

Rochester Institute of Technology

## RIT Digital Institutional Repository

---

### Theses

---

7-2019

## Exploring the Effectiveness of Privacy Preserving Classification in Convolutional Neural Networks

Prathibha Rama  
psr6237@rit.edu

Follow this and additional works at: <https://repository.rit.edu/theses>

---

### Recommended Citation

Rama, Prathibha, "Exploring the Effectiveness of Privacy Preserving Classification in Convolutional Neural Networks" (2019). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact [repository@rit.edu](mailto:repository@rit.edu).

---

# Exploring the Effectiveness of Privacy Preserving Classification in Convolutional Neural Networks

PRATHIBHA RAMA

---

---

# Exploring the Effectiveness of Privacy Preserving Classification in Convolutional Neural Networks

PRATHIBHA RAMA

July 2019

A Thesis Submitted  
in Partial Fulfillment  
of the Requirements for the Degree of  
Master of Science  
in  
Computer Engineering

**R·I·T** | KATE GLEASON  
*College of ENGINEERING*

*Department of Computer Engineering*

---

# Exploring the Effectiveness of Privacy Preserving Classification in Convolutional Neural Networks

PRATHIBHA RAMA

## Committee Approval:

---

Dr. Marcin Łukowiak *Advisor*  
Department of Computer Engineering

Date

---

Dr. Stanisław Radziszowski  
Department of Computer Science

Date

---

Dr. Cory Merkel  
Department of Computer Engineering

Date

## Acknowledgments

They say it takes a village to raise a child, but what they did not say is that it also takes a village to defend a Master's thesis. I am lucky to have so many people in my life that have been there for me throughout this journey.

Thank you to my fellow researchers, both past and present, from the Applied Cryptography and Information Security Lab: Michael Foster, Daniel Stafford, Cody Tinker, Stephanie Soldavani, Andrew Ramsey, Jason Blocklove, and Eric Scheler. Not only have you been so supportive, you have also kept me company when I needed it the most. Also a special shout-out to Cody, Kevin and Yash for responding to my last-minute questions even when I could have answered them myself.

Thank you to everyone at the RIT research cluster for providing me with the resources necessary to gather results. A month before meeting you, I honestly did not think I was going to finish in time. Sidney, Jen, and Andrew, I would have no results without you and for that I am eternally grateful.

Thank you to my committee members for taking the time to sit in on my defense and, more importantly, for providing me with the necessary guidance to succeed. Thank you Dr. Merkel for being the Deep Learning expert in a group of cryptographers; you answered the simplest questions without judgment, making it easy to ask no matter the task. Thank you Dr. Radziszowski for being the inspiring mathematician that you are and for showing me the ropes in Foundations of Cryptography/Advanced Cryptography; I am still in awe at the vast amount of knowledge you have and I look forward to emailing you with my future security queries. To Dr. Lukowiak, I cannot thank you enough for all that you have done to guide me throughout this process; you helped me when I needed it the most and always provided a path to solve the problems I did not think I could solve.

Thank you to my friends who supported me in person or via phone. Neha and Karn, you did not have to sit through an hour-long presentation about applied cryp-

tography, but you did, and it meant a lot.

Last, but certainly not least, I want to give a special thanks to my family. We have had an interesting run these past 7 years and while there were times I wanted to wallow in self-pity, you all found a way to bring me out and cheer me up. Thank you Apoorva for understanding the stress of graduate school; sometimes I just wanted to vent and you were a phone call away. Thank you Dad for providing sound and stable advice; you are a calm voice of reason in an otherwise noisy environment. Finally, thank you Mom for just being there; you have put up with my unpredictable stress, you have stood by my side during my darkest moments, and ultimately I am proud to call you my friend.

## Abstract

A front-runner in modern technological advancement, machine learning relies heavily on the use of personal data. It follows that, when assessing the scope of confidentiality for machine learning models, understanding the potential role of encryption is critical. Convolutional Neural Networks (CNN) are a subset of artificial feed-forward neural networks tailored specifically for image recognition and classification. As the popularity of CNN increases, so too does the need for privacy preserving classification. Homomorphic Encryption (HE) refers to a cryptographic system that allows for computation on encrypted data to obtain an encrypted result such that, when decrypted, the result is the same value that would have been obtained if the operations were performed on the original unencrypted data. The objective of this research was to explore the application of HE alongside CNN with the creation of privacy-preserving CNN layers that have the ability to operate on encrypted images. This was accomplished through (1) researching the underlying structure of preexisting privacy-preserving CNN classifiers, (2) creating privacy-preserving convolution, pooling, and fully-connected layers by mapping the computations found within each layer to a space of homomorphic computations, (3) developing a polynomial-approximated activation function and creating a privacy-preserving activation layer based on this approximation, (4) testing and profiling the designed application to assess efficiency, performance, accuracy, and overall practicality.

# Contents

---

Signature Sheet	i
Acknowledgments	ii
Abstract	iv
Table of Contents	v
Acronyms	viii
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation and Problem . . . . .	1
1.2 Homomorphic Encryption . . . . .	3
1.3 This Work . . . . .	3
<b>2 Background</b>	<b>5</b>
2.1 Privacy Preservation Techniques . . . . .	5
2.1.1 Secure Hardware: Intel SGX . . . . .	5
2.1.2 Secure Multi-party Computing . . . . .	6
2.1.3 Homomorphic Encryption . . . . .	6
2.2 Types of Homomorphic Cryptosystems . . . . .	7
2.3 High Level Fully Homomorphic Encryption (FHE) . . . . .	8
2.3.1 FHE Blueprint . . . . .	9
2.3.2 Noise Growth . . . . .	9
2.4 Convolutional Neural Networks . . . . .	10
<b>3 Related Work</b>	<b>12</b>
3.1 Privacy Preserving Deep Computation Model on Cloud for Big Data Feature Learning . . . . .	13
3.2 CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy . . . . .	14
3.3 Privacy Preserving Classification on Deep Neural Network . . . . .	14
3.4 CryptoDL: Deep Neural Networks over Encrypted Data . . . . .	15



<b>4</b>	<b>Mathematics of Homomorphic Encryption</b>	<b>17</b>
4.1	Mathematic Structures . . . . .	17
4.1.1	Lattices . . . . .	17
4.1.2	Rings . . . . .	19
4.2	Hard Problems . . . . .	20
4.2.1	Shortest Vector Problem . . . . .	20
4.2.2	Learning With Errors . . . . .	20
4.2.3	Ring Learning With Errors . . . . .	22
<b>5</b>	<b>Fully Homomorphic Encryption</b>	<b>24</b>
5.1	Practical FHE . . . . .	24
5.2	FHE Functions . . . . .	24
5.2.1	Homomorphic Addition . . . . .	25
5.2.2	Homomorphic Multiplication . . . . .	25
5.3	Popular FHE Cryptosystems . . . . .	25
5.3.1	FV Cryptosystem . . . . .	26
5.3.2	BGV Cryptosystem . . . . .	28
5.4	HElib vs. Microsoft SEAL . . . . .	30
<b>6</b>	<b>HElib Functions, Security and Parameter Selection</b>	<b>35</b>
6.1	Math Notation . . . . .	35
6.2	Functions . . . . .	36
6.2.1	ContextGen . . . . .	36
6.2.2	Key Generation . . . . .	36
6.2.3	Encryption . . . . .	37
6.2.4	Decryption . . . . .	37
6.2.5	Addition and Multiplication . . . . .	37
6.2.6	Modulus Switching . . . . .	38
6.2.7	Bootstrapping . . . . .	38
6.3	Security . . . . .	38
6.4	Parameter Selection . . . . .	39
6.4.1	Parameters for Security . . . . .	39
6.4.2	Parameters for Functionality . . . . .	40
<b>7</b>	<b>Design</b>	<b>43</b>
7.1	CNN Layers . . . . .	43
7.1.1	Fully Connected . . . . .	43

7.1.2	Convolution . . . . .	44
7.1.3	Activation . . . . .	45
7.1.4	Pooling . . . . .	46
7.2	Layer Design . . . . .	47
7.2.1	HElib Encoding and Functions . . . . .	47
7.2.2	Fully Connected Design . . . . .	48
7.2.3	Convolution Design . . . . .	49
7.2.4	Activation Design . . . . .	50
7.2.5	Pooling Design . . . . .	53
<b>8</b>	<b>Privacy Preserving Logic Gates</b>	<b>55</b>
8.1	Logic Gates . . . . .	55
8.2	Network . . . . .	56
8.3	Test Environment . . . . .	56
8.4	Results . . . . .	57
<b>9</b>	<b>Privacy Preserving CNN</b>	<b>60</b>
9.1	Dataset . . . . .	60
9.2	Network . . . . .	61
9.2.1	Training . . . . .	62
9.2.2	Testing . . . . .	63
9.3	Test Environment . . . . .	63
<b>10</b>	<b>Profiling Results</b>	<b>65</b>
10.1	Timing . . . . .	65
10.2	Scale Variation . . . . .	69
10.3	Security Parameter Variation . . . . .	73
10.4	Level Variation . . . . .	76
10.5	Column Variation . . . . .	79
10.6	Thread Variation . . . . .	81
10.7	Fast Configuration . . . . .	83
<b>11</b>	<b>Conclusion and Future Work</b>	<b>84</b>
	<b>Bibliography</b>	<b>86</b>

# Acronyms

---

**ANN** Artificial feed-forward Neural Networks

**ASVP** Approximate Shortest Vector Problem

**BGV** Brakerski-Gentry-Vaikuntanathan

**CIFAR10** Canadian Institute for Advanced Research

**CNN** Convolutional Neural Networks

**FHE** Fully Homomorphic Encryption

**FV** Fan and Vercauteren

**HE** Homomorphic Encryption

**Intel SGX** Intel Software Guard Instructions

**LWE** Learning with Errors

**MNIST** Modified National Institute of Standards and Technology

**MPC** Secure Multi Party Computing

**OWF** One Way Function

**PHE** Partially Homomorphic Encryption

**PKE** Public Key Encryption

**PRNG** Pseudo Random Number Generator

**ReLU** Rectified Linear Unit

**RLWE** Ring Learning with Errors

**RSA** Rivest Shamir Adleman

**SIS** Short Integer Solution

**SIVP** Shortest Independent Vectors Problem

**SRC** Secure Remote Computation

**SVP** Shortest Vector Problem

**SWHE** Somewhat Homomorphic Encryption

**SYU** Sander Young Yung

**UNUM** Universal Number

**uSVP** Unique Shortest Vector Problem

**YASHE** Yet Another Somewhat Homomorphic Encryption Scheme

# Chapter 1

---

## Introduction

### 1.1 Motivation and Problem

The rise of information technology in the everyday human experience brings forth a new form of currency: privacy of the individual. A search for restaurants near me, while seemingly cost-free, is only possible when the individual searching discloses their current location. Personal data is traded daily and it is only upon close inspection that the potential vulnerability of sharing such information becomes obvious. As a front-runner in technological advancement, that plays a lead role in many modern innovations, machine learning relies heavily on the use of personal data. In machine learning, analytic models are utilized to make informed predictions on provided datasets. Because input datasets can vary from public images of handwritten digits to more sensitive information such as personal medical history, the rise in machine learning naturally leads to an urgency for privacy within specific applications. In addition, many machine learning models need significant computing power to process large amounts of data in an efficient manner. A solution to this conundrum is to take advantage of cloud resources. From a security perspective, a cloud based solution opens the door for a myriad of vulnerabilities. But what if it were possible to have the best of both worlds? Is there potential for taking advantage of cloud resources, while simultaneously maintaining security of the individual? If it is assumed that an

already trained model is hosted on the cloud, can an individual encrypt their data, send the encrypted data to the cloud, process the encrypted data through the model, and receive an encrypted result that only the individual can decrypt? This exact scenario has been proven possible via privacy preserving classification. Understanding the use of encryption within privacy-preserving classification is therefore essential when assessing the confidentiality and efficiency of a system. The privacy-preserving classification problem is concerned with the idea of making encrypted predictions on an encrypted dataset. In machine learning, there are three datasets involved in the creation and execution of a predictive model: training, validation, and testing. During the learning phase, a training dataset is utilized to determine the weights that make up the predictive model. Throughout the learning phase these weights are updated until either a minimum error threshold has been met or a maximum number of iterations has been achieved. A validation dataset is used during the learning phase to fine-tune the architecture and meta-parameters of the model and query the model's performance on unseen data. This fine-tuning helps minimize the potential for over-fitting. Following the learning phase, a testing dataset is used to confirm the predictive power of the final model. This portion is called the classification phase. In the context of privacy-preserving classification, the learning phase works with unencrypted datasets while the inference phase works with encrypted datasets. This scenario assumes a client-server model where the server has already trained the predictive model, but would now like to modify the model to classify encrypted inputs. The learning phase follows the same procedure of updating weights and fine-tuning model architecture using unencrypted training and validation datasets. The difference in process can be observed during the classification phase, where the testing dataset is encrypted with a secret key before it is fed through the model that outputs an encrypted prediction. This output prediction can then be decrypted by the secret key used to encrypt the input data in the first place. Proposed solutions to

the privacy-preserving classification problem are based on various approaches that include Secure Multi Party Computing (MPC), Secure Remote Computation (SRC), and Homomorphic Encryption (HE). Although each approach theoretically provides a viable solution, those that take advantage of HE have been successfully implemented and documented. The focus of this study is to take a closer look at an approach based on HE and observe the intersection of security and ease of use.

## **1.2 Homomorphic Encryption**

Derived from the Greek words for same form, homomorphism is a structure-preserving transformation of one algebraic set into another. In the field of cryptography, HE describes a cryptosystem where the transformation from the plaintext space to the ciphertext space preserves relationships between elements. This property allows for meaningful computation on ciphertexts. Such computations generate an encrypted result that, when decrypted produce the same value that would have resulted from the plaintext computation.

## **1.3 This Work**

Several solutions to the privacy-preserving classification problem are based on HE [1] [2] [3] [4]. While these solutions boast accuracy and efficiency, they are largely unverified and understudied. In fact, with limited documentation and unavailable source code, finding even a simple case-study proves difficult. This research focuses on a more in depth exploration of using HE alongside Convolutional Neural Networks (CNN).

In this study, HElib, an open source cryptographic library based on the Brakerski-Gentry-Vaikuntanathan (BGV) scheme [5], was integrated with CNN. Although it is possible to do both encrypted training and encrypted classification, the primary

focus of this study was to explore the feasibility of encrypted classification and the intricacies involved with HELib.

To work with HE functions, low degree polynomial-approximations of both the Rectified Linear Unit (ReLU) and Sigmoid activation functions were designed. Training of the CNN was done with the original activation functions on unencrypted data and classification was done with the polynomial-approximated activation functions on data encrypted by HELib. In addition to the activation layers, a privacy-preserving convolution layer, privacy-preserving pooling layer, and privacy-preserving fully connected layer were created. To verify the correctness of these privacy-preserving layers, initial tests were done utilizing a simple three-layer network. This was used to predict the output of a specified logic gate based on an encrypted input vector and an unencrypted weights file produced during the training phase. This network was then tested to illustrate that classification on encrypted data is indeed possible and to highlight some basic metrics regarding smaller privacy-preserving classifiers.

Following initial results, a larger eight-layer network was created to perform encrypted classification on the Modified National Institute of Standards and Technology (MNIST) handwritten digit dataset. This application was profiled and tested for accuracy, efficiency, performance, and overall practicality. The results of this experiment illustrate the potential role of HE in many modern information systems, specifically those that utilize CNN.

To analyze the behavior of privacy-preserving classification from a security/cryptographic perspective, the HE parameters were varied to observe the effects of parameter size on efficiency and of noise on accuracy. To analyze the behavior of privacy-preserving classification from a Deep Learning perspective, the effects of varied scaling were noted, specifically the relationship between the number of fixed bits used to represent the weights/biases and overall classification accuracy. How many bits are needed for successful classification? At what point does the classification accuracy deteriorate?



# Chapter 2

---

## Background

### 2.1 Privacy Preservation Techniques

A primary focus for this study is the method of privacy preservation used to secure CNN; not only should the chosen method allow for secure computation, it must also provide a robust level of security. Three popular privacy preservation techniques that were explored in the interest of protecting CNN are SRC, MPC, and HE.

#### 2.1.1 Secure Hardware: Intel SGX

The SRC problem is defined by an individual's ability to execute software on a remote computer while maintaining a level of security [6]. The SRC problem assumes that the remote computer is hosted by a non-trustworthy party and therefore emphasizes the necessity for both data confidentiality and data integrity.

Introduced as a solution to the SRC problem, Intel Software Guard Instructions (Intel SGX) aims to secure user-level code with the use of enclaves or protected regions of memory. Equipped with a special set of CPU instructions, a user can upload data into a secure container where private computations may be executed. Unlike other secure hardware platforms, which use attestation for a considerable amount of code, Intel SGX uses attestation to vouch specifically for the enclave and its contents. On the surface the Intel SGX appears to be the perfect solution to the SRC problem,

but recent studies have shown that this platform has certain drawbacks. In fact, it has been demonstrated that the Intel SGX is particularly vulnerable against cache timing attacks [7]. In addition, according to sources the Intel SGX security claims do not hold for cloud environments [6] [7]. For example, there is a potential scenario where there is co-location between a logical processor running Intel SGX and a logical processor running malicious code. Because the motivation for this study is to increase security in the cloud, secure hardware was ruled out [6].

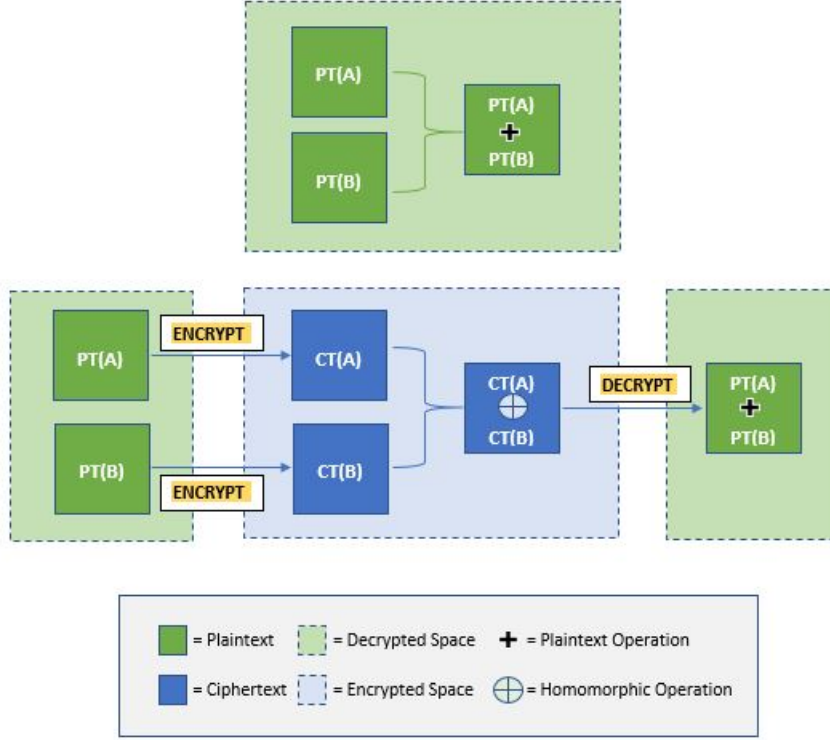
### **2.1.2 Secure Multi-party Computing**

MPC refers to a protocol that grants the ability to calculate functions in a distributed manner. The idea behind MPC is to create a method that allows for several parties to perform computations with one another while maintaining the privacy of each party's input data i.e. collaborative computation without disclosing private data [8]. With MPC, participating parties each provide their input data. This is divided into distinct pieces, each of which are masked with a random value and sent out to various servers. This process ensures the privacy of each individuals personal input data, while allowing for joint computation [9].

