A Platform for Practical Homomorphic Encryption in Neural Network Classification

Mehmood Baryalai

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

Doctor of Philosophy (Ph.D.) in Information Technology,

Massey University, 2021.

Abstract

Convolutional neural networks (CNN) have become remarkably better in correctly identifying and classifying objects. By using CNN, numerous online services now exist that processes our data to provide meaningful insight and value-added services. Not all services are reliable and trustworthy due to which privacy concerns exist. To address the issue, the work presented in this research develops and optimise new techniques to use Homomorphic Encryption (HE) as a solution. Researchers have proposed solutions like the CryptoNets [1], Gazelle [2], and CryptoDL [3]. However, homomorphic encryption is yet to see the limelight for real-world adoption, especially in neural networks. These proposed solutions are seen as a solution only for a particular CNN model and lack generality to be extended to a different CNN model. Moreover, the solutions for HE-CNN integration are seen as unprepared for adoption in a practical and real-world environment. Additionally, the complex integration of hybrid approaches limits their utilization with privacy-preserving based CNN models. For that reason, this research develops the mathematical and practical knowledge required to adopt HE within a CNN. This knowledge of performing encrypted classification for a CNN model is based on a careful selection of appropriate encryption parameters. Furthermore, this study succeeds in developing a dual-cloud system to mitigate many of the technical hurdles for evaluating an encrypted neural network without compromising privacy. Moreover, in the case of a single cloud, this study develops methods for overcoming technical issues in selecting encryption parameters for, and evaluating, a convolutional neural network. In the same context, the novel method of selecting and optimizing encryption parameters based on probability is given. The proposals and the knowledge from this research can aid and advance the strategies of HE-CNN integrations in an efficient and easy way.

In loving memory of my father

Acknowledgements

All the praises and thanks to Allah for blessing me with the potential and ability to complete this manuscript.

I wish to express my profound gratitude and appreciation to my supervisors and my research team of Professor Dianne Brunton, Associate Prof. Winston Sweatman, Dr. Dongxi Liu, and Associate Prof. Julian Jang-Jaccard. Thank you for sharing your wealth of knowledge with me and teaching me scientific methods and research. This PhD dissertation would not have been possible without you. Here I would like to thank all my teachers who taught me in school, college, and university in Pakistan and New Zealand. All of you have helped me to become what I wanted to be.

I would like to thank my fellow Postgrads at Massey University, especially Fahim Zaidi, Ashar Malik, Sibghat Ullah, Rahila Umar, Hooman Alavizadeh, Amir Bashir, Saad Aslam for the invaluable discussions and disruptions.

I would like to thank the Higher Education Commission (HEC) of Pakistan and the Balochistan University of Information Technology, Engineering, and Management Sciences (BUITEMS) for their generous financial assistance as well as the experiences they have allowed me to share in.

Last but not the least; I would like to give a big thank you to my family. I am forever grateful to you all. Especially my mother, my uncles, and my siblings. I owe a very special thank you to my wife for her numerous sacrifices due to my PhD – this thesis would not have been completed without your patience and support.

My apologies to anyone whom I failed to mention.

Contents

List of Figures

List of Tables

Nomenclature

evk	Evaluation key
pk	Public
sk	Secret/Private key
$\lfloor x \rceil$	Represents the scaling down of x to the nearest integer
ω	Weight Matrix. Individual weight is identified by a subscript
i / o	Inputs / Outputs
z	Number of values
d	Input depth (number of channels)
w / h	Width and height of a matrix
w_o / h_o	Output matrix width and height
c / ct	Ciphertext
p	Plaintext
F	Filter
F_z	Number of filters
$F_w \times F_\mathrm{h}$	width and height of a convolutional filter
F_d	The depth of the filter in a 3D conv layer.
X / Y	Input / output
α	Activation functions

L	Total layers
l	Individual layer
z_o	Number of neurons
k	k is the ending number of a counter as in $\sum_{g=1}^{k} C$
v	Inherent Noise Polynomial
v_o	Output inherent noise polynomial after an operation
b	Bias value
K	Target size in relinearization
q	Coefficient modulus
t	Plaintext modulus
n	Degree of polynomial modulus
m	Original message m
s	Random term for security
j	Size of the polynomial representing ciphertext
N_m	the highest degree of the polynomial representing message m
$\|m\|$	The absolute biggest coefficient number for our input message m. *Or in other words:* The maximum of the absolute values of the coefficients of m.

1 Introduction

The internet is an invaluable facility to share data and information. However, the widespread data-breaches and privacy violations have worried businesses as well as the average person discouraging the use of the full potential of internet. Among the potential utilizations, intelligent data classification is progressively increasing. Such intelligent data classifications are made on data that is already stored on the internet or uploaded temporarily for the sake of classification. There are instances where the data items that need to be uploaded may contain sensitive and private information. Due to this sensitive nature, such data items are kept back, and their classification is avoided altogether from the fear of data-breaches and privacy violations.

On the internet, entities can store and host their data in their own datacentres or on a third-party server. Maintaining private datacentres are expensive for many businesses and individuals because it requires financial, hardware, software, and human resources. Therefore, instead of owning a private datacentre, data can be stored and hosted on third-party datacentres after encryption, for which relatively small amounts of rent will be paid. In the same way, if the data is hosted in a private datacentre, it still needs to be encrypted due to security concerns.

In both the cases of storing and hosting our encrypted data on a third party or our private premises, it needs to be processed later at some point. In addition, since the data is encrypted now, we have either to fully decrypt it (either on third party or on our private premises) or use an intelligent way to process without decryption. The intelligent way of processing without decryption is to use the homomorphic

encryption schemes. These schemes become necessary to use in scenarios dealing with sensitive information as discussed in the later section.

1.1 Research Goals

In general, the aim of this research is to improve the state of privacy for online computation. In particular, this thesis serves as a bridge between two domains of knowledge: *artificial neural networks*; and the cryptographic schemes named *homomorphic encryptions*. The central inquiry of this study is to find out and answer the technical issues in adopting homomorphic encryption within neural networks. As a result, solutions will be proposed to make the integration where the artificial neural networks and the homomorphic encryptions work at their best. The steps are as follows:

- Evaluate the state-of-the-art implementations for homomorphic encryption. This involves the study of the weaknesses and strengths of important homomorphic encryption parameters. This incorporates security provision of the implemented scheme, key and ciphertext sizes, complexity of operations, and the processing speed.
- Find the parameters of current homomorphic encryption implementations that pose problems for adoption in applications like neural networks.
- Find the optimum set of encryption parameters for evaluating an encrypted neural network.
- Study the optimisation opportunities between homomorphic encryption and neural networks.

1.2 Thesis Structure

The rest of the dissertation is organized as follows. Chapter 2 presents the background knowledge required in later sections. Chapter 3 presents our system of

a non-colluding dual-cloud model to process a classification task on a neural network. Chapter 4 presents a method for selecting homomorphic encryption parameters for a convolutional neural network. Chapter 5 presents practical case studies and shows how a convolutional neural network can be evaluated homomorphically by going through the important aspects. This chapter uses our parameter selection method for the case studies as practical examples. Chapter 6 presents our research on the probabilistic determination of noise and selecting optimised parameters for practical reasons. Chapter 7 presents conclusions and suggests future work.

1.3 Original Contributions

1.3.1 Proposed a robust non-colluding dual cloud system

We begin by developing a robust non-colluding dual cloud system for encrypted neural network inference. This system is able to perform operations based on a Paillier cryptosystem on one of the cloud systems, and only single plain operations on the other cloud system. A number of protection mechanisms are used to ensure that neither of the clouds learns anything of significance, consequently privacy is ensured. The dual cloud architectural design allows us to perform any sequence of complex operations on the data, including nonlinear functions. Moreover, our system works without the need for a continuous client connection, which is another advantage over similar models. We evaluate the feasibility by implementing and testing on a LAN connection for different encryption key lengths.

1.3.2 Developed generic formulas to estimate homomorphic parameters for a CNN

We developed generic and new mathematical formulas to find the exact circuit depth for any ciphertext processed through an encrypted convolutional neural network. Based on the new formulas, a fresh method is formulated for automatic estimation of encryption parameters for an encrypted feed-forward convolutional neural network. Our parameter estimation platform is better in performance than the standard estimation of SEAL. Moreover, we are the first to show the concrete effects of varying the parameters of a convolutional neural network on the parameters of homomorphic encryption. This helps in designing better-encrypted convolutional neural networks.

1.3.3 Novel knowledge presented using case studies

The proper usage of convolution and pooling layer types in an encrypted convolutional neural network is shown by using case studies. To facilitate practical adoption, distinct optimisation techniques and insights for encrypted convolutional neural networks are given by using case studies.

1.3.4 Proposed a novel approach of approximating and optimising encryption parameters probabilistically

We propose a novel method to approximate and optimise the SEAL encryption parameters probabilistically. New mathematical formulas are created for analysing noise of a ciphertext as a random variable in applications like the convolutional neural network. According to our knowledge, we are the first to use probabilistic optimisation and approximation of homomorphic encryption parameters in a practical application i.e., convolutional neural network. This method can be an extremely effective tool for homomorphic encryption in applied settings.

2 Preliminaries

2.1 Areas Demanding Privacy-Preserving Classification

To stress the need for adopting a privacy-preserving classification mechanism, Armknecht et al. [4] provide convincing cases. Here, we also highlight the following important areas in which we need privacy-preserving classifications:

Forensic Image Classification: There are cases where we need to know the existence of certain objects or people in an image. This is done using object or face-recognition where neural networks match the required data points from the image, such as facial elements with people in an existing database. Here, we need a privacy-preserving approach to detect faces and then match the facial elements with entries in our databases.

Network Traffic Analysis: In classifying malware network traffic, a combination of techniques including neural networks are used. In scenarios where the underlying data is of a private nature, then we need a privacy preserving neural network that can perform deep packet inspection without compromising the privacy of legitimate users as well as classify malicious data.

Medical Data: When dealing with medical data, we must grant doctors or hospitals access to our personal health data. This data can be in the form of medical imagery like X-rays or CT scans, or it can be the records of our medical prescriptions. The combination of all this data is already used by Google's DeepMind project [5] to assist doctors in curing their patients. However, it is also necessary for personal data to be private. Here, privacy-preserving techniques can be used to perform the classifications in the neural network.

Financial Data: For a financial institute or a business, it is desirable to analyse past and current market trends and predict future trends. This becomes very accurate when their internal data can be consolidated and compared with that of other businesses in the market. However, the private data of other financial institutes is never available. This suggests that privacy-preserving technologies are implemented in analysing market trends, where institutes share their private data in encrypted form, for the sake of a more accurate analysis overall.

2.2 Convolutional Neural Networks

The convolutional neural network (CNN) is a class of powerful and complex neural network architecture that has demonstrated par-human accuracy in image classification tasks. CNN is fundamentally a feed-forward neural network, which has proven to be extremely effective in the task of image analysis and classifications. A distinction is made between the non-encrypted execution of a CNN and an encrypted CNN by calling the former a "conventional CNN".

A neural network, in general, is essentially a computational structure that is extremely good in finding the association of an unknown observation of data to already known patterns [6]. The reader may think of identifying a face from an image in which instead of comparing the pixel values for equality with a known image, the neural network matches the pattern in the pixel values with an already analysed pattern from previously seen images of the same face. These (artificial) neural networks are originally inspired by the way biological neural cells operate in our nervous system. Several abstractions from the biological neural cells have been adopted into the present computational neural networks. These shall be discussed.

Neural networks are a combination of interconnected neurons combining input values and, possibly, processing through one of the non-linear functions f, then transmitting the output to the next neuron.

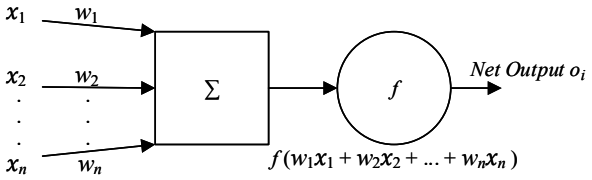


Figure 2-1 Abstraction of a single neuron.

The abstraction of a single neuron is shown in Figure 2-1. This contains n inputs shown by x_i. Each of these input connections to the neuron has an associated weight value shown by w_i. The incoming input x_i is multiplied by the corresponding weight w_i. Both values are expected to be real values. The incoming values, obtained after multiplication, are integrated at the neuron, usually just by adding them all, so that a net input is obtained. The net input of the neuron, possibly, then passed to a non-linear activation function f, generates an output o_i. Details of this activation function will be discussed later.

Based on the primitive neural unit, many models for neural networks exist. Modern networks can comprise of hundreds of layers each comprising of hundreds of neurons. Figure 2-2 shows the topology of a typical neural network where we have layer-wise: 3 inputs (x_1, x_2 x_3), 4 hidden neurons, and 3 output neurons. This network can be written as 3-4-3 due to the number of units in each layer.

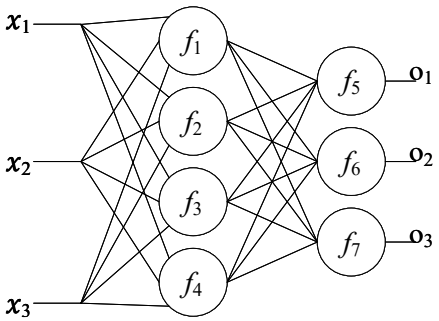


Figure 2-2 An example of a typical 3-4-3 model of an artificial neural network

Neural networks in existence have numerous models. Their differences, as argued by Chang and Lu [6], stem out from three areas that are important for any neural network. They are as follows:

- structure of neurons.
- topology of the network; and
- the learning algorithm used to find the weights of the network.

Structure of neurons: The variations in the structure of a single neuron are primarily due to the activation function. The most common activation functions available in the literature are discussed in section 2.2.6 below:

Topology of the network: After varying in the structure of neurons, neural network models can have further variations based on the topology in terms of the arrangement and interconnections of neurons. The structure of the network comprises various combinations of layers having numerous neurons. The greater the number of layers, the better the network can classify to a range of output classes. In the same way, the greater the number of neurons in the layers, the more accurately the network can classify to classes. In deep neural networks, the classification accuracy is relatively higher due to the increased number of layers as compared to the number of neurons in each layer.

Normally all the number m of neurons from a layer are connected to all the n neurons in the next layer. This will result in $m \cdot n$ interconnections, each having an associated weight w. While classifying, the propagation of values in a network is in the forward direction of the outputs. However, there are exceptions for some networks having interconnections back either into the same neuron or to a different neuron in the previous or current layer.

Learning algorithm used in the network: In a classification task, the flow of values from inputs towards outputs is the forward direction. This is called a forward

propagation. The forward propagation results in the classification inference as an output of the indicated class for the provided inputs. Whereas, before the classification stage, the training phase is necessary to tell the network about the expected output for each kind of input values. The resulting classification of the unknown data record to a known pattern is achieved by training it to already known pieces of data. The process of training is often a challenging task because it involves tweaking internal values of the network to match correct output results. To achieve this, a known data record is provided as an input to the network and the outputs are matched with that of the expected output. Matching values of the expected and the actual output provides the difference as an error measurement value of the network. This error value is propagated in the back direction and the weight values of each input link to each neuron is adjusted according to this error value. Such a process is called the backpropagation training technique. The whole training process is performed by cycling through many records with known patterns. The number and quality of the records in the dataset is one of several factors that affects the accuracy of the whole network.

Learning dataset used for the network: In the training phase of the network, many records of known patterns are processed through the backpropagation technique. The higher the number of records in the training dataset, the more accurate the network becomes in classifying unknown patterns. Similarly, the more versatile the dataset records are, the more confident the network becomes in classifying to different classes.

Figure 2-3 A sample of MNIST dataset images

The main dataset used for training the neural networks in this research was the Mixed National Institute of Standards and Technology (MNIST) dataset [7] shown in Figure 2-3. This selection is based on several reasons. The prime reason of this selection is the comparison of our results with that of a leading paper in our field (i.e., CryptoNets [1]) as they have also used the same dataset. Another reason of using the MNIST dataset is its widespread acceptance as a standard benchmark for classification and computer vision tasks among the researchers. A comprehensive set of machine learning literature is available that use the MNIST dataset which makes it a good choice for research and reference purposes. In this dataset, there are grayscale images of hand-written ten numerals that are 28×28 pixels in dimension. Each pixel can have a value from 0 to 255. Overall, the dataset contains a total of 70,000 images in which 60,000 are for training a neural network and 10,000 are for validating the training of a network.

2.2.1 The composition of a CNN

The processing of a CNN involves a sequence of computations arranged in linear and non-linear transformation layers. To classify an image, a CNN performs computations on its pixel values and outputs one or more numbers representing the statistical equivalence to a known category. The sequence of linear and non-linear layers of computations provide the output of a CNN in the form of potential classes

to which the input image may belong. An example convolutional neural network is shown in Figure 2-4.

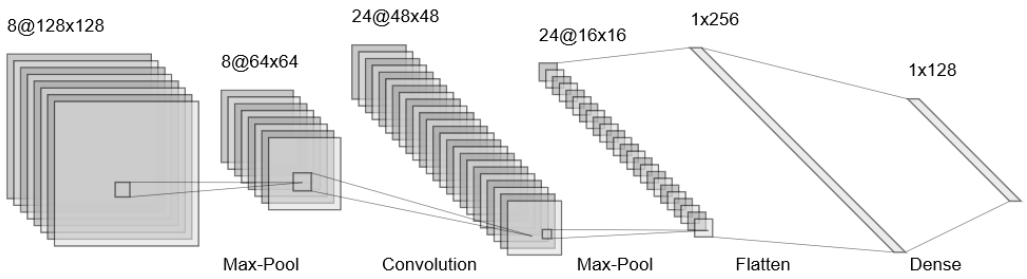


Figure 2-4 Overall architecture of Convolutional Neural Networks

In addition, a hyperparameter is a variable parameter of the network that determines the network structure or the training process. The value of hyperparameters are fixed before the training process begins, for example, the number of convolution layers, or the detail of the filters and neurons.

2.2.2 Linear Layers

In all the artificial neural networks including CNNs, the perceptron is the basic processing unit transforming inputs (x_i) to one output (y). For each input element x_i, there is a corresponding trainable weighted value ω_i. For each perceptron, there is a trainable bias value represented as b. In CNN, a linear layer is a layer of trainable perceptrons that acts on the input elements x by computing $y = \omega \cdot x + b$.

In the context of multi-dimensional inputs, for example an image, there are two kinds of linear layers: the convolutional (conv) and the fully connected (FC). These two kinds of linear layers are based on the arrangement and number of connections from the input elements to the perceptron.

2.2.3 Convolution Layer

The convolution layer reduces the dimensionality of input image data. The dimensions of an input image represented by (w_i,$h_i d_i$) having w_i as width, h_i as height, and d_i as the depth (channels) of the input image i. The convolution layer has trainable perceptrons that in the context of this layer are called filters F. For the classification to be performed in an efficient way, the convolution layers find and convey only the interesting patterns from the original input. Identifying only the interesting patterns is performed by using the filters. After training, these filters look through all the values in an incoming input, and allow only the interesting patterns to the next layer. In this way, the convolution layers eliminate the uninteresting parts of the input, and thus, reduce the dimensionality of the input image by preserving the important patterns for the later layers.

In Figure 2-4, for example, the layers of square blocks show the data matrices. These matrices are passed from left-to-right after processing. Figure 2-4 shows that the convolution-processing layer transforms multi-dimensional input (w, h, d) of $64 \times 64 \times 8$ to the reduced dimensions of $48 \times 48 \times 24$. From the figure, we see the reduction of dimensionality for inputs to outputs by using the convolution processing layers. The data matrices gradually flatten to an array of a single dimension that is then passed on to the fully connected network.

Convolution layers, essentially, are an attempt to extract important features from input images that will help in classification. In the process of extracting only the important features, they discard irrelevant information and thus reduce the dimensionality. The extraction of features is performed using a number of perceptrons arranged in layers called the filters f. The filters process input value matrices by taking a sub-section (represented as $f_w \times f_h \times f_i$) from the incoming input values. Therefore, filters have a defined width and height called the size of

the filter. The width and height restrict the operation of a filter to a specific area at each step, however the depth remains equal to the depth of the input. When processing, we convolve the filter over the pixel values in the image and calculate the dot product of the filter values and related values in the neighbourhood of the pixel.

Each of the filters f also has a bias value b which is added to the outputs after calculating the dot product. The convolution layer has an output of $w_o \times h_o \times d_0$ where the output depth d_o is equal to the number of filters f_i.

The input image is sometimes padded with zeros on the edges to provide a bigger output image size. The term "Same" padding is used to describe edge padding with zeros. Same padding is used to provide the same size of output as the input. Whereas "valid" padding is when there is no input padding used. The width and height of the output in valid padding is calculated as:

$$w_o = w_i - f_w + 1$$
$$h_o = h_i - f_h + 1$$

Sometimes the granularity of convolving the filter over the pixels is specified by a stride size represented here as (s_w,s_h). The stride size specifies how big the filter steps should be across the width and height. When we specify the stride size with valid padding, then the width and height of the output image will be represented as:

$$w_o = \left\lfloor \frac{w_i - f_w + 1}{s_w} \right\rfloor$$
$$h_o = \left\lfloor \frac{h_i - f_h + 1}{s_h} \right\rfloor$$

2.2.4 Fully connected Layers

The fully connected (FC) layer (also called dense layer) takes a one-dimensional vector x of length z_i as input. After processing, the layer provides an output vector y of length z_o. The lengths of the vector x and y, z_i and z_o respectively, are always the same in the case of a FC layer.

The FC layer in a CNN is placed at the end of the network and is shown as dense layers in Figure 2-4. It is called fully connected because every processing unit in this layer is connected to all of the incoming input values. The input is processed by calculating $y = \omega \cdot x + b$. Each input value x_i is multiplied with the corresponding weight ω_i and then all the connections of a neuron are summed up together with a bias value b. Its operation can be viewed mathematically as:

$$y = \omega_1 x_1 + \omega_2 x_2 + \cdots + \omega_z x_z + b$$

The key difference between a convolution layer and a FC layer is the number of input elements they process at a single instance. Convolution layers take a subset of all the input elements to process through a filter whereas the FC layer takes all the input values as input. This makes the FC layers very expensive computationally.

2.2.5 Pooling Layer

Pooling layers reduce the input size for the next layer. They take an image as input i that can be represented as (w_i,$h_i d_i$), having w_i as width, h_i as height, and d_i as the depth (channels) of the input image.

The granularity of the pooling operation is defined in terms of its stride size s. Stride size in a pooling works the same way as in a convolution layer: to define the step size of the pooling filter to perform the next pooling operation.

If the filter size of the pooling filter is $f_w \times f_h$, then we can calculate the output size of the pooling layer in the same way as for the convolution layer:

$$w_o = \left\lfloor \frac{w_i - f_w + 1}{s_w} \right\rfloor$$
$$h_o = \left\lfloor \frac{h_i - f_h + 1}{s_h} \right\rfloor$$
$$d_o = d_i$$

The pooling layers do not have any weight parameters ω to learn during the training of the network. These layers are necessary in order to reduce the size of the input data and continue with the execution in an efficient manner. A 2×2 pooling means that a total of 4 values are pooled together to make a single value.

Conventionally, experts in the machine learning community have been using one of two choices for pooling: average pooling and maximum pooling. In the homomorphic encryption community, Dowlin et al. [1] showed that a sum-pooling is efficient in processing. The pooling operations are performed in such a way that a sub-region of the incoming input is reduced to a single value by taking the maximum, average, or a summation.

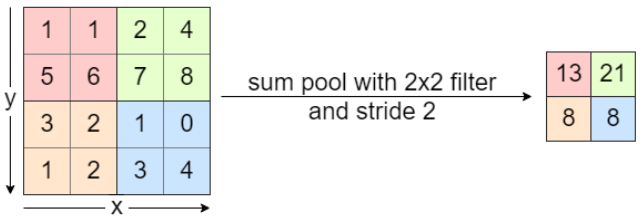


Figure 2-5 Example of a sum pooling operation

From this section, the important observation related to our work is that a sum pooling can also be used as an alternative to the max pooling or average pooling. An example of sum pooling can be seen in Figure 2-5. In the example shown, the left table denotes input values in which a 2×2 filter with a stride of 2 is used. The input values added together are shown by the same colour in the table on the right denoting the output of each pooling step.

2.2.6 Non-Linear Layers

The non-linear layers in a CNN use an activation function (hereon represented by α) to process each input element individually. These activation functions α are non-linear mathematical functions. These do not have any weight parameters ω to learn during the training of the network.

Some of the most common activation functions used in CNN are as follows:

Logistic Sigmoid

Its output lies between 0 and 1. This makes it an ideal candidate to be used for probabilistic results showing the matching probabilities to a known class. The sigmoid functions are mostly used in the fully connected layers to help in better classification.

$$y = \frac{1}{1 + e^{-x}}$$

Hyperbolic Tangent

Its output lies between -1 and $+1$.

$$y = \tanh\, x \;\; = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Heaviside Step

Its output is either 0 or 1.

$$\text{if} \;\; x < 0 \;\; \text{then 0 else if} \;\; x > 0 \;\; \text{then 1}$$

Softmax

It is used in the last layer of a neural network to normalize the outputs because it provides values between 0 and 1 for a single output x, and the sum of all outputs is equal to 1. For example, a layer outputs a total K values, then the softmax activation function is defined as:

$$y_i = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}} \text{ where } i = \{1, \ldots, K\}$$

ReLU

The ReLU function and some of its variants are the most common activation functions used in CNN models. The ReLU functions are mostly used directly after a convolution layer. They are calculated as $y = max(0, x)$ for an input x.

2.2.7 An Example CNN Classification

Let us consider a toy example having just two layers, to explain the key operations involved in convolution layers more concretely. In this example, the input image is a single 4×4 matrix having a single channel (total 16 values) provided to a convolution layer. The convolution layer has a single 2×2 filter.

