampling for estimating the generalization performance of supervised learning," *Journal of Analysis and Testing*, vol. 2, pp. 249–262, Oct. 2018. DOI: 10.1007/s41664-018-0068-2.

[40] G. V. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, pp. 303–314, 1989.

[41] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th International Conference on Machine Learning*, S. Dasgupta and D. McAllester, Eds., ser. Proceedings of Machine Learning Research, vol. 28, Atlanta, Georgia, USA: PMLR, 2013, pp. 1139–1147. [Online]. Available: https://proceedings.mlr.press/v28/sutskever13.html.

[42] A. Botev, G. Lever, and D. Barber, "Nesterov's accelerated gradient and momentum as approximations to regularised update descent," in *2017 International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 1899–1903. DOI: 10.1109/IJCNN.2017.7966082.

[43] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, Jul. 2011.

[44] M. D. Zeiler, "Adadelta: An adaptive learning rate method," *arXiv*, 2012. DOI: 10.48550/ARXIV.1212.5701. [Online]. Available: https://arxiv.org/abs/1212.5701.

[45] G. Hinton, *Lecture 6a overview of mini-batch gradient descent*, 2014. [Online]. Available: https://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf (visited on 06/25/2022).

[46] H. E. Robbins, "A stochastic approximation method," *Annals of Mathematical Statistics*, vol. 22, pp. 400–407, 2007.

[47] X. Qian and D. Klabjan, "The impact of the mini-batch size on the variance of gradients in stochastic gradient descent," *arXiv*, 2020. DOI: 10.48550/ARXIV.2004.13146. [Online]. Available: https://arxiv.org/abs/2004.13146.

[48] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv*, 2016. DOI: 10.48550/ARXIV.1609.04747. [Online]. Available: https://arxiv.org/abs/1609.04747.

[49] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162.

[50] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai, "Privacy-preserving deep learning via additively homomorphic encryption," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 5, pp. 1333–1345, 2018. DOI: 10.1109/TIFS.2017.2787987.

[51] J. Ma, S.-A. Naas, S. Sigg, and X. Lyu, "Privacy-preserving federated learning based on multi-key homomorphic encryption," *arXiv*, 2021. DOI: 10.48550/ARXIV.2104.06824. [Online]. Available: https://arxiv.org/abs/2104.06824.

[52] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, "BatchCrypt: Efficient homomorphic encryption for Cross-Silo federated learning," pp. 493–506, Jul. 2020. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/zhang-chengliang.

[53] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," 2019. DOI: 10.48550/ARXIV.1902.04885. [Online]. Available: https://arxiv.org/abs/1902.04885.

[54] A. G. Sébert, R. Sirdey, O. Stan, and C. Gouy-Pailler, "Protecting data from all parties: Combining fhe and dp in federated learning," 2022. DOI: 10.48550/ARXIV.2205.04330. [Online]. Available: https://arxiv.org/abs/2205.04330.

[55] Dataiku, *Dataiku*, 2022. [Online]. Available: https://www.dataiku.com/ (visited on 06/30/2022).

[56] D. Maclaurin, D. Duvenaud, M. Johnson, and J. Townsend, *Autograd*, 2015. [Online]. Available: https://github.com/hips/autograd.

[57] J. Bradbury *et al.*, *JAX: Composable transformations of Python+NumPy programs*, version 0.3.13, 2018. [Online]. Available: http://github.com/google/jax.

[58] *Lorica cybersecurity*, 2022. [Online]. Available: https://www.loricacyber.com/ (visited on 09/01/2022).

[59] W. Jakob, J. Adler, L. Burns, S. Corlay, and E. Cousineau, *Pybind11*, 2022. [Online]. Available: https://github.com/pybind/pybind11.

[60] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[61] F. Grisoni, V. Consonni, and D. Ballabio, *Qsar androgen receptor data set*, 2019. [Online]. Available: https://www.kaggle.com/datasets/ishandutta/qsar-androgen-receptor-data-set?resource=download.

[62] M. Chase *et al.*, "Security of homomorphic encryption," 2017. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2018/01/security_homomorphic_encryption_white_paper.pdf (visited on 08/20/2022).

[63] Y. Gu, Y. Bai, and S. Xu, "Cs-mia: Membership inference attack based on prediction confidence series in federated learning," *Journal of Information Security and Applications*, vol. 67, p. 103 201, Jun. 2022. DOI: 10.1016/j.jisa.2022.103201.

[64] U. C. Bureau, "UCI: Census-income (kdd) data set," 1994. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Census-Income+%28KDD%29 (visited on 06/20/2022).

[65] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Curran Associates, Inc.*, pp. 8024–8035, 2019. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Appendix A: Metrics

A.1 Binary Classification

In the context of a binary classification problem (a problem in which there are only two classes), a confusion matrix is a 2-by-2 matrix that quantifies the performance of a predictive model using the number of correct and incorrect positive and negative predictions. Figure A.1 illustrates the form of a confusion matrix. The following defines the quantities in the confusion matrix.