### **2.1.3 Homomorphic Encryption**

A form of encryption that allows computation on ciphertexts, HE has many potential applications. In a broad sense, HE cryptosystems function like many other Public Key Encryption (PKE) cryptosystems, where data is encrypted with a public key and decrypted with a private key. Unlike other cryptosystems, once the data is encrypted with the public key, HE allows for valid arithmetic operations on encrypted data. For example if an operation, say homomorphic addition, is performed between two encrypted values, the output will be the encrypted result of the unencrypted values added together. Operations done within the ciphertext space therefore mimic

operations done in the plaintext space. This can be observed in figure 2.1.



**Figure 2.1:** High Level Diagram of Homomorphic Encryption

Of the three privacy-preserving techniques mentioned, solutions based on Fully Homomorphic Encryption (FHE) have been successfully implemented and documented. For the purpose of this study, the BGV encryption scheme was chosen as it is the most effective FHE scheme for polynomial evaluations [10]. An open source implementation of BGV, HELib was integrated alongside a CNN.

## 2.2 Types of Homomorphic Cryptosystems

There are three types of HE: Partially Homomorphic Encryption (PHE), Somewhat Homomorphic Encryption (SWHE), and FHE.

PHE is a cryptosystem that allows for one type of operation on encrypted data. This operation can be performed an unlimited number of times within the cipher-

text space. Famous examples of PHE cryptosystems include Rivest Shamir Adleman (RSA), which allows for unlimited multiplication, El-Gamal, which allows for unlimited multiplication, and Paillier, which allows for unlimited addition [11]. Practical uses for PHE have manifested in the form of electronic voting, where votes are homomorphically added, and Private Information Retrieval, where values are homomorphically compared [11].

SWHE is a cryptosystem that allows for a limited number of operations on encrypted data. These operations can only be performed a limited number of times within the ciphertext space. Examples of SWHE include BGN, which allows for unlimited addition and one multiplication, Polly Cracker, which allows for arbitrary additions/multiplication, and Sander Young Yung (SYU), which allows for unlimited ANDs and one OR/NOT [11]. Practical uses for SWHE are generally seen when the depth of the evaluation operation is constant [11].

FHE is a cryptosystem that allows for an unlimited number of operations on encrypted data. These operations can be performed an unlimited number of times within the ciphertext space. Examples of FHE include Gentry's FHE scheme and BGV, both of which allow for unlimited addition and unlimited multiplication. Because FHE cryptosystems allow for an unlimited number of operations an unlimited number of times, they can theoretically be used for any application [11].

## 2.3 High Level FHE

FHE is a type of HE that allows for an unlimited number of operations on encrypted data. The first FHE scheme was introduced in 2009 by Craig Gentry [12]. Based on the mathematics of ideal lattices, Gentry's FHE scheme is comprised of two steps. The first step is to start with a Somewhat HE scheme (SWHE). SWHE is a type of HE that allows for both homomorphic addition and homomorphic multiplication a fixed number of times on encrypted data. This fixed number of operations is a result

of how SWHE schemes are constructed. Built on the Learning with Errors (LWE) problem, each ciphertext has some noise that hides the original message. The primary limitation with this construction is the inevitable noise growth that results from arithmetic operations; once the noise reaches a certain threshold, the original message is irretrievable. The second step is to add a Bootstrapping mechanism to the SWHE scheme to refresh the ciphertext. Bootstrapping essentially consists of homomorphically evaluating the decryption circuit for noise reduction. Although this method is both secure as well as functionally correct, it is not considered practical. This lack of practicality is largely because of high computation cost and high memory cost. Following Gentry's 2009 scheme, several other FHE schemes have been developed. These include schemes based on ideal lattices (Gentry's scheme)[12], schemes based on (Ring) LWE (BGV)[5], and schemes based on integers (Van Dijk's scheme)[13].

### **2.3.1 FHE Blueprint**

PKE schemes contain the following three functions: KeyGen, Encrypt, and Decrypt. KeyGen is used to generate both the secret key and the public key, Encrypt is used to encrypt the plaintext data into a ciphertext, and Decrypt is used to decrypt the ciphertext data into a plaintext. FHE schemes contain the following four functions: KeyGen, Encrypt, Decrypt, and Evaluate. KeyGen, Encrypt, and Decrypt are the same as any other PKE scheme, but the addition of an Evaluation function allows for computations on ciphertexts. Evaluation performs some function with a set of ciphertexts as inputs and outputs a ciphertext that corresponds to a functional plaintext. For FHE schemes, the evaluate function will consist of Addition and Multiplication.

### **2.3.2 Noise Growth**

The base construction of practical FHE schemes focuses on the concept of noisy ciphertexts, where each ciphertext has noise that hides the message. This concept

is like hidden error correcting codes, where the intended message is the codeword, but the sent message is the codeword with some error. If the noise is small, then the receiver can use the knowledge of a hidden code to remove the noise. However, if the noise is large, decryption is impossible for the receiver.

When performing any mathematical operation on noisy ciphertexts, noise growth is inevitable. Thus, both addition and multiplication will increase noise. While addition adds the noise vectors, multiplication multiplies the noise vectors, making the noise growth extremely large. Recall that for a cryptosystem to be considered fully homomorphic, the operations must have the ability to be performed an unlimited number of times. If noise growth renders the ciphertext meaningless after only a few multiplications, the cryptosystem is not fully homomorphic. This implies that for multiplication to be considered meaningful for an unlimited number of times, noise growth must be managed.

## **2.4 Convolutional Neural Networks**

Neural networks or Artificial feed-forward Neural Networks (ANN) refer to computer systems that are inspired by the human brain. Comprised of multiple layers, the nodes found within neural networks are interconnected like neurons found within the brain. Neural networks constitute three types of layers: input layer, hidden layers, and output layer. The input layer takes in the various features of an input data point and relays these features to the hidden layer. The hidden layer then computes a function over values gained from the previous layer and passes the calculated values to the next layer. The output layer then performs the final computation on values gained from the hidden layer [14]. CNN are a type of Neural Network used specifically for image recognition. Like ANN, CNN consist of an input layer, hidden layer, and output layer. In addition to the common types of layers observed in ANN, CNN also have one or more convolutional layers. This new layer is created with spatial

convolutional filtering in mind and therefore facilitates image processing. At a high-level CNN take an input image and then, through a series of layers, transforms this data into an output of label scores. Layers within CNN are ordered one after another, where each one is linked to a layer before and a layer after. In this case, the output of a function applied to the neurons of the current layer will be the input neurons to the next layer [15].

## Chapter 3

---

### Related Work

Although privacy preserving deep learning is a relatively new area of research, recent studies have proven successful with the integration of cryptography and deep learning. The primary metric for success in most of these studies is the ability to preserve the accuracy of the original Neural Network even after the introduction of HE.

It should also be noted that there is a distinction between encrypted training and encrypted classification. Like CNN, designing a privacy preserving CNN involves both a training phase and a classification phase. In the context of encrypted data, this means the privacy preserving CNN can be trained on encrypted data and classified on unencrypted data, trained on unencrypted data and classified on encrypted data, or trained on encrypted data and classified on encrypted data. The motivation behind encrypted training is to avoid model leakage; if the model is trained on encrypted data, it is difficult to infer anything about the data even with heavy statistical analysis. The motivation behind privacy preserving classification is to ensure individual privacy while maximizing the efficiency of cloud computing; if a server is hosting an already trained model can it be modified so that the client can secure their data before sending it for classification?

Prior to privacy preserving deep learning, efforts were made to integrate HE with basic machine learning classifiers. In a 2015 study, Bost et. al created three private classifiers with the Hyperplane, Decision Tree, and Naive Bayes classifiers [16]. Each



private classifier proved both robust and efficient when tested on relevant datasets. This study provided the necessary groundwork for future attempts at incorporating HE with Machine Learning.

### 3.1 Privacy Preserving Deep Computation Model on Cloud for Big Data Feature Learning

In a study done by Chen et. al [1] privacy preserving deep computation is explored during the training phase. The goal of this experiment is to improve the efficiency of training by offloading expensive operations to the cloud. Input data is encrypted using the BGV encryption scheme and then uploaded to the cloud where the high-order back propagation can be performed. Because HE does not support exponentiation, the primary modification introduced is the Taylor series approximation of the Sigmoid Activation function. Timing and accuracy results were gathered for both the original high-order back propagation algorithm outlined in the paper and the modified privacy preserving back propagation algorithm which incorporated the computing power of the cloud. Results of this experiment clearly show an improvement in efficiency, with the privacy preserving scheme being two times more efficient, with regards to timing, than the non privacy preserving scheme. This comparison was done . At the same time, the privacy preserving scheme introduces a 2% accuracy degradation when compared to the conventional scheme. Although this experiment does not focus specifically on privacy preserving CNN, it does examine how to integrate HE with a deep learning model by using a Taylor Series representation of the Sigmoid activation function.

### **3.2 CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy**

One of the first commercial examples of integrating HE with Neural Networks, Microsoft's CryptoNets [2] presents a method for converting learned neural networks to neural networks that can be applied to encrypted data. The goal of this experiment is to improve the efficiency of Neural Network classification using the cloud. Input data is encrypted using the Yet Another Somewhat Homomorphic Encryption Scheme (YASHE) encryption scheme and then uploaded to the cloud, where the privacy preserving neural network classifies the encrypted data. Because HE only supports addition and multiplication, the primary modification introduced is the replacement of the non-linear layers. The ReLU activation layer is replaced with the square function and the Max Pooling layer is replaced with Sum Pooling. Using the modified privacy-preserving Neural Network, classification accuracy on the MNIST dataset is 98.95%, where as state of the art accuracy is 99.77%. This experiment is tested on a small CNN that has a total of 9 layers with two activation layers. This study shows how to implement a small-scale privacy preserving CNN with the modification of non-linear layers.

### **3.3 Privacy Preserving Classification on Deep Neural Network**

It is important to note that because the square activation function has an unbounded derivative, too many of these activation layers will lead to unstable training. This means CryptoNets becomes largely ineffective for large CNN where there are many activation layers or non-linear layers in general. To overcome this drawback, Chabanne et. al [3] suggests improvements to the CryptoNets solution with the introduction

of batch normalization (Ioffe and Szegedy) to both the training and classification phase. During the training phase, the original ReLU function is used, max pooling is replaced with sum pooling, and a batch normalization level is added before each activation layer. During the privacy-preserving classification phase, the ReLU function is replaced with a low-degree polynomial approximation, max pooling is replaced with sum pooling, and a batch normalization level is added before each activation layer. Initial accuracy result show that this approach while successful on a light CNN (9 total layers, 2 activation layers) shows a fair amount of accuracy degradation on a deep CNN (24 total layers, 6 activation layers). Following initial accuracy analysis, improvements are made by building new polynomial approximations learned from a distribution close to output distribution of batch normalization. Results show that non-private classification accuracy (ReLU) is 99.59% while private classification accuracy is 99.30%. This study highlights how batch normalization and low degree polynomial approximation of the ReLU activation function can be used to improve the accuracy of privacy preserving CNN.

### 3.4 CryptoDL: Deep Neural Networks over Encrypted Data

A study done by Hesamifard et. al [4] takes into consideration the aforementioned pitfalls and attempts to improve the accuracy of privacy preserving CNN by studying the behavior of approximated activation functions. The methods for approximating the ReLU function include: numerical analysis, Taylor series, standard Chebyshev polynomials, modified Chebyshev polynomials, and their approach based on the derivative of the ReLU function. Using the best method of approximation, ReLU derivative, a privacy preserving CNN was implemented and tested on both the MNIST and Canadian Institute for Advanced Research (CIFAR10) datasets. The model achieved a classification accuracy of 99.52% for the MNIST dataset. Because the approximation based on the derivative of the ReLU function yielded the greatest accuracy, this is

the approximation utilized for this study.

# Chapter 4

---

## Mathematics of Homomorphic Encryption

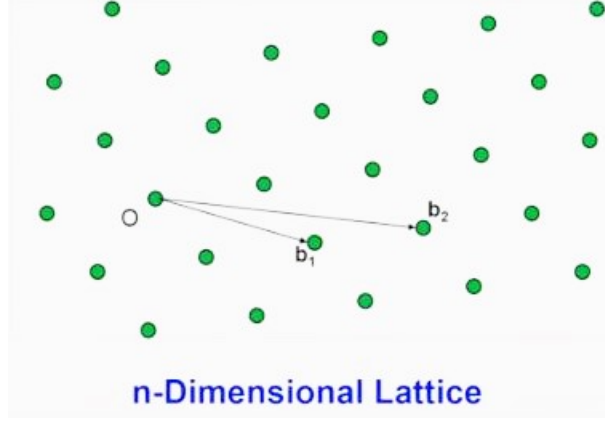
The foundation of all FHE schemes relies on the mathematics of lattices. Because the mathematics of lattices contain various hard problems, it lends itself to the field of cryptography. Two popular hard problems include the Short Integer Solution (SIS), used to create One Way Function (OWF) and collision resistant hashing, and LWE, used to create Pseudo Random Number Generator (PRNG) and PKE [17]. Based on these hardness assumptions as well as others, existing FHE schemes can be divided into four categories: Ideal Lattice-Based, Integer-Based, (Ring) LWE, and NTRU-Based. Currently, Ring Learning with Errors (RLWE)-Based cryptosystems are the primary focus of FHE.

### 4.1 Mathematic Structures

#### 4.1.1 Lattices

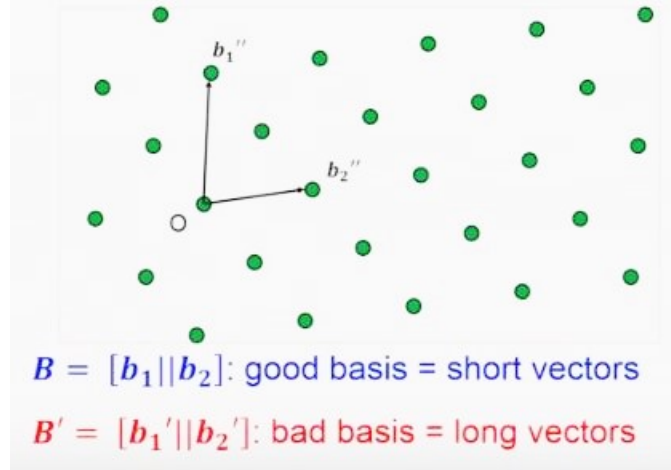
An abstract structure in mathematics, an  $n$ -dimensional lattice is all integer linear combinations of  $n$  basis vectors  $b_1, b_2, \dots, b_n$ . Depending on the basis, the same lattice can be generated in multiple different ways. An  $n$ -dimensional lattice can be observed in figure 4.1.

Generally speaking, short vectors are considered good basis while long vectors are considered bad basis [17]. An example of a good basis versus a bad basis for the same



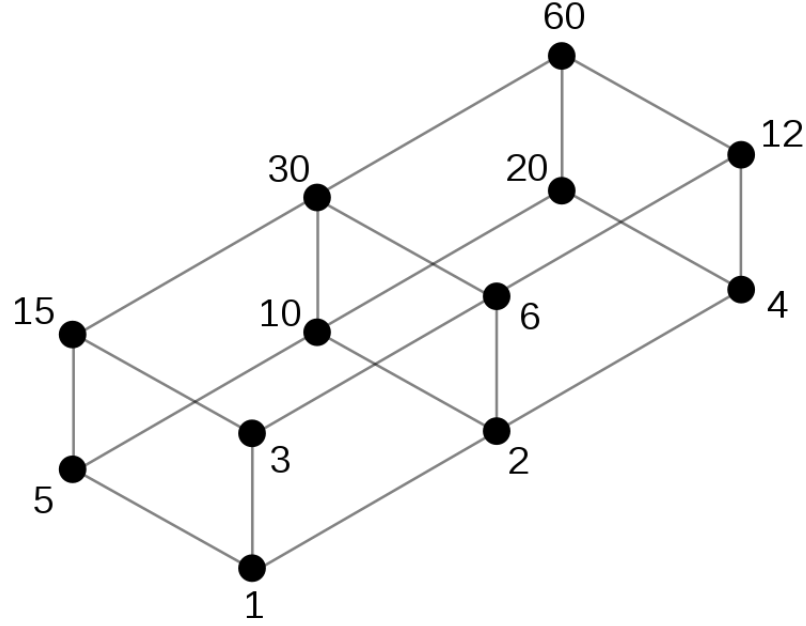
**Figure 4.1:** n-dimensional Lattice [17]

lattice can be observed in figure 4.2.



**Figure 4.2:** Good basis vs Bad basis for the same Lattice [17]

In simple terms lattices are partially ordered sets where each pair of elements has a unique combination comprised of an upper bound and a lower bound. A popular example of a lattice is the natural numbers where the lower bound between two elements is the greatest common divisor and the upper bound between two numbers is the least common multiple. The order relation in this example would be divisibility [18]. This order relation can be observed in figure 4.3.



**Figure 4.3:** Lattice based on natural numbers with divisibility as the order relation: GCD, LCM [19]

#### 4.1.2 Rings

A ring  $R$  is defined as a set of elements with two operations: addition and multiplication. Therefore, if two elements are added/multiplied within a ring, it will produce another element in the ring. A ring is an abelian group under addition: addition is both associative and commutative and there exists an additive identity and additive inverses. This property of a ring makes subtraction possible. A ring is a monoid under multiplication: multiplication is associative, but not commutative and there usually exists a multiplicative identity, although it is not required. This property of a ring makes division impossible. In a ring, multiplication is distributive with respect to addition [20]. To summarize, a ring is a set of elements that contains addition, subtraction, and non-commutative multiplication, but does not contain division. Rings are useful for generalizing structures such as matrices. For example, the  $2 \times 2$  matrices with real numbers form a ring as matrix multiplication is not commutative [21].

## 4.2 Hard Problems

In cryptography the computation hardness assumption refers to the hypothesis that a practical size problem cannot be solved in polynomial time, making it impractical for a computer to solve. The mathematics of lattices contains many such hard problems.

### 4.2.1 Shortest Vector Problem

The Shortest Vector Problem (SVP) is one of the most commonly known hardness problems. This problem states that given a basis, find a shortest non-zero vector, where  $\lambda_1$  is the length of the shortest non-zero vector. An example of the SVP in 1-Dimensional space is finding the greatest common denominator between two elements. On a small scale, this problem seems easy, but as many cryptosystems have shown this problem can prove difficult to solve. From SVP stems two permutations: Approximate Shortest Vector Problem (ASVP) and Shortest Independent Vectors Problem (SIVP) [22].