The output of the convolution is connected to a fully connected layer having a vector of 4 neurons. These two layers can be seen in Figure 2-6 connected by a black arrow whose direction represents the flow of values. In this example, we have not used any activation functions and the fully connected layer is a simple weighted sum of values for each of the 4 neurons. The output of the 4 neurons will be treated as the output of the network.

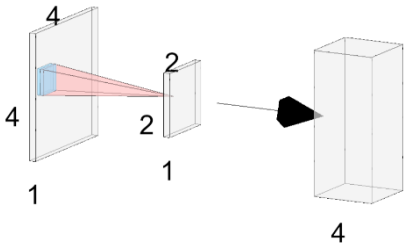


Figure 2-6 Toy example for classification using convolutional neural network.

In this network, the 2×2 filter is first set on the top left 4 values of the input. These values are multiplied by the 4 weight values of the filter and summed up

together to provide the first output value. Then the filter is moved two pixels to the right to the top right input values to multiply them by the 4 weight values of the filter and sum them to get the second output value. Then the filter is moved again to the 4 values in the lower left of the input image and processes them to get the third output value. The filter is then moved to the lower right 4 values to get the final fourth output value of the convolution layer.

After processing the convolution layer, the 4 output values are provided to the fully connected layer as input. Since the output from the convolution layer is a 2-dimensional matrix, but the fully connected layer expects a 1-dimensional array as input, we have to convert the incoming values to an array. At the end, the converted linear vector is provided to a fully connected layer to process as expected. After the processing of the fully connected layer that is our last layer, we interpret the output as our prediction result from the network.

This toy example highlights the major parts of a CNN where we perform the main predictive operations.

2.3 Homomorphic Encryption

This section aims to provide the preliminary level of understanding for homomorphic encryption in general, and then the important aspects that are relevant to this research.

In general, encryption is the method of converting information, known as plaintext, into a seemingly random and garbled form known as ciphertext. The conversion, however, is predictable and only reversible by the party holding a secret piece of information called the secret key.

Homo-morphism is derived from Greek words that mean to have *same form*. The word homomorphism in algebra refers to a structure-preserving map between two

algebraic structures of the same type as rings or vectors. Consequently, Homomorphic encryptions are cryptosystems where the conversion from plaintext to ciphertext preserves relationship between their elements.

The aim of creating homomorphic encryption schemes is to perform arithmetic operations directly on the encrypted data. This exclusive property enables us to perform any arbitrary operations on the ciphertexts without the need of decryption. In a traditional encryption technique, we can only convert our sensitive plaintext data to scrambled ciphertext. Traditionally, the scrambled ciphertexts do not allow any computational operations on the data other than its decryption back to the original plaintext. This suggests a non-interactive property of the traditional ciphertexts in which we cannot interact with the original content scrambled into it. However, the ciphertext generated through homomorphic encryption schemes has an interactive property through which we can perform arithmetic operations on the ciphertext itself. The results of the operations are the same as if we were performing the same operations on the content inside the ciphertext itself.

In the most recent homomorphic schemes, the supported arithmetic operations available are addition and multiplication. Subtraction can be performed by negating the second number and adding this to the first. The division operation, however, is not possible in a straightforward way. In the same way, since comparison of two numbers is a logical task, this is also not straightforward.

In 2009, Gentry proposed the first fully homomorphic encryption (FHE) scheme [8]. The key to allowing arbitrary computations is that an FHE scheme allows both homomorphic addition and multiplication operations on the encrypted data. Previous homomorphic encryption schemes only provided one of the two operations and therefore were not fully homomorphic. Ciphertexts of current FHE schemes

inherently contain a certain amount of noise, which grows during homomorphic operations. This noise ‘‘pollutes’’ the ciphertext and if it grows too large, makes correct decryption impossible, even with the legitimate decryption key.

Fully homomorphic schemes have at their core a somewhat homomorphic encryption scheme that can handle a certain amount of homomorphic computation. To enable an unlimited number of operations, ciphertexts need to be refreshed by a costly re-encrypt procedure called bootstrapping.

In Gentry's initial work and many follow-up papers, the standard way of encrypting data is bitwise. This means that the encryption procedure takes each bit of the data separately and produces a corresponding ciphertext. Addition and multiplication of bits (modulo 2) corresponds to bitwise XOR and AND operations and thus allows us to evaluate any Boolean circuit, i.e., carry out arbitrary computation, by first expressing the computation in XOR and AND gates. Nevertheless, breaking down a computation into bit operations can quickly lead to a complicated and deep circuit that cannot be handled by the somewhat homomorphic scheme and requires bootstrapping.

Homomorphic Encryption Schemes: A seminal work on classifying and defining various homomorphic encryption schemes based on their properties was conducted by Armknecht et al. in 2015 [1]. Their work has resulted in adopting precise terminologies and setting a trajectory for our study into current homomorphic encryption schemes. The following definitions are an adoption of their work in which explanations of the definitions are provided wherever necessary.

Current homomorphic encryption schemes can evaluate two types of Boolean functions in the form of logic gates: addition and multiplication. Messages in such schemes are represented in a plaintext space of $P = \{0,1\}$, which can be used to represent any message. The two Boolean functions can be used to form a logic

circuit C that can describe computation involving addition ($+$) and multiplication ($\times$) by using binary XOR and AND operators.

2.3.1 Partially Homomorphic Encryption (PHE)

Prior to Gentry's work in 2009, PHE schemes showed how to compute a single type of homomorphic operation over the encrypted values. Among them, some did allow multiple types of operations but in a very limited way. Notable ones are shown against their supported type of encryption as follows:

Table 2-1 Partially Homomorphic Cryptosystems

PHE Cryptosystem	Supported Homomorphic Encryption
El Gamal [9]	Multiplicative
Paillier [10]	Additive
RSA [11]	Multiplicative
Goldwasser-Micali [12]	Additive, but it can encrypt only a single bit
Boneh-Goh-Nissim [13]	Unlimited number of additions but only one multiplication

As opposed to the recent Boolean circuit based evaluations, Dowlin et al. [1] discuss PHE schemes which have their plaintext messages and their corresponding ciphertexts contained in a related algebraic structure, often a group or a ring. This typically limits the function f to be an algebraic operation associated with the structure of the plaintexts. PHE schemes are more efficient than the later discussed SHE and FHE.

The preliminary knowledge of Paillier cryptosystem for our work in Chapter 3 is reproduced here from our original publication in [14].

Paillier cryptosystem

Homomorphic Encryption (HE) [8] deals with the mathematical phenomena of performing operations on garbled text (also known as ciphertext). This involves encryption to be done in such a way that if the resultant ciphertext is transformed back to plaintext then the result will be the same as if the operation was performed in plaintext.

In mathematics, a homomorphism is a structure-preserving transformation. To explain, Dowlin et al. [1] provides an example. Consider the map Φ: $\mathbb{Z} \rightarrow \mathbb{Z}_7$ such that $\Phi\ z\ = z \bmod 7$. This map Φ preserves both the additive and multiplicative structure of the values in the sense that for every $z_1,\ z_2\ \in \mathbb{Z}$, we have that $\Phi(z_1 + z_2) = \Phi(z_1) \oplus \Phi(z_2)$ and $\Phi(z_1 \bullet z_2) = \Phi(z_1) \otimes \Phi(z_2)$ where $\oplus$ and $\otimes$ are the addition and multiplication operations in $\mathbb{Z}_7$.

The Paillier cryptosystem is a probabilistic asymmetric algorithm for public key encryption. It was first invented by Pascal Paillier [10] and then later improved by constructing more generalized version [15]. The scheme is an additive homomorphic cryptosystem which can compute the encryption of $m_1 + m_2$ if it is given a public key and the encryptions of m_1 and m_2.

The scheme works as follows:

Key generation

This algorithm generates a set of key pairs K where $K = PK \times SK$ with a public key PK and a secret key SK. These keys are generated by selecting p, q as big primes. Then $\lambda = lcm(p - 1, q - 1)$ is the secret key SK. Moreover, let $n = pq$ and g in $\mathbb{Z}_{n^2}^*$ be an element of order αn for some $\alpha \neq 0$. (n, g) is the public key PK.

Encryption

This algorithm is responsible for generating ciphertext C from a plaintext message $M < n$ by using the PK in such a way that a random value $r < n$ gives us $C = g^M r^n \bmod n^2 = Enc_{pk}(M)$.

Decryption

This algorithm is responsible for generating the plaintext message M in such a way that $Dec_{sk}(C) \rightarrow M$. Here, $C < n^2$ is the ciphertext, and $Dec_{sk}(C) = \frac{L(C^\lambda \bmod n^2)}{L(g^\lambda \bmod n^2)} \bmod n$, where $L(x) = (x-1)/n$.

2.3.2 Somewhat Homomorphic Encryption (SHE)

SHE is defined as a Boolean circuit-evaluation scheme that has correct decryption and correct evaluation on the ciphertext, with no limits enforced on the length of the generated ciphertexts, and having support for a limited number of consecutive operations for evaluation.

An example of SHE is the Brakerski, Gentry, and Vaikuntanathan (BGV) scheme [16] which bases its security on the Ring Learning With Errors (RLWE) problem. RLWE is considered to form a fundamental foundation for future public-key cryptography in a post-quantum computational world in which cryptanalysis of traditional problems, like integer factorization and discrete logarithm, will be much easier to solve. Although these schemes have no limit intrinsically on the manipulated ciphertext [17], the BGV showed that such schemes can be feasible to implement and adopt practically for relative efficiency and reasonably short ciphertexts [18]. SHE schemes also serve to build up later schemes like FHE and LHE, after modifications.

2.3.3 Fully Homomorphic Encryption (FHE)

FHE is defined as a Boolean circuit evaluation scheme in which: the length of the ciphertext does not grow through homomorphic operations, the ciphertext is correctly decrypted to the corresponding plaintext without error, and which supports all possible circuits for evaluation through the evaluation key evk. In addition, the length of the output after arithmetic evaluation has an imposed limit, which in the case of SHE is non-existent.

It is important to note here that the key contribution of Gentry's work [19], [20], was the development of the bootstrapping technique. FHE schemes based on his work can evaluate a circuit by adding noise to the ciphertext. This noise however increases with each homomorphic evaluation and requires decryption to eliminate it. If we do not decrypt, then, after a certain threshold on the number of homomorphic evaluations, the growing noise in the ciphertext results in a failure to decrypt correctly to its corresponding plaintext. Gentry solved the problem through a re-encryption step by converting the decryption algorithm for his scheme into a circuit for homomorphic evaluation. In addition, encrypting the noisy ciphertext again to make it doubly encrypted. In this way, he could perform decryption of the inner encryption in a homomorphically-encrypted way by passing the doubly encrypted ciphertext and an encrypted version of the private key to the decryption circuit. The output of his re-encryption was the outermost ciphertext of the same plaintext that was doubly encrypted, but with the noise gone. Furthermore, this recryption ability was made a part of the evaluation algorithm along with the ability to evaluate at least a single logic gate so that any desired circuit can be fully evaluated.

2.3.4 Levelled Homomorphic Encryption (LHE)

LHE is defined as a Boolean circuit evaluation scheme in which the auxiliary input α limits the maximum depth, d, of circuits can be evaluated. Along with the requirements of correct decryption and a practical limit on the output from arithmetic evaluation, it is necessary that the length of the evaluation output is not dependent on the depth of the Boolean circuit.

The limit on the depth of the Boolean circuits is usually a confusing difference between LHE and SHE. The depth of the Boolean circuits in SHE can be increased through the choice of parameters that in turn increases the length of the output ciphertext. In LHE on the other hand, the depth of the Boolean circuit is an input parameter that is independent from the length of the output ciphertext.

2.3.5 Introduction to SHE and its implementation in SEAL

Homomorphic encryptions are able to perform computations directly on ciphertext. However, based on the properties of the various variants of HE, not all are suitable for every task. Somewhat homomorphic encryption (SHE) is a variant of HE in which one is able to perform many encrypted operations sequentially, but the number of total encrypted operations is limited by the initialization parameters of the scheme.

In our work, we have used the SEAL[1] implementation that is a practical homomorphic library for use in higher-level applications. The library is written in C++, with a wrapper for languages like C#, and the library code is available for

[1] https://github.com/microsoft/SEAL

analysis and improvement under the open-source MIT licence on GitHub, which is sufficient for our work.

One of the underlying schemes implemented in SEAL is the Fan-Vercauteren (FV) scheme [21] along with some improvements as discussed in their documentation [22]. The FV scheme is a somewhat homomorphic encryption scheme. The security of this scheme is based on the Ring Learning with Errors (RLWE) problem that is considered a quantum-secure problem providing high security. Besides the FV scheme, SEAL now includes the CKKS scheme [23] as well.

For somewhat homomorphic encryptions, the distinguishing property from other homomorphic encryption types is that they are not compact. Their ciphertext length is not bounded for the output of any homomorphic evaluation operation [4]. This means that if we use a somewhat homomorphic encryption scheme, then sequential use of the homomorphic operations will result in a bigger and bigger ciphertext.

2.3.6 Description of the FV Scheme

The homomorphic encryption scheme used in the SEAL library is the FV scheme. This scheme is based on the algebraic ring structure. The algebraic rings are basically mathematical sets of elements within a modulus and support the two binary operations of addition and multiplication. The FV scheme requires our original plaintext numbers, those that we want to encrypt, to lie in the ring structure R_t. The ring R_t is defined such that $R_t = \mathbb{Z}_t[x]/\ x^n + \ 1$, which means that only those integer numbers from $\mathbb{Z}$ are included for which there exists a polynomial of degree smaller than n with coefficients reduced modulo t. Here, the scheme is initialized by specifying the key initialization parameters of the plaintext modulus t, ciphertext modulus q, and the degree of polynomial modulus n. The ring structure allows polynomials of degree less than n with coefficients modulo t. The t is called the plaintext modulus and the $x^n + 1$ is called the polynomial modulus.

Both of these moduli are specified as encryption parameters before performing the encryption itself.

For one to encrypt a number under this scheme the number has to lie within the ring structure R_t; therefore, we first encode each of our original numbers to become a member of the ring structure and to make it encrypt-able. The ring R_t specifies that any number, either an integer or a rational number, should be encoded into a plaintext polynomial in R_t and then it can be encrypted under the scheme. Once the required numbers are encoded into R_t, they are encrypted into a ciphertext array of at least two polynomials in the ring structure R_q, where q is called the coefficient modulus and is specified as an encryption parameter before performing the encryption itself. See section 2.3.7 below for setting the initialization parameters.

The precise definitions of the encryption and decryption in the FV scheme are stated in this section for completeness. Here, by $a \xleftarrow{\$} R_2$ it is denoted that a is sampled uniformly from the finite set R_2. The main algorithms of the scheme are:

- Generate secret key sk, public key pk, and evaluation keys evk using the *SecretKeyGen*, *PublicKeyGen*, and *EvaluationKeyGen* algorithms.
- *Encryption*(pk, m): For $m \in R_t$, let the public key $pk = (p_0, p_1)$. Sample $u \xleftarrow{\$} R_2$, and e_1,$e_2 \leftarrow \chi$.
    - Compute $ct = ([\Delta m + \ p_0 u + e_1]_q, [p_1 u + e_2]_q)$ to get the encrypted ciphertext ct.
- *Decryption*(sk, ct): Set $s = sk$, $c_0 = \mathrm{ct}[0]$, and $c_1 = \mathrm{ct}[1]$.
    - Compute $m' = \left[\left\lfloor \frac{t}{q} [c_0 + c_1 s]_q \right\rceil \right]_t$ to get the decryption of m into m'.

2.3.7 Key generation

The encryption step converts a plaintext number to a ciphertext number by using the secret (also called the private) and public key pair. The key generation is performed in two steps. The first step is to generate the secret key s. The generation of the secret key is performed by generating a random polynomial having n terms. Moreover, each of the coefficients is sampled uniformly from the set $\{-1,0,1\}$. Next, the public key pk is generated by first taking another temporary random polynomial (called the a polynomial) from the ciphertext space, i.e., a polynomial having its coefficients modulo the q variable. The coefficients are sampled uniformly from the entire range of the q. The number of terms in the temporary random polynomial a will also be equal to n as in the secret key. Next, we will need a random error polynomial called e for the public key, so we sample n coefficients from a discrete Gaussian distribution of relatively much smaller values as compared to q. The public key is then defined by the two polynomials ($pk_0 = [-as + e]_q$ and $pk_1 = a$) in the following form:

$$\text{pk} = ([-as + e]_q, a)$$

Now, after generation of the s and the pk keys, we are able to perform encryptions. Therefore, next, we will see how the encryption step is performed in the encrypt-process-decrypt routine.

2.3.8 Encryption

Recall that the encryption step converts a plaintext polynomial to a pair of ciphertext polynomials. Where the plaintext polynomial has n terms, each having its coefficient modulo t. Moreover, the ciphertext polynomial pair also has n terms, but the coefficients are modulo q. In order to perform the encryption, we will have to generate three more small polynomials similar to those used in the generation of

the public key. Two error polynomials e_1 and e_2 will be sampled from the same discrete Gaussian distribution used in the generation of e in the public key. Along with e_1 and e_2, we will also generate the third polynomial, u, having coefficients uniformly sampled from the same set as the secret key, i.e., the set $\{-1,0,1\}$.

$$e_1 = a'_{n-1}x^{n-1} + a'_{n-2}x^{n-2} + \cdots + a'_1x^1 + a'_0$$

$$e_2 = a''_{n-1}x^{n-1} + a''_{n-2}x^{n-2} + \cdots + a''_1x^1 + a''_0$$

$$u = a'''_{n-1}x^{n-1} + a'''_{n-2}x^{n-2} + \cdots + a'''_1x^1 + a'''_0$$

After generating the three polynomials e_1, e_2, and u, the two ciphertext polynomials ($ct_0 = [pk_0 \cdot u + e_1 + \lfloor \frac{q}{t} \rfloor \cdot m]_q$ and $ct_1 = [pk_1 \cdot u + e_2]_q$) are calculated as:

$$ct = \left(\left[pk_0 \cdot u + e_1 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m \right]_q, [pk_1 \cdot u + e_2]_q \right)$$

The ciphertext ct calculated above hides our message m in the combination of random noise values successfully. Since our original message m is the plaintext polynomial having coefficients of the modulus t variable, it is scaled up first by $\lfloor \frac{q}{t} \rfloor$, and then hidden by summing up with $pk_0 \cdot u + e_1$. Although, the e_1 is sampled from a discrete Gaussian distribution, the term $pk_0 \cdot u$ hides our message effectively making it indistinguishable from random noise. It is precisely because of $pk_0 \cdot u$, that the same plaintext message will result in a different ciphertext every time.

By further analysis of the calculations for any single encryption step, we can distinguish five components that go into a ciphertext. They are the following:

1. public key
2. private key
3. message
4. mask
5. noise

The five components of the encryptions can be observed in the mathematical expansion of the public key in the encryption step as follows:

The ciphertext is a combination of two polynomials as

$$ct_0 = \left[pk_0 \cdot u + e_1 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m\right]_q$$
$$ct_1 = [pk_1 \cdot u + e_2]_q$$

Similarly, the public key pk is a combination of the following two polynomials as

$$pk_0 = [-as + e]_q$$
$$pk_1 = a$$

So, expanding pk_0 in ct_0 will give us

$$ct_0 = \left[\ -as + e\ \cdot u + e_1 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m\right]_q$$
$$ct_0 = \left[-aus + eu + e_1 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m\right]_q$$

Similarly, expanding pk_1 in ct_1 will give us

$$ct_1 = [a \cdot u + e_2]_q$$

By combining the expanded form of ct_0 and ct_1 for ct, we get distinct components as emphasized in equation 2.1.

$$ct = \left(\left[\overbrace{-\underbrace{a}_{mask}\ \underbrace{u}}^{pk_0} \quad \underbrace{s}_{secret} + \underbrace{\overbrace{e}^{pk_0}\ u + e_1}_{noise} + \underbrace{\left\lfloor \frac{q}{t} \right\rfloor \cdot m}_{message} \right]_q , \left[\underbrace{\overbrace{a}^{pk_1} \cdot u}_{mask} + \underbrace{e_2}_{noise} \right]_q \right) \tag{2.1}$$

2.3.9 Decryption

After understanding the encryption step in the encrypt-process-decrypt routine, we move to understanding the decryption process. In order to decrypt a ciphertext ct in the FV scheme, we first remove the masking by adding the two polynomials of the ciphertext to get a polynomial as follows:

$$
\begin{aligned}
[ct_0 + ct_1 \cdot s]_q &= \left[-aus + e_0u + e_1 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m\right]_q + [a \cdot u + e_2]_q \cdot s \\
&= \left[-aus + e_0u + e_1 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m\right]_q + [a \cdot u \cdot s + e_2 s]_q \\
&= \left[e_0u + e_1 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m\right]_q + [e_2 s]_q \\
&= \left[e_2 s + e_0u + e_1 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m\right]_q
\end{aligned}
$$

The above expansion of the terms shows that, besides our message m scaled by $\lfloor \frac{q}{t} \rfloor$, extra information called the inherent noise, v, exists in the ciphertext. This inherent noise can be defined from the equation above as

$$v = [e_2 s + e_0 u + e_1]_q \tag{2.2}$$

Next, for the decryption to work correctly, we calculate by scaling down the ct polynomial back to the values in modulo t. Meanwhile, the noise terms v will be removed by rounding off. For this to work, the noise terms need to be small enough so that they are rounded off otherwise the decryption will fail. This scaling down step is performed by first multiplying with $\frac{t}{q}$, and then the small noise terms are rounded off as follows:

$$m' = \left[\left\lfloor \frac{t}{q} \left[e_2 s + eu + e_1 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m \right]_q \right\rfloor \right]_t$$

On the other hand, we can write this by highlighting the noise polynomial as:

$$m' = \left[\left\lfloor \frac{t}{q} \left[v + \left\lfloor \frac{q}{t} \right\rfloor \cdot m \right]_q \right\rfloor \right]_t$$

In the above equation, our plaintext message m is decrypted to its equivalent plaintext message m'. If the ciphertext was not operated on, then $m' = m$, otherwise m' will be the resultant value of the operation being performed.

More importantly, the coefficients of the noise polynomials represented by v need to be small enough so that they are scaled down by $\frac{t}{q}$ and are rounded off. On the contrary, if the noise coefficients are bigger, then they will end up closer to a different integer than their correct one and will silently produce an incorrect result. This observation means that the difference q/t gives us the liberty to handle an equivalent amount of noise. The larger the difference between the q and the t, the larger the amount of noise that can be tolerated in the decryption process.

2.3.10 Noise budget (Circuit Depth)

Each ciphertext can support a limited number of homomorphic operations on it. This limit is called the noise budget by SEAL but is normally referred to as the circuit depth by other researchers in the homomorphic encryption community. We will use the nomenclature of "noise budget" as well. As we perform homomorphic operations, this noise budget decreases towards zero. After the zero limit of the noise budget is reached, all the homomorphic operations will result in garbage values because the coefficients of the polynomial representing the ciphertext will exceed the coefficient modulus q, an encryption parameter. Thus, the decryption algorithm will not be able to decrypt the ciphertext within the set encryption parameters.

In our work, the noise budget is the most important aspect to follow. This is because the noise budget allows or restricts a computing party to perform any further homomorphic operations on a ciphertext.

It is worth mentioning here that if we combine a ciphertext, having a zero or insufficient noise budget, in an arithmetic operation with another ciphertext having ample noise budget, then the noise budget for the resultant ciphertext will be zero. This clarifies that if either of the ciphertext operands have insufficient noise budget, then the output will not be decrypted and decoded correctly.

The initial noise v_i in a ciphertext is determined by the following formula as shown in the SEAL documentation [22]:

$$v_i = \frac{q \bmod t}{q} \cdot \|m\| \cdot N_m + \frac{7nt}{q} \cdot \min\ \text{noiseMaxDeviation}, 6 \times \text{noiseStandardDeviation}$$

In the formula for calculating the initial noise budget above, we have our original message represented as m, the encryption parameters as n, t, q, and the highest degree of polynomial m as N_m. The random noise distribution is defined by the standard deviation and the maximum deviation of the sample. We see from the initial noise formula above that for the same message m, the initial noise budget is determined by the initialization parameters t, q, and n of the encryption scheme. These parameters are defined in the following section.

2.3.11 Parameter Selection (t, q, n)

The homomorphic operations are strongly dependent on the encryption initialization parameters. The initialization parameters affect the actual encryption, decryption, performance, and the result of the operations. These parameters need to be set before encrypting any numbers or performing any homomorphic operations. Based on these encryption parameters, the security keys (the public key, private key, and the evaluation keys) of the scheme are generated. Three main encryption parameters are explained as below:

2.3.11.1 Plaintext Modulus (t)

The plaintext (coefficient) modulus can be any positive integer that determines the size of the plaintext data that can be encrypted. Importantly, it affects the initial noise budget in a freshly encrypted ciphertext, as well as the consumption of the

noise budget in homomorphic multiplications. It is essential to keep the t value as small as possible for a good performance without affecting the noise budget [22].

2.3.11.2 Ciphertext Modulus (q)

The ciphertext (coefficient) modulus in the FV scheme is a product of one or more small prime numbers. The size of the coefficient modulus should be thought of as a significant factor in determining the noise budget. A ciphertext should have its noise value smaller than the q value to get decrypted properly. The decryption algorithm will not be able to decode a ciphertext that has a noise value larger than the q value.

If a large noise budget is required for complicated computations, a large coefficient modulus needs to be used. However, researchers have shown that a larger coefficient modulus q also lowers the security level of the scheme. This reduction in the security level can be recovered by simultaneously increasing the polynomial modulus n when increasing q [22].

When we talk about the size of the coefficient modulus, we mean the bit length of the product of one or more smaller prime numbers. In SEAL, the coefficient modulus is a positive composite number that is the product of distinct primes limited in size to less than 60 bits.

Performance is primarily affected by the size of the polynomial modulus n, and the number of prime factors in the coefficient modulus. Thus, based on our experiments as shown later in section 4.6, we will recommend using as few factors in the coefficient modulus is possible for good performance.