True Positives (TP): the model predicts "positive" correctly.

True Negatives (TN): the model predicts "negative" correctly.

False Positives (FP): the model incorrectly predicts "positive". The actual data sample was "negative".

False Negatives (FN): the model incorrectly predicts "negative". The actual data sample was "positive".

Using these quantities, various metrics that quantify the performance of a classification model can be extracted. Firstly, the accuracy is the ratio of the number of accurate predictions to the total number of predictions expressed as a percentage. The accuracy has been expressed in terms of the quantities in a confusion matrix in Equation A.1.

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \times 100\% \tag{A.1}$$

The precision is the ratio of the number of accurate predictions to the number of total predictions for a given class. For example, the "positive" class precision is the

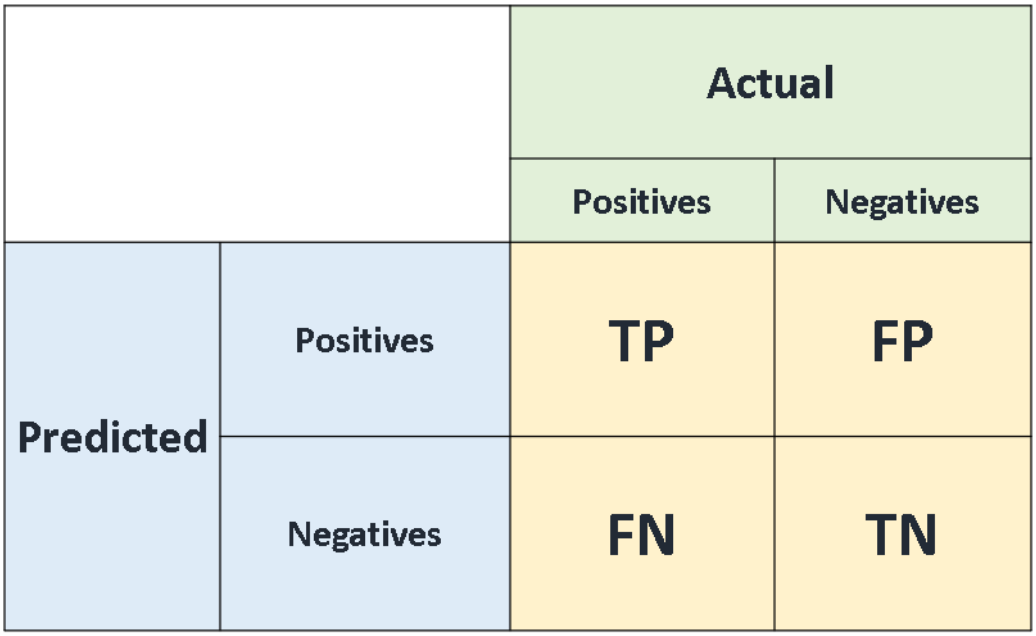

Figure A.1: Confusion matrix illustration

ratio of true positives to the total number of positive predictions. The formula for calculating precision is detailed in Equation A.2.

$$Precision = \frac{TP}{TP + FP} \tag{A.2}$$

The recall, or sensitivity, is the complement of precision. It is the ratio of the number of true positives to the total number of positive instances. This quantity is delineated in Equation A.3.

$$Recall = \frac{TP}{TP + FN} \tag{A.3}$$

The F1-score is the harmonic mean of the precision and the recall and is defined in Equation A.4.

$$F_1 = \frac{TP}{TP + \frac{1}{2}\,(FP + FN)} \tag{A.4}$$

This measure gives equivalent importance to precision and recall. Practically, it is common to weigh each of these measures uniquely across different problem contexts.

A.2 Regression

As mentioned previously, a continuous quantity is the target in a regression. The classification metrics discussed above are, therefore, not applicable. We cannot calculate the accuracy for a regression model. Instead, we utilize metrics that quantify how close predictions were to their expected values on average. Two examples of these metrics are delineated below.

Mean Squared Error (MSE):

The MSE is the mean of the squared error between the predicted quantities and the target quantities.

$$MSE = \frac{1}{N} \sum_{i}^{N} (y_i^{hat} - y_i^{true})^2 \tag{A.5}$$

Since this measure squares the error between predicted and true values, when the difference between y_i^{hat} and y_i^{true} is large, this error is amplified. Conversely, when the difference between the predicted and true values is small, the error is minimized. When using this metric, it is common to employ some sort of preliminary outlier filtering method to prevent these outliers from skewing the perceived quality of the model.

Mean Absolute Error (MAE):

The MAE is the mean of the absolute error between the predicted and target quantities.

$$MAE = \frac{1}{N} \sum_{i}^{N} |y_i^{hat} - y_i^{true}| \tag{A.6}$$

Unlike the MSE, the MAE does not amplify the error contribution of outliers as a squared transformation is not being applied.