The ASVP states that given a basis, find  $\alpha$ -approximate shortest vector. In other words, find a non-zero vector of length at most  $\alpha\lambda_1$ . This permutation suggests that the goal is not to find the exact shortest vector, but vector that is relatively close to the shortest [23].

The SIVP states that given a basis, find  $n$  vectors of length at most  $\lambda_n$ , where  $\lambda_n = \min \{ r : \text{there are } n \text{ linearly independent lattice vectors of length } \leq r \}$  [23]

### 4.2.2 Learning With Errors

The hard problem used in PKE is LWE. LWE takes the easy problem of solving a system of linear equations and transforms it into the hard problem of solving a system of approximate linear equations [23]:



Original:

$$\vec{s} * \begin{vmatrix} 5 & 1 & 3 \\ 6 & 2 & 1 \end{vmatrix} = \begin{vmatrix} 11 & 3 & 9 \end{vmatrix} \rightarrow Find \vec{s}$$

Modified:

$$\vec{s} * \begin{vmatrix} 5 & 1 & 3 \\ 6 & 2 & 1 \end{vmatrix} + \begin{vmatrix} e_1 & e_2 & e_3 \end{vmatrix} = \begin{vmatrix} 11 & 3 & 9 \end{vmatrix} \rightarrow Find \vec{s}$$

where  $e_1, e_2, e_3$  are small values.

The slight perturbation caused by the error vector is what makes this problem computationally difficult to solve. At a high-level, LWE states that given many noisy equations on a secret  $s$ , it is impossible to find  $s$ . Formally, LWE can be defined as:

$$(A, s^T A + e^T) \rightarrow Find \vec{s} \quad (4.1)$$

where  $A \in Z_q^{m \times n}$ ,  $\vec{s} \in Z_q^n$ ,  $e$  is a "small" error vector.

A variant of the LWE problem, that is as hard as LWE, is the decisional-LWE problem. At a high-level, decisional-LWE states that given many noisy equations on a secret  $s$ , it is impossible to distinguish them from random values [23]. Formally, decisional-LWE can be defined as:

$$(A, s^T A + e^T) = (A, b) \quad (4.2)$$

where  $A \in Z_q^{m \times n}$ ,  $\vec{s} \in Z_q^n$ ,  $e$  is a "small" error vector,  $b$  is uniformly random.

In cryptography, the hardness of LWE naturally lends itself to both a OWF and

a PRNG. Given the function:

$$g_A(s, e) = (s^T A + e^T) \quad (4.3)$$

If LWE is assumed, then  $g_A$  is an OWF and if decisional-LWE is assumed, then  $g_A$  is a PRNG.

### 4.2.3 Ring Learning With Errors

Another example of a ring are the polynomials with integer coefficients. In this case, addition, subtraction and multiplication will successfully produce another polynomial with integer coefficients, while division will not.

Polynomial Ring  $K[X]$  in  $X$  over a field  $K$ :  $a_0 + a_1x + \dots \text{mod } a_i \in K$

Quotient Ring  $K[X]/[b_0 + b_1x + \dots + b_nx^n]$ :  $a_0 + a_1x + \dots \text{mod } b_0 + b_1x + \dots + b_nx^n$

RLWE is an extension of LWE that utilizes ring elements:

$$(A, s^T A + e^T) \rightarrow \text{Find } \vec{s} \quad (4.4)$$

where  $e^T$  is a "small" error vector.

RLWE is used over LWE because it is more efficient to compute and store ring elements. The following example illustrates addition and multiplication with ring elements on  $Z_q[x]/x^n + 1$ :

Say that  $q = 17$  and  $n = 4$  then:

$$a := 15 + 2x + 4x^2 + 7x^3 \in Z_{17}[x]/(x^4 + 1)$$

$$b := 8 + 9x + 3x^2 + 4x^3 \in Z_{17}[x]/(x^4 + 1)$$

Addition:

$$\begin{aligned} a + b &= ((15 + 2x + 4x^2 + 7x^3) + (8 + 9x + 3x^2 + 4x^3)) \mod (17, x^4 + 1) \\ &= 23 + 11x + 7x^2 + 11x^3 \mod (17, x^4 + 1) \\ &= 6 + 11x + 7x^2 + 11x^3 \mod (17, x^4 + 1) \end{aligned}$$

Observe that the coefficients of the polynomial are bounded by 17 while the polynomial itself is bounded by  $x^4 + 1$ . This is the reason why 23 becomes 6 in the final two lines as  $23 \mod 17 = 6$ .

Multiplication:

$$\begin{aligned} a * b &= ((15 + 2x + 4x^2 + 7x^3) * (8 + 9x + 3x^2 + 4x^3)) \mod (17, x^4 + 1) \\ &= (120 + 151x + 95x^2 + 158x^3 + 83x^4 + 37x^5 + 28x^6) \mod (17, x^4 + 1) \\ &= (120 + 151x + 95x^2 + 158x^3 + 83(-1) + 37(-x) + 28(-x^2)) \mod (17, x^4 + 1) \\ &= (37 + 114x + 67x^2 + 158x^3) \mod (17, x^4 + 1) \\ &= (3 + 12x + 16x^2 + 5x^3) \mod (17, x^4 + 1) \end{aligned}$$

Similar to the addition example, the coefficients of the polynomial are bounded by 17 while the polynomial itself is bounded by  $x^4 + 1$ . Because  $x^4 = -1 \mod (x^4 + 1)$ ,  $x^4$  is replaced with -1,  $x^5$  is replaced with  $-1x$ , and  $x^6$  is replaced with  $-1x^2$ .

# Chapter 5

---

## Fully Homomorphic Encryption

### 5.1 Practical FHE

Considered the Holy Grail of HE, FHE is a cryptosystem that allows for an unlimited number of operations an unlimited number of times within the ciphertext space. While theoretically suitable for any application, practically FHE faces certain limitations. To understand these limitations, the underlying structure of well-known FHE schemes must first be explored.

### 5.2 FHE Functions

From a high-level perspective, most HE schemes are constructed as follows [24]:

1. Let  $m$  be a Plaintext message
2. Let a shared public key be a random odd integer  $p$
3. Choose a random large integer  $q$ , small  $r$ ,  $|r| \leq p/2$
4. Ciphertext  $c = pq + 2r + m$  (Ciphertext  $c$  is close to multiple of  $p$ )
5. Perform homomorphic addition/multiplication as required
6. Decrypt  $m = (c \bmod p) \bmod 2$

FHE schemes contain the following four functions: KeyGen, Encrypt, Decrypt, and Evaluate. KeyGen, Encrypt, and Decrypt are the same as any other PKE scheme, but the addition of an Evaluation function allows for computations on ciphertexts.

Evaluation performs some function with a set of ciphertexts as inputs and outputs a ciphertext that corresponds to a functional plaintext. In PHE schemes the evaluation function allows for either homomorphic addition or homomorphic multiplication, in FHE schemes, the evaluation function allows for both homomorphic addition and homomorphic multiplication.

Using the same high-level example, the corresponding homomorphic addition and homomorphic multiplication operations can be seen below [24]:

### 5.2.1 Homomorphic Addition

$$c_1 = q_1 * p + 2 * r_1 + m_1$$

$$c_2 = q_2 * p + 2 * r_2 + m_2$$

$$c_1 + c_2 = (q_1 + q_2) * p + 2 * (r_1 + r_2) + (m_1 + m_2)$$

### 5.2.2 Homomorphic Multiplication

$$c_1 = q_1 * p + 2 * r_1 + m_1$$

$$c_2 = q_2 * p + 2 * r_2 + m_2$$

$$c_1 * c_2 = ((c_1 * q_2) + q_1 * c_2 * q_1 * q_2) * p + 2(2 * r_1 * r_2 + r_1 * m_2 + m_1 * r_2) + m_1 * m_2$$

## 5.3 Popular FHE Cryptosystems

Two popular FHE cryptosystems are Fan and Vercauteren (FV), and BGV. Both FV and BGV are built on the RLWE hardness assumption. As a result the plaintext and ciphertext spaces are defined with regard to some ring  $R$ . In this case, the ring is defined as the polynomials with integer coefficients where addition, subtraction and multiplication successfully produce another polynomial with integer coefficients. Formally,  $R$  is defined as  $Z[x]/\phi_d(x)$ , where the polynomial degree is less than  $n = \phi(d)$ . Generally these polynomials can be represented as a vector of coefficients. Ciphertext coefficients are reduced modulo  $q$  and mapped into the range  $[-q/2, q/2]$ ,

where  $[\cdot]_q$  represents the modulus operation itself and  $R_q$  represents ring elements with coefficients modulo  $q$ . Plaintext coefficients are reduced modulo  $t$ , where  $t < q$ . Additional notation seen in these cryptosystems includes  $l_{w,q}$ , where  $w$  is an integer used in a radix- $w$  system and  $l_{w,q} = \lceil \log_w(q) \rceil + 1$  [25].

The following two functions can be seen in one or both of the cryptosystems:

**PowersOf:** This serves as a mapping function, where ring elements are converted to a vector of  $l_{w,q}$  elements. Each mapped ring element has coefficients scaled by the radix integer. In this case the radix integer is iteratively exponentiated based on the value states after PowersOf, i.e Powersof2 [25].

**WordDecomp:** This serves as a mapping function, where ring elements are converted to a vector of  $l_{w,q}$  elements. Each mapped ring element has coefficients that are the word decomposition of the original coefficients [25].

### 5.3.1 FV Cryptosystem

The FV cryptosystem is a FHE scheme that allows for both addition and multiplication. Like most FHE schemes, FV encryption is based on noisy ciphertexts, where each ciphertext has noise that hides the message. A modification of Brakerski's scale-invariant FHE scheme, the FV scheme operates under the RLWE hardness assumption [26]. A generalized version of the FV scheme, detailed in a study done by Lepoint et al., can be seen in the example below:

#### 5.3.1.1 Parameter Generation

**FV.ParamsGen( $\lambda$ ):** Given the security parameter  $\lambda$ , fix a positive integer  $d$  that determines  $R$ , moduli  $q$  and  $t$  with  $1 < t < q$ , distributions  $\chi_{key}$ ,  $\chi_{err}$  on  $R$  and an integer base  $w > 1$ . Output  $d, q, t, \chi_{key}, \chi_{err}, w$  [25].

### 5.3.1.2 Key Generation

$\text{FV.KeyGen}(d, q, t, \chi_{key}, \chi_{err}, w)$ : Sample  $s \leftarrow \chi_{key}$ ,  $a \leftarrow R_q$  uniformly at random, and  $e \leftarrow \chi_{err}$  and compute  $b = [-(as + e)]_q$ . Sample  $a \leftarrow R_q^{l_{w,q}}$  uniformly at random,  $e \leftarrow \chi_{err}^{l_{w,q}}$ , compute  $((\text{PowersOf}_{w,q}(s^2) - (e + a * s))_q, a) \in R^{l_{w,q}}$  and output  $(pk, sk, evk) = ((b, a), s, \gamma)$  [25].

### 5.3.1.3 Encrypt

$\text{FV.Encrypt}((b, a), m)$ : This message space is  $R/tR$ . For a message  $m + tR$ , sample  $u \leftarrow \chi_{key}$ ,  $e_1, e_2 \leftarrow \chi_{err}$ , and output the the ciphertext  $c = ([\delta[m]_t + bu + e_1]_q, [au + e_2]_q) \in R^2$  [25].

### 5.3.1.4 Decrypt

$\text{FV.Decrypt}(s, c)$ : Decrypt a ciphertext  $c = (c_0, c_1)$  by  $m = [[t/q * [c_0 + c_1 * s]_q]]_t \in R$  [25].

### 5.3.1.5 Add

$\text{FV.Add}(c_1, c_2)$ : Given ciphertexts  $c_1 = (c_{1,0}, c_{1,1})$  and  $c_2 = (c_{2,0}, c_{2,1})$ , output  $c_{add} = ([c_{1,0}, c_{2,0}]_q + [c_{1,1}, c_{2,1}]_q)$  [25].

### 5.3.1.6 ReLin

$\text{FV.ReLin}(c_{mult}, evk)$ : Let  $(b, a) = evk$  and let  $c_{mult} = (c_0, c_1, c_2)$ . Output the ciphertext [25]

$$[c_0 + < \text{WordDecomp}_{w,q}(c_2), b >]_q, [c_1 + < \text{WordDecomp}_{w,q}(c_2), a >]_q$$

### 5.3.1.7 Mult

$\text{FV.Mult}(c_1, c_2, evk)$ : Output the ciphertext  $c_{mult} = \text{FV.ReLin}(c_{mult}, evk)$ , where [25]

$$c_{mult} = (c_0, c_1, c_2) = ([t/q * c_{1,0} * c_{2,0}]]_q, [[t/q * (c_{1,0} * c_{2,1} + c_{1,1} * c_{2,0})]]_q, [[t/q * c_{1,1} * c_{2,1}]]_q)$$

### 5.3.2 BGV Cryptosystem

The BGV cryptosystem is a FHE scheme that allows for both addition and multiplication. Like most FHE schemes, BGV encryption is based on noisy ciphertexts, where each ciphertext has noise that hides the message. A generalized version of the BGV scheme, detailed in a study done by Lepoint et. al, can be seen in the example below [25]:

#### 5.3.2.1 Parameter Generation

BGV.ParamsGen( $\lambda, L$ ): Given the security parameter  $\lambda$ , fix a positive integer  $d$  that determines  $R$  and a distribution  $\chi$  on  $R$ . For  $j = L$  down to 0, generate a decreasing ladder of moduli  $q_i$ . Output  $d, q_i, \chi$  [25].

#### 5.3.2.2 Key Generation

BGV.KeyGend,  $q_i, \chi$ : For  $j = L$  down to 0, sample  $s'_i \leftarrow \chi$  and set  $s_1 = (1, s'_i)$ . Sample  $a'_i \leftarrow R_{q_i}$  and an element  $e_i \leftarrow \chi$  and set  $b_i = a'_i s'_i + 2e_i$ . Set  $a_i = (b_i, -a'_i)^T$ . Set  $s'_j = s_j x s_j \in R_{q_j}^{(2/2)}$ . Set  $b_i = a_i + Powersof2(s_i)$  (Add  $Powersof2(s_1) \in R_{q_i}^{[log_2(q_i)]}$  to  $a$ 's first column). Set  $\tau_{s'_{j+1} \rightarrow s_j} = b_i$  except for when  $j = L$ . Set the secret key  $sk$  to a vector of  $s_i$  and the public key  $pk$  a vector of  $a_i$  and a vector of  $\tau_{s'_{j+1} \rightarrow s_j}$  is a public parameter [25].

#### 5.3.2.3 Encrypt

BGV.Encrypt $pk, m$ : To encrypt a message  $m \in R_2$ , set  $m = (m, 0) \in R_2^2$ . Sample  $r \leftarrow \chi$  and  $e \leftarrow \chi^2$  output the ciphertext [25]

$$c = m + 2 * e + a_L^T * r \in R_{q_L}^2$$



#### 5.3.2.4 Decrypt

BGV.Decrypt $sk, c$ : Suppose the ciphertext  $c$  is encrypted under  $s_j$ . To decrypt  $c$ , compute [25]

$$m = [[\langle c, s_j \rangle]_{q_j}]_2$$

#### 5.3.2.5 Switch Key

BGV.SwitchKey $\tau_{s'_{j+1} \rightarrow s_j}, c, q_j$ : Output the new ciphertext [25]

$$c_1 = \text{BitDecomp}(c)^T * b_j$$

#### 5.3.2.6 Refresh

BGV.Refresh $\tau_{s'_{j+1} \rightarrow s_j}, c, q_j$ : Suppose the ciphertext is encrypted under  $s'_j$ . Do the following [25]:

1. Switch Keys: Set  $c_1 \leftarrow \text{BGV.SwitchKey}\tau_{s'_{j+1} \rightarrow s_j}, c, q_j$  for modulus  $q_j$ .
2. Switch Moduli: Set  $c_2 \leftarrow \text{BGV.Scale}c_1, q_j, q_{j-1}, 2$ , a ciphertext under  $s_{j-1}$  for modulus  $q_{j-1}$ .

#### 5.3.2.7 Add

BGV.Add $pk, c_1, c_2$ : Takes two ciphertexts encrypted under the same key  $s_j$  [25].

$$c_3 = c_1 + c_2 \quad c_{add} = \text{Refresh}(c_3, \tau_{s'_j \rightarrow s_{j-1}}, q_j, q_{j-1})$$

#### 5.3.2.8 Mult

BGV.Mult $pk, c_1, c_2$ : Takes two ciphertexts encrypted under the same key  $s_j$  [25].

$$c_3 = L_{c_1, c_2}^{long}(x * x) \quad c_{mult} = \text{Refresh}(c_3, \tau_{s'_j \rightarrow s_{j-1}}, q_j, q_{j-1})$$

## 5.4 HELib vs. Microsoft SEAL

To date, there are few practical implementations of FHE cryptosystems. A popular implementation of the FV scheme is the Microsoft SEAL library. SEAL supports Python and C++ development and can be used for a myriad of applications. A popular implementation of the BGV scheme is HELib. Like the Microsoft SEAL library, HELib is also suitable for Python and C++. Both libraries have similar capabilities, so when selecting which library to utilize the comparison below was considered [27] [28].

Initial comparison was done on the basic features that each library provides. These features include asymmetry, serialization/deserialization, negative computations, and encryption parameter/ciphertext size [24]. This can be observed in table 5.1.

**Table 5.1:** Basic Features

Basic Features	SEAL	HELlib
Asymmetric	Yes	Yes
Serialization and Deserialization of keys and ciphertexts	Yes	Yes
Negative Computations Support	Yes	No
Ciphertext size (less than 1MB for 1 input)	No	No
Can run on less than 2GB RAM	No	Yes

Microsoft SEAL and HELib provide asymmetry or implement a PKE scheme. Recall that PKE cryptosystems have both a public key, that is used to encrypt the plaintext data, and a private key, that is used to decrypt the ciphertext data. The public key can be shared with various users while the private key remains a secret and is held only by the individual authorized to decrypt the ciphertext data. Both libraries also provide for the Serialization and Deserialization of keys and ciphertext.

This means the developer does not have to implement an API for local storage and retrieval when it comes to input/output for the encryption schemes. The first difference between Microsoft SEAL and HELib can be observed with Negative computations. In integer arithmetic, certain operations can result in a negative value. One such operation is subtraction, where the second operand is greater than the first operand. While HELib does not have the ability to encode for negative values, Microsoft SEAL provides an Integer Encoder or Fractional Encoder that supports negative computation. Both libraries have a ciphertext size of a least 1MB for 1 input. This size is primarily due to choice of input encryption parameters. These parameters include the plain modulus, coefficient modulus, polynomial modulus, etc. Size of these parameters not only affects ciphertext size, but RAM requirements as well. Although HELib can still effectively run on less than 2GB of RAM, Microsoft SEAL cannot.

Following the basic feature comparison, the advanced features of both libraries were considered. These features include noise budget, reryption, ciphertext packing, relinearization, and multithreading [24]. This can be observed in table 5.2.