2.3.11.3 Polynomial Modulus (n)

The polynomial modulus is the maximum number of terms that can be used in a polynomial to represent a plaintext or a ciphertext. The value n should be thought

of as mainly affecting the security level of the scheme. A larger polynomial modulus makes the scheme more secure. In order to be able to encode numbers into the ring R_t properly, the value of n must be a power-of-2 cyclotomic polynomial, i.e., a polynomial of the form $(1 \cdot x^{(power\ of\ 2)} + 1)$, for example $(1 \cdot x^{2048} + 1)$. The prominent effect of a bigger polynomial modulus n is that it makes ciphertext sizes larger because of the higher number of coefficients, and consequently all operations are slower. SEAL's documentation recommends n takes values of 1024, 2048, 4096, 8192, 16384, or 32768, based on security and performance reasons, for common computation scenarios.

2.3.12 Relinearization

In the FV [21], and other similar homomorphic encryptions schemes, multiplications result in increasing the number of polynomials in a ciphertext. *Relinearization* is the method of reducing the number of polynomials back to a certain acceptable value to control the noise growth.

Since each ciphertext is an array of polynomials, the array contains at least two polynomials (c_0 and c_1) as represented below where s is the random number for security:

$$c = (c_0 + c_1 s)$$

Relinearization is necessary due to several reasons. These include that a bigger polynomial takes more time to process as compared to a smaller polynomial. Further, convolutional neural networks are very computationally intensive algorithms, involving operations, like multiplication of many numbers, in several layers, to produce the result. During our research, we found another interesting reason to use relinearization is that smaller ciphertexts lead to a smaller increase in the noise. The increase of noise is dependent on the size of the ciphertext operands

that can be improved by using relinearization after every multiplication in the CNN. This can be seen by a simulation of the noise growth when two ciphertexts are multiplied together, one with an increasing number of polynomials and the other with a fixed polynomial count size of 2. The noise in the result of the multiplication is calculated by using the relinearization formula provided in the SEAL documentation for multiplication [22].

$$v_o = v_i + \frac{2t}{q} \cdot \min\{B, 6\sigma\} \cdot \ J - K \ \cdot n \cdot \ l + 1 \ \cdot z \qquad 2.3$$

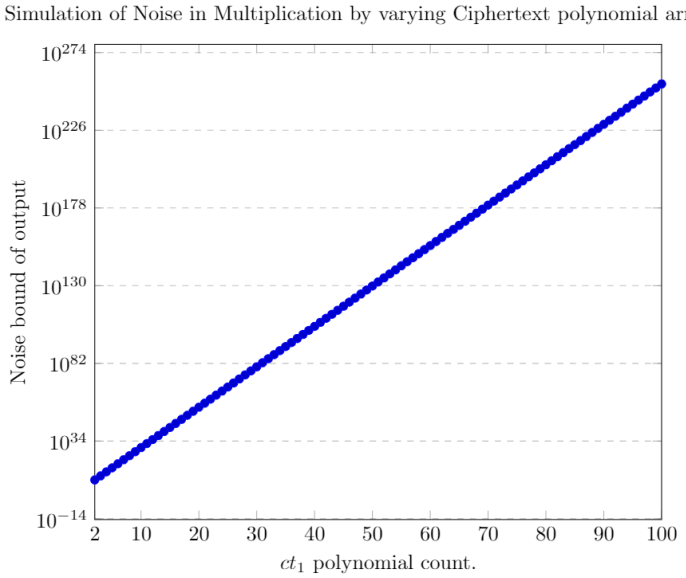


Figure 2-7 Effect of ciphertext polynomial count on the noise in the results

The example simulation in Figure 2-7 shows the estimation of the upper bound of the noise in the product of ciphertexts $ct_1 \times ct_2$ where the polynomial count size of ct_2 is always 2. On the x-axis, we have the varying size of the ciphertext c_1 from two polynomials to 100 polynomials. On the y-axis, we have the noise growth estimates for the product of ct_1 and ct_2.

This estimation is based on the noise bound formula for ciphertext multiplication given in SEAL documentation. The formula is reproduced in Appendix A. In the formula, the sizes j_1 and j_2 represents the polynomial counts of the ciphertext

operands $ct_1(c_0, c_1, ..., c_{j_1})$ and $ct_2(c_0, c_1, ..., c_{j_2})$ respectively. Here, c represents a single polynomial of the ciphertext. In addition, in the formula, v_1 and v_2 represent the existing noise in these ciphertext operands, respectively. The formula for the noise bound tells us the extent of the noise in the resultant ciphertext after multiplying ciphertexts of sizes $j_1 + 1$ and $j_2 + 1$ respectively. By keeping all other values static and varying the size of an operand, say j_1 for ct_1 in the multiplication formula, we will know whether multiplication with a smaller size ciphertext inherits smaller noise or not. Using this formula, we can estimate the maximum theoretical noise that can be inherited in the resultant ciphertext. Note that this is not giving us the exact noise value but rather gives a maximum bound on the noise that is possible but is unlikely to occur in practice.

We conclude from this simulation that there is an exponential growth of noise from small to large ciphertexts.

It is notable that this behaviour occurs for the SEAL library because it allows ciphertexts of size greater than 2 to delay a relinearization step for flexibility. However, we do not recommend delaying the relinearization step in order to keep the noise of the result to a minimum.

2.3.13 Number encoding

The numbers on which we want to perform computations in SEAL have to be encoded in a polynomial of the form $x^n + 1$. Here n is a power of two and x is the coefficient of the polynomial. The $x^n + 1$ is the polynomial modulus as discussed in section 2.3.11.3 above. The SEAL library encodes integers and fractions in a slightly different way as described below.

Integer Encoding

We introduce integer encoding using an example. If the encoding base $x = 2$, then the integer $26 = 2^4 + 2^3 + 2^1$ which is encoded as the polynomial $1x^4 + 1x^3 + 1x^1$. In the same way, when the encoding base $x = 3$, then the integer $26 = 3^3 - 3^0$ is encoded as the polynomial $1x^3 - 1$.

Fractional Encoding

Fractional encoding is done using fixed-precision rational numbers in a manner where the integral part is handled similarly to integer encoding but the fractional part is handled slightly differently. It expands the number in a given base x, possibly truncating an infinite fractional part to finite precision, for example,

$$26.75 = 2^4 + 2^3 + 2^1 + 2^{-1} + 2^{-2}$$

where $x = 2$ as an example. For purpose of illustration, suppose the polynomial modulus is $1x^{1024} + 1$. The integer part of the number is represented in the same way as in encoding an integer, but the fractional part instead is moved to the highest degree part of the polynomial with the signs of the coefficients changed. Since we are dealing with ring structures with only positive integers, therefore, the negative coefficients are always encoded as a remainder of the plaintext coefficient modulus t. In this example, with $t = 8$ and $n = 1024$, we would represent 26.75 as the polynomial below:

$$26.75 = 7x^{1023} + 7x^{1022} + 1x^4 + 1x^3 + 1x^1.$$

2.3.14 Noise within the Ciphertext

In a homomorphic operation, random numbers called the "noise" are added to the resultant ciphertext for security reasons. The practice of combining random numbers with our original message m is common in cryptography to make the ciphertexts unpredictable. These random noise numbers are taken from a

combination of random number distributions. To get our original message m back, from the scrambled ciphertext, it is necessary to remove all the random noise values. The removal of the random noise is achieved through the help of the secret keys, which only the authorized parties hold.

In homomorphic encryption schemes, unlike their traditional counterparts, the ciphertexts are dynamic as they have the ability to interact with other ciphertexts, resulting in an entirely new ciphertext. The value of the noise is taken randomly; therefore, its exact value cannot be known. However, the random distribution from which we sampled the noise value is known. SEAL documentation [22] provides a close estimation for the upper bound and the extent of the added noise. This estimation of noise is used in our work described later in section 4.5.2 below to get the encryption parameters for a convolutional neural network.

2.3.15 An example of HE operations

The mechanisms and concepts behind a homomorphic encryption scheme can be overwhelming at a first view. Here we will present an example to connect and elaborate the HE preliminary mechanisms and concepts described above. For illustrative purposes, much smaller numbers will be used in the example to keep it well-defined. However, numbers in real usage of HE need to be much larger than in this example due to security reasons. For due reference, the idea for this explanation is inspired by Stephen Hardy's article online[2]. Unfortunately, the website is now offline and not available on the internet.

[2] https://blog.n1analytics.com/homomorphic-encryption-illustrated-primer (Accessed: October 22, 2019)

We will start with the most basic numerical structure (i.e., rings) used in these schemes to hold data. The ring structure holds the data contents of both the encrypted ciphertext as well as the unencrypted plaintext. Any ciphertext or plaintext is essentially a normal polynomial, but apart from the ring limits applied to it. To understand the concept of rings and ring limits further, consider the example of a normal polynomial such as $8x^2 + 4x + 1$.

The polynomial in a ring structure will have two differences from a normal polynomial. First, all the coefficients (i.e., 8, 4, and 1) are whole numbers and the remainder of some other whole number $\boldsymbol{t}$ called the plaintext modulus (see section 2.3.11.1). If $t = 12$ then this will essentially be a 12-hour clock. Adding 6 to 9 in such a case will give us a value of 3.

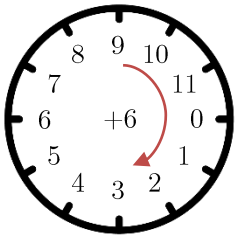


Figure 2-8 Remainder example of a single coefficient

Second, this idea of using only the remainders is extended to the complete polynomials themselves and not limited to their coefficients only. The polynomial modulus is represented by n. Here, the n restricts the polynomials to only the remainders after division with a specific form of n. Notably, the FV scheme [21] used in this research requires n in the specific form of $x^n + 1$ where n is a power of 2. If we take the value of $n = 16$, then the polynomial modulus will be $x^{16} + 1$. So, each polynomial considered will range from 0 to 15 where each coefficient will be from 0 to $t - 1$. This is illustrated in Figure 2-9 where we have 16 rings (similar to Figure 2-8), and each ring represents a single coefficient.

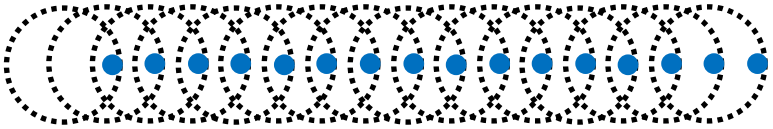


Figure 2-9 Ring illustration of a polynomial

The illustration in Figure 2-9 is represented by the polynomial in the following form,

$$a_{15}x^{15} + a_{14}x^{14} + a_{13}x^{13} + a_{12}x^{12} + a_{11}x^{11} + a_{10}x^{10} + a_9x^9 + a_8x^8 + a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

Now that we have understood the underlying to hold the data of a plaintext and ciphertext in an FV scheme, we will see how encryption and decryption is performed. To encrypt a plaintext to a ciphertext, we will need a pair of public and private keys. The public key is used to perform the conversion from a plaintext to a ciphertext. Whereas the reverse process of decryption to get a plaintext from a ciphertext is performed using the private key.

The plaintext is represented by a polynomial from the ring with its coefficient modulus t. The encryption of this plaintext is represented by two polynomials from the ring but a coefficient modulus q, which is much larger than t. For example, if we take the values $n = 16, t = 7$, and $q = 874$, then the generation of private and public keys is the first step which is done as follows.

The private (or secret) key represented as s is generated by taking a random polynomial with coefficients from the set {−1,0,1}. For example,

$$s = x^{15} - x^{13} - x^{12} - x^{11} - x^9 + x^8 + x^6 - x^4 + x^2 + x - 1$$

After generating the private key, we generate the public key by taking a random polynomial a with coefficients from the ciphertext space i.e., coefficients modulus q.

$$a = 42x^{15} - 256x^{14} - 393x^{13} - 229x^{12} + 447x^{11} - 369x^{10} - 212x^9 + 107x^8 + 52x^7 + 70x^6 - 138x^5 + 322x^4 + 186x^3 - 282x^2 - 60x + 84$$

In the next step of generating public key, we take another random polynomial e with its coefficients drawn from a discrete gaussian distribution. The coefficients are relatively much smaller, and this polynomial is used only once here for the generation of public key and is then discarded.

The public key is defined by a pair of polynomials pk_0 pk_1 as $pk = (\lceil -as + e \rceil_q,\ a)$, for which we now have all the components. Therefore, the first polynomial pk_0 of pk is constructed as $pk_0 = \lceil -a \times s + e \rceil_q$, which gives us

$$\begin{aligned} pk_0 = -285x^{15} - 431x^{14} - 32x^{13} + 86x^{12} - 83x^{11} - 142x^{10} - 41x^9 + 430x^8 \\ + 26x^7 - 158x^6 - 281x^5 + 377x^4 + 110x^3 - 234x^2 - 113x + 252 \end{aligned}$$

Now that we have both the private and the public keys generated, we will see the encryption phase. The encryption coverts a plaintext polynomial with coefficients modulo t, to a pair of polynomials with coefficients modulo q. For example, if our original message to encrypt is $m = 3 + 4x^8$, then the encryption step is as follows.

The encryption phase requires three more small polynomials. Two of them are similar random error polynomials (e_1 and e_2) with their coefficients taken from the same discrete gaussian distribution as above. And another polynomial u with its coefficients drawn from the set $\{-1,0,1\}$. Let's say we get the following

$$\begin{aligned} e_1 &= -5x^{15} - 2x^{14} + 3x^{13} - x^{12} - 4x^{11} + 3x^{10} + x^9 + 4x^8 + 4x^7 + 5x^6 - 4x^5 \\ &\quad - 3x^4 - 3x^3 + 2x^2 - 6x + 4 \\ e_2 &= -7x^{15} + 2x^{14} - 4x^{13} + 5x^{11} + 2x^{10} - x^9 + 4x^8 + 4x^7 - 3x^6 + 2x^5 - 2x^4 + x^3 \\ &\quad - 4x^2 - 2x + 2 \\ u &= x^{14} + x^{13} + x^{12} - x^8 - x^5 - x^3 + 1 \end{aligned}$$

Then the encryption process converts m to a pair of ciphertext polynomials ct_0 , ct_1 as $ct = ([pk_0 \cdot u + e_1 + \lfloor \frac{q}{t} \rfloor \cdot m]_q\ ,\ [pk_1 \cdot u + e_2]_q)$. The original message m is present here with no change except just a scaling. All the other terms are here only for hiding m. Due to this phenomena, homomorphic encryption enables the

additions and multiplications of our original message even though in an encrypted form. Here, evaluating the first element of ct gives us

$$ct_0 = 217x^{15} - 53x^{14} + 13x^{13} - 249x^{12} - 392x^{11} - 238x^{10} + 252x^9 + 115x^8 + 5x^7 + 184x^6 - 201x^5 - 258x^4 - 247x^3 + 144x^2 + 23x + 42$$

Similarly, evaluating the second element of ct gives us

$$ct_1 = 25x^{15} + 225x^{14} - 12x^{13} + 270x^{12} + 350x^{11} - 24x^{10} + 56x^9 - 330x^8 + 386x^7 + 225x^6 - 332x^5 + 68x^4 - 20x^3 - 26x^2 - 91x + 380$$

Now that we have converted a plaintext message m to an encrypted ciphertext ct, we will see how it can be converted back through decryption. To decrypt, we will need the secret key s to calculate $[ct_0 + ct_1 s]_q$. This gives us

$$ct_1 s + ct_0 = 13x^{15} - 2x^{14} + 17x^{13} + 22x^{12} - 32x^{11} - 23x^{10} + 19x^9 - 380x^8 + 9x^7 + 10x^6 - 13x^5 - 3x^4 - 2x^3 - 12x^2 + 7x + 393$$

After this, we rescale the polynomial back to the original range of modulus t by q/t, then we have

$$\frac{13x^{15}}{128} - \frac{x^{14}}{64} + \frac{17x^{13}}{128} + \frac{11x^{12}}{64} - \frac{x^{11}}{4} - \frac{23x^{10}}{128} + \frac{19x^9}{128} - \frac{95x^8}{32} + \frac{9x^7}{128} + \frac{5x^6}{64} - \frac{13x^5}{128} - \frac{3x^4}{128} - \frac{x^3}{64} - \frac{3x^2}{32} + \frac{7x}{128} + \frac{393}{128}$$

Finally, after rounding off our polynomial, we get back our original message m as

$$m = 3 + 4x^8$$

After understanding rings, encryption, decryption, and keys, we look at the addition process of two ciphertexts a and b encrypting plaintext messages m_1 and m_2 respectively.

$$\mathbf{a} = \left(\left[\mathbf{pk}_0 u_1 + e_1 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m_1 \right]_q, [\mathbf{pk}_1 u_1 + e_2]_q \right)$$
$$\mathbf{b} = \left(\left[\mathbf{pk}_0 u_2 + e_3 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m_2 \right]_q, [\mathbf{pk}_1 u_2 + e_4]_q \right)$$

Here, note that both the ciphertexts are encrypted using the same public key. However, the small polynomials of u, and $e_1 ... e_4$ are all different for both ciphertexts.

The addition of two ciphertexts is procedurally simple and involves the addition of their two corresponding polynomial elements. In our example, this will be

$$\boldsymbol{a}+\boldsymbol{b} = [\boldsymbol{pk}_0\ u_1+u_2\ +\ e_1+e_3\ +q\ m_1+m_2\ /t]_q, [\boldsymbol{pk}_1\ u_1+u_2\ +\ e_2+e_4\ ]_q$$
$$\boldsymbol{a}+\boldsymbol{b} = [\boldsymbol{pk}_0\ u_3\ +\ e_5\ +q\ m_1+m_2\ /t]_q, [\boldsymbol{pk}_1\ u_3\ +\ e_6\ ]_q$$

So, from the above addition step, we see that the two messages as well as the small error terms are all added, and we get a resultant ciphertext with similar polynomials that can be decrypted as above. After decryption, we will get a plaintext message that is the sum of m_1 and m_2.

Similarly, the multiplication involves taking the product c of the two corresponding polynomial elements but with some further detail to consider. Consider the same ciphertexts a and b encrypted using the same public key and having their corresponding masking terms as follows

$$\mathbf{a} = [\mathbf{pk}_0 u_1 + e_{11} + qm_1/t, \mathbf{pk}_1 u_1 + e_{12}]$$
$$\mathbf{b} = [\mathbf{pk}_0 u_2 + e_{21} + qm_2/t, \mathbf{pk}_1 u_2 + e_{22}]$$

When we multiply two ciphertexts a and b, here instead of ending up with two polynomials as in addition, we will get three polynomials as the product outcome. This is shown below.

$$\text{mult}\ \boldsymbol{a},\boldsymbol{b}\ = \boldsymbol{c}_0 + \boldsymbol{c}_1 s + \boldsymbol{c}_2 s^2$$

$$c_0 = \left[\frac{t}{q} a_0 b_0\right]_q$$

$$\mathbf{c}_1 = \left[\frac{t}{q}\ \mathbf{a}_1\mathbf{b}_0 + \mathbf{a}_0\mathbf{b}_1\ \right]_q$$

$$c_2 = \left[\frac{t}{q} a_1 b_1\right]_q$$

From a decryption point of view, the third extra polynomial in the product is accommodated by expanding the decryption process to include the extra term as follows:

$$\text{decrypt } a \times b \ = m_3 = \left[\left\lfloor \frac{t}{q} [\mathbf{ct}_0 s^0 + \mathbf{ct}_1 s^1 + \mathbf{ct}_2 s^2]_q \right\rceil \right]_t$$

2.4 Homomorphic Encryption Code Libraries

Practical implementations of PHE are common. A simple search on GitHub for "paillier" returned more than 90 results for repositories of project source codes in languages like Java, Python, C, JavaScript etc. Similarly, the search for other PHE schemes returned many results such as "elgamal" which returned 161 repositories in various languages. Among these repositories, we found several well-maintained source code projects that are usable in the real world.

On the other hand, most of the FHE are of a theoretical nature and exist mostly on paper at present. Only a few major implementations of FHE exist. Two of them are: Halevi and Shoup's HElib [24]; and Microsoft's SEAL [25]. An up-to-date list of implementations is maintained online by the homomorphic encryption consortium[3].

HElib is an implementation of the HE scheme by Brakerski, Gentry, and Vaikuntanathan (BGV) [16], in which the library uses several optimizations such as the ciphertext packing techniques for single instruction, multiple data (SIMD) [26] and the bootstrapping for FHE as detailed in [27]. Currently the HElib has very

[3] https://homomorphicencryption.org/introduction/

low-level routines and is intended only for researchers working on HE. These low-level routines (e.g., SHIFT, ADD, MUL, etc.) are considered as the assembly language for HE. It is written in C++ and uses the NTL mathematical library[4] that is often used with the GNU Multi-Precision library[5] (GMP) for faster multiplications.

The Simple Encrypted Arithmetic Library (SEAL) is an implementation of the HE scheme by Fan and Vercauteren (FV) [21], in which, the security is again based on the Ring Learning With Errors (RLWE) problem like the BGV. RLWE is considered highly secure, and its ciphertext noise growth properties are considered better primarily due to a smaller secret key size.

A notable optimization library using CUDA GPU acceleration is the cuHE [28]. This is designed to boost homomorphic encryption schemes such as the BGV [16], López, Tromer, and Vaikuntanathan (LTV) [29] and Doröz, Hu, and Sunar (DHS) [30], by taking advantage of the greater parallel computing power and high memory bandwidth offered by GPUs. It also takes advantage of an efficient implementation of the bootstrapping algorithm [31] making it an ideal support for our work.

2.5 Related Work in Privacy-Preserving Classification.

Several researchers have worked on the preservation of privacy of data that includes the adoption of homomorphic encryption. In this regards, researchers have tried techniques like differential privacy in [32] and [33]. Differential privacy tries to achieve data privacy by summarising or omitting information. Nevertheless, the wider adoption of homomorphic encryption in neural networks is yet to occur and

[4] http://www.shoup.net/ntl/

[5] https://gmplib.org/

will require developing more user-friendly code libraries and toolsets. In this regard, several research works and codes exist to integrate homomorphic encryption with neural networks.

A parallel work of Elsloo et al. [34] presents a framework on neural network inference on encrypted data. Their research abstracts most of the encryption related details for quick prototyping of an encrypted neural network inference task.

Another parallel work by Dathathri et al. [35] presents another similar framework in the form of a domain-specific compiler for fully-homomorphic neural network inferencing. Their work evaluates the circuit depth required for a neural network and then optimizes the underlying encryption scheme parameters for it.

Another development by Dowlin et al. [1] introduces CryptoNets which demonstrates a levelled homomorphic encryption scheme that is applied in the execution of a trained neural network. In their work the authors applied the levelled-homomorphic encryption scheme of Bos et al. [36], inspired by the original fully homomorphic encryption scheme suggested by Gentry [19]. Their method is applied to the 28×28 images of handwritten digits from the Mixed National Institute of Standards and Technology (MNIST) dataset [7]. In their work, they avoided the heavy bootstrapping method of re-encryption of the original scheme and, instead, they estimated the number of operations required to compute an arbitrary function homomorphically over encrypted data without the loss of accuracy in the final decrypted result. In addition, once this estimation was done for the encryption scheme in NN, the input values are limited to a fixed precision. They have reported an average of 51,000 predictions per hour on an ordinary PC. That is an improvement over similar techniques. This approach has helped to reduce the required processing power and time significantly. However, it is worth mentioning

that their approach has limitations in terms of computing the neural network activation functions and the pooling operations.

Recently, plenty of works in encrypted inferencing like the [37], [38], [39], [40], and [41] are being produced to advance the area of private inferencing using homomorphic encryption.

2.6 Issues in Privacy-Preserving Classification

The classification task can be listed as a set of arithmetic operations along with a few modifications to the neural network. In view of the discussion above, the following issues need to be solved to achieve privacy-preserving classification.

2.6.1 Evaluating Activation Function

The crucial part in evaluating a single neuron is the evaluation of the activation function by using only the multiplication and the addition operations. This is problematic when we do not have support for the division operator.

Functions such as those presented in section 2.2.6 above, however, can be approximated by polynomial expressions using integer values, and do not necessarily need to be expressed in a bitwise manner. Some of the more efficient FHE schemes allow encrypting polynomials that can encode such integer values. The advantage of this approach is that a single ciphertext now contains much more information than just a single bit of plaintext, but restricts the possible operations to arithmetic circuits in these polynomials. Furthermore, these functions are often simple enough such that the expensive bootstrapping procedure can be avoided. Evaluating functions like the logistic sigmoid $\alpha(x) = \frac{1}{1+e^{-x}}$ requires us to approximate it using Taylor series. This is because in the encrypted domain, we cannot raise the value of e to an encrypted number x. So, the normal polynomial terms from McLaurin

series (i.e., a case of Taylor series in the region near $x = 0$) for approximating the logistic sigmoid are stated in equation 2.4.

$$f\ x\ = \frac{1}{2} + \frac{1}{4}x - \frac{1}{48}x^3 + \frac{1}{480}x^5 - \frac{17}{80640}x^7 + \frac{31}{1451520}x^9 - \cdots \qquad 2.4$$

The more the number of terms we use to approximate the function the more accurate will be the result, but the more computational power will be needed.

2.6.2 Number Encoding

Homomorphic encryption schemes support only integers denoted by the set $\mathbb{Z}$. On the other hand, neural networks are considered to work best on real numbers denoted by the set $\mathbb{R}$. The clear mismatch between the integer and decimal datatypes in terms of practical implementation is an issue faced in three separate parts of a neural network, which are:

- Input values for the neural network.
- Optimal weights of neurons after training.
- Output of the activation function after processing each neuron.

One way to overcome the datatype mismatch is by adopting an encoded evaluation mechanism that helps to execute the neural network on integers even if the inputs or the weights are non-integer values.

To allow real numbers as inputs, we can multiply the input number x by 10^d to obtain an integer representation, where d is the number of accurate digits we require after the decimal. However, an issue arises when we want to get our required number back after the completion of evaluation through the network. This is because to get the output back to expected range, we must divide the output by 10^d, and in our case we do not have the divide operation in homomorphic encryptions.