**Table 5.2:** Advanced Features

Advanced Features	SEAL	HELlib
Noise affected after each computation	Yes	Yes
Reryption	No	Yes
Relinearization	Yes	Yes
Ciphertext packing	Yes	Yes
Multithreading	Yes	Yes

Because Microsoft SEAL and HELib are built upon RLWE cryptosystems, they are both noise affected after each computation. Recall that LWE, in this case RLWE, cryptosystems work by hiding the plaintext with noise and that, after a certain threshold, a noise encrypted plaintext will not decrypt to it's original state. When these

ciphertexts are operated upon, the noise grows. With addition/subtraction, noise growth is fairly manageable, but with multiplication, noise growth can become extremely large. One way to manage such noise growth is reencryption. The process of reencryption converts bounded depth homomorphism to unbounded depth homomorphism, resetting the ciphertext noise. HElib allows for reencryption, while Microsoft SEAL does not. In addition to reencryption, relinearization can be used to manage noise growth. Relinearization focuses on reducing the size of the output ciphertext of the multiplication operation. Microsoft SEAL and HElib both provide a method for relinearization.

For the purpose of speed-up Microsoft SEAL and HElib have a ciphertext packing feature. This technique takes advantage of the Chinese Remainder Theorem and packs multiple plaintext values into a single ciphertext vector. Operations can then be performed on the entire vector resulting in faster computation. This process is called Single Instruction Multiple Data or SIMD. For additional speedup, both libraries have multithreading capabilities. Microsoft SEAL contains many functions that are thread-safe by default, whereas HElib can be made thread-safe by setting a few flags.

Table 5.3 illustrates the basic operations that both libraries provide [24].

**Table 5.3:** Operations

Operations	SEAL	HElib
Addition/Subtraction	Yes	Yes
Multiplication	Yes	Yes
Comparison	No	No
Division	No	No
Boolean Op.	No	No
Bitwise Op.	Yes	Yes
Matrix Op.	Yes	Yes
Exponentiation	Yes	Yes
Square	Yes	Yes
Negation	Yes	Yes
Addition/Subtraction/Multiplication Plain	Yes	No

Microsoft SEAL and HElib allow for basic addition, subtraction, and multiplication between ciphertexts. Neither library allows for comparison, division, or boolean operations between ciphertexts. Both libraries also provide the capability of bitwise operations, matrix operations, exponentiation, square, and negation of ciphertexts. The only difference between the two libraries with regards to operations is that Microsoft SEAL supports addition, subtraction, and multiplication between a ciphertext and a plaintext.

Looking at the comparison between Microsoft SEAL and HElib, both libraries provide similar functionality and operations. The primary difference between the two libraries is that Microsoft SEAL has negative computation support and explicitly defined functions for addition/subtraction/multiplication between plaintexts and ciphertexts while HElib can run on less than 2GB RAM and provides reryption support. Although HElib does not have negative computation support, it is still possible

to differentiate between negative and positive values. Take the plaintext modulus,  $p$ , any value less than  $p/2$  is considered positive and any value greater than  $p/2$  is considered negative. HELib also has two functions, `addConstant` and `multiplyByConstant`, that produce the same functionality as basic arithmetic between ciphertexts and plaintexts. In this case, the plaintext is just treated as a constant vector. Unfortunately, while it is possible for HELib to mimic the missing functionality seen in Microsoft SEAL, it is not possible for Microsoft SEAL to run on less than 2GB of RAM or replicate reryption. In addition, while there is limited documentation on both libraries, when it comes to privacy preserving CNN, HELib appears to be the library of choice. For these reasons, the library chosen for this study is HELib.

# Chapter 6

---

## HElib Functions, Security and Parameter Selection

### 6.1 Math Notation

The HELib implementation uses polynomial rings over integers modulo an irreducible, cyclotomic polynomial. This is represented as  $R = \mathbb{Z}_q[x]/\phi(x)$ , where  $\phi_m(x)$  is the  $m^{th}$  cyclotomic polynomial. In the case of a composite integer  $q$  the polynomial ring is defined as  $R_q = \mathbb{Z}[x]/(\phi_m(x), q)$  where  $A_q$  is the set of integer polynomials of degree up to  $\phi_m(x)$  modulo  $q$ .

The plaintext space is binary polynomials  $R_2$ . The ciphertext and key space are the vectors defined over polynomial ring  $R$ . Also plaintext  $a$  is in the coefficient representation, where  $a = \langle a_0, a_1, \dots, a_{\phi(m)-1} \rangle \in \mathbb{Z}/q\mathbb{Z}^{\phi(m)}$  is a list of the coefficients in the polynomial  $a(X) = \sum_{i < \phi(m)} a_i X^i$ .

Below is a list of parameters that will be used to describe the functions found in HELib.

$\lambda$ : security parameter, representing  $2^\lambda$  security against unknown attacks

$n$ : dimension

$q$ : current integer modulus

$\chi$ : noise distribution

$N$ : additional integer parameter

## 6.2 Functions

The following section describes the basic functions provided by HELib [29].

### 6.2.1 ContextGen

Context Generation refers to the process of calculation the ciphertext modulus  $q$ , the variance error distribution  $\sigma$ , and the dimension  $n$ . In order to successfully calculate these three variables, context generation requires the plaintext modulus  $p$ , the multiplicative depth  $L$ , and the security parameter  $\lambda$ .

### 6.2.2 Key Generation

Secret Key: To generate the current secret key  $sk$

$$s' \leftarrow \chi^N \tag{6.1}$$

$$sk = s \leftarrow (1, s'[1], \dots, s'[n]) \in R_q^{n+1} \tag{6.2}$$

Public Key: To generate the current public key  $pk$

$$A' \leftarrow R_q^{N \times n} \tag{6.3}$$

$$e \leftarrow \chi^N \tag{6.4}$$

$$b \leftarrow A's' + 2e \tag{6.5}$$



### 6.2.3 Encryption

To encrypt a message  $a \in R_2$

$$a \leftarrow (a, 0, \dots, 0) \in R_q^{n+1} \quad (6.6)$$

$$r \leftarrow R_2^N \quad (6.7)$$

$$c \leftarrow a + A^T r \in R_q^{n+1} \quad (6.8)$$

Here, the transpose of the public key  $A^T$  is multiplied by a sample  $r$  and then added to the message  $a$ .

### 6.2.4 Decryption

To decrypt a ciphertext  $c$

$$a \leftarrow [[\langle c, s \rangle \bmod \phi_m(x)]_q]_2 \quad (6.9)$$

Here, the inner product between  $c$  and the secret key  $s$  over the polynomial ring  $A_q$  is computed, where  $q$  is the current modulus. This result is then reduced once more modulo 2.

### 6.2.5 Addition and Multiplication

Homomorphic addition is done by simply adding two ciphertext vectors over  $R_q$  with respect to the same secret key and modulus  $q$ . Homomorphic multiplication is done by taking the tensor product of two ciphertext vectors over  $R_q$  with respect to the same secret key and modulus  $q$ . This operation changes the current key.

### 6.2.6 Modulus Switching

To manage the inevitable noise growth that comes with homomorphic operations BGV achieves FHE with the use of recursive modulus switching. Modulus switching allows for the transformation of ciphertext  $c \bmod q$  to ciphertext  $c' \bmod p$  by scaling the original ciphertext by a factor of  $p/q$  and rounding the result accordingly. This process reduces the magnitude of the noise [30].

### 6.2.7 Bootstrapping

An alternative way to manage noise growth, Bootstrapping is used in BGV as an optimization to allow for unlimited homomorphic operations. Bootstrapping is defined as the process of refreshing a ciphertext by homomorphically evaluating the decryption function. Refreshing is done by encrypting the ciphertext with a second layer and decrypting the first layer homomorphically. In order to decrypt the ciphertext homomorphically, the user must encrypt the secret key and use it as an input to the Evaluation function alongside the ciphertext to be refreshed [31].

## 6.3 Security

Since HELib is as an implementation of the BGV cryptosystem and the BGV cryptosystem is based on the RLWE, the security of HELib will be based on attacks against RLWE. Based on current research, the best known attacks against RLWE schemes are those used against the LWE problem.

Currently, there are three well known attack algorithms against LWE. These attacks are: The Unique Shortest Vector Problem (uSVP) attack, the decoding attack, and the dual attack. The uSVP attack works by taking several LWE sample vectors and translating them into a matrix where each row represents a lattice. This matrix contains information that reveals the secret errors from each vector such that if the

shortest vector is discovered, the secret can be recovered as well. To find the shortest vector, the iterative block-wise algorithm for basis reduction also known as BKZ can be utilized. The decoding attack works to solve the search-LWE problem. This problem is solved by treating it as the Bounded Distance Decoding problem, where the BKZ basis reduction is utilized followed by the recursive Nearest Plane algorithm. The dual attack is used for 'small' secrets by aiming to solve the decisional problem and not the secret. This attack utilizes BKZ to find the shortest non-zero vector. This short vector is then used to distinguish the samples based on the sample size [32].

## 6.4 Parameter Selection

Parameter selection within HELib plays an important role when it comes to both the security and functionality of the overall system. This section details recommended parameters for both security and functionality.

### 6.4.1 Parameters for Security

To achieve a minimum level of security, is important to note that certain input parameters need to be initialized appropriately. One such parameter is the ciphertext modulus  $q$ . In a study done by Chase et al. researchers use the uSVP attack, decoding attack, and dual attack to determine the necessary size of  $q$  to achieve a specific security level for a given dimension  $n$ . In this case,  $q$  value recommendations were made for 3 different security levels: 128 bits, 192 bits, and 256 bits. In figure 6.1 a recommended  $\log_2 q$  is given for  $n = 2^{10}$  to  $n = 2^{15}$  for each security level. Estimated running time for the uSVP attack, decoding attack, and dual attack are given in bits [32].

distribution	n	security level	log(q)	uSVP	dec	dual
uniform	1024	128	31	130.6	133.8	147.5
		192	22	203.6	211.2	231.8
		256	18	269.9	280.5	303.6
	2048	128	59	129.5	129.7	139.2
		192	42	194.0	197.6	212.4
		256	33	263.8	270.7	289.9
	4096	128	113	131.9	129.4	136.8
		192	80	192.7	193.2	203.2
		256	63	260.7	263.6	277.6
	8192	128	222	132.9	128.9	134.9
		192	157	195.4	192.8	200.6
		256	124	257.0	256.8	266.7
	16384	128	440	133.9	129.0	133.0
		192	310	196.4	192.4	198.7
		256	243	259.5	256.6	264.1
	32768	128	880	134.3	129.1	131.6
		192	612	198.8	193.9	198.2
		256	480	261.6	257.6	263.6

**Figure 6.1:** Security Parameter Recommendations [32]

#### 6.4.2 Parameters for Functionality

When utilizing HELib, each parameter is associated with a different functionality. Below is a detailed explanation of what each parameter represents [33].

$m$  represents the specific modulus or ciphertext base. This value is the same  $q$  value mentioned in the 'Parameters for Security' section. As detailed before, this value is important when it comes to the security of the overall cryptosystem. In HELib, the FindM function takes  $k, L, c, p, d, s$  and outputs an appropriate  $m$ . The value for  $m$  can also be manually set.

$p$  represents the plaintext base. This value needs to be a prime number and is used as the coefficient modulus. In other words, computations are done modulo  $p$ .

$r$  represents the lifting value. This value is also part of the native plaintext space and the default is  $r = 1$ . When  $r = 1$  computations are done modulo  $p$ . In the case

$r! = 1$ , computations are done modulo  $p^r$ .

$L$  represents the number of levels in the modulus chain. Levels are an integral part of the underlying cryptosystem used by HELib. From a high-level perspective, levels refer to a non-fixed ciphertext space and are used to reduce the noise inside ciphertexts. This noise reduction is done using the modulus switching technique mentioned earlier. Levels are normally changed after ciphertext multiplication as this is the operation that generates the most noise. This means the level value is largely dependent on the evaluation function [34].

$c$  represents the number of columns in the key-switching matrix. The number of columns plays a key role in the relineralization process. Recall that the purpose of relinearization is to reduce the overhead in ciphertext multiplication and can be used to manage noise growth. Ultimately this value is also dependent on the multiplication depth of the evaluation function. The default value is  $c = 3$  [34].

$w$  represents the Hamming weight of the secret key.

$d$  represents the degree of field extension. The default value is  $d = 1$ .

$k$  represents the security parameter. This is the same  $\lambda$  value mentioned in the 'Parameters for Security' section. The default value is  $k = 80$ . Setting  $k = 128$  is considered equivalent to the security of AES-128.

$s$  represents the minimum number of slots. This value is used for ciphertext packing and allows for SIMD. Recall packing refers to the process of combining several messages into one ciphertext. This is generally used to reduce the overall number

of ciphertexts and to speedup computation time. In HELib there are two types of packing: pack into coefficients, and pack into subfields (CRT based packing) [34].

# Chapter 7

---

## Design

In this study, an approach to integrate HE and CNN was explored. HE was utilized to perform the necessary classification on an encrypted dataset, such that when the information is decrypted, the decrypted result matches the would be result of classification on unencrypted data. To successfully create a privacy preserving CNN, it is important to study the building blocks of these networks: layers. Prior to creating a privacy preserving CNN each layer was studied, implemented, and tested in both the plaintext space and the ciphertext space. The plaintext layers were used as a baseline comparison to ensure that the encrypted classification was indeed correct.

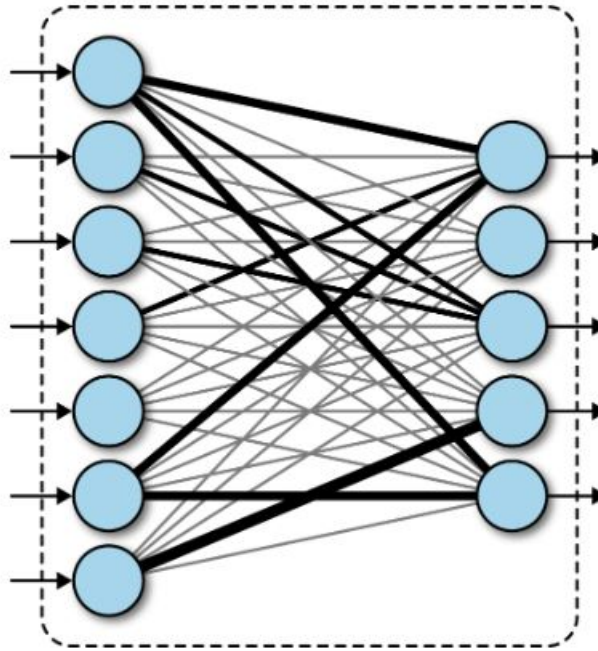
### 7.1 CNN Layers

CNN are made up of cascading layers that take an input layer of image data and transform it into an output layer of label scores. The four common layers used in CNN include: Fully Connected Layer, Convolutional Layer, Pooling Layer, and Activation Layer [35].

#### 7.1.1 Fully Connected

Fully Connected layer is the layer where each neuron is connected to all the neurons in the previous layer. In this layer the total number of weights is equivalent to the product of the total number of neurons in the previous layer and the total number

of neurons in the current layer. In the context of CNN, the Fully Connected layer appears as the final layer which outputs an  $N$  dimensional vector where  $N$  is the number of classes the program can choose from [35]. This can be seen in figure 7.1.

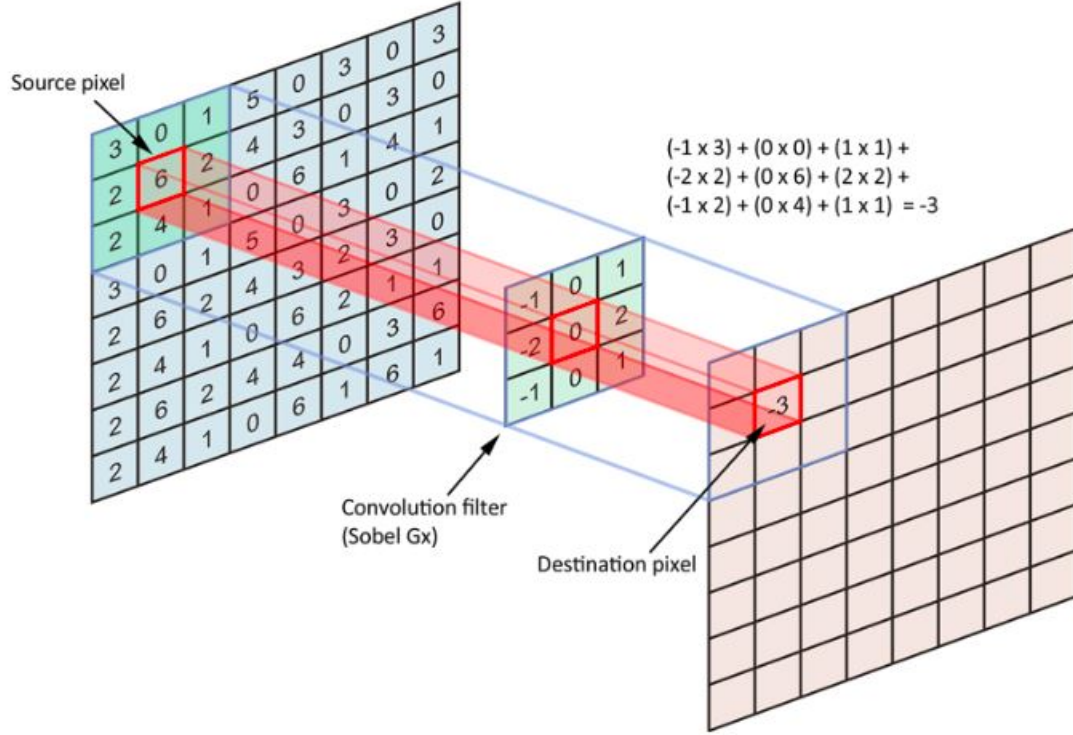


**Figure 7.1:** Fully Connected Layer [36]

### 7.1.2 Convolution

Convolution layer is the layer that applies a sliding filter to an input image and outputs the sum of elementwise multiplications between filter weights and neurons from the previous layer. The purpose of using a filter is to extract certain characteristics from an image, thus several filters can be used in the same layer to extract different characteristics. The sliding filter(s) used in this layer are 3 dimensions and contain a set of weights that are learned during the training phase. The output of applying the 3-dimensional filter is a 2-dimensional matrix which is then stacked with all other filter outputs to create a 3-dimensional result. This layer is unique to CNN and is based on the technique of convolutional filtering found in image processing [35]. This can be seen in figure 7.2.