Due to this reason, we can get precise results for only a single layer of an already trained neural network having pre-defined weights. This is because the output of the single layer is much higher in value than expected by the next layer.

This problem demands that we either perform the complete execution of the neural network in binary form (base-2 numbers) using digital logic circuits, or we match the high input values with those of the associated connection weights so that the aggregated input to the neuron permits proper evaluation. This also suggests that we can adjust the optimal weights to match the encoded input.

Moreover, in our case the encoding of input values only will not suffice for the complete execution of the neural network. This is because the output from the first layer are again real numbers, and the culprit now is the activation function. To overcome this problem, we used the Maclaurin Series that give us an approximating polynomial to estimate the sigmoid function. We raise the whole polynomial to get integers as output instead of real numbers. For this, we again multiply the required number of terms from the polynomial with 10^d. It should be noted that in this case the value of d is not necessarily to be the same as in the case of encoding input values.

For example, to make the approximation of a logistic sigmoid function as shown in equation 2.4 to output integers, its terms will be stated as:

$$f\ x\ = \frac{10^d}{2} + \frac{10^d}{4 \times 10^d} x - \frac{10^d}{48 \times 10^{(d \times 3)}} x^3 + \cdots$$

These three terms are tested using HElib [24] and the following observations were made:

- The effective range of input for three terms from the series is from -4 to +4 inclusive. Beyond this range, the output is distorted.

- Comparing the normal evaluation versus the encrypted evaluation of the three terms, when our encoding value for d is 6, the output for the encrypted form reveals that the approximation is giving a correct result within the specified range. For example:

  A normal approximation for an input of value 1 for x will be as follows:

  $$\begin{aligned} &= (0.500000) + (0.250000) - (-0.020833) \\ &= (0.729167) \end{aligned}$$

- Whereas, the same approximation in encrypted form for the input value of 1 for x will be:

  $$\begin{aligned} &= (500000) + (250000) - (200154760) \\ &= (729167) \end{aligned}$$

- The average calculation time for a single neuron in the above evaluation using HElib was close to 1 second on a PC having a processor of Intel Core i7 and 8 GB of RAM.

2.6.3 Interpreting the Final Result

The output values from the neurons in the last layer are the confidence values of the network. These output neurons and their values correspond to the various classes that can be predicted about. Normally the last layer uses the softmax activation function, which provides all the values in probabilities for each class. However, the probabilities are just a representational form to help in the understanding of the output.

There is no direct method to evaluate the softmax function using the existing fully homomorphic encryption schemes. In addition, because the output values represent the confidence of the network for each class, we cannot take the max of all the encrypted values to determine the class. Several ways exist to find the max value for the encrypted results; it can be either of the following:

- Get the max in the binarized form of the encrypted result.
- Send the values to the client to decrypt and get the result against each class.
- Use a multiparty computation protocol to find the maximum of the values.

3 Towards Privacy Preserving Classification in Neural Networks

This chapter presents a multi-cloud non-colluding platform for encrypted inference for a neural network. This chapter is reproduced from our study published in [14] with minor emendations.

3.1 System Goals

To address privacy concerns and evaluate all the operations in a neural network, we propose a non-colluding dual cloud system that utilizes Paillier cryptosystem. We illustrate how our proposal would allow us to compute non-linear functions while preserving privacy. Such an achievement could make our proposed system an ideal solution to use for real world applications.

3.1.1 Secure Outsourcing of Computation

Maintaining a neural network (NN) requires a significant expertise and sophistication in understanding the way the neural network runs and produce results (i.e., the classification). Rather than leaving this burden to the client, our model proposes to outsource the NN computation to a cloud (let us call it a Cloud A).

However, this approach could potentially create a security problem. For example, Cloud A may gain critical unauthorized knowledge (e.g., input data, weighted

measure at different neurons, final classification outcomes etc.) if an appropriate security mechanism is not in place.

Our model provides a number of protection mechanisms in which Cloud A never learns anything that it is not supposed to learn. First, any data passed from the client to Cloud A is always encrypted, and Cloud A cannot decrypt this data by any means. Second, the computation run by Cloud A is only performed on the encrypted data and this effectively removes any possibility from Cloud A snipping into the computation.

3.2 Related Work

An implementation of using FHE for classification tasks was presented by Bost et al. [42] in their library integrating building blocks for construction of classifiers. Their work consists of a set of protocols using homomorphic encryption and multi-party computation (MPC), through which they built three major classifiers, i.e., Hyperplane Decisions, Naïve Bayes, and Decision trees. Their protocols, however, are only designed to accommodate two parties i.e., the cloud with the classifiers and the client with the data. In such a scenario, the client needs to establish a continuous connection and actively participate in the computational process of the classifier.

Orlandi et al. [43] also explored the evaluation of a neural network by using obscurity and mingling of the weights and data transfer through homomorphic encryption among multiple parties. In this, they are leaking much of the information on weights of the network to the other party. To protect the data, they propose the data being sent to the NN is encrypted using the generalized Paillier cryptosystem. The Paillier cryptosystem they used is a modification from [15]. In this work, the authors have shown that neither the owner of the NN is able to acquire any information from the data inputs nor the client is able to know about the

architecture, hidden weights, and the activation functions of the NN model the cloud runs.

The architecture with two non-colluding clouds has been used in encrypted data processing before. For example, the architecture of two clouds is proposed to cluster encrypted data from multiple parties [44]. The existing work shows that the architecture of two clouds can be used to deal with complex functions over encrypted data. As shown in [45], this architecture is used to check complex query conditions over encrypted databases. The activation functions that we use however are more complex than query conditions in databases since they usually include exponentiation operations and divisions.

However, we have not found any other work that utilises two non-colluding clouds to classify encrypted data with neural networks.

3.3 Autonomous Computation

Further, our model also provides an additional advantage over other similar techniques by allowing Cloud A to run independently without having to request any keys (i.e., those required for cryptographic operations) from the client. Other techniques usually require evaluation keys from the client. This adds another layer of unwanted complexity although it aims to provide security for the data inputs.

3.4 Noise-Reduction

The noise is a random number added into the ciphertext while encrypting, during HE to guarantee the security of the cryptosystems. The noises are typically sampled from a distribution (e.g., from a set of polynomials with coefficients in {-1, 0, 1}).

The level of noise generated is proportional to the number of homomorphic encryption computations (i.e., additions and multiplications). As the number of neurons (and hidden layers) in Cloud A increases, the noise level will increase as additional hidden layers participate in the homomorphic encryption operation. The level of noise introduced by the NN will become too large if the number of neurons grows too large. At some point the decryption will no longer work as the amount of the noise will exceed the maximum value set by the underlying cryptographic system. To resolve this, Cloud A requires a way to minimize the number of HE operations to be carried out by itself and to share it with another entity in a secure way.

This leads us to the model of a dual cloud, that is, two clouds working together to run homomorphic encryption operations at different stages. With the dual cloud model we propose, Cloud A only runs HE that relates to input data to a neuron while the additional cloud (let us call it a Cloud B) runs homomorphic encryption relating to the output results of each hidden and output layer.

The input values required for Cloud B to compute the outputs, as a result of running an activation function, are encrypted. The Cloud B cannot learn anything from the activation function even though it receives the input results. Cloud A cannot learn anything from the outputs received from the Cloud B as the outputs are encrypted using the key that Cloud A does not know.

3.5 Our Proposed System

The preliminary knowledge for our work is given in 2.2 and 2.3.1. Here, we discuss the details of our proposed system.

3.5.1 Overview

The proposed system is depicted in Figure 3-1. Our system is comprised of three parties: the client, Cloud A and Cloud B. The client owns the data. When the client wants to run a NN for a classification task, it outsources Cloud A to run the neural network-based classification on behalf of the client. Cloud A then collaborates with Cloud B to produce the classification results in a secure way. The classification result is finally then passed back to the client.

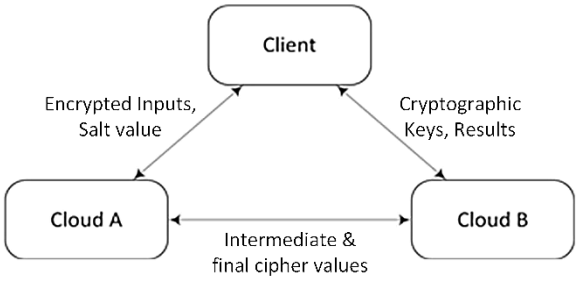


Figure 3-1 Architecture of our dual-cloud approach [14]

3.6 Components

3.6.1 Client

In our proposed system, the client is the data owner who outsources the neural network-based classification tasks from a cloud service.

The client as the data owner has the full knowledge of the data, and its class (e.g., data attributes such as ages, salaries, zip code etc.). When the data owner wants to know a classification of an input, the data owner has the knowledge about the attributes the client needs to feed into the neural network.

Before the feed, all attributes are encrypted homomorphically using the underlying Paillier cryptosystem with a secret key shared between the owner and Cloud B.

This is done so that the computational run at the NN is being processed over the encrypted data.

After sending the encrypted input, the client now waits for the result (i.e., classification). Upon receiving the result, the client decrypts it using the shared secret key and knows the classification of the input value it fed earlier to NN.

3.6.2 Cloud A

Cloud A is an outsourced cloud service where the actual NN model resides. The complete architecture of the NN, which has a pre-determined number of neurons in a pre-determined number of hidden layers, is only known by Cloud A. For each neuron, the corresponding weights are also known only to the cloud A. It may fine-tune these from time to time.

The only thing known publicly about the neural network is the total number of inputs and their expected input type, and the total number of outputs.

We assume that this neural network has been pre-trained. This implies that the overall neural network architecture and the individual weights are set before it has started providing the classification service to the client. The client benefits from the technical expertise of the up-to-date and pre-trained neural network.

For our model to work, however, Cloud A has to make a number of modifications to a conventional neural network processing mechanism in order to provide the service in a privacy-preservation way. The modifications made by Cloud A for our proposal are as follows:

- A change is made in the calculation of each perceptron. The weighted sum of all the inputs to each neuron in every layer is done homomorphically.
- The homomorphically encrypted weighted sum of each neuron in every layer is sent to Cloud B for applying an activation function.

- A random number, called the salt in cryptographic community, is added homomorphically to each of the final outputs before sending it to Cloud B to prevent Cloud B from guessing the value therefore reducing the attack surface.
- Encrypted outputs from the last layer are sent to Cloud B. At this stage, the last output is posterior probabilities that cannot be read by the client directly. Cloud B runs another and final Softmax activation function to calculate the output in a better readable form on behalf of the client.

3.6.3 Cloud B

There are number of distinct functions performed by Cloud B. To start with, Cloud B generates a shared key for each client it is interacting with. The client uses the shared key to encrypt the data before sending it to Cloud A. Now Cloud A cannot decrypt the data sent by the client, as it does not have the key to decrypt. Different clients will have a different key.

During the execution of the neural network operation at Cloud A, Cloud B receives the homomorphically encrypted weighted sum of each neuron from each layer. Cloud B decrypts the encrypted weighted sum as it holds the shared secret key exchanged with the client earlier. After the decryption, Cloud B runs an activation function. Cloud B encrypts the result of the activation function using the shared secret key and sends these back to Cloud A. These encrypted outputs become a new input feed for the next neuron to run in the hidden layer.

After the final output layer at Cloud A, the final output values are sent to the Cloud B to produce the final result (i.e., classification). The result is encrypted again using the shared secret key and sent to the client. The client decrypts the result using the shared secret key and thus knows the classification value.

3.7 System Design

3.7.1 Feeding Data Encryption

Each input (i.e., different data attribute) is encrypted to prevent any unwanted access (e.g., man-in-the-middle who may eavesdrop the communication between the client and the cloud). For that, the client first obtains a shared secret key from Cloud B. Here we assume that the client and Cloud B run the key exchange based on [46] to produce ephemeral shared secret keys. Using the shared key, the client now encrypts each attribute using a homomorphic encryption scheme according to the underlying cryptographic algorithm implementation (i.e., Paillier cryptography). The data is homomorphically encrypted so that any subsequent computations can be done on an encrypted data without having to decrypt it first.

After encrypting all different data attributes, the client combines the encrypted inputs as an indexed cipher data array. This cipher data array is then encrypted once again using the public key of Cloud A as a whole, independently from the earlier encryption. The encrypted cipher array is then sent to the Cloud A where it is decrypted to the indexed cipher data array using its private key. This array acts as the input to the neural network.

3.7.2 Homomorphic Weighted Sum

The basic unit of a neural network is a perceptron as described earlier. The perceptron is interconnected using weighted connections in such a way that the output value from one perceptron is the input value to each perceptron in the next layer shown in Figure 3-2. The number of layers between the input layer and the output layer are called hidden layers and are determined solely by Cloud A.

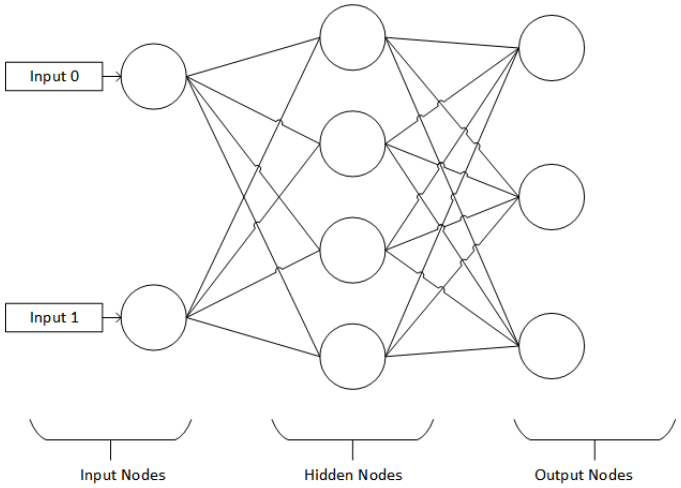


Figure 3-2 A neural network with one hidden layer [14]

In Paillier cryptosystem, the addition operation is homomorphic. Consider two ciphertext messages $[\![m_1]\!]$ and $[\![m_2]\!]$, where $[\![\ ]\!]$ indicates the encryption and the value inside is the plaintext value. The function Dec shows the simplified decryption algorithm. The underlying cryptographic system computes as a sum of both the ciphertexts by multiplying $\bullet$ as:

$$Dec_{sk}\ [\![m_1]\!] \bullet [\![n_1]\!]\ = m_1 + n_1$$

The multiplication operation is done homomorphically over encrypted data by simplifying the same addition property as:

$$Dec_{sk}\ [\![m_1]\!]^{n_1}\ = Dec_{sk}\left(\prod_{i=1}^{n_1} [\![m_1]\!]\right) = m_1 \times n_1$$

m_1 is encrypted and n_1 is not encrypted. In our case n is the weight of the connection to the perceptron. The Cloud A already knows this weight value so there is no need to encrypt it. In other words, $[\![m_1]\!]$ is added to itself n_1 times. Moreover, combining the above two operations we get the weighted sum for each perceptron using:

$$Dec_{sk}\left(\prod_{i=0}^{t} [\![m_i]\!]^{n_i}\right) = Dec_{sk}\left(\prod_{i=0}^{t}\prod_{q=1}^{n_i} [\![m_i]\!]\right) = \sum_{i=0}^{t} m_i n_i$$

Here m is the input value to the perceptron, n is the corresponding weight, and i is the index of the connection with t, the total number of connections.

3.7.3 Activation Functions

The weighted sum is sent to Cloud B to calculate the activation function. Note that at this stage, the weighted sum is a ciphertext and needs to be decrypted for any further processing. Cloud B does the decryption using the key it created earlier.

We assume that the activation function that is to run has already determined been by Cloud A although it does not run it due to concerns with the computational overhead and the possibility of increasing noise. To minimise the number of addition and multiplication operations, Cloud B runs the Activation function.

By sharing neural network execution, Cloud B is freed from computational overhead concerns hence is flexible to support different activation functions depending on the output requirements of applications. Common activation functions are given in section 2.2.6 above.

3.7.4 Dealing with the Output Layer

In determining the class of an unknown input dataset, the output values from the last layer need to be interpretable categorically as posterior probabilities. Thus, these values need to be between zero and one, and their sum equal to one. To serve this requirement, Cloud B needs to run the softmax function on behalf of the client.

As these values could give too much information as to what the final classification is, the values must be only known to the client and neither to Cloud A nor to Cloud B. Cloud A, however, cannot guess what these values could mean because the values are already in ciphertext. To protect from Cloud B guessing, the client adds a random salt to each of the output values homomorphically and then sends them to

Cloud B. Cloud B applies the softmax function. The result is encrypted using the shared key and sent to the client.

3.7.5 Decrypting Classification

The final values at Cloud B will be the probabilities of each attribute. These values are sent to the client for further processing. The client decrypts them using the shared secret key it created with Cloud B earlier. Once decrypted, the client will have access to final values of each class it requires. The attribute with the matching maximum value will be the class for a final classification.

3.8 Security and Correctness Analysis

Our solution is effective against honest-but-curious adversaries (also known as passive adversaries). In this model, we assume adversaries are honest and follow the security protocol. They do not try to gain any other information other than which can be deduced from the protocol (e.g., they do not have the ability to steal any cryptographic primitives such as secret keys or salts). However, they do have an intention to peek into or store the private information (i.e., input and output feeds to NN) for their own gain. The following table illustrates what can be known or unknown to the adversaries if they have access to the major components of our system.

Table 3-1 Adversarial Matrix [14]

	Known	**Unknown**
Client	• Input Data • Number and data types of Inputs for the neural network. • Number and data types of Outputs from the neural network. • Secret Key for encryption and decryption.	• Weights and architecture of the neural network.
Cloud A	• Weights and architecture of the neural network.	• Input Data • Secret Key for encryption and decryption
Cloud B	• Secret Key for encryption and decryption	• Weights and architecture of the neural network. • Input Data • Number and data types of Outputs from the neural network.

We illustrate different defence mechanisms we have in place in our proposed system to thwart the adversaries who may gain access on different components of our system.

3.8.1 Attack on Communication between the Client and Cloud A

We assume that the adversary has gained access to the communication channel between the client and the Cloud A where client's input is being sent. Even if the adversary obtains the input data somehow, access to the data is denied because the data is encrypted and the adversary cannot decrypt the data, as it does not know the secret key the data is encrypted with.

Further, the adversary does not know the neural network model and the weights. Therefore, it is not possible for the adversary to know the final classification even if the input data is intercepted and decrypted.

3.8.2 Attack on Cloud A (or dishonest Cloud A)

We presume that either an adversary has taken over the Cloud A (e.g., using a malware infection) or Cloud A is a dishonest entity which tries to gain an unauthorized access to the client's data for personal gain.

Fortunately, the adversary cannot tap into the data because the data sent to the Cloud A is encrypted. This data is never decrypted at Cloud A because Cloud A (or the adversary) does not possess the key the data is encrypted with.

Even if, at this stage, the adversary knows the weights at each node and the architecture of the neural network model, it cannot guess the eventual classification as it does not have the input. In addition, the actual weighted sum between the weights and input values is done homomorphically and that makes unauthorized access almost impossible.

The dishonest Cloud A knows which activation function to run and therefore may attempt to run weighted sums as well as activations by itself without sharing them with Cloud B. However, if a fully homomorphic encryption scheme is used, then Cloud A may generate too much noise and at some stage Cloud A would not be able to run homomorphic operations.

3.8.3 Attack on Cloud B (or dishonest Cloud B)

We presume again that either an adversary has taken over the Cloud B or Cloud B is dishonest.

The available data to the adversary (or dishonest Cloud B) is either the intermediate result or the final array of output values from the output layer of the neural network.

The intermediary value is of no use to the adversary as it is only used as another input to the neural network that is run at Cloud A. Without having knowledge of the weights and the neural network model (like Cloud A), the dishonest Cloud B would be unable to obtain the classification information.

The final array of output values could hint to the adversary what the eventual classification could mean. We protect against this by adding a random salt to output values before they are sent to the Cloud B. Now the adversary does not know the exact value as this already has been garbled.

3.8.4 Attacks on Communication between Cloud A and Cloud B

Let us presume the adversary is the man in the middle between Cloud A and Cloud B and has intercepted the data exchanged between the Clouds.

What happens here is Cloud A sending the weighted sum of each neuron to Cloud B. Even if the adversary intercepts the weighted sum, it cannot access the data as it does not hold a secret key to decrypt the data being intercepted.

Another data transfer occurs when Cloud B sends the activation result back to Cloud A. If the adversary intercepts this result, it cannot access it because the result is encrypted using the secret key that only the client (and Cloud B) knows.

3.8.5 Communication between Cloud B and Client

Let us presume the adversary is tapping the communication between Cloud B and the client.

At this stage, Cloud B sends the results of the final activation function. If this result is intercepted, the adversary cannot decrypt it because it does not possess the secret key it needs. Further, the adversary would never know the true value of the classification because the result is garbled and only the client knows the salt.

3.9 Findings and Summary

In this chapter, a different approach of using non-colluding dual-cloud in evaluating a privacy-preserving neural network was given as part of this research [14]. The approach works by segregating the activation function to a second cloud in such a way that it will be computed in plaintext, whereas the first cloud hosts the weights and the topology of the neural network. The first cloud receives encrypted inputs from the client by using a Paillier Cryptosystem which is an additively homomorphic scheme. The non-colluding nature of the two clouds and the computational mechanism of the neural network provides results after preserving the privacy of the client input data. Such an arrangement is very effective in obtaining classification results without compromising on privacy. However, the results will show that it is relatively slower.

By using a dual cloud, one cloud solely handles input data to each node in the neural network while the other cloud handles the output of the neurons. Because of this work divide, the clouds do not suffer from heavy computational overheads. In addition, when a fully homomorphic encryption scheme is used, our system provides a way to control the noises in intermediate cipher texts.

The notable insight for an implementation within a LAN connection concerned the overall computing time of the neural network for different encryption key lengths. This is shown in Figure 3-3. For this experiment, the Mixed National Institute of Standards and Technology (MNIST) dataset [47] was used to provide handwritten

digits for the classification task. The dataset that was chosen contains digits in the form of labels along with pictorial representations in images of 28×28 pixels. It contains 60,000 images for training and 10,000 for testing.

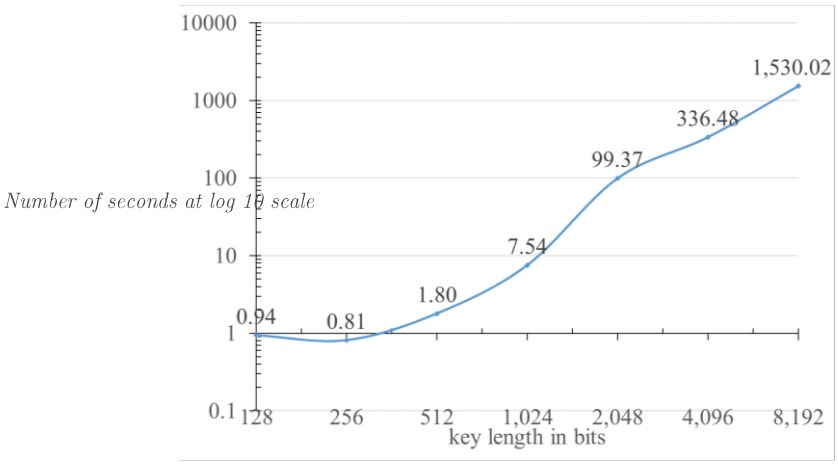


Figure 3-3 Total Roundtrip Time until classification results against different key length

4 Parameter Selection for Homomorphically Encrypted Neural Networks

Dowlin et al. [1] showed us how to perform secure and privacy-preserving inference for a convolutional neural network using a single server setup. They called their setup as CryptoNets. However, the study lacked generality and focused only on a single architecture of a convolutional neural network. Although extensive research has been carried out on using homomorphic encryptions in neural networks, much of the research up to now is descriptive in nature. To cover this gap, this chapter focuses on generating the knowledge for obtaining secure inferences from a wide variety of convolutional neural networks. This is achieved by deriving comprehensive mathematical formulas for selecting the encryption parameters for evaluating a convolutional neural network homomorphically. These mathematical relations are then used to show the concrete effects of changing any of the important parameters for either the encryption scheme or the neural network under consideration.

4.1 Motivation

The beneficiaries of this work are researchers in the fields of machine learning and homomorphic encryption, as well as application developers that need the practical knowledge. These beneficiaries need a clear understanding of the effects of integrating homomorphic encryption in their solutions involving convolution neural networks.

Based on the understanding our work provides, AI researchers can create better convolutional neural network models and structures for encrypted classification of data. Likewise, homomorphic encryption researchers obtain a clearer picture of how to develop improved homomorphic encryption schemes for a higher-level application like convolutional neural network. Similarly, application developers can easily transform an existing conventional convolutional neural network solution into an encrypted convolutional neural network by using our work.

Our method can be extended to all the code libraries implementing homomorphic encryption in a similar way. Therefore, application developers gain benefit from our work with the knowledge of what to expect when they change the underlying homomorphic encryption library.

4.2 Major contributions

In general, this study provides important insights for advancing our knowledge of private and privacy-preserving computations in an untrusted environment. More precisely, this study makes an original contribution through the following important aspects:

1. A clear understanding for practical encrypted classifications in a single cloud environment is provided that supports a wide variety of CNN models.
2. Deriving mathematical formulas to relate a CNN model with the BFV HE parameters. This method is extensible to all the major homomorphic encryption schemes.
3. Providing a method to find the minimal theoretical homomorphic encryption parameters for any feed-forward convolutional neural network. This method is extensible to various NN architectures of feed-

forward propagation. In addition, our noise estimation is better in performance than the standard estimation of SEAL given in [48].

4. Showing the effects of varying the parameters of the homomorphic encryption scheme and the neural network on each other.

4.3 Related Work

Although, several studies have been done to show how one can perform privacy-preserving inferences using a neural network, the issue of setting proper homomorphic encryption parameters for more complicated neural networks have still not been addressed. For practical reasons, not only is this an important issue to address, but it is also a much needed one. The research to date has mainly focused on how to frame the computation of a neural network in the form of encrypted data, or the working of an encryption scheme to support a neural network. This has only been successful in providing theoretical frameworks and protocols, but not in providing the ability to extend the approach to other complex neural networks. To support complex neural networks, the effect of setting proper homomorphic encryption parameters with respect to the neural network has been an open problem.

Among these studies, an important contribution is Microsoft's CryptoNets [1], in which a protocol is given to perform encrypted classification using a convolutional neural network. In their research, they fixed encryption parameters for the network and gave their reasons for selecting them; however, this is not sufficient to be generalized and adopted into other convolutional neural networks. In their work, they primarily showed the SIMD ability of evaluating a convolutional neural network for a batch of input images using the SEAL library [25]. The SIMD (single instruction multiple data) allows us to perform classification of multiple images in

a single execution. Moreover, they trained their network on a bigger convolutional neural network and then merged several of the layers to make the network smaller for encrypted classification without losing any classification accuracy.

The Gazelle [2] framework is a set of two-party protocols through which one is able to perform encrypted convolutional neural network classifications. The underlying encryption scheme used by Gazelle is a packed additively homomorphic encryption (PAHE) and garbled circuits (GC). The two-party computation in Gazelle involves the client to process some parts of the computations. This has shown great improvements in terms of the execution speed over CryptoNets. However, the authors did not provide a detailed account of extending their work to convolutional neural network applications. More importantly, they concluded by stating their future intention to build a compiler that allows the computations to factor into the encryption scheme and two-party primitives to select their encryption parameters according to the convolutional neural network. For building the compiler for selecting the encryption parameters according to a convolutional neural network, this chapter might serve as a helpful resource.

Preceding Gazelle, three similar works involving multi-party computations have suggested similar kinds of protocols for evaluating an encrypted CNN: the MiniONN [49], the SecureML [50], and the DeepSecure [51]. All three have presented related ways of enabling an encrypted classification but the three of them are deficient in the same area, which is the selection of parameters according to the neural network.

In a another line of work similar to CryptoNets, CryptoDL [3] shows improved performance and support for convolutional neural network operations. CryptoDL's main contribution is to provide the mechanism to approximate the lowest degree polynomials representing a function. Their work is also focused on enabling an

encrypted classification efficiently, however, it lacks in providing the specific details of selecting encryption parameters for a convolutional neural network.

In places where research studies address parameter selection for homomorphic encryptions, tend to focus only on the security aspect. Therefore, the knowledge gap of setting homomorphic encryption parameters by considering not only the security but also the efficiency of the neural network needs attention. This issue is highlighted in the following section.

4.4 Problem Description

The convolutional neural networks have been a very important tool in data analysis. They have become the most reliable mechanism of classifying unseen images to known classes with very high accuracy. Therefore, such reliability has allowed convolutional neural network to provide meaningful insights through all sorts of services. However, due to the inherent concern of privacy, people have been reluctant in sharing their private datasets with convolutional neural network-based services. This reluctance exists even if the convolutional neural network services are providing very valuable and meaningful insights from the private datasets. For example, a hospital, serving numerous patients, can have a silo of medical imagery data, but they will be comfortable in sharing those images with a third-party convolutional neural network service to get automated analysis.

This section discusses the problems and provides detail of why the proposed solutions are neither generic nor readily adoptable yet.

4.4.1 Importance of appropriate HE parameters

In the cryptographic community, the setting up of parameters for a cryptosystem is considered a very crucial step [48]. The choice of parameters for a homomorphic

encryption scheme controls several factors like the depth of calculations, execution performance, and the security of the encrypted values. When we want to perform encrypted classification in a convolutional neural network, we must choose the parameters of the cryptosystem by considering all the factors they control.

The security of any encryption scheme is dependent on mathematically hard problems against attackers guessing our secret keys and data. If we fail to consider the attacks against our encryption schemes, then our encryption can be broken by using available attack methods and our valuable data will be exposed. Studies in the literature (for example [52]–[54]) have assessed the encryption algorithms against certain encryption parameters and have shown the scenarios of attackers guessing our secrets. These studies have provided a combination of parameter ranges against all the known attack methods that can be used. By using these secure parameter ranges, we can be sure of the security of our data as well as private processing.

The requirements of the encryption security, and the computational depth, apply to the computation of a privacy-preserving neural network, because we want to enable a secure as well as complete execution of a convolutional neural network privately. In previous years, many frameworks and protocols for privacy-preserving neural networks have been proposed as shared in the related work section. However, researchers have argued that the extension and customization of these approaches to more complicated neural networks is still an open problem.

Among researchers that have posed the problem of extending existing work is Riazi and Koushanfar [55]. These researchers have assessed the current privacy-preserving framework and protocols for deep learning and inference, in a systematic manner. After the systemization of knowledge, they have shared the conclusion that the customization of the privacy-preserving approaches for deep learning and inference

is a viable research direction that is going to open doors for more efficient solutions. Their conclusion rightly points to the customization of such approaches as the foremost important step. The customization of the homomorphic encryption in this regard is twofold. First, develop special purpose encryption schemes for neural networks. Second, utilize the existing homomorphic encryption schemes by selecting appropriate encryption parameters.

To design an encrypted CNN, the "noise budget" (also called circuit depth) requires attention. In the homomorphic encryption literature, noise budget defines the number of maximum successive operations starting from a freshly encrypted ciphertext. Beyond the limit of the noise budget, the resultant ciphertext will result in an incorrect plaintext after decryption as shown in Figure 4-1 below. This is an important subject for our attention before deciding on the design of an encrypted CNN due to an important reason. That is, CNN have all their calculations arranged in the form of successive operations in several layers demanding a strong consideration of the noise budget. However, the noise budget can be "extended" to accommodate the whole of the CNN until the result. The extension of the noise budget is possible by choosing appropriate encryption parameters as discussed in the subsequent section.

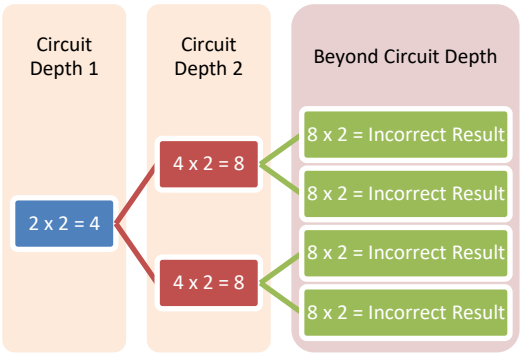


Figure 4-1 A depiction of circuit depth

Figure 4-1 shows that if we use a ciphertext beyond its noise budget (circuit depth) then we get an incorrect answer from the operation. In this depiction, we start by multiplying ciphertexts of value 2 with 2 and get a resultant ciphertext of value 4. In the next step, we use the same ciphertext of value 4 to do another multiplication with 2 and get a ciphertext of value 8. Assuming this is the maximum depth that the encryption parameters support, if we perform another multiplication of the ciphertext 8 with a value, say two, then we will get a garbage value.

A simple experimental evaluation can depict the phenomena of noise budget as follows by setting encryption parameters fixed to a value. We test for the maximum number of operations (also called the depth of the evaluation circuit) supported by performing a successive set of operations on a ciphertext. This test was performed by performing the following operation successively:

$$= \left(\left(\ 2 \times 1 \ \ \times 1 \right) \times 1 \right) \ldots \qquad 4.1$$

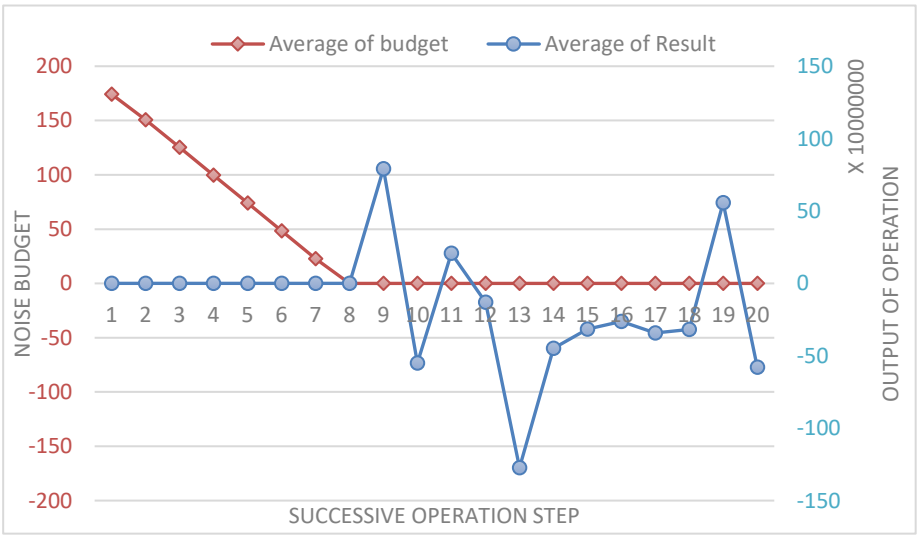


Figure 4-2 Output of an operation before and after the maximum circuit depth of 8 (i.e., noise budget)

Figure 4-2 shows the output of the simple test to check the circuit depth by performing successive multiplication operations as in equation 4.1. Theoretically,

when we subtract 2 from the output there should always be a zero. However, the output of the multiplication after the eighth successive multiplication is always a random garbage number. This is evident from the blue coloured line with round markers. Until step 8, the output is zero according to our evaluated formula, which means that eight successive multiplications is the multiplicative depth in this case.

For a homomorphic scheme, the encryption parameters are the determinants of how many successive operations we can perform using the same ciphertext.

In the FV scheme [21], as implemented in SEAL [22], the current support of arithmetic operations over encrypted data is limited to two operands only. Within this limit, complex operations have to be computed successively. Each intermediate result is used for the next operation. For example, a cube operation of x^3 will require us to perform two separate multiplications as $(x \cdot x) \cdot x$. This limitation is more prominent in convolutional neural networks because of operations, like the weighted sum (calculation of $\sum \omega_l \cdot x_l$), arranged in the form of layers.

4.4.2 Complexity of setting appropriate HE parameters

Moore et al. [56] surveyed the approaches for making homomorphic encryption more practical. After surveying, they concluded that the foremost problem in adopting homomorphic encryption in a real-world scenario is the parameter selection. They have also discussed how complexity arises because each scheme must specifically select parameters based on the existing attacks and the computational limits.

To set up a cryptosystem for any computation, a user must go through an extremely complex process of selecting parameter settings by considering the required computational depth, security, and performance of the executing algorithm. The complexity becomes more prominent in algorithms: like a convolutional neural

network where we have many operations to be performed on values. However, we have a limit on the number of operations that can be performed successively on those values.

4.5 Parameter Selection Method

A very high-level view of our approach can be listed as a 5-step process:

1. Calculate the total number of encrypted operations required to produce a *single* CNN output.
2. Assume an initial set of small encryption parameters for calculating the noise.
3. Based on the initial encryption parameters, calculate the maximum noise threshold of the network using the known theoretical bounds of the encryption scheme.
4. Check the ability of set initial encryption parameters for successfully decrypting a ciphertext with the calculated maximum noise threshold.
5. If the decryption is not successful, then gradually increase the encryption parameters to find the smallest secure encryption parameters sufficient for the maximum noise. Otherwise, provide the set encryption parameters for evaluating the neural network homomorphically.

These steps are detailed in the following sections.

4.5.1 Calculating encrypted operations in a CNN

In order to estimate the encryption parameters, we have to know certain important things about the operations involved in the evaluation of the neural network. The neural network architecture that we will be focusing upon in this study is a convolutional neural network. In this regard, the evaluation of the convolutional neural network needs to be broken down into basic arithmetic operations. This

needs to be done so that we can calculate the total incurred noise for each value after processing. The calculation of the noise is performed by using the noise estimates for basic encrypted arithmetic operations given in the SEAL documentation [22].

After breaking the evaluation into basic arithmetic operations, the operands of the operations need to be categorised as plaintext or ciphertext. A homomorphic operation always outputs a ciphertext, but it may take two ciphertexts or a ciphertext and a plaintext as input. Identifying the operands is necessary because the noise growth in an operation involving ciphertexts is greater than that of an operation involving a ciphertext and a plaintext operand. This will be shown representing the plaintext operand with P and the ciphertext operand with C. For example, if a multiplication involves a ciphertext and a plaintext then it will be written as $C \times P$.

Moreover, after breaking the evaluation into basic arithmetic operations, another distinction about the sequence of the homomorphic operations is necessary. The sequence of homomorphic operations is identified as sequential or discrete. This distinction will identify the accumulation of noise in the case of sequential operations. Fundamentally, any homomorphic operation is a binary operation involving only two operands to produce the result. In the case of many operands, all the operands have to be operated one by one to produce the result. There are functions available in the SEAL library to provide more than two ciphertext values to perform an operation, for example, there is a function to add up many ciphertexts together. However, these functions perform the calculation in the same way by only taking two values at any single time and then loop through the many ciphertexts. Here, we make the distinction of homomorphic operations being sequential or

discrete based on the number of operands. We call those operations involving more than two operands a sequential operation, because if we want to perform an operation involving more than two values, then the operation will be performed sequentially only using two values at any one time. On the other hand, if an operation involves only two values, then the operation is a discrete operation.

This distinction can become clearer after taking the example processing of a single neuron shown in section 2.2 Figure 2-1. This is a single neuron, consider having 3 input values represented by x_i. These input values are multiplied with their corresponding weight values ω_i, and then summed together to form a single output value. A bias value b is then added to this output value, and the result is passed to either the activation function f or to the next layer directly. In here, we can see that there are two arithmetic operations involved to get the output, i.e., multiplication and addition.

In summary, by representing a ciphertext with C and a plaintext with a P, the steps required to process a single neuron will sequentially be as follows:

1. Multiplying with a weight value: $C \times P$.
2. Calculating the weighted sum: $C + C + \cdots$.
3. Adding a bias value: $C + P$.

In the first step, a single input value is multiplied by a single weight value making a total of 2 values for the operation. Therefore, the multiplication part involves distinct operations. After the multiplication, we will get the exact number of weighted-input values as before the multiplication. We already know that the operands of these multiplications are a ciphertext and a plaintext represented as $C \times P$. Here, we take the input values as encrypted and the weight values as plaintext values.

In the second step of evaluating a single neuron, as in section 2.2 Figure 2-1, all the weighted-input values need to be added together to make one value. This summation is a sequential operation and not a discrete operation. It involves adding up three values. This is done by adding the first 2 values, and then adding the third value to the earlier sum of the 2 values. We also know that the operands of these additions are all ciphertexts and so the operation will incur two additions to sum-up three values. The operation is represented as $y = [(c_0 + c_1) + c_2]$.

The third step of evaluating a single neuron involves adding a plaintext bias value b to the weighted-sum. This is a discrete operation involving the addition of a ciphertext with a plaintext. This can be represented by $C + P$. The outcome is either passed to an activation function f or to the next layer for further calculations.

4.5.1.1 Encrypted operations in a Convolution Layer

A convolution layer works on multi-dimensional data as compared to a fully connected layer where only single-dimensional vector data is processed. In the convolution layers, instead of neurons, we have filters processing multi-dimensional data such as images. A single filter takes a subset of input values across the width and height of an image. However, an image can have more than 1 matrix of pixels. The matrices are called the depth channels, which in a normal RGB image, there are 3 channels each for the red, green, and blue colours. The channel depth will be represented by d hereon. A filter can take values across all of the channels, or it can process through the channels one by one. If the movement of the filter is specified in three axes, then it is called a 3-D filter. Similarly, if the movement of the filter needs on two axes, then it is called a 2-D filter. Internally, the filter in a convolution layer performs its execution in the same way as a neuron in a fully connected layer.

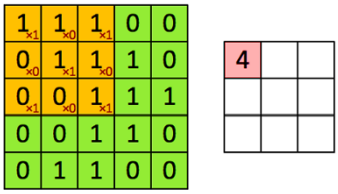


Figure 4-3 Processing of a filter in the convolution layer

The nature of the operations involved in a convolution layer can be analysed by using the example shown in Figure 4-3. For each value in the output matrix, the filter works on a specific area from the input matrix. This area for each filter is determined by the filter width and height represented by F_w and F_h, respectively.

In Figure 4-3, the green coloured matrix shows the input matrix having a width and a height of 5×5. The yellow region in the input matrix represents the filter that has a width and a height of 3×3. There is only a single *Input Channel* in this example; therefore, the filter processes a total of nine values from the input matrix in a single step. Every step produced a single value in the output matrix, shown on the right-hand side of Figure 4-3, having width and height of 3×3. The filter is moved a single pixel at a time from left to right and top to bottom to process the next input region. This movement of the filter is called the stride value and it is one for this example. For each step, the filter performs the following calculations to get a single output value for the output matrix. Here ω is the weight, x is the input value, b is the bias of the filter, and a is the final outcome from the filter.

$$a = b + \ \omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3 + \omega_4 x_4 + \omega_5 x_5 + \omega_6 x_6 + \omega_7 x_7 + \omega_8 x_8 + \omega_9 x_9$$

We see that there are distinct multiplications between the weight ω and the input x. They are distinct because every multiplication will involve two operands. They are represented as $C \times P$ as we know that the input value is a ciphertext and the weight value is a plaintext.

From these observations, we derive that, for every output of any filter in a convolution layer, the number of multiplications will be the width times the height times the depth of the filter $F_w \times F_h \times d$.

Furthermore, the additions will be sequential because they involve more than two operands to produce a single output. In the example, there are eight addition operations, each involving 2-ciphertext operands. Therefore, there will be a total of $(F_w \times F_h \times d) - 1$ addition operations for every output of each filter.

Once we obtain the weighted sum, we have to add a plaintext bias value b to get the output of the filter. This will be a single distinct operation involving a ciphertext and a plaintext.

Therefore, for a single encrypted output, a 3-dimensional convolution layer will have the following numbers of homomorphic operations for each filter. In other words, a single ciphertext output from a 3-dimensional convolution layer will be acquired after the following processing steps.

- Number of $C \times P$ operations is $F_w \times F_h \times d$.
- Number of $C + C$ operations is $(F_w \times F_h \times d) - 1$.
- Number of $C + P$ operations is 1.

These operations will be carried out by taking the depth d of the input data as the channels so a single filter will be performing processing on a multi-channel input and transforming it into a single channel output. However, when we want to perform a two-dimensional convolution on a per channel basis, then the number of input channels will only be 1 instead of d. This is because each filter is performing operations on each channel independent of the others.

After understanding the encrypted operations in a convolution layer, the important observation here is the precise computations required for an encrypted convolutional neural network.

From a different perspective on the processing of a convolution or pooling layer, it could be argued that the total number of operations performed on each pixel are different. This perspective is based on the pixel location in the input matrix. From the perspective of an input matrix, the edge pixels get used fewer times in the calculation of the outputs than the centre pixels. This phenomenon of the convolution method is an intrinsic trait already known in the machine learning community. However, this phenomenon does not change our calculations since our study already takes the maximum count for an operation to occur in a layer, therefore this is not significant. The preference for using the maximum count is due to the reason that the encryption parameters must be set according to the output ciphertext with the highest number of operations. In this way, all other ciphertexts with lesser a number of operations will automatically be accommodated.

4.5.1.2 **Encrypted operations in a Fully Connected Layer**

In the fully connected layer, we have to know the number of operations that result in an output ciphertext so that we can calculate the homomorphic noise. We represent the total number of input values as z_i, and the total number of neurons in the layer as z_o, then the number of distinct multiplications in the layer will be $z_i \times z_o$. A fully connected layer will always have the following number of homomorphic operations to provide a single output ciphertext. Here, P is a plaintext, C is a ciphertext, z_i is the number of inputs, and z_o is the number of outputs.

- Number of discrete $C \times P$ operations is $z_i \times z_o$.
- Number of sequential $C + C$ operations is $z_i - 1$.
- Number of $C + P$ operations is 1.

In this section, the important aspect to consider for our work is the total number of additions and multiplications. In a fully connected layer, the number of multiplications performed will be $z_i \times z_o$.

In general, fully connected layers are very costly in terms of processing requirement. In comparison to a convolution layer, the FC layer requires a considerably higher number of multiplications and additions for the same number of inputs. If we do not use convolution, and instead feed all the values of an image directly into a fully connected layer, then this makes for a very high number of connections to process. A small image having dimensions of 200×200 being fed to merely a 10-neuron layer will result in $200 \times 200 \times 10 = 400{,}000$ interconnections, just for the single layer. However, if we use a convolution layer with a single filter of size 5×5 with a stride 2 then the same 200×200 input will be reduced to a 98×98 input to the fully connected layer. Such a reduced input will result in $98 \times 98 \times 10 = 96{,}040$ interconnections, which is a significant reduction of 76%. Due to such a high number of interconnections, the fully connected layer is used at the end of a convolutional neural network where the number of values has decreased dramatically from the earlier values, so that the network can learn or process in an efficient manner.

4.5.1.3 **Encrypted operations in Pooling Layers**

A pooling layer, in general, performs sub-sampling of the incoming values and reduces the dimension. In a *sum-pooling* layer, a region in each channel of the input matrix is summed to use a single value in its place. A simple sum pooling operation can be seen in section 2.2.5 Figure 2-5 where we have a 4×4 input matrix reduced

to a 2×2 output matrix. The filter size is 2×2 which means that four values are summed at every step to make a single output value. The filter has stride 2 which means that the filter is moved 2 pixels from left to right or top to bottom. For each step, the filter performs the following operation where the x represent the input and the a is the output.

$$a = \ x_1 + x_2 + x_3 + x_4$$

We observe that only sequential additions are required between ciphertexts. The total number of these additions are determined by the size of the filter. Therefore, we obtain the total number of operations to produce a single output value in a sum-pooling layer as:

- Number of $C + C$ operations is $(F_w \times F_h) - 1$.

Similarly, an *average-pooling* layer performs sub-sampling of the incoming value and reduces the dimension in the same way as a sum pooling. In an average-pooling layer, a region in each channel of the input matrix is summed-up and then divided, by the total number of elements, to produce a single value in its place. For each step, as in the example in the previous section, the filter performs the following operations, where the x represent the input and the a is the output,

$$a = \frac{x_1 + x_2 + x_3 + x_4}{4} = \ x_1 + x_2 + x_3 + x_4 \ \times 0.25$$

We observe that only sequential additions are required between ciphertexts and a final multiplication. The total number of these additions to produce a single output are determined by the size of the filter. Therefore, we obtain the total number of operations for each output value in an average-pooling layer are:

- Number of $C + C$ operations is $\ F_w \times F_h \ - 1$.
- Number of $C \times P$ operations is 1.

4.5.1.4 Encrypted operations in Activation Layers

The square function is the smallest non-linear function where we compute the square of a number x [1]. When we want to perform the square activation function on a homomorphic encryption, we multiply the ciphertext with itself to obtain the output. This is mathematically represented below, where a is the output and x is the input,

$$a = x^2 = x \cdot x$$

We can see that there is a single multiplication involved. For which, both the operands are the same ciphertext. Noting that both the operands are the same ciphertext is important, and this will be discussed when we consider the calculation of noise for the square activation layer. For each output from the square layer, the homomorphic operation required is:

- Number of $C \times C$ operations is 1.

In this section we have laid out the exact number of operations required to produce a single output for a feed forward convolutional neural network. Next, we will describe the process to find the noise threshold based on the number of operations that any ciphertext goes through in the convolutional neural network under consideration.

4.5.2 Estimating the total noise in a CNN

Let us recall the properties of noise as implemented in the SEAL library for the FV scheme. There is an acceptable range [25] for the growth of inherent noise v_{inh} which is characterized by the ciphertext modulus q and the plaintext modulus t shown in Lemma 35 of Player [57] as:

$$\|v_{\text{inh}}\| < \frac{\text{q}}{2\text{t}} - \frac{\text{t}}{2}$$

The noise v_{inh} grows in a known manner whenever a ciphertext goes through homomorphic operations. Eventually, when the noise has grown more than the limit, then it will not be possible to decrypt the ciphertext to a correct value as shown with the example in section 4.4.1 Figure 4-2. Therefore, the noise is a key determinant for the proper integration of the convolutional neural network and the homomorphic encryption.

Although we know the growth factor for the noise term after a homomorphic operation, it is still random. The SEAL documentation [22] has shared mathematical formulas for calculating the maximum incurred noise value for all of the homomorphic operations. Based on these formulas, we know the maximum noise term for an operation like multiplication or addition.

To calculate the maximum noise, we have to understand the relationship between the noise and the encryption parameters. The determination of maximum noise for the output of any homomorphic operation is determined by the encryption parameters that encrypted the input ciphertext operands. This means that, to know the maximum noise of an operation, we have to know the encryption parameters first. Figure 4-4 below, provides a visual summary of the relationship between the noise and the encryption parameters. For a homomorphic operation, with each variation of the encryption parameter set, the maximum noise in the result will differ. Not only the maximum noise, but also the initial noise and the rate of noise growth in a ciphertext are affected by the encryption parameters. The maximum noise value is a key factor in deciding the correct encryption parameters.

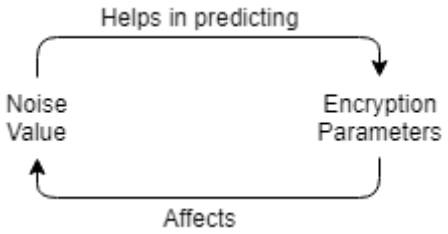


Figure 4-4 Relationship of Noise value and encryption parameters

In our approach, we always know the exact nature of operations, as detailed in 4.5.1 above. Therefore, here, we will be using the noise estimation formulas of SEAL to find the smallest encryption parameter set, which can decrypt the maximum possible noise for a network.