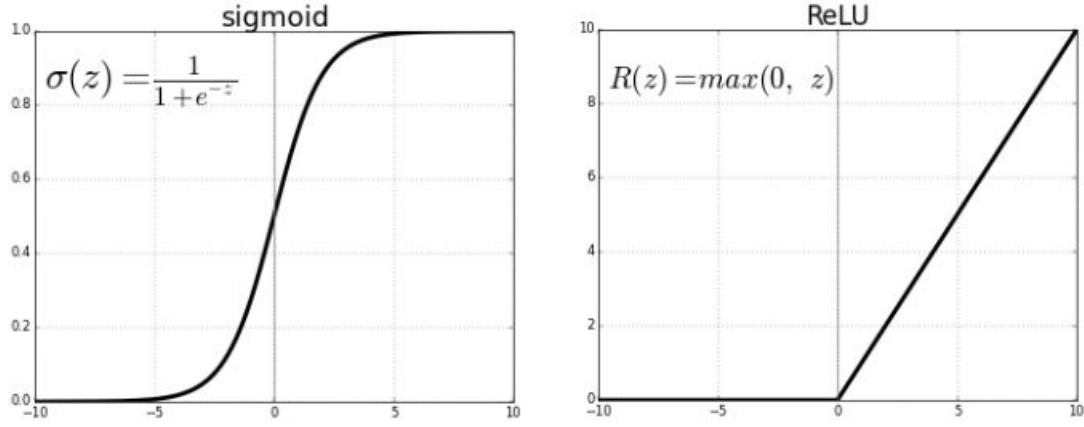




**Figure 7.2:** Convolution Layer [37]

### 7.1.3 Activation

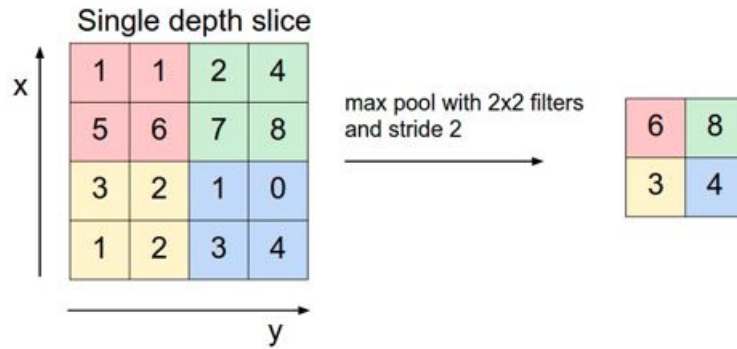
Activation layer is the layer that applies a nonlinear activation function to each neuron of the previous layer. Activation layers allow ANN and CNN to solve more complex classification problems by introducing a non-linear component. In fact, without the addition of this layer or the Pooling layer CNN are only able to classify linearly. Two popular activation functions are the ReLU function and the Sigmoid function. The ReLU activation applies  $f(x) = \max(0, x)$  to then input neuron and the Sigmoid activation function applies  $f(x) = \frac{1}{1+e^{-x}}$  to then input neuron. Generally, the Activation layer is found after the Fully Connected layer or the Convolutional layer [35]. This can be seen in figure 7.3.



**Figure 7.3:** Sigmoid Activation and ReLU Activation [38]

#### 7.1.4 Pooling

Pooling layer is the layer that applies a function on non-overlapping subsections from the previous layer to output one neuron. Like the Activation layer, the Pooling layer is a non-linear layer. In addition to non-linearity, the pooling layer is also used to reduce the total number of neurons by reducing spatial size. Two of the most common pooling layers are the Max Pooling layer and the Average Pooling layer. The Max Pooling layer outputs the maximum value within the subsection and the Average Pooling layer outputs the average of the values within the subsection. Generally, the Pooling layer is found after the Activation layer [35]. This can be seen in figure 7.4.



**Figure 7.4:** Pool Layer [37]

## 7.2 Layer Design

The focus of this study is only on encrypted classification, therefore the layers were only created with the feed-forward phase in mind. Backpropagation was not included. HELib only supports additions and multiplications. This means that not only does HELib work best when computing low-degree polynomials, it cannot compute inverses or exponents. Thus, the primary challenge was dealing with the non-linear layers, specifically the activation layer and the max-pooling layer. To combat this challenge, the same approach as the experiment done by Hesamifard et. al [4] was implemented. Layers in the plaintext space were designed in C++. These layers take inputs that are vectors of data type long and outputs vectors of data type long. Layers in the ciphertext space were designed in C++ utilizing HELib. These layers take inputs that are vectors of data type Ctxt and outputs vectors of data type Ctxt.

### 7.2.1 HELib Encoding and Functions

In the creation of each privacy preserving layer, HELib played a major role. Two major considerations when working with HELib were how to encode the input image and what functions to utilized in order to achieve the desired results.

#### 7.2.1.1 Encoding

To encode any input values for HELib encryption, the value was simply converted to the polynomial ZZ<sub>X</sub> form. This was done by utilizing the toZZ<sub>X</sub> function provided by the NTL library. It is important to note that the toZZ<sub>X</sub> function is unable to operate on floating point values, but because each input image consisted of pixels represented by a positive integer within the range of [0-255] this was not an issue. Thus each pixel was simply encoded with the toZZ<sub>X</sub> function before being encrypted. In the case that the input value was a floating point number, this number would have

to be scaled into an integer before utilizing the `toZZX` function.

### 7.2.1.2 Functions

To create a successful privacy preserving CNN there were certain high-level functions from HElib that were utilized. These functions are detailed below:

1. **Encrypt:** This function takes as input a public key and a value to be encrypted. This value must first be encoded in the format polynomial ZZ<sub>X</sub>. This function outputs a ciphertext value. The public key is generated during the initialization phase and is represented by the `FHEPubKey` class.

2. **Decrypt:** This function takes as input a secret key and a value to be decrypted. This function outputs a plaintext value in the format polynomial ZZ<sub>X</sub>. The secret key is generated during the initialization phase and is a subclass of `FHEPubKey`.

3. **AddConstant:** This function takes as input a plaintext value and a ciphertext value. This function adds the values together and outputs a ciphertext.

4. **MultiplyByConstant:** This function takes as input a plaintext value and a ciphertext value. This function multiplies the values together and outputs a ciphertext.

5. **Ciphertext Addition:** Denoted simply by the '+' symbol, this function takes as input two ciphertext values. This function adds the values together and outputs a ciphertext.

6. **Ciphertext Multiplication:** Denoted simply by the '\*' symbol, this function takes as input two ciphertext values. This function multiplies the values together and outputs a ciphertext.

## 7.2.2 Fully Connected Design

The input to this function is a 1-Dimensional weights vector holding type long and a 1-Dimensional input vector holding either type long or `Ctxt` depending on if the input is unencrypted/encrypted. The output of this function is also a 1-Dimensional vector

holding either type long or Ctxt depending on if the input is unencrypted/encrypted. This layer consists of performing the dot product between the input vector and weight vector. The bias vector is then added elementwise to the result. In the plaintext fully connected layer, the input vector, weights vector, bias vector, and output vector are all unencrypted. In the ciphertext fully connected layer, the input vector and output vector are encrypted while the weights vector and bias vector are unencrypted. HElib provides the ability to perform addition and multiplication between plaintexts and ciphertexts with the AddConstant function and MultiplyByConstant function.

---

**Algorithm 1** Fully Connected Layer

---

**Input:**  $in, nInput, nOutput, weight, bias, scale$   
**Output:**  $out$   
 $inSize \leftarrow nInput$   
 $outSize \leftarrow nOutput$   
**for**  $i = 0, 1, \dots, outSize$  **do**  
    **for**  $j = 0, 1, \dots, inSize$  **do**  
         $tmp \leftarrow in[j]$   
         $tmp \leftarrow tmp * weight[j][i]$   
        **if**  $j = 0$  **then**  
             $out[i] \leftarrow tmp$   
        **else**  
             $out[i][j][k] += tmp$   
     $out[i] \leftarrow b[i] * scale$

---

### 7.2.3 Convolution Design

The input to this function is a 4-Dimensional weights vector holding type long and a 3-Dimensional input vector holding either type long or Ctxt depending on if the input is unencrypted/encrypted. The output to this function is also a 3-Dimensional vector holding either type long or Ctxt depending on if the input is unencrypted/encrypted. This layer consists of performing the dot product between the input vector and weight vector within the confines of a sliding filter or kernel. The number of kernels applied to the input vector determines the depth of the output vector. The bias vector is then

added elementwise to the result. In the plaintext convolution layer, the input vector, weights vector, bias vector, and output vector are all unencrypted. In the ciphertext convolution layer, the input vector and output vector are encrypted while the weights vector and bias vector are unencrypted. HElib provides the ability to perform addition and multiplication between plaintexts and ciphertexts with the `AddConstant` function and `MultiplyByConstant` function.

---

**Algorithm 2** Convolution Layer

---

**Input:**  $in, inHeight, inWidth, depth, kernelHeight, kernelWidth, nkernels, weight, bias, scale$   
**Output:**  $out$   
 $count \leftarrow 0$   
 $outHeight \leftarrow inHeight - (kernelHeight - 1)$   
 $outWidth \leftarrow inWidth - (kernelWidth - 1)$   
**for**  $k = 0, 1, \dots, nkernels$  **do**  
    **for**  $y = 0, 1, \dots, outHeight$  **do**  
        **for**  $x = 0, 1, \dots, outWidth$  **do**  
            **for**  $c = 0, 1, \dots, depth$  **do**  
                **for**  $ky = 0, 1, \dots, kernelHeight$  **do**  
                    **for**  $kx = 0, 1, \dots, kernelWidth$  **do**  
                         $tmp \leftarrow in[y + ky][x + kx][c]$   
                         $tmp \leftarrow tmp * weight[ky][kx][c][k]$   
                        **if**  $count = 0$  **then**  
                             $out[y][x][k] \leftarrow tmp$   
                             $count++$   
                        **else**  
                             $out[y][x][k] += tmp$   
                             $count++$   
                        **if**  $count = (kernelHeight * kernelWidth * depth)$  **then**  
                             $count = 0$   
                         $out[y][x][k] \leftarrow b[k] * scale$

---

#### 7.2.4 Activation Design

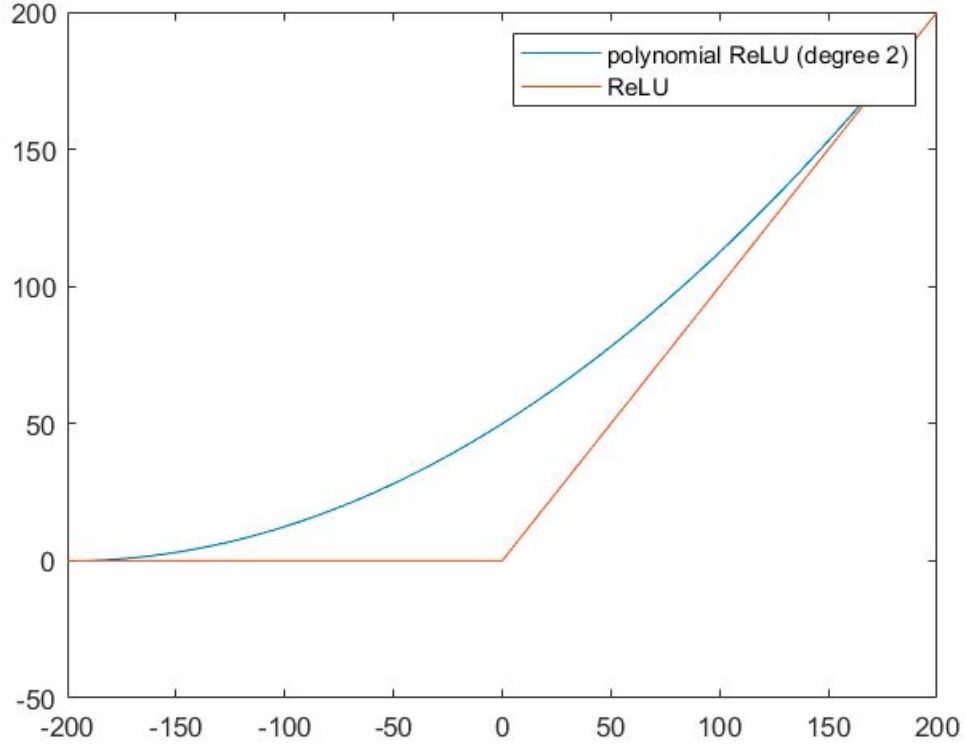
HElib provides only linear operations, therefore any non-linear layer had to be modified. Because the activation layer is by definition non-linear, the functions needed to be modified accordingly. The input to both activation functions are a 1-Dimensional/3-Dimensional input vector holding either type long or Ctxt depending on if the input

is unencrypted/encrypted. The output to both activation functions are also a 1-Dimensional/3-Dimensional vector holding either type long or Ctxt depending on if the input is unencrypted/encrypted. In the plaintext activation layer, the input vector and output vector are unencrypted. In the ciphertext activation layer, the input vector and output vector are encrypted.

#### 7.2.4.1 ReLU

For the ReLU activation layer, the approximation approach taken by Hesamifard et. al [4] was implemented. In this experiment authors decided to take a different approach and approximate the derivative of the ReLU function. This approach was taken because of the derivative's impact on both error calculation and weight updates. Authors noted that a simulation of the ReLU derivative, the Step function, mimics the behavior of the Sigmoid function. From this observation, authors calculated the integral of the polynomial approximation of the Sigmoid function. This integral was then used to approximate the ReLU function. The polynomial approximation used is:  $0.0012x^2 + 0.5x + 52$ . Results from the study done by Hesamifard et al. indicate that this function yielded the best approximation of the ReLU when compared to other methods: numerical analysis, Taylor series, standard Chebyshev, and modified Chebyshev. The comparison of this approximation and the ReLU function can be seen in the figure below.

HElib is unable to operate on floating point values. As a result, the coefficients seen in the polynomial approximation of the ReLU activation function had to be scaled by a factor of 10000. This allows each coefficient to be treated as an integer instead of a float. This scaling needs to be taken into account when observing the final output.



**Figure 7.5:** Approximation of ReLU function

#### 7.2.4.2 Sigmoid

For the Sigmoid activation layer, a Taylor series approximation was implemented. Because HELib only works well with lower degree polynomials, the Taylor series approximation is limited to a degree 3 polynomial. The polynomial approximation used is:  $-0.002x^3 + 0.25x + 0.5$ .

HELlib is unable to operate on floating point values. As a result, the coefficients seen in the polynomial approximation of the Sigmoid activation function had to be scaled by a factor of 10000. This allows each coefficient to be treated as an integer instead of a float. This scaling needs to be taken into account when observing the final output.



---

**Algorithm 3** ReLU Layer

---

**Input:**  $in, nInput, nOutput$   
**Output:**  $out$   
 $scale = 10000$   
 $c0 = 520000$   
 $c1 = 5000$   
 $c2 = 12$   
 $inSize \leftarrow nInput$   
**for**  $i = 0, 1, \dots, inSize$  **do**  
     $out[i] \leftarrow in[i] * in[i] * c2 + in[i] * c1 + c0$

---

---

**Algorithm 4** Sigmoid Layer

---

**Input:**  $in, nInput, nOutput$   
**Output:**  $out$   
 $scale = 10000$   
 $c0 = 5000$   
 $c1 = 2500$   
 $c2 = -200$   
 $inSize \leftarrow nInput$   
**for**  $i = 0, 1, \dots, inSize$  **do**  
     $out[i] \leftarrow in[i] * in[i] * in[i] * c2 + in[i] * c1 + c0$

---

### 7.2.5 Pooling Design

HElib does not provide any comparison operation, therefore the Max Pool layer had to be modified. In this case the max pool layer was replaced with a sumpool layer. Instead of outputting the largest value within a sliding window, sum pool just adds all the values within the sliding window and outputs that value. The input to this function is a 3-Dimensional input vector holding either type long or Ctxt depending on if the input is unencrypted/encrypted. The output to this function is also a 3-Dimensional vector holding either type long or Ctxt depending on if the input is unencrypted/encrypted. This layer consists of adding the values within the confines of a sliding window. In the plaintext sum pool layer, the input vector and output vector are unencrypted. In the ciphertext sum pool layer, the input vector and output vector are encrypted. HElib provides the ability to add ciphertexts with one another, thus this functionality was used in the ciphertext sum pool layer.

---

**Algorithm 5** SumPool Layer

---

**Input:**  $in, inHeight, inWidth, depth, pooly, poolx$ **Output:**  $out$  $count \leftarrow 0$  $outHeight \leftarrow inHeight / pooly$  $outWidth \leftarrow inWidth / poolx$ **for**  $c = 0, 1, \dots, depth$  **do**    **for**  $y = 0, 1, \dots, outHeight$  **do**        **for**  $x = 0, 1, \dots, outWidth$  **do**            **for**  $i = 0, 1, \dots, pooly$  **do**                **for**  $j = 0, 1, \dots, poolx$  **do**                     $tmp \leftarrow in[y * pooly + i][x * poolx + j][c]$                     **if**  $count = 0$  **then**                         $out[y][x][c] \leftarrow tmp$                          $count++$                     **else**                         $out[y][x][c] += tmp$                          $count++$                     **if**  $count = (pools * poolx)$  **then**                         $count = 0$ 

---

# Chapter 8

## Privacy Preserving Logic Gates

To test the basic functionality of privacy preserving classification, a small neural network was created to predict the output of logic gates. This neural network works by taking an input vector and weights file, feeding them through the network, and predicting the output based on the logic gate specified.

### 8.1 Logic Gates

Logic gates are the basic building blocks of any digital system. An electronic circuit that has one or more inputs and only one output, logic gates provide the perfect system for a basic neural network classifier. For the purpose of this experiment, the focus will be on two-input logic gates with the exception of the NOT gate, which has only one input. Figure 8.1 details the truth table for the logics gates to be classified by the privacy preserving neural network.

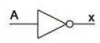






Name	NOT	AND	NAND	OR	NOR	XOR	XNOR																																																																																																
Alg. Expr.	$\overline{A}$	$AB$	$\overline{AB}$	$A+B$	$\overline{A+B}$	$A\oplus B$	$\overline{A\oplus B}$																																																																																																
Symbol																																																																																																							
Truth Table	<table><tr><th>A</th><th>X</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	X	0	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	B	A	X	0	0	0	0	1	0	1	0	0	1	1	1	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	B	A	X	0	0	1	0	1	1	1	0	1	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	B	A	X	0	0	0	0	1	1	1	0	1	1	1	1	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	B	A	X	0	0	1	0	1	0	1	0	0	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	B	A	X	0	0	0	0	1	1	1	0	1	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	B	A	X	0	0	1	0	1	0	1	0	0	1	1	1
A	X																																																																																																						
0	1																																																																																																						
1	0																																																																																																						
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	1																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					

Figure 8.1: Logic Gate Truth Tables [39]

Looking at the various truth tables, the privacy preserving neural network will work by taking in an input vector consisting of the values in columns B and A and output a vector consisting of the values in column X.

## 8.2 Network

The privacy preserving neural network used to predict the output of the logic gate consists of two layers: Fully Connected layer followed by a Sigmoid Activation layer. The idea behind using such a small network is to imitate the behavior of a basic perceptron. Here the input data is converted to an array and the weights data is converted to an array. A dot product is then performed between these two arrays and the result is fed through the Sigmoid activation function. A high level diagram of this network can be seen in the figure 8.2.