We have to remember that there is an initial noise in each of the ciphertexts. This initial noise is estimated based on the given formula for SEAL reproduced in Appendix A:

$$v_i = \frac{q \bmod t}{q} \cdot \|m\| \cdot N_m + \frac{7nt}{q} \cdot \min \begin{matrix} \text{noiseMaxDeviation}, \\ 6 \times \text{noiseStandardDeviation} \end{matrix} \qquad 4.2$$

From the above formula, that is explained in section 2.3.10, we can calculate the initial noise v_i for any ciphertext. We will use the formulas for different kinds of layers to calculate the total noise incurred in the output ciphertext. After their derivation, the formulas for each of these layers will be summarized later in Table 4-6.

4.5.2.1 Estimating noise in Convolutional and Fully Connected Layers:

The noise calculation formulas for both the convolutional layer and the fully connected layer are the same. The reason is that the nature of operations in these two layers is the same. The key difference between these two layers is the number of inputs that are processed at a single instance of the calculation. When we are working with the fully connected layer, we take all the incoming values, represented as the number of inputs i, and process these in a neuron. Whereas, when we are working with a convolution layer, we only take a subset of all the incoming values and process these through a filter.

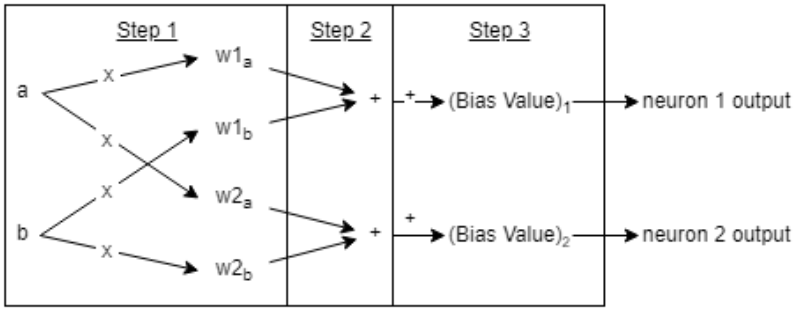


Figure 4-5 Operations in a fully connected layer

The noise calculation can be understood using an example. Consider a two-neuron fully connected layer having only two inputs a and b. If we want to calculate the noise for this layer, then we will require the operations shown in Figure 4-5.

Since there are two neurons, the calculations of the neurons are independent of each other as they work on the inputs separately. The operations that will be performed by a neuron, as shown in the Figure 4-5, are as follows:

1) Multiply input a with the weight $w1_a$ and multiply input b with the weight $w1_b$ to get weighted a and b for neuron 1.
2) Add the resultant weighted a and weighted b of neuron 1 together.

3) Add a plain bias value to get the output.

In the same way, the operations are performed by neuron 2 for its corresponding weight values independently of neuron 1, as shown in Figure 4-5.

Therefore, in total, there are eight operations for a two-neuron layer having two inputs and two outputs. The sequential calculation of a neuron in terms of the nature of homomorphic operations can be stated as:

$$[\ C_a \times P_a\ +\ C_b \times P_b\ ] + P_{bias}$$

Here C represents the encrypted ciphertext of the inputs a and b, respectively, and the P represents the plaintext values for corresponding weights and the bias value. The same sequential calculation will be performed by neuron 2 but independently of any interaction with the calculation of neuron 1. The only difference between neurons exists in their weight values, for which we know the maximum limit. Therefore, the maximum noise for both neuron 1 and neuron 2 will be exactly the same, so we can calculate for either of them and it will be equally applicable to both.

For a single neuron, we can enlist the formulas for noise calculation from Table 4-1.

Table 4-1 Calculating the total noise for a fully connected / convolution layer.

Step	Description	Nature of operation	Maximum Noise Estimate
1	Multiplication of an input with the weight value	$C \times P$	$N_m \lVert m \rVert v_i$
2	Summation of all the weighted inputs	$C + C = \sum_{k=1}^{i} C$	$v_3 = v_1 + v_2$
3	Addition of the bias value	$C + P$	$v_o = v_3 + \frac{q\ mod\ t}{q} \times N_m \times \lVert m \rVert$

In Table 4-1, we have the formulas for estimating the maximum noise incurred after the corresponding homomorphic operation in a fully connected or a convolution layer. After these three steps in the layer, the maximum noise estimate v_o will be the final noise for a single neuron / filter. Here the noise v_i represents the initial noise already present in the input ciphertext C. When we multiply the initial noise v_i with the N_m and the $\|m\|$ (the highest degree of the polynomial representing our message m, and the biggest number that the ciphertext can represent, respectively), we get the noise estimate for the $C \times P$.

In the second step in the layer for performing the $C + C$ estimation, we have i ciphertexts. To estimate the maximum noise, we have to add their corresponding noise values v as shown in Table 4-1. In our example, we have 2 inputs, so we will have 2 different ciphertexts having noise v_1 and v_2 each. By adding these 2 we will get the intermediate maximum noise v_3. The final noise of the output ciphertext of a single neuron will be known after the estimation in step 3.

To make the process efficient, we can skip the calculation values and still be able to get a good noise estimate. Since we are estimating for the maximum noise, once we calculate noise for the biggest number in our input, then other smaller numbers can also be processed homomorphically within the same homomorphic parameters. This estimation for the first step of multiplying the ciphertext with the weight value, can be performed on all the inputs independently of each other. Thus, out of all the inputs, the maximum number can represent all other smaller input numbers in the noise calculation. Therefore, if we calculate noise in the first step for only the maximum input number and the maximum plaintext weight value P, then all the smaller numbers will also be accommodated.

Based on the discussion above, we make the process of finding the maximum possible noise in a fully connected / convolution layer significantly efficient. We

calculate the first step estimating $C \times P$ only once, using the maximum possible C and P. This will be sufficient for all the inputs i and so will significantly improve estimation performance.

After the first step, the second step estimates the noise for only the biggest input value and weight value. Therefore, for step two in the layer, we can add this estimated noise to itself i times. This will give us the maximum noise for a neuron / filter. This is summarized below.

We know that the nature of homomorphic operations in a fully connected or a convolution layer will be:

$$[\ C_a \times P_a\ +\ C_b \times P_b\ + \cdots +\ C_i \times P_i\ ] + P_{bias}$$

where the total number of summands is equal to the total number of inputs i. In our example of two inputs, this will be $C_a \times P_a\ + (C_b \times P_b)$ for inputs a and b. However, we know that if we calculate for the biggest input number and weight value, for example a, then it will be sufficient for all the inputs. If we combine the estimation of all the three steps above for the biggest input, then we will obtain the following:

For step 1, we have for each input k:

$$v_1 = N_m \times \|m\| \times v_k$$

For step 2, we have:

$$v_2 = v_1 + v_1 + \cdots + v_1$$

After putting in the value of v_1, for step 2 we get:

$$v_2 = \sum_1^i N_m \times \|m\| \times v_k \qquad \text{or} \qquad v_2 = i \cdot (N_m \times \|m\| \times v_k)$$

For step 3, we have:

$$v_o = v_2 + \frac{q \ mod \ t}{q} \times N_m \times \|m\|$$

After putting in the value of v_2 from step 2, we get:

$$v_o = [i \cdot (N_m \times \|m\| \times v_k)] + \frac{q \ mod \ t}{q} \times N_m \times \|m\|$$

After simplification, we obtain the final formula for estimating the total noise in a fully connected or convolution layer as:

$$v_o = N_m \times \|m\| \cdot \left(\frac{q \ mod \ t}{q} + i \cdot v_k\right) \tag{4.3}$$

Where noise v_o will be the maximum noise in all the resultant ciphertexts from the layer, v_i is the initial noise before processing the layer, and i is the number of total inputs to a neuron or a filter.

An important thing to note here is that $\frac{q \ mod \ t}{q}$ is always:

- 1 when $q < t$.
- 0 when $q = t$.
- Some fractional number when $q > t$.

This is important because it means that in general the bigger q is relative to t, the smaller the noise will be. Moreover, it is interesting to see that the derivative of the formula for the estimation is:

$$\frac{\partial}{\partial v}\left(N_m \times \|m\| \cdot \left[\frac{q \ mod \ t}{q} + i \cdot v\right]\right) = i \cdot N_m \cdot \|m\|$$

This shows that, in terms of the rate of change of noise v, the dependence is on the number of inputs i, and the size of the input numbers represented by $N_m \cdot \|m\|$.

4.5.2.2 **Estimating noise in Pooling Layers**

The noise estimation in a sum-pooling layer is based on the same methodology of calculation as the fully connected or convolutional layers. However, the calculation is much simpler. In simple addition of ciphertexts, their noise values are added together to get the maximum noise of the resultant ciphertext. In this layer, we only have $C + C$ operations, with ciphertexts as operands, to be summed. As an example, as shown in Figure 4-6, if we have a 3×3 input to a pooling layer of windows size 2×2, then we will obtain a 2×2 output from the layer. For each output, we will have to add 4 ciphertexts together. The number of ciphertexts to sum is dependent on the pooling window size, which in the case of our example is $2 \times 2 = 4$.

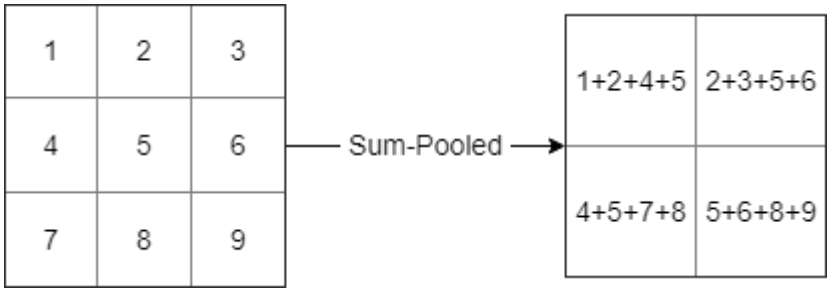


Figure 4-6 Operations in a sum-pooling example

The exact nature of operations in this layer is shown in Table 4-2.

Table 4-2 Calculating the total noise for a sum-pooling layer.

Step	Description	Nature of operation	Maximum Noise Estimate
1	Summation of inputs in the pooling window	$C + C = \sum_{k=1}^{i} C$	$v_o = v_1 + v_2 + v_3 + v_4$

Here, i is the total number of ciphertexts in the pooling window, which in our example is four. In simple addition of ciphertexts, their noise values are added

together. Therefore, the formula for maximum noise estimation is written to accommodate four of the noise values of our four ciphertexts.

For optimizing this estimation, we can take a subset of all the calculations. We take the largest input number and weight value, to get the maximum noise estimate for the layer. We can take the estimation of a single pooling operation in the layer, and this estimate will represent all the incoming inputs because it will be the estimation for the maximum number.

The general formula for the estimation of noise after a sum pooling is:

$$v_o = v_1 + v_2 + \cdots + v_i$$

Which in our case will become as:

$$v_o = \sum_{k=1}^{i} v_k = i \cdot v_k$$

In a similar fashion, the method of noise estimation in an average-pooling layer is similar to that of the sum pooling. In the average-pooling operation of ciphertexts, their noise values are added together to get the maximum noise of the resultant ciphertext, and then divide them with the number of ciphertexts being pooled. Due to the underlying encryption scheme, we are limited to multiplication; therefore, the divide operation is carried out by multiplying with a decimal number.

The exact nature of operations in this layer is shown in Table 4-3.

Table 4-3 Calculating the total noise for an average-pooling layer.

Step	**Description**	**Nature of operation**	**Maximum Noise Estimate**
1	Summation of inputs in the pooling window	$C + C = \sum_{k=1}^{i} C$	$v_5 = v_1 + v_2 + v_3 + v_4$
2	Division by the total number of elements	$C \times P$	$v_o = N_m \| m \| v_5$

Here, i is the total number of ciphertexts in the pooling window, which in our example is four. In simple addition of ciphertexts, their noise values are added together and then the summation is multiplied by the end number. Therefore, the formula for maximum noise estimation is written to accommodate the four noise values of our four ciphertexts.

For optimizing this estimation, we can take a subset of all the calculations. We take the largest input number and weight value, to get the maximum noise estimate for the layer. The maximum noise for all the outputs of the convolution or fully connected layers will be the same because of usually the same homomorphic operations. Since the pooling layers are used after a convolution layer, we can take the estimate for a single pooling window. This estimate will represent all the incoming inputs and will be the maximum.

The general formula for the estimation of noise after average pooling is:

$$v_o = v_1 + v_2 + \cdots + v_i$$

Which in our case will become:

$$v_o = N_m \cdot \|m\| \cdot \sum_{g=1}^{k} v_i = N_m \cdot \|m\| \cdot i \cdot v_k \tag{4.4}$$

4.5.2.3 **Estimating noise in Activation Layers:**

The activation functions are infamous for not allowing straightforward encrypted operation or estimation. This has been due to the use of the exponential function in activation functions. However, researchers such as Dowlin et al. [1] have shown that the square function is the smallest usable activation function in a CNN. Therefore, here, we will estimate the noise for the square activation function.

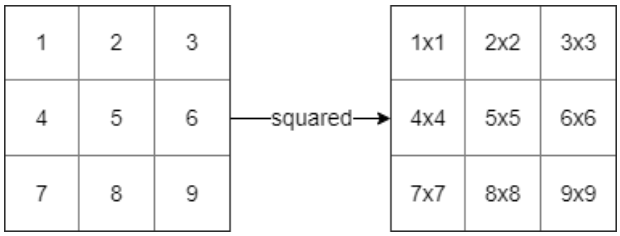


Figure 4-7 Operations in a square activation layer

The method of maximum noise calculation in the square activation layer follows the same method as in the other layers. In a square layer, all the ciphertexts are multiplied by themselves to get the squared result. The same number of outputs will be provided as the number of inputs. For example, Figure 4-7 shows a simple square activation of a 3×3 input that provides the same 3×3 output. However, each output is the result of a multiplication of the corresponding input with itself. Therefore, the homomorphic operations involved in this layer will only be $C \times C$.

Table 4-4 Calculating the total noise for a square activation layer.

Step	**Description**	**Nature of operation**	**Maximum Noise Estimate**
1	Square Activation Layer	$C^2 = C \times C$	$v_o = t\ \sqrt{3n}\ (2^{j+1} \cdot 3^{j/2} \cdot v_i \cdot n^{j/2}\ +\ 12^j \cdot n^j)$ Or if ciphertext size is always kept to 2, then: $v_o = t \cdot \sqrt{3n}\ (144n^2 + 24nv_i) + v_{relin}$

In Table 4-4, we can see the maximum noise estimation formula, reproduced from the SEAL documentation as in Appendix A, for the multiplication of two ciphertexts C_1 and C_2. Beside the encryption parameters t and n, the size j of the polynomial array representing the ciphertext is also required. This formula tells us the final noise v_o based on the previous noise v_i in the ciphertext. Therefore, we can further simplify this estimate.

Based on the following estimation formula for multiplying two ciphertexts, given in the SEAL documentation, we can derive our simplified formula for the square activation layer.

$$v_o = t \cdot \sqrt{3n} \cdot \left[\ 12n^{\ \frac{j_1}{2}} \cdot v_2 + \ 12n^{\ \frac{j_2}{2}} \cdot v_1 + \ 12n^{\ \frac{j_1+j_2}{2}} \right]$$

By replacing both the sizes j and the noises v with the same value, because the ciphertexts are the same, we obtain:

$$v_o = t \cdot \sqrt{3n} \cdot \left[\ 12n^{\ \frac{j}{2}} \cdot v_i + \ 12n^{\ \frac{j}{2}} \cdot v_i + \ 12n^{\ \frac{j+j}{2}} \right]$$

After simplification, we obtain for a ciphertext of size j and noise v_i:

$$v_o = t \cdot \sqrt{3n} \ \left(2^{j\,+\,1} \cdot 3^{\mathrm{j}/2} \cdot v_i \cdot n^{\mathrm{j}/2} \ + \ 12^j \cdot n^j \right) \tag{4.5}$$

If we are keeping the ciphertext size $j = 2$, which is the smallest possible, then this formula can be further simplified by putting the size j as 2 to obtain

$$\boxed{v_o = \mathrm{t} \cdot \sqrt{3n} \ (144n^2 + 24nv_i) + v_{relin}} \tag{4.6}$$

Based on the formulas derived above, we can estimate the maximum noise possible in a square activation layer. This maximum value will be the same for all of the outputs from this layer.

If we intend to keep the size of the ciphertext at two, then we have to relinearize each ciphertext back to size 2 after every $C \times C$ operation. This relinearization step also has an associated noise v_{relin} to add. In the SEAL documentation reproduced in Appendix A, this is estimated as

$$v_{relin} = v_i + \frac{2t}{q} \cdot \min \ B, 6\sigma \ \cdot \ J - K \ \cdot n \cdot \ l + 1 \ \cdot z$$

Here, $\min \ B, 6\sigma$ is the minimum of the maximum noise deviation B and the noise standard deviation 6σ. The J is the current size of the ciphertext and K is the

target size of the ciphertext (which in our case is always 2). $(l+1)$ corresponds to the evaluation key size. Here, z is the number base for the relinearization.

If we perform the relinearization, the final noise of the ciphertext, by combining with equation 4.6, is

$$v = v_o + v_{relin}$$

4.5.3 Estimating the encryption parameters

In this section, we combine the previous two steps and provide the required encryption parameters for a convolutional neural network. First, we calculate the exact number of operations that any ciphertext goes through in a convolutional neural network in section 4.5.1. Then we calculate how much noise would change in each of a convolutional neural network layer, in section 4.5.2.

While evaluating a convolutional neural network, we can trace a single input through to the output and add the noise estimates for each operation along the way. We use the maximum noise that can be incurred for each input value until we get our output. This maximum noise value is going to be very helpful in determining the encryption parameters required for the encrypted evaluation of the convolutional neural network. The maximum noise value, along with the upper bound of the input values, will determine exactly what the encryption parameters must be.

For clarity, and distinction from our work, we note that the earlier versions of the SEAL library [25] also included a noise estimation module for simple operations. However, our work takes the calculation path for only a single ciphertext output and generalizes this over an entire convolutional neural network. Whereas the SEAL module would have simulated the entire convolutional neural network for every input in every layer. This makes the SEAL estimation mechanism less efficient, and

it often crashed because of code bugs. The SEAL estimation module is currently removed from their latest versions of the code.

In the SEAL estimation module, if we were to estimate a convolutional neural network, then all the individual values in all of the layers would be saved along with their history of operations. So, a final noise estimate is calculated based on all the operations. If the parameters are not able to perform the required operation, then the parameters are updated accordingly, and the entire estimation is rerun. At the end, while updating the parameters for each required calculation, if the parameters are able to decrypt all the outputs, then they are selected as the output of the parameter estimation. Otherwise, if the updated parameters are not able to decrypt all of the results, then the parameter estimation fails. Figure 4-8 gives the flowchart of the parameter selection available previously in the SEAL library.

It is also important to know that the SEAL estimation primarily used integer encodings and the default set of encryption parameters provided with the library. This default set of encryption parameters are based on the underlying security of the encryption algorithm. Therefore, this default set of encryption parameters is always limited in its range of suggested parameters for any application. If an application, in our case the convolutional neural network, requires more encrypted operations than those supported in the default set, then the estimation process will clearly fail in acquiring a suitable set of encryption parameters. In essence, if the estimation process cannot find the parameters from the default set, then the estimation will fail in finding a suitable set of encryption parameters. This kind of failure only means that there are not enough combinations of the default parameters in the library, and we are able to modify and add values according to our own needs. To add new values to the default set of parameters, we have to consider the

security implications. SEAL recommends consulting the standardization document [58]. Besides the SEAL library, this standardization document also mentions certain recommended values that can be set for encryption. These recommendations are all based on the literature [59] for better security and execution.

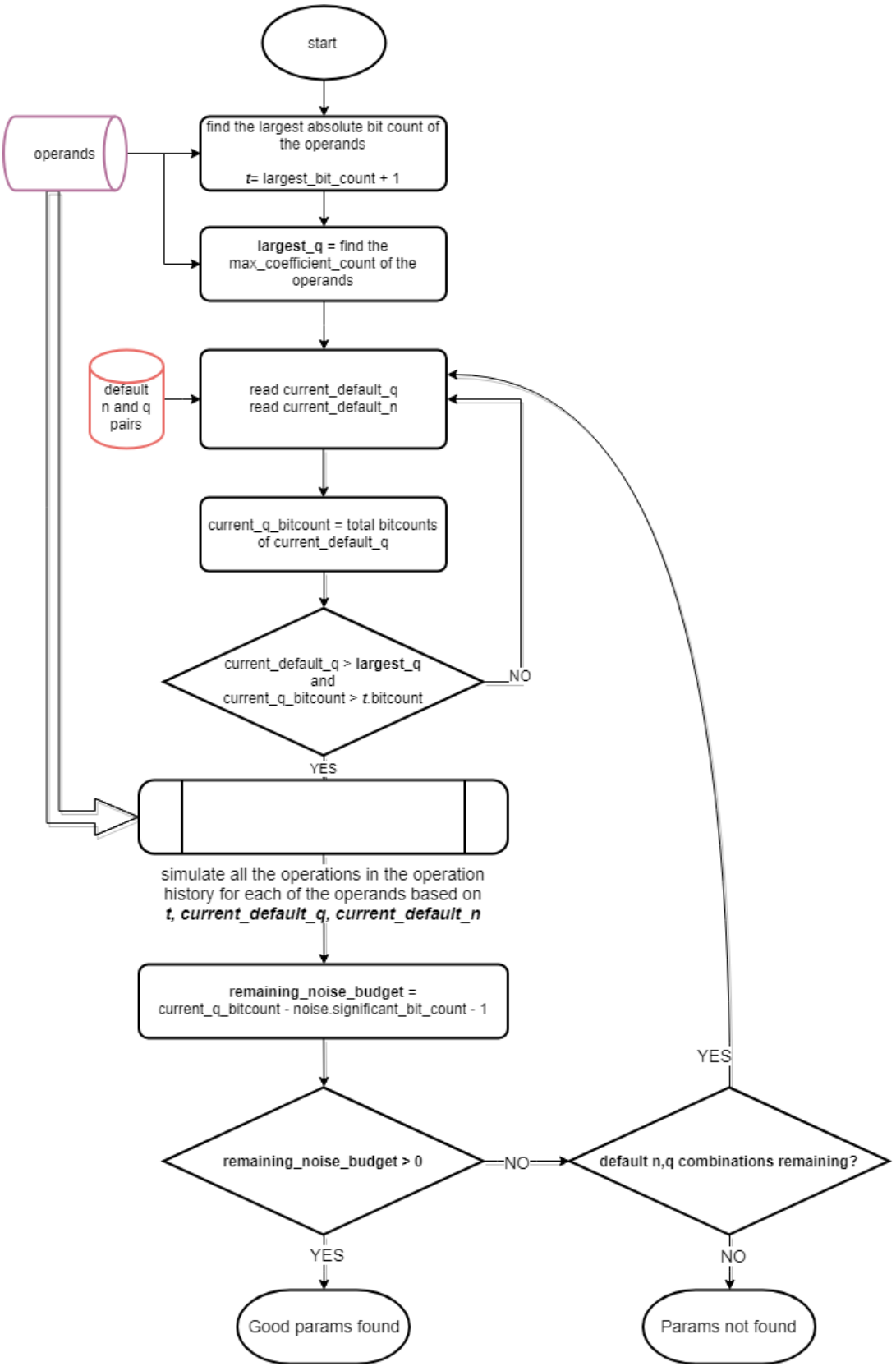


Figure 4-8 Flowchart for the Estimation of Encryption Parameters in SEAL

In Figure 4-8, a flowchart of the parameter estimation mechanism, found in the earlier versions of SEAL library [25], is presented. Sequentially, the flow starts with the estimation of the t and q values based on the expected maximum of input. These two values are specified before starting the estimation. Based on the values of t and q, a default combination of q and n are tried for a simulation on all the operations resulting in the output ciphertext. This simulation will give the final noise for the individual outputs based on the t, q, and n. After calculating the final noise for all the outputs, the remaining noise budget is calculated. This remaining noise budget should be greater than 0 in order to select the t, q_{default}, and n_{default} as the final parameters. Otherwise, if there are no more default combinations of q_{default}, and n_{default} remaining, then the parameter estimation fails.

Our work extends the existing module of parameter selection in SEAL by estimating the encryption parameters optimised and designed for a complete convolutional neural network. The extension to the SEAL library is made by developing our own methods of estimating the homomorphic encryption noise for individual convolutional neural network layers.

The method of selecting the individual encryption parameters is provided below.

4.5.3.1 **Estimating *t***

In SEAL, all the numbers for applying homomorphic operations are processed as polynomials of the form "$x^n + 1$", where x is the number base for the coefficients having polynomial degree less than n. The coefficients are modulo plaintext-modulus t which controls the number of non-zero coefficients in a polynomial. This plaintext modulus t determines the size of the plaintext data, therefore, the bigger the plaintext number, the bigger value for t that is required. Moreover, due to the coefficients being modulo t, each non-zero coefficient in a polynomial can expand up to t before a reduction can occur.

Therefore, to estimate t, we have to do the reverse and determine two quantities: the maximum length of the polynomial; and the maximum absolute value of polynomial coefficients. The maximum coefficient length, and the maximum absolute value, needs to be checked for all the operands and the resultant value after performing the operation so that a reduction, modulo t, is avoided. We note that we do not necessarily need a real encrypted operation; instead, we can calculate directly using the underlying polynomials to find the two quantities.

4.5.3.2 **Estimating n and q**

Once the estimation settles to a good value for t, the estimation for determining the values for the polynomial modulus n and the coefficient modulus q is performed. In the SEAL library, a set of n and q pairs of combinations are provided for ease of selection. This set of pairs, which can be seen in the SEAL documentation [22], is used to estimate the q_{default}, and n_{default} values during the estimation. The estimated noise of the operations in SEAL are calculated through the formulas given in the SEAL documentation [22] (reproduced in Appendix A). The combination of n and q is selected as shown in Figure 4-7 above by looping through them sequentially and testing them against the final noise. In this way, we will get our required encryption parameters for any feed-forward convolutional neural network.