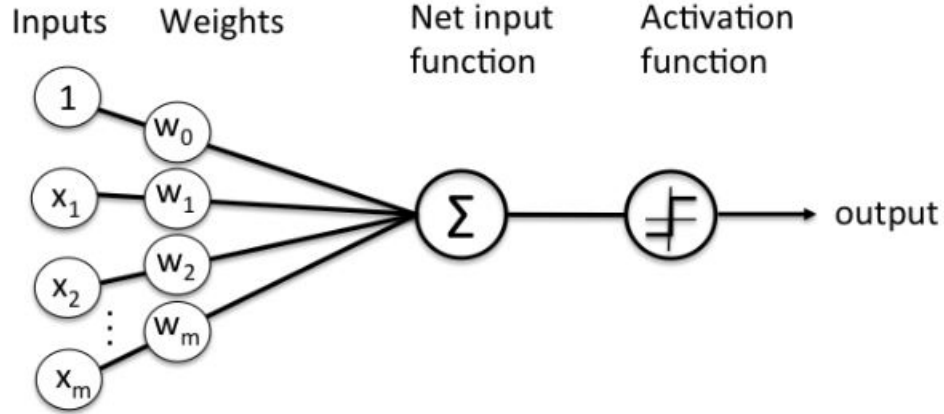


Figure 8.2: Perceptron [40]

## 8.3 Test Environment

Computations were run on a computer with 4GB RAM, Intel Core i3 processor, 2.4 GHz and Ubuntu 16.04.

## 8.4 Results

For the purposes of this small example, there was no training portion done to output a weights file. Instead, a predetermined weights file was fed through the privacy preserving classifier just to observe the performance of the encrypted arithmetic.

Table 8.1 details the output of the encrypted classifier when given the input data file and input weights file for the NOT gate.

**Table 8.1:** NOT

A	actual	expected
0	1.0	1
1	0.0	0

Table 8.2 details the output of the encrypted classifier when given the input data file and input weights file for the AND gate.

**Table 8.2:** AND

A	B	actual	expected
0	0	0.0	0
0	1	0.0	0
1	0	0.0	0
1	1	1.0	1

Table 8.3 details the output of the encrypted classifier when given the input data file and input weights file for the OR gate.

**Table 8.3:** OR

A	B	actual	expected
0	0	0.0	0
0	1	1.0	1
1	0	1.0	1
1	1	1.0	1

Table 8.4 details the output of the encrypted classifier when given the input data file and input weights file for the NAND gate.

**Table 8.4:** NAND

A	B	actual	expected
0	0	1.0	1
0	1	1.0	1
1	0	1.0	1
1	1	0.0	0

Table 8.5 details the output of the encrypted classifier when given the input data file and input weights file for the NOR gate.

**Table 8.5:** NOR

A	B	actual	expected
0	0	1.0	1
0	1	0.0	0
1	0	0.0	0
1	1	0.0	0

Table 8.6 details the output of the encrypted classifier when given the input data file and input weights file for the NOR gate.

**Table 8.6:** XOR

A	B	actual	expected
0	0	0.0	0
0	1	1.0	1
1	0	1.0	1
1	1	0.0	0

It can be noted that all gates performed as expected, yielding the correct output with both the encrypted input as well as the Sigmoid approximation. The example was used as a proof-of-concept before proceeding to a larger network for image classification.

# Chapter 9

---

## Privacy Preserving CNN

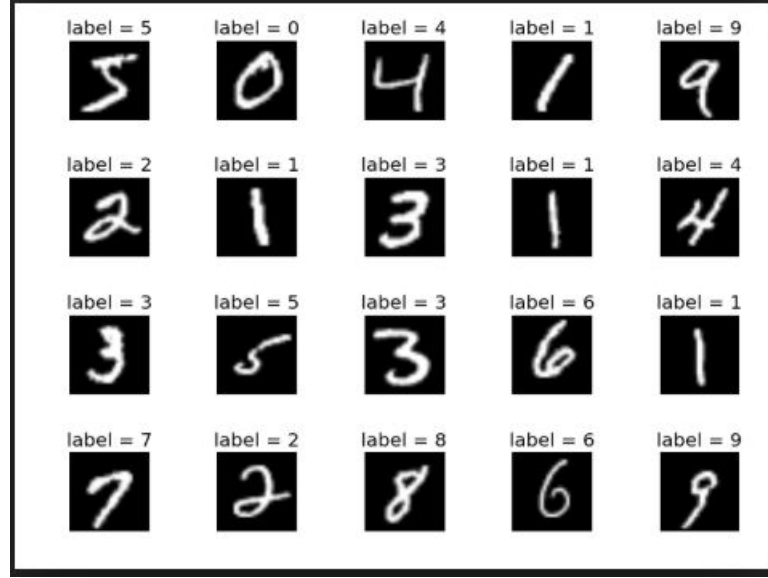
The primary motivation behind the creation of privacy preserving CNN is to maintain a level of information anonymity for all parties involved. While privacy preserving CNN are not needed in everyday scenarios, there are certainly situations that call for such measures. For example, when dealing with medical data, oftentimes the privacy of a patients personal information is of extreme importance or the model utilized by the hospital to predict a certain diagnosis can be proprietary information. In such a situation, privacy preserving CNN can allow patients to send personal information and receive a diagnosis, where both the information and the diagnosis are inaccessible to all parties except for the patient. In addition to this, the hospital can keep their model private from the patients, while still utilizing their classifier on the encrypted data.

### 9.1 Dataset

The privacy preserving CNN was trained and tested using the MNIST data set. This datasets was specifically chosen because of it's wide use in the deep learning community. This allowed for accuracy comparison with existing studies. This dataset consists of 60,000 images, with 50,000 image for the training portion and 10,000 images for the testing portion. Images in the MNIST database are 28x28 pixel arrays. Each pixel is a positive integer within the range of [0-255]. An example of images



from the MNIST dataset can be seen in figure 9.1.



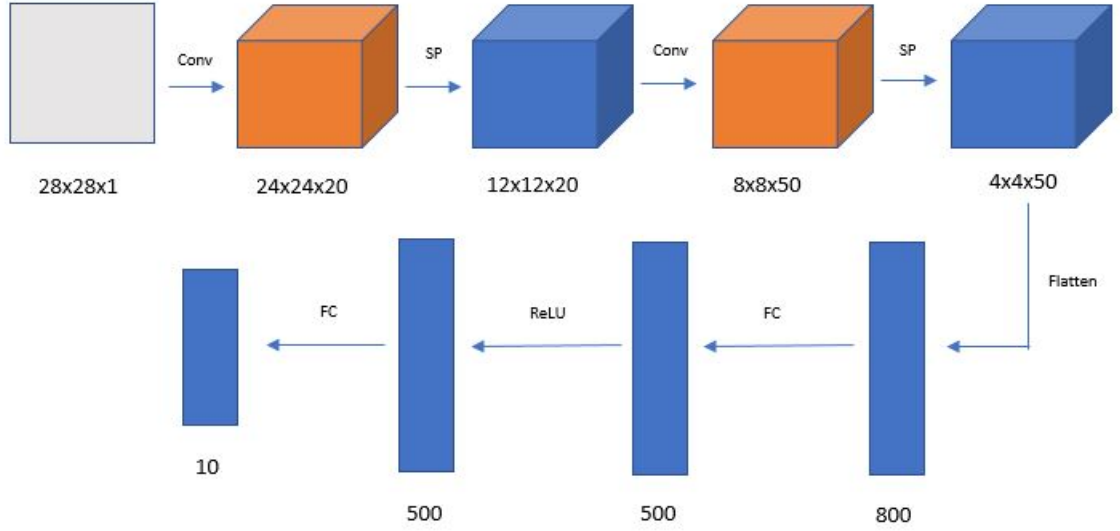
**Figure 9.1:** MNIST image sample

## 9.2 Network

The Network that was created to train and classify the MNIST data set can be seen in figure 9.2.

The following is a description of the Network seen in the figure 9.2.

1. Convolution Layer: Input image is  $28 \times 28 \times 1$ . The convolution has 20 kernels of size  $5 \times 5$  and a stride of  $(1,1)$ . The output of this layer is  $24 \times 24 \times 20$ .
2. Sum Pool Layer: Input is  $24 \times 24 \times 20$ . The stride is  $(2,2)$ . The output of this layer is  $12 \times 12 \times 20$ .
3. Convolution Layer: Input is  $12 \times 12 \times 20$ . The convolution has 50 kernels of size  $5 \times 5$  and a stride of  $(1,1)$ . The output of this layer is  $8 \times 8 \times 50$ .
4. Sum Pool Layer: Input is  $8 \times 8 \times 50$ . The stride is  $(2,2)$ . The output of this layer is  $4 \times 4 \times 50$ .
5. Flatten Layer: Input is  $4 \times 4 \times 50$ . The output of this layer is 800.



**Figure 9.2:** Privacy Preserving Convolutional Neural Network

6. Fully Connected Layer: This layer fully connects the incoming 800 nodes to the outgoing 500 nodes or is equivalently a multiplication by a 800x500 matrix.
7. ReLU Activation Layer: Takes the ReLU of the value at each input node.
8. Fully Connected Layer: This layer fully connects the incoming 500 nodes to the outgoing 10 nodes or is equivalently a multiplication by a 500x10 matrix.

### 9.2.1 Training

During the training phase, the original ReLU function was used and max pooling was replaced with sum pooling. The CNN was trained with the Keras framework with a Tensorflow backend on the MNIST database. Training was done on batches of size 128 for a total of 10 epochs.

The optimization algorithm used for training was Adam or the Adaptive Moment Estimation. The reason this was chosen is because Adam has low memory requirements and works well with little tuning of hyperparameters. Default parameters provided by Keras were utilized [41]:

Learning Rate = 0.001

beta1=0.9

beta2=0.999

epsilon=1e-8

### 9.2.2 Testing

During the privacy-preserving classification phase, the ReLU function was replaced with a low-degree polynomial approximation and max pooling was replaced with sum pooling.

The privacy preserving CNN takes as input a PNG image file representing a handwritten digit from 0-9 and the weights file computed during the training phase. The privacy preserving CNN then encrypts the image, classifies the encrypted image, and decrypts the output of the final layer. This decrypted vector contains 10 values each associated with a digit from 0-9. Whichever value from 0-9 is associated with the highest value found in the vector is the classifiers prediction.

HElib is unable to operate on floating point values. As a result, the values within the weights/bias file had to be scaled appropriately. Scaling was done simply by multiplying the input value by some large integer value ranging from 1-512. Additionally, because operating over encrypted data takes a significant amount of time and memory, images were not classified in batches. Instead each image was individually processed by the privacy preserving classifier.

## 9.3 Test Environment

Initial attempts were made to run computations on the same test environment used for the privacy preserving logic gates. Unfortunately, this environment did not have enough memory to handle the privacy preserving CNN. As a result, tests were run on the Rochester Institute of Technology research computing cluster. This environment

provides 2304 cores and 24 TB RAM. The entire privacy preserving CNN application utilizes about 300000 MB/300 GB.

# Chapter 10

---

## Profiling Results

When assessing the practicality of any cryptosystem, two of the most important factors to take into consideration are accuracy and timing: does this correctly classify the image and how long does it take to classify the image? In the interest of exploring the capabilities of HELib, the privacy preserving CNN was tested under various conditions to observe the effects of different parameters on both accuracy and timing. For timing, the number of seconds it took to encrypt/decrypt the image and execute each layer was measured. For accuracy, normally the value is calculated by running the privacy preserving CNN over the entire test dataset, but because of limited resources a very basic test had to be implemented. Instead of testing all 10,000 images, a random image was selected from the testing dataset and run through the privacy preserving CNN. The entire privacy preserving CNN application utilizes about 300000 MB/300 GB.

### 10.1 Timing

During initial stages of testing, it was observed that the classification of one encrypted image can take up to three hours. In the interest of exploring where the classification may be spending most of its time, timing measurements for encryption/decryption and each layer were noted. The two tables below detail the time it took to read the image, encrypt the image, calculate the output of each layer, and decrypt the final

result. All timing values listed below were calculated with the thread capabilities disabled, scale set to 128, security parameter set to 80 bits, columns set to 3, and levels set to 11.

Table 10.1 illustrates the time it took for each layer to execute.

**Table 10.1:** Breakdown of Running Time for CNN Model

Layer	Time (seconds)
Convolution Layer (20 feature maps)	1176.8500
SumPool Layer	12.4079
Convolution Layer (50 feature maps)	6084.8600
SumPool Layer	2.3229
Flatten Layer	0.4046
Fully Connected Layer	1529.1300
ReLU Layer	23.3203
Fully Connected Layer	27.2122

From the timing results in Table 10.1, it can be seen that the convolution layers take the longest to calculate followed by the fully connected layers, the ReLU layer and the sum pool layers.

Recall that the convolution layer consists of taking the dot product between the previous layer and a 3-dimensional sliding filter(s) of weights. This process therefore involves a combination of homomorphic multiplication with a constant and homomorphic addition with a constant. The first convolution layer takes as input the encrypted image of dimension  $28 \times 28 \times 1$  and has 20 feature maps of size  $5 \times 5$  with a stride of 1. With this information, the number of dot product operations can be calculated as  $(20) \times (28 - (5 - 1)) \times (28 - (5 - 1)) \times (1) \times (5) \times (5) = 288000$ . Because this layer takes 1176.85 seconds, it can be inferred that one dot product computation takes roughly 0.00408 seconds. The second convolution layer takes as input the output of the first

sum pool layer with the dimension 12x12x20 and has 50 feature maps of size 5x5 with a stride of 1. With this information, the number of dot product operations can be calculated as  $(50) \cdot (12 - (5 - 1)) \cdot (12 - (5 - 1)) \cdot (20) \cdot (5) \cdot (5) = 1600000$ . Because this layer takes 6084.86 seconds, it can be inferred that one dot product computation takes roughly 0.00380 seconds, which is fairly consistent with the first convolution layer. It should be noted that the convolution layer also includes the addition of a bias vector following the dot product computation, but this has been omitted from the estimate as it is fairly negligible.

The fully connected layered also takes the dot product between an input layer and a 1-dimensional weights vector, but because the convolution layer consists of several 3-dimensional filters, it takes less time to compute. None-the-less, the fully connected layers take the second longest to compute after the convolution layer. The first fully connected layer connects 800 input neurons to 500 output neurons resulting in a total of 400000 dot product operations. Because this layer takes 1529.13 seconds, it can be inferred that one dot product computation takes roughly 0.00382 seconds. The second fully connected layer connects 500 input neurons to 10 output neurons resulting in a total of 5000 dot product operations. Because this layer takes 27.2122 seconds, it can be inferred that one dot product computation takes roughly 0.00544 seconds. Like the convolution layer, the fully connected layer also includes the addition of a bias vector following the dot product computation, but this has been omitted from the estimate as it is fairly negligible.

The ReLU layer is the most computationally intensive layer as it estimates the activation function with the following equation  $0.0012x^2 + 0.5x + 52$ . This computation involves both homomorphic multiplication and homomorphic addition. The ReLU layer takes as input the output of the first fully connected layer with the dimension of 500 neurons. Because this layer takes 23.3203 seconds, it can be inferred that one polynomial computation takes roughly 0.4664 seconds.

The sum pool layer consists of only homomorphic addition operations. The first sum pool layer takes as input the output of the first convolution layer with the dimension of  $24 \times 24 \times 20$  and has a pool size of  $2 \times 2$  with a stride of 2. With this information, the number of addition operations can be calculated as  $(24/2) * (24/2) * (20) * (2) * (2) = 11520$ . Because this layer takes 12.4079 seconds, it can be inferred that one addition computation takes roughly 0.001077 seconds. The first sum pool layer takes as input the output of the second convolution layer with the dimension of  $8 \times 8 \times 50$  and has a pool size of  $2 \times 2$  with a stride of 2. With this information, the number of addition operations can be calculated as  $(8/2) * (8/2) * (50) * (2) * (2) = 3200$ . Because this layer takes 2.32292 seconds, it can be inferred that one addition computation takes roughly 0.000726 seconds.

Looking at both convolution layers and fully connected layers, it takes about 0.0038-0.0054 seconds to perform one homomorphic dot product operation, where the multiplication is between a ciphertext and a constant and the addition is between two ciphertexts. Next, looking at the ReLU layer, it takes about 0.4664 seconds to compute one polynomial computation. Finally, looking at both sum pool layers, it takes about 0.0007-0.001 seconds to perform one homomorphic addition operation. These results are consistent with the idea that homomorphic multiplication is the most expensive operation as the layer that involves the multiplication between ciphertexts, ReLU layer, takes the longest per individual computation. Ultimately the convolution layers take the longest to calculate because of the sheer volume of computations.

Table 10.2 illustrates the time it took for reading the image, encrypting the image, and decrypting the image.



**Table 10.2:** Breakdown of Running Time for Encryption/Decryption

Operations	Time (seconds)
Read Image	0.0006
Encryption	16.7634
Decryption	0.2503

From the timing results in Table 10.2, it can be seen that the time it takes to read the image is quite small at 0.000595 seconds. Encryption takes 16.7634 seconds, so to encrypt one pixel it takes about  $16.7634/(28*28*1) = 0.2137$  seconds. Decryption takes 0.250251 seconds, so to decrypt one value it takes about  $0.250251/(10) = 0.025$  seconds.

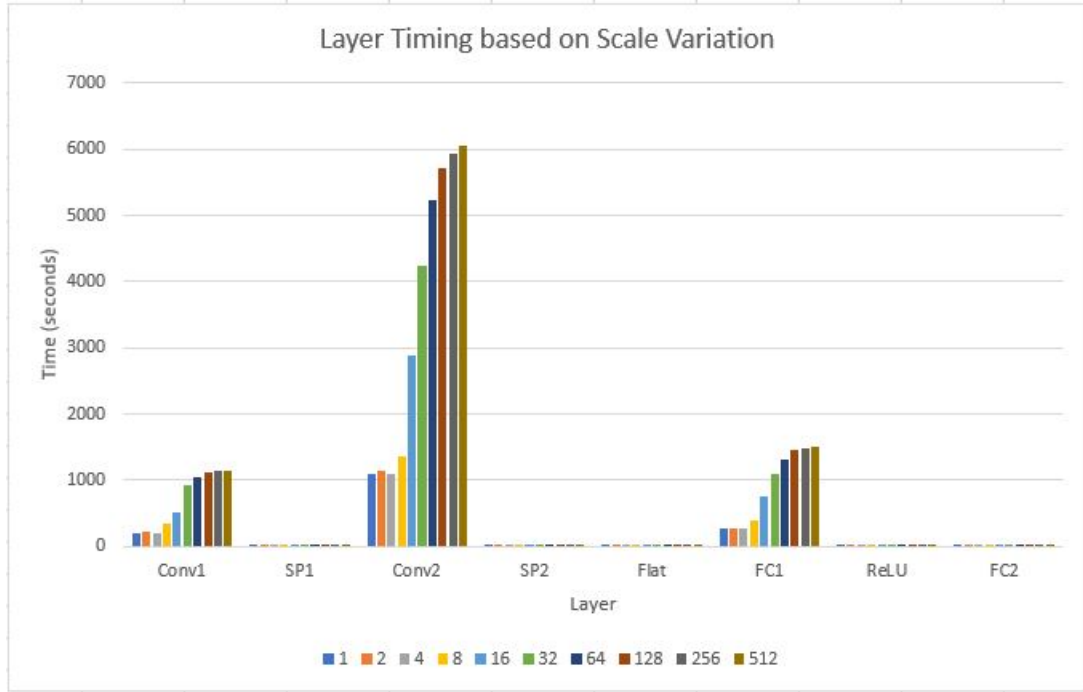
Based on the initial timing results seen in Table 10.1 and Table 10.2, it is clear that the bottleneck in computation is from the Convolution Layer and Fully Connected Layers.