4.5.3.3 **Numerical Limits of the Proposed System**

In connection with the inference of a neural network, it is essential that we explicitly describe the limits to the numbers used throughout the system. We describe these limits from a combined view of homomorphic encryption and convolutional neural network hereafter.

Numerical limits pertaining to the convolutional neural networks are relatively easier to understand when we see them according to their usage. In our encrypted neural network system, we have two views of the numbers involved: the unencrypted numbers; and the encrypted numbers.

In the training phase, the convolutional neural network is basically a normal neural network, and no special limits are applied to the numbers involved. This means that the limits applied to any normal neural network are applicable to the training networks in our system as well.

However, in the inference phase we have both unencrypted as well as encrypted numbers involved. The unencrypted numbers in the inference phase are only the trained weight values of each neuron/filter derived during the training phase. Whereas the encrypted numbers are the inputs and the outputs to each layer in the network.

There are important numerical limits applicable in the inference phase of an encrypted neural network. The unencrypted numbers (i.e., the weight values) needs to be encoded into the ring structure of the encryption scheme and this encoding can affect the classification accuracy of the neural network. For this purpose, we followed the decimal precision used by Dowlin et al. [1] for comparative reasons. We found that encoding the weight value using 10 coefficients (i.e., 10-bit precision) of a plaintext polynomial gives an encoding accuracy of 3 decimal places. This has been enough to maintain our classification accuracy unchanged from an unencrypted network to an encrypted network. In addition to the unencrypted numbers, the inputs that need to be encrypted must also go through encoding first. Therefore, all input values were encoded with the same encoding setting of 10-bit precision.

For the sake of completeness, we also mention that the encrypted operations have a different precision value as compared to the unencrypted values. This is because the modulus for the plaintext numbers is t, whereas the modulus for the encrypted numbers is q. Therefore, once the numbers are encrypted, then the numbers remain within the range of the selected n and q pair used in the initial encryption. These numbers can be different for different networks. For security hardness and simplicity reasons, the most commonly required pairs of n and q are given in the SEAL documentation [22]. Luckily, we do not have to manually select this pair for a neural network because our system is able to suggest this as explained in above sections.

4.6 Evaluation of the parameter selection method

From a top-level view, the estimation of encryption parameters for a complete convolution neural network requires some important considerations. In a convolution neural network, the number of calculations can quickly become very high and complicated to manage, if we are not careful about the arrangement of layers or the number of neurons or filters in the layer. This becomes even more important to consider in an encrypted environment. Therefore, here, we present and evaluate our parameter estimation method based on the variations of convolutional neural network hyperparameters. The evaluation will be shown for the most important aspects:

1) The number of fully connected layers connected in succession.
2) The number of neurons in a fully connected layer.
3) The number of sequential convolution layers.
4) The number of filters in a convolution layer.
5) The number of pooling layers.
6) The size of a filter in a convolution layer.

7) The stride size of a filter in a convolution layer

In estimating the optimal encryption parameters, we developed a module for estimating the sequence of operations in each layer based on the discussion in 4.5.1 above. The results of implementing and testing the estimation will be shared in the current section.

Hereon, the results from estimating the encryption parameters against the hyperparameters will be presented based on our proposed method. Each section shows the effect in three graphs, for t, q, and n, respectively. The t will be shown in log-scale because of their exponential growth. The value for q will be a value showing the significant-bit of the product of all the primes in the coefficient modulus. By knowing the significant-bit of the product, we can use any combination of primes to make the product within the range of bits. Each of the primes in the product, however, needs to be within the 60-bit limit of the SEAL library. Subsequently, the values for n are representing the highest degree of the ciphertext polynomials.

4.6.1 Effects of fully connected layers

The number of fully connected layers affects the accurate classification. However, the number of fully connected layers in a convolutional neural network is usually set by trying different variations and choosing what works best for the particular classification problem.

The following estimations were performed by providing a set of four input values, and using a single neuron in sequential layers from 1 to 10. The effect on each parameter is shown below.

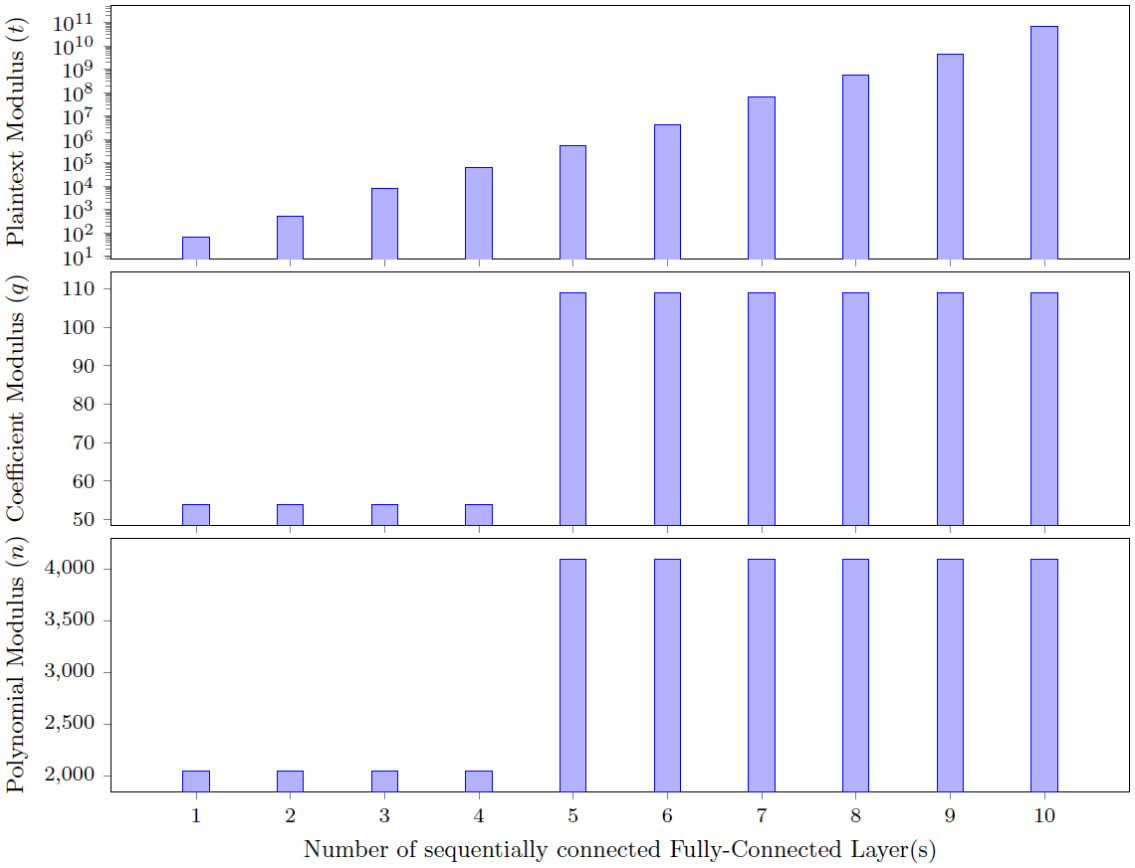


Figure 4-9 Effects of sequentially connected FC Layers on the estimation of encryption parameters.

From the estimation shown in Figure 4-9 above, the value of t needs to be increased exponentially if we want to add sequential fully connected layers. This effect is because the result from each layer is used as an input for the next layer, and they are multiplied by a weight value. Therefore, the encrypted values will grow and will require a bigger value of the plaintext modulus t to represent them in the encrypted form.

From the estimation, we see that the value of q is quite consistent across the sequential FC layers. The only time it changed, from a 54 bit to almost its double of 109 bits, is after 4 layers. This is because the polynomial modulus n and the value of t need to be increased to accommodate bigger numbers.

From the estimation of n, we see that the sequential execution of up until 4 FC layers will result in numbers within the range of polynomials having 2048 coefficients. After this, the degree of polynomial needs to be increased to accommodate bigger numbers.

This sequence of the increasing encryption parameters continues in a similar way for more than 20 layers.

4.6.2 Effects of Neuron Count in fully connected layers

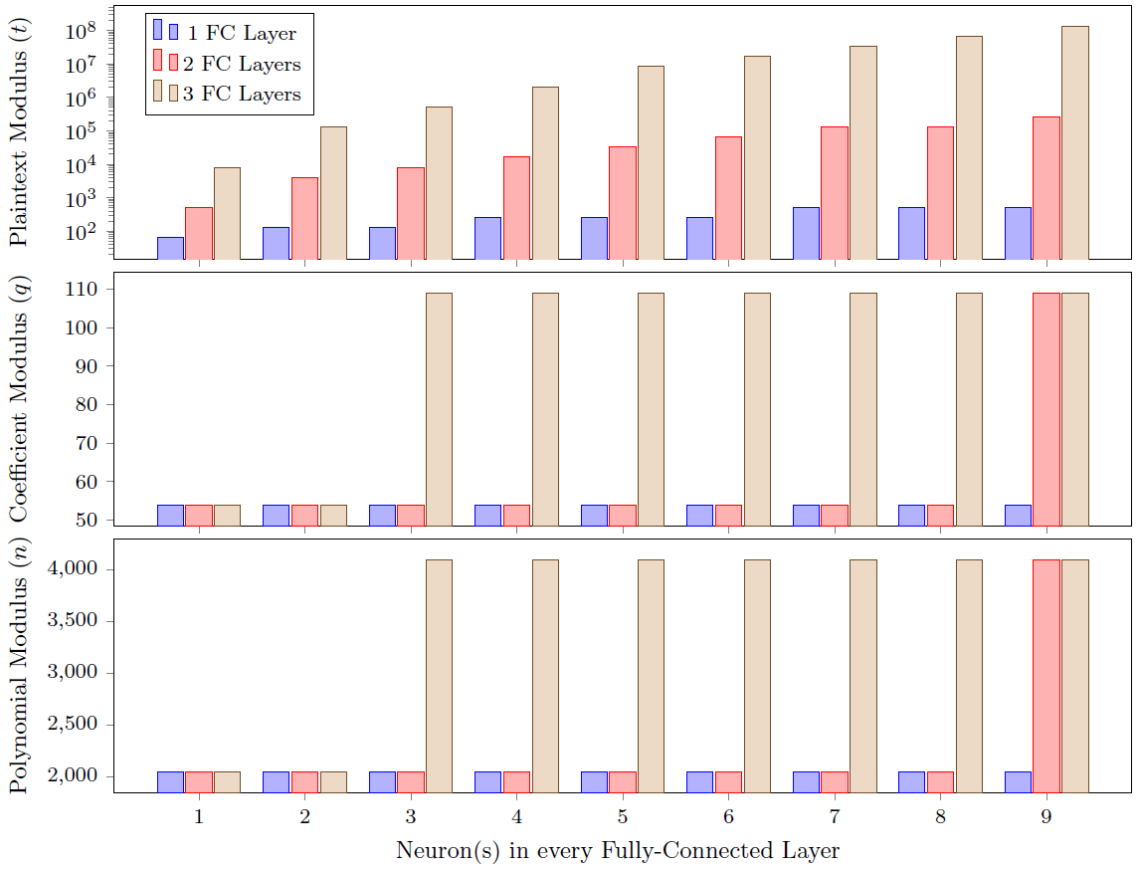


Figure 4-10 Effects of neuron count on the estimation of encryption parameters in sequentially connected FC Layers.

The estimation in Figure 4-10 shows the effect of the number of neurons in a fully connected layer. The estimation was performed by providing a set of four inputs to one, two and three sequentially connected layers to see the effect more clearly.

From the estimation shown, we observe that the number of neurons have little impact in a single layer. This is because the values are not used in any further calculations. However, the impact is high for performing the same estimation when 2 or 3 sequential FC layers are used and the neuron count is increased gradually from 1 to 9.

From the estimation shown, there is no effect on q when using a single layer, again, because the values are not used further. Even when using 2 or 3 sequential layers, the effect is not a major one. The value is almost doubled when using 9 neurons in 2 sequential layers because of the doubling of the value of n.

From the estimation shown, we see that the only change in the value of n is required when the underlying numbers are getting bigger. In two sequential fully connected layers, this can be seen from 9 neurons.

4.6.3 Effects of Convolution Layers

In these estimations, the effect of sequential convolution layers is observed by providing an initial input of 10×10 matrix. All the convolution layers have a single filter of size 2×2 with a stride of 1.

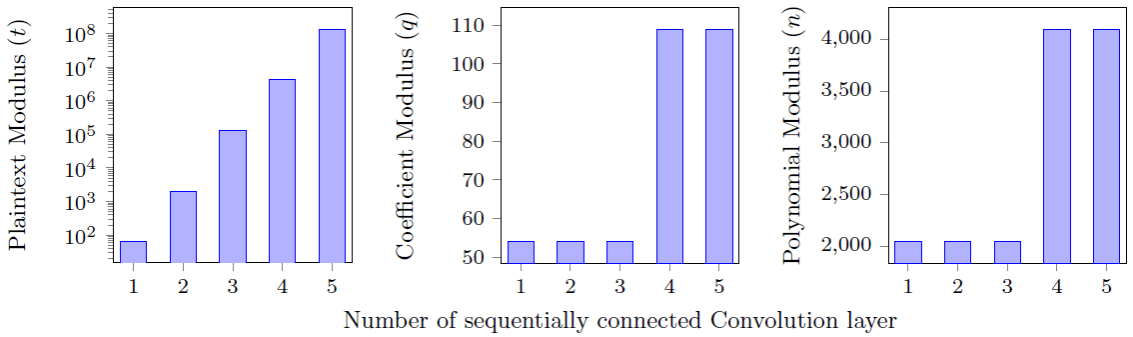


Figure 4-11 Effects of sequentially connected Convolution Layers on the estimation of encryption parameters.

From the estimation shown in Figure 4-11, we observe the effect of sequential convolution layers on the value of t. Here, the t value needs to be increased exponentially more rapidly as compared with the t value from the FC layer count. This effect may be due to the fact that the FC layers had an input of four values, whereas the convolution layers start with a 10×10 input matrix making a total of 100 values.

The effects on q are very similar to those of a fully connected layer. This means that there is less effect on q value whether we increase the fully connected layers or convolution layers.

As with the estimation of q, the estimation of n, also shows similarity between the counts of fully connected layers or convolutional layers. This means that there is less effect on the n value whether we increase the FC layers or convolution layers.

4.6.4 Effects of Filter Count in Convolution Layers

Filters in a convolution layer act as the neurons to find patterns. The effect is important to note because it is considered as an important factor for the prediction accuracy of a convolution network. For this estimation, an input matrix of 5×5 was provided as input to the convolution layer.

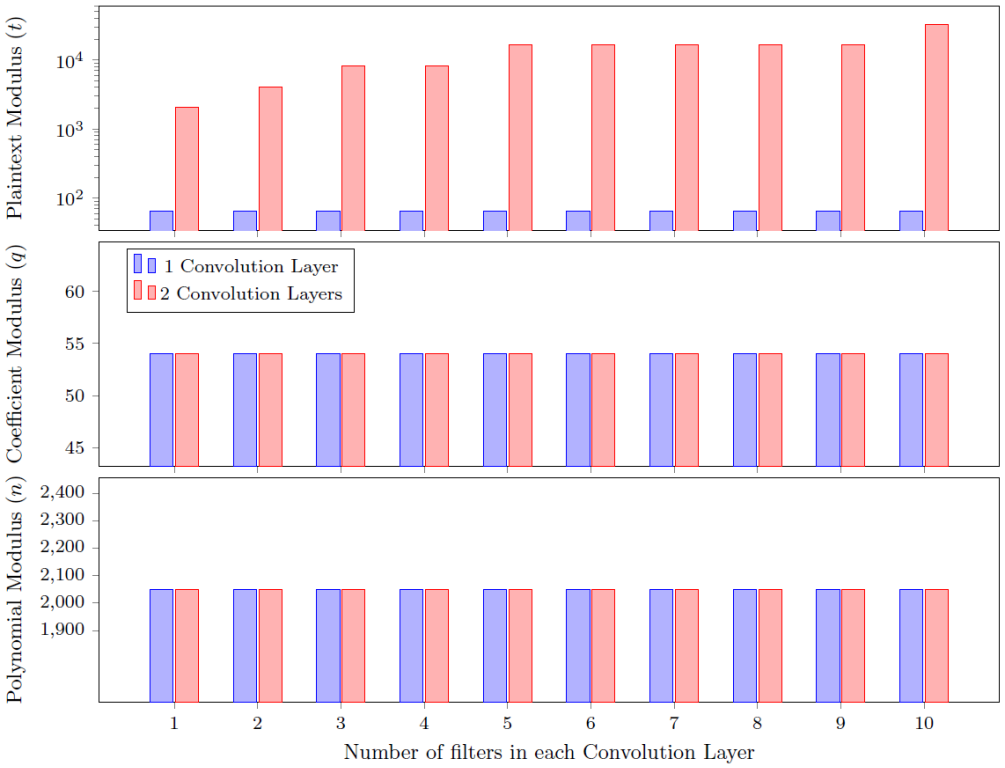


Figure 4-12 Effects of Filter count on the estimation of encryption parameters in Convolution Layers.

There were filters from 1 to 19 having a size of 2×2 with stride 1. Estimation was observed for a single layer as well as for 2 sequential convolution layers. The effects can be seen in Figure 4-12.

From the estimation shown in Figure 4-12, we observe that if the values are not used in a subsequent layer, then the number of filters has no effect on t value. This is because each filter performs processing independently on the same set of input values. However, it is important to note that once the output values are being used in another layer, then the number of filters will affect the t value. If we compare this estimation of filter count with that of the neuron count in the fully connected layer, then we see little difference.

From the estimation shown, there is no difference for q as the number of filters is increased if they are in sequential convolution layers.

Similarly, to the effect on q, the estimation shows us that the number of filters in a convolution layer has no effect on the value of n.

4.6.5 Effects of Filter Size in Convolution Layers

Filter size is the width and height of the region in the input matrix for which the filter processes values in a single step. The bigger the filter size, the bigger the region of values processed at once. In a convolution neural network, the filter size determines the size of the pattern for which the filter has to look for. In this estimation, observation was made by providing an input of 10×10 to a single filter in a single convolution layer having a stride of 1. The following effects were observed for the estimation as shown in Figure 4-13.

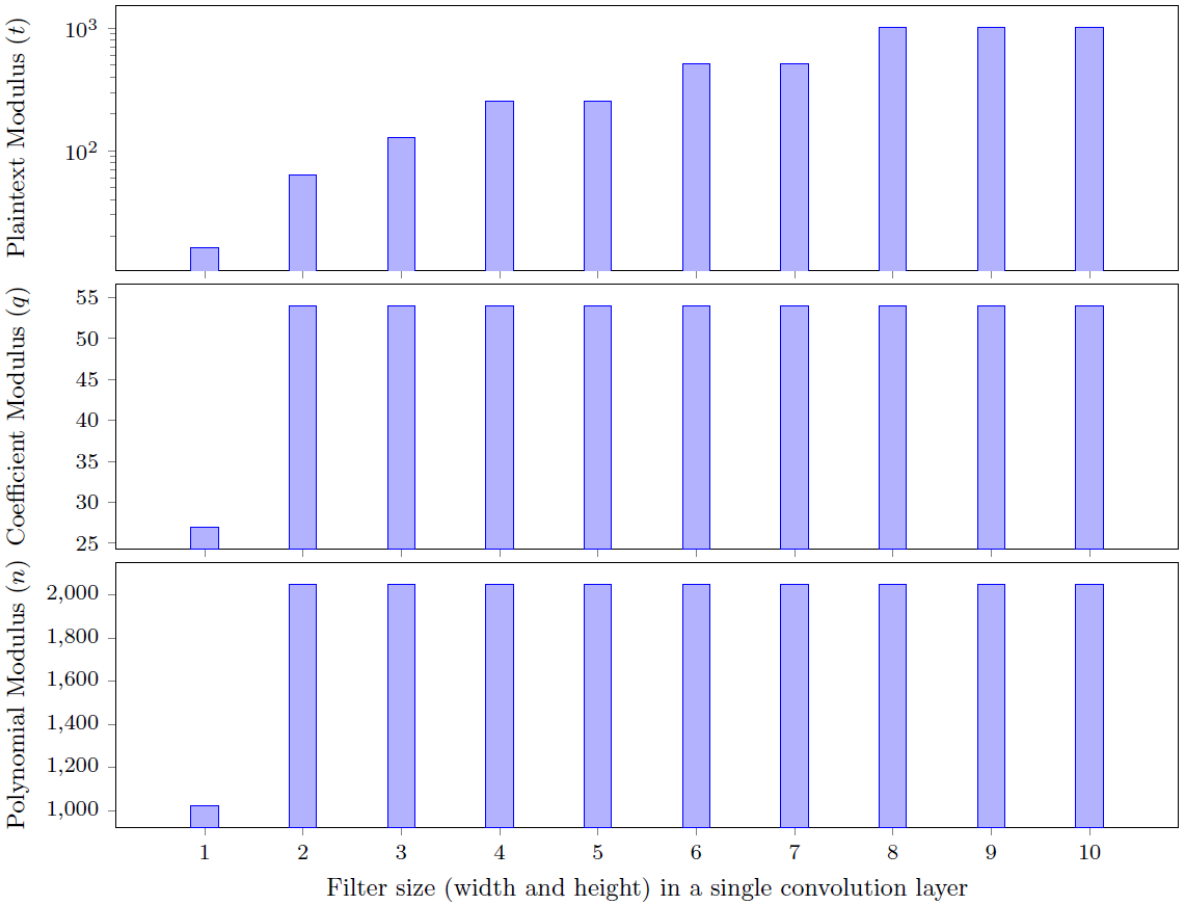


Figure 4-13 Effects of Filter size on the estimation of encryption parameters in Convolution Layers

The only difference due to changing the filter size was observed for the value of t. The estimation shows the effect that the bigger the filter size, the bigger t is required to accommodate the accumulation of many values into a single one. Since the filter size indicates the dimension of the filter region taking more input values for processing, the bigger the size of the filter, usually the double the t value is required.

Unlike the t value, the filter size has no effect whatsoever on the q value. From the estimation, it can be noted that a filter size of 1 means that the filter is taking a 1×1 region for processing, which means only a single value. However, the rest of the filter sizes shows no effect on the q. Similarly, the estimation shows that the n value is not affected by the size of the filter used in a convolution layer.

4.6.6 Effects of Filter Stride Size in Convolution

The stride size of a filter is the amount of movement from one-step to the next. The stride value determines how coarsely or finely the filter can look for patterns. Since the stride value determines the total output dimension of the filter as well as the number of individual operations in processing filter, its effect is analysed here.

For this estimation, a stride from 1 to 5 is used for a single filter of size 2×2 in a single convolution layer. A 10×10 matrix of input values is provided. The effects are observed in Figure 4-14.

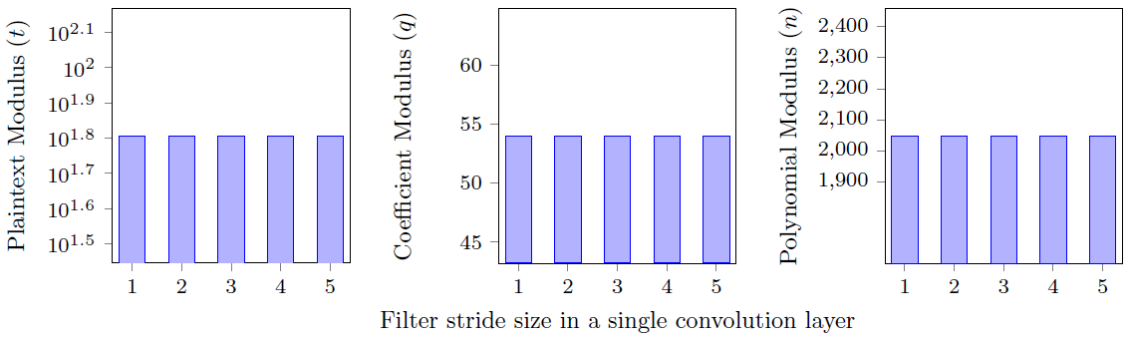


Figure 4-14 Effects of Filter stride size on the estimation of encryption parameters in Convolution Layers

From the estimation shown in Figure 4-14, no effect was seen on the t value. This clears the concern, mentioned in section 4.5.1.1 above, about using the edge pixels less than the centre pixels in a convolution layer.

As for the t value, there is no effect of stride on the q and the n values.

4.6.7 Effects of Sum-Pooling Size

The pooling size is the size of the window for taking a subset of values from the original input matrix. The bigger the pooling size, the higher the number of values processed in a single pooling step. A pooling layer essentially reduces the dimension of the input matrix. In this estimation, the effect of pooling size on the encryption parameters was observed. This estimation was performed on an input matrix of size $10 \times 10 \times 1$ having a stride size of 1.