## 10.2 Scale Variation

HElib is unable to operate on floating point values. As a result, the values within the weights/bias file had to be scaled appropriately. Scaling was done simply by multiplying the input value by some large integer value ranging from 1-512 (1-9 bits). In this section, the scale was varied to observe the speedup in overall computation time. All timing values were calculated with the thread capabilities disabled, security parameter set to 80 bits, columns set to 3, and levels set to 11. In addition, for the sake of consistency, the network was retrained every time the scale value was changed to accommodate for the parameter change. This means a different weights file was used for each encrypted classification.

Figure 10.1 illustrates the time it took for each layer to execute based on variation

in scale.

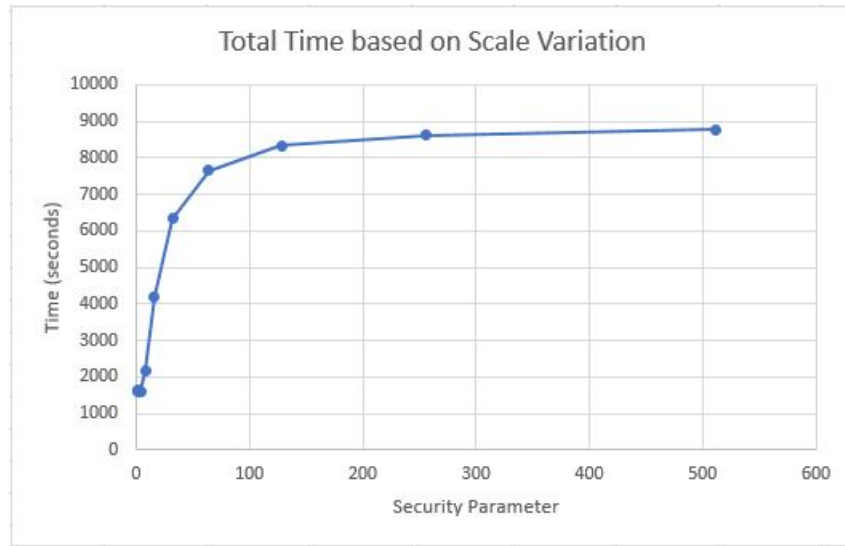


**Figure 10.1:** Timings for each Layer based on Scale Variation

From the results in Figure 10.1, as the scale grows so too does the amount of time it takes to evaluate each layer. Looking at the layers that take the longest to compute (Conv1/Conv2/FC1) from a scale of 1 (1 bit) to a scale of 512 (9 bits), there is a 5.5x increase in computation time.

Figure 10.2 illustrates the total time to execute the network based on variation in scale.

From the results in Figure 10.2, it can be seen that while the overall time does indeed increase as the scale increases, the growth is not linear. Instead it appears to rapidly grow from 1-8 (0-3 bits) and plateau around 256-512 (8-9 bits). That being said, the total time it takes with a scale of 1 (1 bit) is roughly 1600 seconds and the total time it takes with a scale of 512 (9 bits) is roughly 8800 seconds, which is a difference of 2 hours. This brings into question timing at the cost of accuracy, specifically how much accuracy one is willing to sacrifice for faster computation.



**Figure 10.2:** Total Time based on Scale Variation

Table 10.3 illustrates the time it took for reading the image, encrypting the image, decrypting the image, and if the image was correctly classified based on the scale. Recall that for accuracy, normally the value is calculated by running the privacy preserving CNN over the entire test dataset, but because of limited resources a very basic test had to be implemented. Instead of testing all 10,000 images, a random image was selected from the testing dataset and run through the privacy preserving CNN.

**Table 10.3:** Scale Variation

Scale	Correctly Predicted?	Read Image(s)	Encrypt(s)	Decrypt(s)
1	No	0.0006	15.6955	0.01749
2	No	0.0006	16.0526	0.0177
4	No	0.0006	16.0523	0.1160
8	No	0.0006	16.0651	0.2226
16	Yes	0.0006	16.2280	0.2229
32	Yes	0.0006	16.3002	0.2217
64	Yes	0.0006	16.1879	0.0040
128	Yes	0.0006	16.2058	0.2222
256	Yes	0.0006	16.1387	0.2221
512	Yes	0.0006	16.2236	0.2227

From the timing results in Table 10.3, it can be seen that scale variation does not have much effect on the time it takes to read the image or the time it takes to encrypt/decrypt the image. On the other hand, scale variation does appear to have an effect on accuracy. It is important to note that because the entire testing set could not be processed with the available computation power, the accuracy metric seen here is in no way indicative of how the network would perform over all of the images. That being said, for the few random images tested if the scale was set between 1-8 (0-3 bits), the classifier was unfortunately unable to correctly predict the value in the encrypted image. If the scale was set between 16-512 (4-9 bits), the classifier was able to correctly predict the value in the encrypted image. A potential explanation for this difference in predictive capability could be the loss of precision with the smaller scale values. Because the scaling was done simply by multiplying the input value with an integer, if the input floating point from the weights file was a value much smaller than 1, multiplying it by a scale value of 1-8 may not have been enough. Recall that

HElib operates only on integer values, thus if the scale did not round to a value larger than 0, the weight would be considered 0. A weight with the value 0 changes the forward propagation calculation, leading to a potential misclassification.

Something important to note is when utilizing the non privacy preserving layers, unencrypted classification yields the same results as encrypted classification. This is because the non privacy preserving layers were built with the same mathematical modifications as the privacy preserving layers. Although it was not tested, if the original non-modified CNN layers were utilized to classify unencrypted images, scaling may have had a similar effect on the accuracy results. This is because while the images were tested with the scaled values, they were not trained with the scaled values.

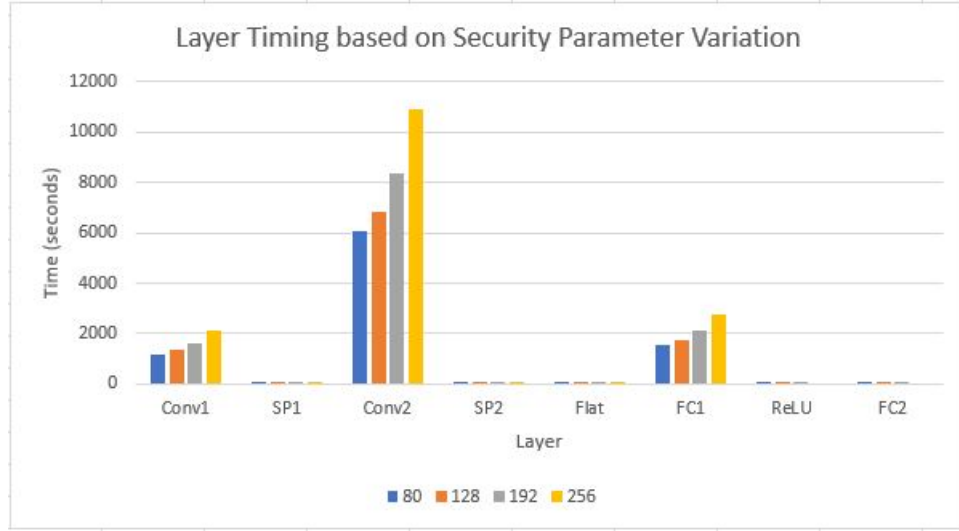
### 10.3 Security Parameter Variation

HElib has a few parameters that are essential when it comes to the security of the privacy preserving CNN. One such parameter is  $k/\lambda$  also known as the security parameter. The default value is  $k = 80$  and is the value that has been used for the other experiments. In this section, the security parameters were varied to observe the threshold for calculation and overall security. All timing values were calculated with the thread capabilities disabled, scale set to 128, columns set to 3, and levels set to 11.

For HElib, setting  $k = 128$  is considered equivalent to the security of AES-128, setting  $k = 192$  is considered equivalent to the security of AES-192, setting  $k = 256$  is considered equivalent to the security of AES-256. As a result, these are the three security parameters tested aside from the default  $k = 80$ .

Figure 10.3 illustrates the time it took for each layer to execute based on variation in security parameter.

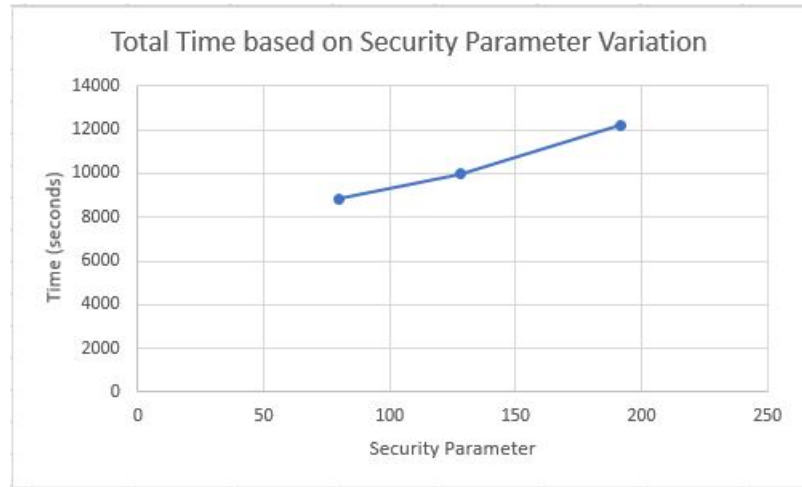
From the results in Figure 10.3, as the security parameter grows so too does the amount of time it takes to evaluate each layer. Looking at the layers that take the



**Figure 10.3:** Timings for each Layer based on Security Variation

longest to compute (Conv1/Conv2/FC1) from a security parameter of 80 bits to a security parameter of 256 bits, there is a 1.8x increase in computation time.

Figure 10.4 illustrates the total time to execute the network based on variation in security parameter.



**Figure 10.4:** Total Time based on Security Parameter Variation

From the results in Figure 10.4, it can be seen that the larger the security parameter, the larger the computation time. Based on the graph presented, the relationship between the security parameter value and the total time for classification appears to

be linear with a gradual slope. With a security parameter of 80 bits the total time it takes to classify an encrypted image is 8856.51 seconds, with a security parameter of 128 bits it takes 9977.31 seconds, and with a security parameter of 192 bits it takes 12209.61 seconds. It is important to note that the security parameter of 256 was unable to completely classify the encrypted image as the noise growth was too large and there were not enough levels to accommodate the security parameter. The time difference between using a security parameter of 80 bits vs a security parameter of 192 bits is roughly 40 minutes. This brings into question timing at the cost of security: are there scenarios where one would be willing to wait longer to ensure a higher level of security?

Table 10.4 illustrates the time it took for reading the image, encrypting the image, decrypting the image, and if the image was correctly classified based on the security parameter.

**Table 10.4:** Security Variation Timings Encrypt/Decrypt

Sec Param	FindM	Read Image(s)	Encryption(s)	Decryption(s)	Correctly Predicted?
128	11987	0.0006	19.9479	0.2279	Yes
192	15179	0.0005	24.5930	0.3600	Yes
256	18281	0.0053	31.0589	N/A	No

From the timing results in Table 10.4, it can be seen that, unlike scale variation, security parameter variation does have an effect on the time it takes to encrypt/decrypt the image. As the security parameter gets larger, so too does the amount of time it takes to encrypt the image. This difference in time is only a few seconds and therefore negligible in the grand scheme of things.

Security parameter variation also appears to have an effect on accuracy. While scale variation incorrectly classifies the encrypted image because of the bit precision of the weights, the security parameter of 256 bits incorrectly classifies because there

were not enough levels provided to support 256 bits of security. To solve this issue, the network was run again with the security parameter set to 256 bits and the levels set to 15.

## 10.4 Level Variation

Another important parameter found in HELib is  $L$  or the number of levels in the modulus chain. Levels are normally changed after ciphertext multiplication as this is the operation that generates the most noise. This means the level value is largely dependent on the evaluation function. There is no default value suggested for the number of levels, so  $L = 11$  is the value that has been used for the other experiments as it is the minimum value to successfully classify an encrypted image. In this section, the levels were varied to observe the threshold for calculation and overall timings. All timing values were calculated with the thread capabilities disabled, security parameter set to 80 bits, scale set to 128, and columns set to 3.

Figure 10.5 illustrates the time it took for each layer to execute based on variation in number of levels.

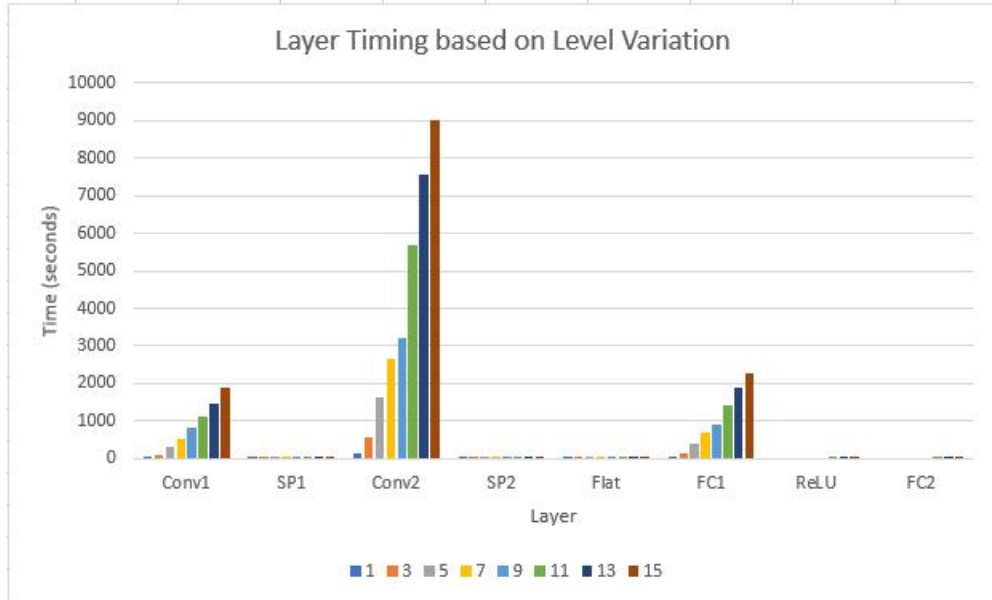
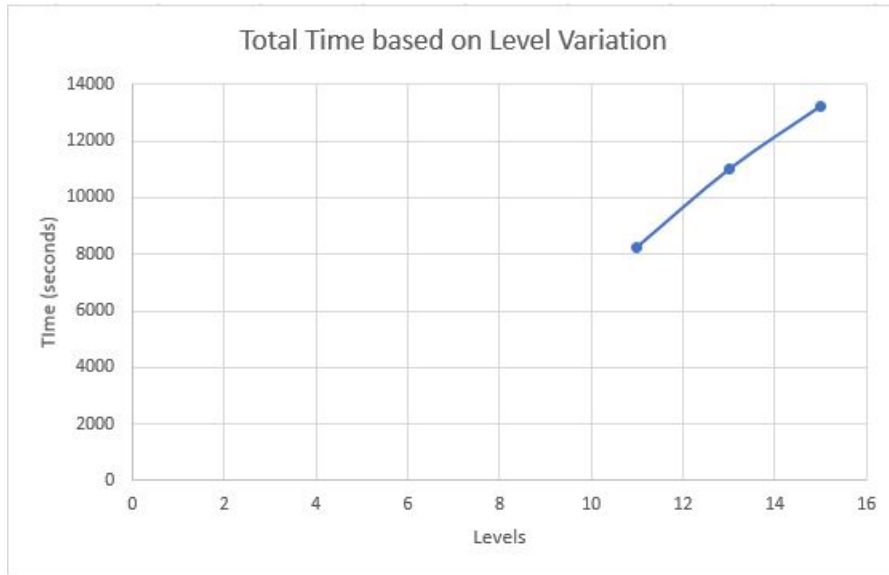


Figure 10.5: Timings for each Layer based on Level Variation



From the results in Figure 10.5, as the number of levels grow so too does the amount of time it takes to evaluate each layer. Looking at the layers that take the longest to compute (Conv1/Conv2/FC1) from number of levels set to 1 to number of levels set to 15, there is a 65x-70x increase in computation time.

Figure 10.6 illustrates the total time to execute the network based on variation in number of levels.



**Figure 10.6:** Total Time based on Level Variation

From the results in Figure 10.6, it can be seen that the greater the number of levels, the greater the computation time. Based on the graph presented, the relationship between the number of levels and the total time for classification appears to be linear with an average slope. It is important to note that if the number of levels was less than 11, the network was unable to completely classify the encrypted image as the noise growth was too large. The time difference between using 11 levels vs 15 levels is roughly 80 minutes. In addition, the slope in Figure 10.6 is much steeper than the slope in Figure 10.4. The for a faster computation time, it makes more sense to minimize the number of levels as much as possible over the security parameter.

Table 10.5 illustrates the time it took for reading the image, encrypting the image,

decrypting the image, and if the image was correctly classified based on the number of levels.

**Table 10.5:** Level Variation Timings Encrypt/Decrypt

Levels	Read Image(s)	Encryption(s)	Decryption(s)	Correctly Predicted?
1	0.0005	0.7648	N/A	No
3	0.0005	2.2317	N/A	No
5	0.0005	5.1763	N/A	No
7	0.0006	8.1466	N/A	No
9	0.0006	12.7384	N/A	No
11	0.0005	16.1540	0.2225	Yes
13	0.0006	21.0840	0.2870	Yes
15	0.0006	25.6875	0.2819	Yes

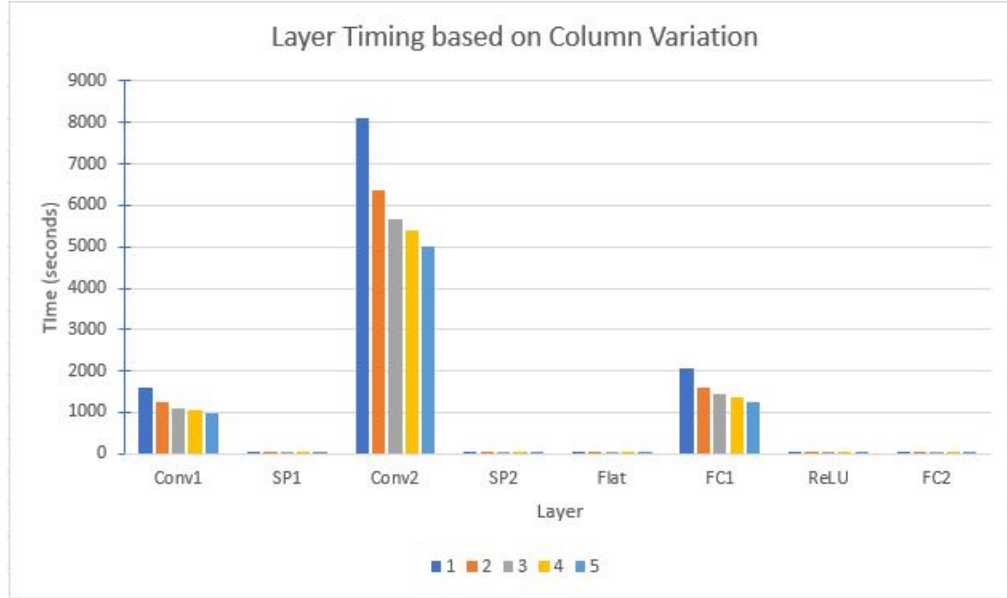
From the timing results in Table 10.5, it can be seen that, unlike scale variation, level variation does have an effect on the time it takes to encrypt/decrypt the image. As the number of levels increase, so too does the amount of time it takes to encrypt the image. This difference in time is only a few seconds and therefore negligible in the grand scheme of things.

Level variation also has effect on accuracy. There is clearly a minimum number of levels needed to successfully classify the encrypted image and manage noise growth. While there is no easy way to compute the necessary number of levels, generally number of levels corresponds with the number of multiplications in the evaluation circuit. The network tested in this experiment contains 4 dot products and a degree two polynomial calculation. Because of this, initial experiments were done with levels set to 6. When results showed an error message, guess and check was done to find the minimum number of levels. In this case, the minimum number of levels necessary for a successful classification is 11.

## 10.5 Column Variation

Another important parameter found in HELib is  $c$  or the number of columns in the key-switching matrix. The number of columns plays a key role in the relinerization process and can be used to manage noise growth. Like the number of levels, this value is also dependent on the multiplication depth of the evaluation function. The default value is  $c = 3$  and is the value that has been used for the other experiments. In this section, the columns were varied to observe the threshold for calculation and overall timings. All timing values were calculated with the thread capabilities disabled, security parameter set to 80 bits, scale set to 128, and levels set to 11.

Figure 10.6 illustrates the time it took for each layer to execute based on variation in number of columns.

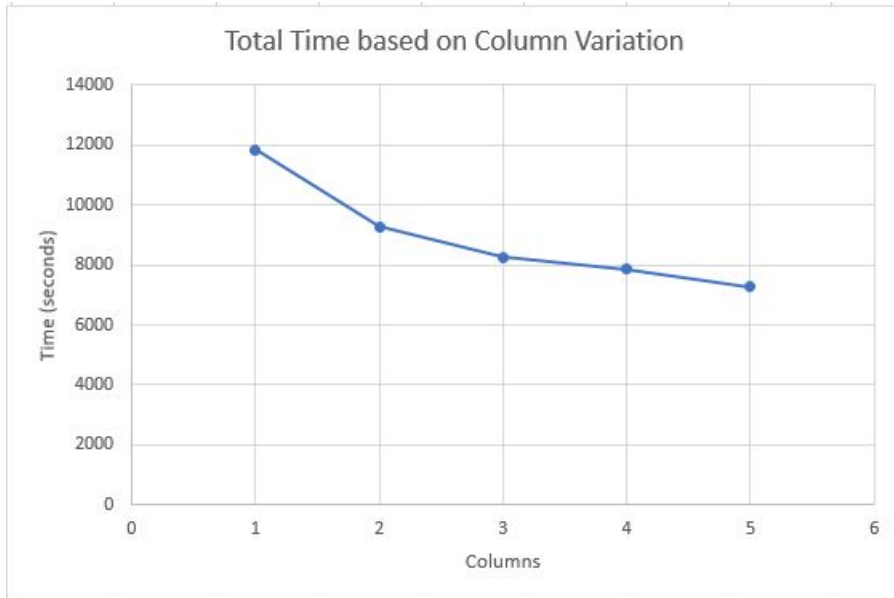


**Figure 10.7:** Timings for each Layer based on Column Variation

From the results in Figure 10.6, unlike the security parameter value and number of levels, as the number of columns grow the time it takes to evaluate each layer actually decreases. Looking at the layers that take the longest to compute (Conv1/Conv2/FC1) from number of columns set to 1 to number of columns set to

5, there is a 0.5x-0.6x decrease in computation time.

Figure 10.7 illustrates the total time to execute the network based on variation in number of columns.



**Figure 10.8:** Total Time based on Column Variation

From the results in Figure 10.7, it can be seen that the greater the number of columns, the less the computation time. Based on the graph presented, the relationship between the number of columns and the total time for classification appears to be linear with an average slope. The time difference between using 5 columns vs 1 column is roughly 76 minutes.

Table 10.6 illustrates the time it took for reading the image, encrypting the image, decrypting the image, and if the image was correctly classified based on the number of columns.

**Table 10.6:** Column Variation Timings Encrypt/Decrypt

Columns	Read Image(s)	Encryption(s)	Decryption(s)	Correctly Predicted?
1	0.0005	23.6943	0.3644	Yes
2	0.0005	18.6674	0.2673	Yes
3	0.0006	16.2449	0.2235	Yes
4	0.0006	15.3336	0.2153	Yes
5	0.0005	14.1503	0.1621	Yes

From the timing results in Table 10.6, it can be seen that column variation does have an effect on the time it takes to encrypt/decrypt the image. As the number of columns increase, the amount of time it takes to encrypt the image decreases. This difference in time is only a few seconds and therefore negligible in the grand scheme of things. Table 10.6 also shows that, at least in this case, column variation has no effect on accuracy. In the interest of reducing overall computation time, it makes sense to maximize the total number of columns.

## 10.6 Thread Variation

In addition to the basic functionality provided, HElib has an option for multithreading. In order to allow for this capability, HElib had to be rebuilt with `NTL THREADS=on`. In addition certain changes needed to be made in the layer design to allow for the NTL thread macro.

In this section, the thread count was varied to observe the speedup in overall computation time. All timing values were calculated with the security parameter set to 80 bits, scale set to 128, and levels set to 11 and columns set to 3.

Table 10.7 illustrates the time it took for each layer to execute based on variation in number of threads.

**Table 10.7:** Thread Variation Timings for each Layer

Threads	Conv1	SP1	Conv2	SP2	Flatten	FC1	ReLU	FC2
1	1176.8500	12.4079	6084.8600	2.3229	0.4046	1529.1300	23.3203	27.2122
4	343.8590	3.6240	1777.1200	0.3925	0.6780	446.5910	6.8110	7.9470
12	130.7450	1.3780	675.7100	0.4162	0.2580	169.8000	2.5890	3.0220
36	61.3830	0.6468	317.2350	0.4032	0.1210	79.7214	1.2160	1.4190

From the results in Figure 10.7, threading clearly helps with computation time. As the total number of threads increase, the computation time for each layer significantly decreases. In fact going from 1 thread to 36 threads has a speedup of 20x, more than any parameter variation provides.

Table 10.8 illustrates the time it took for reading the image, encrypting the image, decrypting the image, and if the image was correctly classified based on the number of threads.

**Table 10.8:** Thread Variation Timings Encrypt/Decrypt

Number of Threads	Read Image (s)	Encryption (s)	Decryption (s)
1	0.000572	16.1563	0.223557
4	0.000645	16.6239	0.249485
12	0.0006	16.2086	0.228941
36	0.000626	16.7332	0.264578

From the results in Figure 10.8, multithreading has no effect on the time it takes to read the image or encryption/decryption time. Regardless, the significant improvement in computation time makes threading extremely valuable in the process of encrypted classification. In the case that it is possible to enable multithreading, the scale value, security parameter value, number of levels, and number of columns can be

set to any value within reason. Of course, with regard to the scale value and number of levels, the accuracy of the evaluation circuit must be taken into consideration.

## 10.7 Fast Configuration

Based on the aforementioned results, a final test was done to gather timing for the fast configuration. Fast configuration refers to a combination of the parameters that had the best timing, while still maintaining a correct prediction. In this section timing values were calculated with the security parameter set to 80 bits, scale set to 128, levels set to 11, columns set to 5, and threads set to 36.

**Table 10.9:** Fast Configuration Timings for each Layer

Conv1	SP1	Conv2	SP2	Flatten	FC1	ReLU	FC2
51.128	0.445	249.716	0.085	0.020	62.957	0.746	1.139

Table 10.9 shows that the fast configuration does indeed achieve the best time for this network, with the timing for each layer reaching an all time low.

# Chapter 11

---

## Conclusion and Future Work

One significant limitation to this study was the available computing power, specifically number of cores and overall memory. Although it was eventually possible to access greater resources with the help of RIT research computing cluster, time constraints led to the creation of a privacy preserving CNN that, while successful, was only able to classify a small number of images from the MNIST dataset. With greater computing power and more time, not only will it be possible to classify the entire MNIST dataset, a larger privacy preserving classifier could also be implemented. Additionally, a larger dataset could be trained and tested. One such dataset is CIFAR10.

While this study did not generate results on overall classification accuracy, various studies indicate that the drawback of privacy preserving CNN is the loss in accuracy [1] [2] [3] [4]. Results show that when HE is integrated with Deep Learning, the classification accuracy is not comparable state-of the art classification accuracy. To improve classification accuracy, a future study could be done to explore the potential of a deep learning number system known as Universal Number (UNUM). This could be integrated with the implementation presented by Hesamifard et. al [4]. Similar to the floating point format, UNUM was proposed by John Gustafson as a replacement to the IEEE format. Type III UNUM or Posits perform well with regard to accuracy in the range near one. This quality makes Posits particularly useful in the realm of deep learning [42].



For even greater computation speedup, a future study could take advantage of other advanced functionalities HElib provides. One such functionality is Ciphertext packing or Single Instruction Multiple Data. Using this feature take advantage of a HElib feature that combines several messages into one ciphertext. This process would reduce the overall number of ciphertexts and speedup the computation time. Of course this would involve changing the structure of the inputs to each layer as well as the way each weights file is processed and stored.

Overall, this study proved to be a successful proof of concept with regards to encrypted image classification. Results showed that it is not only possible to utilize HElib alongside a CNN to create a privacy preserving classifier, it is possible to create various types of evaluation circuits as well. Although a significant speedup was achieved towards the end of experimentation with the help of multithreading, in some cases multithreading is not possible. If multithreading is not possible, then a simple change in scale value, security parameter value, number of levels, and number of columns can have an effect on the overall computation time of the privacy preserving CNN. Some parameters, such as number of levels and security parameter value, have a greater effect on timing while others, such as number of levels and scale value, have a distinct effect on accuracy. It is thus extremely important to note that when selecting parameters in any HElib evaluation circuit, each value plays a significant role with regards to computation time and overall accuracy.

## Bibliography

---

- [1] Q. Zhang, L. T. Yang, and Z. Chen, “Privacy preserving deep computation model on cloud for big data feature learning,” *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1351–1362, May 2016.
- [2] N. Dowlin, R. Gilad-Bachrach, K. L. Sin, K. Lauter, M. Maehrige, and J. Wernsing, *CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy*, February 2016. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/cryptonets-applying-neural-networks-to-encrypted-data-with-high-throughput-and-accuracy/>
- [3] H. Chabanne, A. de Wargny, J. Milgram, C. Morel, and E. Prouff, “Privacy-preserving classification on deep neural network,” *IACR Cryptology ePrint Archive*, vol. 2017, p. 35, 2017.
- [4] E. Hesamifard, H. Takabi, and M. Ghasemi, “Cryptodl: Deep neural networks over encrypted data,” *CoRR*, vol. abs/1711.05189, 2017. [Online]. Available: <http://arxiv.org/abs/1711.05189>
- [5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “(leveled) fully homomorphic encryption without bootstrapping,” *ACM Trans. Comput. Theory*, vol. 6, no. 3, pp. 13:1–13:36, Jul. 2014. [Online]. Available: <http://doi.acm.org/10.1145/2633600>
- [6] V. Costan and S. Devadas, “Intel sgx explained,” 2016, <https://eprint.iacr.org/2016/086>.
- [7] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, “Cache attacks on intel sgx,” in *Proceedings of the 10th European Workshop on Systems Security*, ser. EuroSec’17. New York, NY, USA: ACM, 2017, pp. 2:1–2:6. [Online]. Available: <http://doi.acm.org/10.1145/3065913.3065915>
- [8] S. Samet and A. Miri, “Privacy-preserving classification and clustering using secure multi-party computation,” in *Privacy-Preserving Classification and Clustering Using Secure Multi-Party Computation*, 2008.
- [9] J. Colarossi and C. Soler. What is secure multiparty computation? [Online]. Available: <https://www.bu.edu/research/articles/secure-multiparty-computation/>
- [10] L. Barthelemy. A brief survey of fully homomorphic encryption, computing on encrypted data. [Online]. Available: <https://blog.quarkslab.com/a-brief-survey-of-fully-homomorphic-encryption-computing-on-encrypted-data.html>

- [11] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, “A survey on homomorphic encryption schemes: Theory and implementation,” *ACM Comput. Surv.*, vol. 51, no. 4, pp. 79:1–79:35, Jul. 2018. [Online]. Available: <http://doi.acm.org/10.1145/3214303>
- [12] C. Gentry, “A fully homomorphic encryption scheme,” Ph.D. dissertation, Stanford University, 2009.
- [13] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, “Fully homomorphic encryption over the integers,” in *Proceedings of the 29th Annual International Conference on Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 24–43. [Online]. Available: [http://dx.doi.org/10.1007/978-3-642-13190-5\\_2](http://dx.doi.org/10.1007/978-3-642-13190-5_2)
- [14] O. Davydova. A beginner’s guide to understanding convolutional neural networks. [Online]. Available: <https://medium.com/@datamonsters/artificial-neural-networks-for-natural-language-processing-part-1-64ca9ebfa3b2>
- [15] A. Deshpande. A beginner’s guide to understanding convolutional neural networks. [Online]. Available: <https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/>
- [16] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser, “Machine learning classification over encrypted data,” *IACR Cryptology ePrint Archive*, vol. 2014, p. 331, 2014. [Online]. Available: <http://eprint.iacr.org/2014/331>
- [17] V. Vaikuntanathan. The mathematics of lattices i. [Online]. Available: [https://www.youtube.com/watch?v=LlPXfy6bKIY&list=PLgKuh-lKre139cwM0pjuxMa\\_YVzMeCiTf](https://www.youtube.com/watch?v=LlPXfy6bKIY&list=PLgKuh-lKre139cwM0pjuxMa_YVzMeCiTf)
- [18] C. Manwani. Mathematics—partial orders and lattices. [Online]. Available: <https://www.geeksforgeeks.org/mathematics-partial-orders-lattices/>
- [19] Lattice of integer divisors of 60, ordered by ”divides”. [Online]. Available: [https://en.wikipedia.org/wiki/Lattice\\_\(order\)#/media/File:Lattice\\_of\\_the\\_divisibility\\_of\\_60.svg](https://en.wikipedia.org/wiki/Lattice_(order)#/media/File:Lattice_of_the_divisibility_of_60.svg)
- [20] T. Gupta. Mathematics—rings, integral domains and fields. [Online]. Available: <https://www.geeksforgeeks.org/mathematics-rings-integral-domains-and-fields/>
- [21] M. Harrison and K. Harrison. Abstract algebra: The definition of a ring. [Online]. Available: <https://www.youtube.com/watch?v=6RC70C9FNXI>
- [22] L. Barthelemy. The brief survey of fully homomorphic encryption, computing on encrypted data. [Online]. Available: <https://blog.quarkslab.com/a-brief-survey-of-fully-homomorphic-encryption-computing-on-encrypted-data.html>

- [23] V. Vaikuntanathan. The mathematics of lattices ii. [Online]. Available: [https://www.youtube.com/watch?v=SZkTJMOrxnM&list=PLgKuh-lKre139cwM0pjuxMa\\_YVzMeCiTf&index=2](https://www.youtube.com/watch?v=SZkTJMOrxnM&list=PLgKuh-lKre139cwM0pjuxMa_YVzMeCiTf&index=2)
- [24] S. S. Sathya, P. Vepakomma, R. Raskar, R. Ramachandra, and S. Bhattacharya, "A review of homomorphic encryption libraries for secure computation," *CoRR*, vol. abs/1812.02428, 2018.
- [25] T. Lepoint and M. Naehrig, "A comparison of the homomorphic encryption schemes fv and yashe," vol. 8469, 05 2014.
- [26] K. Hariss, M. Chamoun, and A. E. Samhat, "On dghv and bgv fully homomorphic encryption schemes," in *2017 1st Cyber Security in Networking Conference (CSNet)*, Oct 2017, pp. 1–9.
- [27] H. Chen, K. Laine, and R. Player, "Simple encrypted arithmetic library - seal v2.1," 04 2017, pp. 3–18.
- [28] M. S. H. Cruz. 2+3 using helib. [Online]. Available: <https://mshcruz.wordpress.com/2016/06/17/2-3-using-helib/>
- [29] S. Halevi and V. Shoup, "Algorithms in helib," 08 2014.
- [30] H. P. Blog. The bgv scheme. [Online]. Available: <http://heat-h2020-project.blogspot.com/2015/04/the-bgv-scheme.html>
- [31] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *Electronic Colloquium on Computational Complexity (ECCC)*, vol. 18, p. 111, 01 2011.
- [32] M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, J. Hoffstein, K. Lauter, S. Lokam, D. Moddy, T. Morrison, A. Sahai, and V. Vaikuntanathan, *Security of Homomorphic Encryption*, July 2017. [Online]. Available: [http://homomorphicencryption.org/white-papers/security\\_homomorphic\\_encryption\\_white\\_paper.pdf](http://homomorphicencryption.org/white-papers/security_homomorphic_encryption_white_paper.pdf)
- [33] T. M. DuBuisson. Secure computation with helib. [Online]. Available: <https://tommd.github.io/posts/HELib-Intro.html>
- [34] W. Lu. Ring-learning with errors and helib. [Online]. Available: <https://www.slideshare.net/ssuser4c5f79/h-elib>
- [35] K. Ujjwal. An intuitive explanation of convolutional neural networks. [Online]. Available: <https://ujjwalkarn.me/2016/08/11/intuitiv>
- [36] R. B. Zadeh and B. Ramsundar. Fully connected deep neural networks. [Online]. Available: <https://www.oreilly.com/library/view/tensorflow-for-deep/9781491980446/ch04.html#ch4-fclayer>

- [37] D. Cornelisse. An intuitive guide to convolutional neural networks. [Online]. Available: <https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/>
- [38] S. Sharma. Activation functions in neural networks. [Online]. Available: <https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>
- [39] P. VV. How to teach logic to your neural networks. [Online]. Available: <https://medium.com/autonomous-agents/how-to-teach-logic-to-your-neuralnetworks-116215c71a49>
- [40] F. Camillo. Neural representation of logic gates. [Online]. Available: <https://towardsdatascience.com/neural-representation-of-logic-gates-df044ec922bc>
- [41] J. Brownlee. Gentle introduction to the adam optimization algorithm for deep learning. [Online]. Available: <https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/>
- [42] Gustafson and Yonemoto, “Beating floating point at its own game: Posit arithmetic,” *Supercomput. Front. Innov.: Int. J.*, vol. 4, no. 2, pp. 71–86, Jun. 2017. [Online]. Available: <https://doi.org/10.14529/jsfi170206>