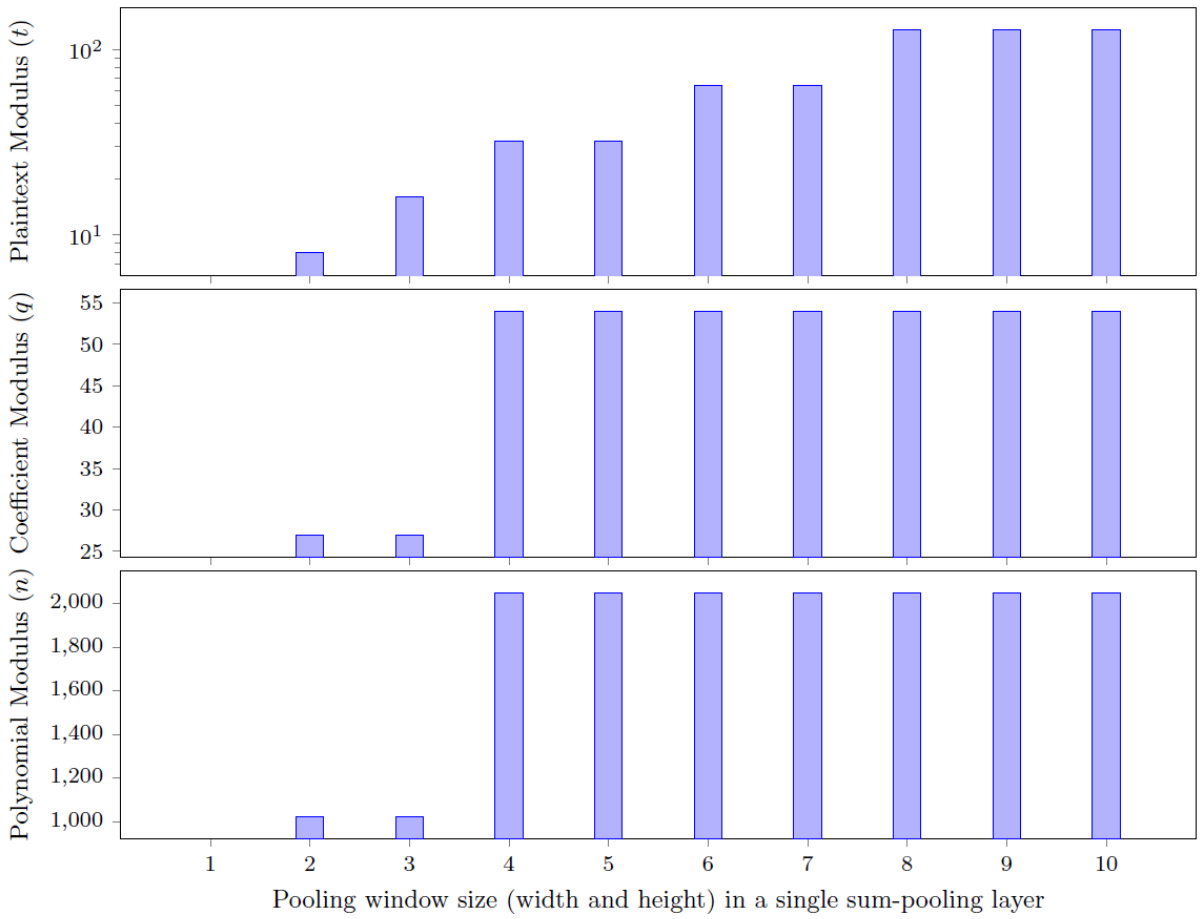


Figure 4-15 Effects of pooling size on the estimation of encryption parameters in Sum-Pooling Layers

Figure 4-15 shows the effect of pooling size on the value of t. We can see that the bigger the pooling size, the bigger the value of t that is required, to accommodate many values.

In this estimation, we see that the value of q is mostly consistent. This means that the pooling size does not have a direct effect on the value of q.

As for the q value, the pooling size is not the main determinant of the value of n and the value can be seen to be mostly consistent.

4.6.8 Effects of Sum-Pooling Stride

The pooling stride is the amount of movement for the pooling window after every operation. The bigger the value of the stride, the more the number of inputs that are skipped after each step. For this estimation, two experiments with different input matrix size were performed: 10×10 and 100×100 input matrices. In the experiments, a pooling size of 2×2 was used. The effects are shown in Figure 4-16.

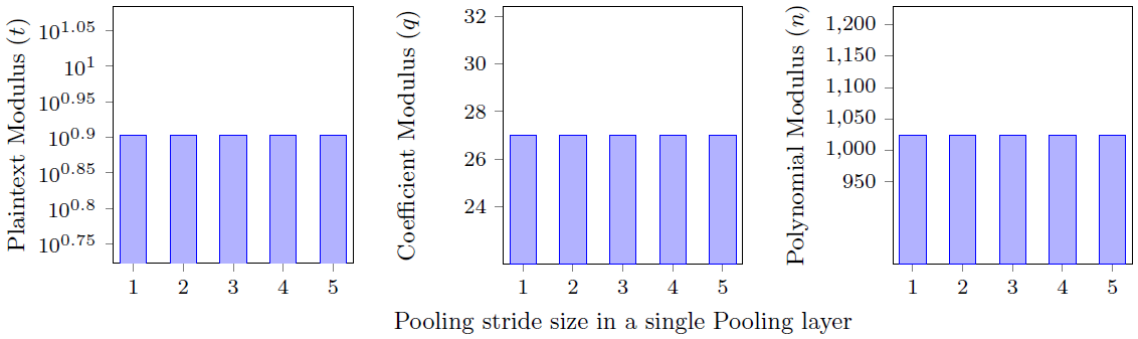


Figure 4-16 Effects of Pooling stride size on the estimation of encryption parameters in Sum-Pooling Layers

In Figure 4-16, we can see that the pooling stride is irrelevant here to the value of t, q, and n. These values are the same for all the stride sizes.

4.7 Findings

We have described methods used to find the relationship between the homomorphic encryption and convolutional neural network parameters. The first key finding of our research is the method and the formulas for finding the exact circuit depth of an encrypted convolutional neural network. In Table 4-5 below, a summary of the sequence and the type of homomorphic operations is presented for future reference. Their calculation method is detailed in section 4.5.1 above. In the summary Table 4-5, each row represents a typical layer type found in a convolutional neural network. Each column represents the type of operation. The columns show the type of operations between a ciphertext C and a plaintext P.

Table 4-5 Nature of homomorphic operations in a convolutional neural network

	$\boldsymbol{C \times P}$ **operations**	$\boldsymbol{C + C}$ **operations**	$\boldsymbol{C + P}$ **operations**	$\boldsymbol{C \times C}$ **operations**
Fully Connected	$z_i \times z_o$	$z_i - 1$	1	None
3D-Convolution	$F_w \times F_h \times d$	$(F_w \times F_h \times d) - 1$	1	None
2D-Convolution	$F_w \times F_h \times 1$	$(F_w \times F_h) - 1$	1	None
Sum-Pooling	None	$(F_w \times F_h) - 1$	None	None
Average-Pooling	1	$(F_w \times F_h) - 1$	None	None
Square Activation	None	None	None	1

The second key finding is the derivation of formulas for calculating the maximum noise incurred for a certain convolutional neural network layer. A summary of these noise-calculating formulas is given in Table 4-6. To calculate the maximum noise for a network, we first calculate the initial noise a after fresh encryption. Besides the initial noise, our derived formulas from the estimations of SEAL for each network layer type are:

Table 4-6 Noise estimates for Layers in Convolution Neural network based on SEAL's heuristic estimation.

Layer type	**Maximum noise v_r after the layer**
Convolution Layer	$v_o = N_m \times \|m\| \cdot \left(\frac{q \bmod t}{q} + i \cdot v_k\right)$
Fully Connected Layer	$v_o = N_m \times \|m\| \cdot \left(\frac{q \bmod t}{q} + i \cdot v_k\right)$
Sum - Pooling Layer	$v_o = i \cdot v_k$
Average - Pooling Layer	$v_o = N_m \cdot \|m\| \cdot i \cdot v_k$
Square Activation Layer	$v_o = t\ \sqrt{3n}\ (2^{j+1} \cdot 3^{j/2} \cdot v_i \cdot n^{j/2}\ +\ 12^j \cdot n^j)$ Or if ciphertext size is always kept to 2, then: $v_o = t \cdot \sqrt{3n}\ (144n^2 + 24nv_i) + v_{relin}$

In Table 4-6, we summarize our derived formulas for estimating the maximum change in noise for any ciphertext passing through a convolutional neural network layer. Here, $\|m\|$ is the absolute biggest number that can be provided as input. N_m is the highest degree of the polynomial representing our message m. The j is the ciphertext size, this usually will be 2, when relinearizing after every ciphertext-ciphertext multiplication. The i represents the number of inputs to the layer. The v_i represents the noise already in the ciphertext before the layer. The v_o is the noise after the layer. The t, q, and n represent the encryption parameters of plain modulus, coefficient modulus, and polynomial modulus, respectively.

Another key finding is the concrete establishment of the effects of individual hyperparameters of a CNN on the HE parameters (see section 4.5.3.3). In here, we conclude from the estimations that some hyperparameters are affecting the encryption parameters, whereas some are not. This can be summarized in the following Table 4-7.

Table 4-7 Convolutional neural network Hyperparameters affecting encryption parameters.

	Affects t	**Affects q**	**Affects n**
FC Layer Count	Yes	Yes	Yes
FC Neuron Count	Yes	Yes	Yes
Conv Layer Count	Yes	Yes	Yes
Filter Count	Yes	No	No
Filter Size	Yes	No	No
Stride size	No	No	No

Table 4-7 summarizes the effects of CNN hyperparameters on the HE parameters where yes means a strong link and no means a weak or non-existent link. A strong link with any of the convolutional neural network hyperparameters and any of the encryption parameter shows a directly proportional link. In such a case, if we increase or decrease the hyperparameter, then the encryption parameter will also need to be increased or decreased, respectively. The amount to which the encryption parameter requires to be increased or decreased, can be estimated using our derived relationships shown in Table 4-6. In comparison to a strong link, a weak link, such as that in the case of stride size, shows that there is no need for increasing or decreasing the encryption parameter with respect to that particular hyperparameter. This means that, the particular hyperparameter can be changed without the need to change the encryption parameter. In such a case, the noise in the output ciphertext will remain the same, irrespective of the change in the hyperparameter.

4.8 Summary

Throughout this chapter, we demonstrate that the solution to preserve privacy, while analysing data through a third party encrypted convolutional neural network, can be made generic to accommodate a variety of convolutional neural network models. We discuss the mechanism for extending encrypted classification capability, to any feed-forward convolutional neural network model by using our parameter estimation method. Moreover, we show how to derive our noise estimation formulas for the individual convolutional neural network layers. Based on our derived noise calculation formulas, we saw the effects of different hyperparameters of a convolutional neural network that define the core structure of the network.

5 Practical Case Studies

The application of homomorphic encryption within a convolutional neural network is a desirable functionality for developers. The enabling mechanism of this functionality is discussed in the previous chapter on selecting the encryption parameters. Here, we describe and learn from important cases of convolutional neural network models by applying our mechanism for selecting encryption parameters. These case studies help to acquire in-depth insights about the common convolutional neural network models. Among which, the case studies will illustrate the parameter selection method more eloquently. In addition, these cases show that the methodology, suggested by us, is a generic method applicable to a wide variety of convolutional neural network models, and not just to one specific convolutional neural network. Moreover, these case studies help us in knowing the exact limitations of the overall research work, and highlight the required improvements to enable a privacy-preserving convolutional neural network classification task. These case studies have been very effective in terms of defining the limitations and the edge cases of our work.

5.1 Selection of Case Studies

For the purpose of this investigation, we have limited our observations to 2 case studies. The selection of case studies is always a crucial step to properly present and discuss issues. Therefore, we aim to select the most prominent convolutional neural network models available. Since the CryptoNets [1] is based on the SEAL library, it is important to compare our work with this, therefore the CryptoNets was the first case study. Besides the CryptoNets, we shortlisted 4 convolutional

neural network models from Canziani et al. [60] that could be good candidates to fulfil our objective of a case-study. Canziani et al. [60] provide a thorough analysis of the state-of-the-art 14 different deep convolutional neural networks (DNNs). Their analysis was based on several important metrics for practical applications such as accuracy, memory footprint, parameters, operation counts, inference time and power consumption. The shortlisted candidates for detailed analysis in our case studies were:

1. CryptoNets
2. AlexNet
3. VGG-16
4. LeNet-5
5. GoogleLeNet / Inception

AlexNet [61] and VGG-16 [62] both require millions of encrypted calculations. The initial analysis using our parameter estimation technique showed that their encryption parameters could be estimated easily within a second on our computer having Intel Core i7 7700 with 16GB RAM. However, due to their huge number of encrypted operations, they were not practical to evaluate. Therefore, they could not be selected for further analysis.

GoogleLeNet [63], also called the Inception Networks, are also interesting to analyse. The main differentiating factor in the Inception Networks from other convolutional neural network models is the inception module. The inception module first takes the input from each layer and passes it through several parallel convolutions. Then the results of all these convolutions are combined together to provide a single output to pass to the next layer. This reason also eliminated GoogleLeNet from our study, as it will not be practical to evaluate and analyse.

Beside the limitations of the case studies themselves, another factor related to the estimation of the parameters is the selection of the maximum absolute value and the maximum coefficient count. During these case studies, we used a fixed coefficient count of 10 for all the ciphertext encryption as well as the plaintext encodings. However, these values will depend on the scale of the input numbers and the decoding step for the decrypted outputs. The reason for the values that we have selected is that a count of 10 coefficients in the ciphertext polynomials provide 3-digit accuracy for the fractional part of a number in base 2. Moreover, the maximum absolute value was set to 1.

5.2 Microsoft's CryptoNets Case

The use of Homomorphic Encryption in Convolutional Neural Networks has been an active research area. In this regard, Dowlin et al. [1] shared their research of incorporating a state-of-the-art HE scheme in a convolutional neural network model which they called CryptoNets.

These researchers shared high throughput and accuracy for classifying encrypted images through a convolutional neural network. CryptoNets show that homomorphic encryption could be efficient and accurate enough for real world applications. Unlike the previous criticism against the use of homomorphic encryption in real world applications, CryptoNets demonstrates that a single classification step of encrypted input data can be processed on a single PC in 570 seconds (9.5 minutes). In a normal classification setup, only one input record is provided to the neural network to produce one predicted output. However, the underlying encryption scheme used in CryptoNets allows one to classify more than one input record to produce multiple predicted outputs. CryptoNets is primarily aimed at classifying more than one image at a time by encoding multiple images in

one single encrypted ciphertext. The high throughput claimed by CryptoNets is due to this batch processing ability. It has been demonstrated that CryptoNets can process 8192 input records in a single classification step to produce 8192 predicted outputs simultaneously without any lag in processing time [1]. This ability of batch processing is certainly a very welcomed aspect of CryptoNets. However, not all applications will require the classification of multiple input records in a single step.

5.2.1 Novelty in CryptoNets

5.2.1.1 Usage of the sum-pooling

In CryptoNets, the researchers have modified some parts of a traditional convolutional neural network to enable it to be used for homomorphic operations. One of the modifications is the use of a sum-pooling operation instead of the commonly used max pooling or the average-pooling operations. CryptoNets shows that this change results in fewer homomorphic operations without any loss in the classification accuracy [1].

In our observation, the usage of sum pooling can be justified for use in neural networks with a fewer number of layers but not for deeper networks. However, there exists the option of an average pooling which can be used in an encrypted environment with minimal effect on the inherent noise or computational load. It is noteworthy that the sum-pooling operation adds several incoming input numbers to make a single number. This can result in a rather large number for the last layers. Therefore, the sum pooling should be used with consideration of the growth of the numbers within a neural network. The issue with very huge numbers is that they require bigger encryption parameters, and hence, with unnecessary reduction in computational performance.

5.2.1.2 The two forms of CryptoNets

Another modification CryptoNets has is the usage of two different forms of their convolutional neural network. One form of the neural network was termed as the full form of the network for the purpose of training it, and the other one was the reduced form of the network for performing inferences only.

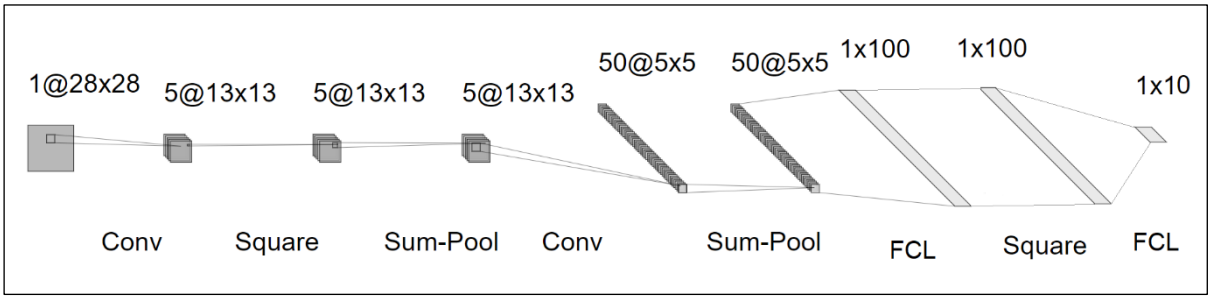


Figure 5-1 The full form of CryptoNets

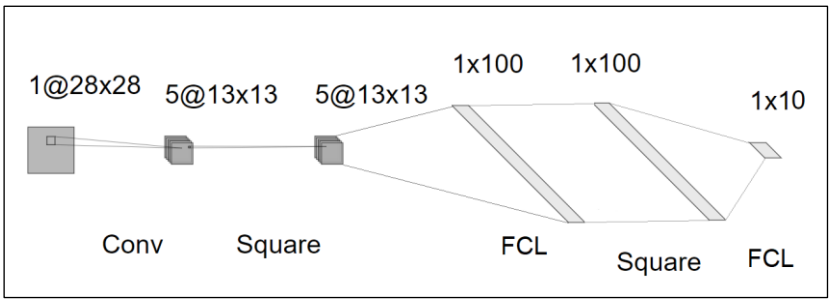


Figure 5-2 The reduced form of CryptoNets

The two forms of the CryptoNets CNN model for training and inference purposes can be seen in Figure 5-1 and Figure 5-2, respectively. The training network had 8 layers (excluding the last sigmoid layer) and the reduced network had 5 layers. In these figures, the numbers above the layers indicate the transformation of the dimensions representing the pixels. The prime difference comes from the merging of layer 3 to 6 in such a way that the pooling layers are removed, and the weights of the 3rd layer (a convolution layer) is merged with that of the 6th layer (a fully connected layer). By the merging of the convolution and the fully connected layers,

a single fully connected layer is shown to work for both of these layers. This arrangement is easily understood in the following Table 5-1.

Table 5-1 Layer Variation in CryptoNets

Layer Number	**The full form**	**The reduced form**
1	Convolution	Convolution
2	Square	Square
3	Sum	FCL
4	Convolution	
5	Sum	
6	FCL	
7	Square	Square
8	FCL	FCL

From Table 5-1, we have an important observation about the arrangement of the layers. The observation is about the logical arrangement of the first 3 layers where the pooling layer is expected to come before the activation layer. Conversely, the activation layer precedes the pooling layer in CryptoNets.

5.2.1.3 Usage of 3-dimensional convolution instead of 2-dimensional convolution

The CryptoNets used a 3-dimensional (3D) convolution filter type in their network instead of the commonly used 2-dimensional (2D) convolution filters. The basic difference between these two convolutions arises from the way they perform a single convolution step on the incoming input values. In a single convolution step, a 3D convolution filter can take values from a single input channel whereas the 2D convolution filter takes inputs from all the input channels. The 3D convolution

layers, as in CryptoNets, convolve filters over the input data volume in 3-dimensional directions including the depth direction. This means that, whenever we use a 3D convolution filter, the stride size will be specified in 3 directions as the width F_w, height F_h, and the depth F_d of a filter F. In this way, we specify the number of channels to process in a single convolution step. For example, we can specify the filter depth F_d as one, which means that in a single convolution step the filter will take values from only a single channel.

Unlike the 3D convolution, a 2D convolution filter actually takes values from all the input channels. The convolution is performed in only the width and height directions but not the depth. In this way, 2D convolution will use a stride specification of only width F_w and height F_h, but not the depth F_d of a filter F. The 2D convolution filters take values from all the input channels constrained by the filter dimension for processing.

The filter depth F_d in CryptoNets was set to one, so, each filter convolves over a single channel of the input at a time. In this way, the convolution continues over the next input channel after finishing the current one. The next section provides some insight for deciding to have a 2D convolution over all the channels versus a 3-dimensional convolution over each of the channels separately.

5.2.2 Arithmetic operations in CryptoNets

In order to find the encryption parameters, it is necessary to understand the underlying CNN model. Therefore, the two forms of the CryptoNets are studied.

Table 5-2 Operations in CryptoNets (Left: full form, Right: reduced form)

Layer No.	Layer Type	Sum Ops	Mul Ops
1	Conv	21,125	21,125
2	Square	0	845
3	Sum	5,760	0
4	Conv	6,250	6,250
5	Sum	1,280	0
6	FCL	25,000	25,000
7	Square	0	100
8	FCL	1,000	1,000
Total		**60,415**	**54,320**

Layer No.	Layer Type	Sum Ops	Mul Ops
1	Conv	21125	21125
2	Square	0	845
3	FCL	84500	84500
4	Square	0	100
5	FCL	1000	1000
Total		**106625**	**107570**

In Table 5-2, the left and right-hand tables represent the full-form and the reduced-form of the CryptoNets models respectively. Against each of the layer types, the summation and the multiplication operations are shown to give an overview of what to expect.

During the observation of CryptoNets as a case, a necessary improvement was found to the architecture of the neural network model. To make CryptoNets more suitable in terms of execution performance, the convolution layers can be changed from 3D convolutions to 2D convolutions. If we only consider the encryption parameters required for the 3D or the 2D convolution layers, then the estimated parameters will be the same. However, the number of heavy multiplication calculations being performed will be significantly less in the case of 2D. Even more, the number of homomorphic multiplications in the full form of the network, having 8 layers, will

be less than that of the reduced form, with only 5 layers. To be exact, the difference is shown in the following Table 5-3.

Table 5-3 Difference of operations between 3D and 2D convolutions in the full form of CryptoNets

	Total $C \times P$	**Total $C + C$**
3D convolution	302,375	188,330
2D convolution	77,375	84,210
Difference:	**+225,000**	**+104,120**

The interesting reason for this difference in the number of operations is that when a convolution layer has filters that are applied on a single input channel for the 3D convolution, then two things happen. First, the total number of trainable weight values that each filter has, are reduced, because the filter only has width and a height but not the similar depth as the 2D. Secondly, a greater number of output values for the 3D convolution, in contrast to the 2D convolution, will be passed on to the next layer.

5.2.3 Detail of encrypted operations in CryptoNets

The total number of encrypted operations in an encrypted CNN can be similar to that of an unencrypted plaintext version of the same CNN model. Nevertheless, it is important to note that this may not always be the case. Sometimes the encrypted version may have more operations because HE operations support only two operands in a single instance. Whereas, in a plaintext version, a single operation can have more than two operands at a time. Therefore, the number of encrypted operations can be different.

The total number of encrypted operations for the CryptoNets for the full form of the network and the reduced form of the network are given in the following table.

Table 5-4 Total encrypted operations for the full form of CryptoNets

Layer Number	**Layer Type**	**Total $C \times C$**	**Total $C \times P$**	**Total $C + C$**	**Total $C + P$**
1	Conv2D	0	21,125	20,280	845
2	Square	845	0	0	0
3	Sum	0	0	5,760	0
4	Conv2D	0	156,250	30,000	1,250
5	Sum	0	0	6,400	0
6	FCL	0	125,000	124,900	100
7	Square	100	0	0	0
8	FCL	0	0	990	10
Grand Total		**945**	**302,375**	**188,330**	**2,205**

In the same way, the total number of encrypted operations in the reduced form of the CryptoNets model are:

Table 5-5 Total encrypted operations for the reduced form of CryptoNets

Layer Number	**Layer Type**	**Total $C \times C$**	**Total $C \times P$**	**Total $C + C$**	**Total $C + P$**
1	Conv2D	0	21,125	20,280	0
2	Square	845	0	0	0
3	FCL	0	84,500	84,400	100
4	Square	100	0	0	0
5	FCL	0	0	990	10
Grand Total		**945**	**105,625**	**105,670**	**110**

In Table 5-4 and Table 5-5, we see the total number of encrypted operations as calculated by using the following formulas for all of the filters in the convolution layers and the neurons in the fully connected layers.

The total number of operations can only tell us how fast the execution performance can be, but not the exact number of operations that an individual ciphertext goes through. The number of operations for a single ciphertext is essential to estimate the encryption parameters rather than the total number of operations in a layer. When an input is passed through a convolutional layer, then each of the input pixels only has an effect on some of the output values in that layer and not all of them.

For the encrypted version of CryptoNets, the number of encrypted operations that a single ciphertext goes through, are as follows.

Table 5-6 Individual encrypted operations for the full form of CryptoNets

Layer Number	Layer Type	Individual $C \times C$	Individual $C \times P$	Individual $C + C$	Individual $C + P$
1	Conv2D	0	1	24	1
2	Square	1	0	0	0
3	Sum	0	0	8	0
4	Conv2D	0	1	24	1
5	Sum	0	0	8	0
6	FCL	0	1	1,249	1
7	Square	1	0	0	0
8	FCL	0	1	99	1
Grand Total		**2**	**4**	**1,412**	**4**

In the same way, the number of individual encrypted operations for the reduced form of CryptoNets are as follows.

Table 5-7 Individual encrypted operations for the reduced form of CryptoNets

Layer Number	Layer Type	Individual $C \times C$	Individual $C \times P$	Individual $C + C$	Individual $C + P$
1	Conv2D	0	1	24	1
2	Square	1	0	0	0
3	FCL	0	1	844	1
4	Square	1	0	0	0
5	FCL	0	1	99	1
Grand Total		**2**	**3**	**967**	**3**

The number of encrypted operations that any single ciphertext goes through in the CryptoNets are shown in Table 5-6 and Table 5-7. These were calculated by using our formulas shown in Table 4-5.

5.2.4 Practical Implementation Details

For our research, we used and customised several software tools. Here we provide a brief summary of the various tools and their combinations. The block diagram in Figure 5-3 below shows a summarized view for visual explanation.

Since this research is focused on CNN, therefore, the open-source ConvNetSharp[6] deep learning library was customised for our use. The customisation involves the integration of SEAL library within the ConvNetSharp code. This integration enabled us to perform SEAL encrypted classification tasks of any neural network that we desire. Moreover, the ConvNetSharp library already has several training algorithms for a neural network that we used. The customised integration works in such a way that we train a neural network using a plaintext dataset to get plaintext weight values. Afterwards, we can use the trained weight values both in a plaintext as well as encrypted neural network.

In addition to the above customisation of the ConvNetSharp library, we also added several new methods within the SEAL library itself. The new methods allowed us to estimate the encryption parameters for any feed forward neural network such as the CryptoNets. Details of these methods are added in the Appendix C.

[6] https://github.com/cbovar/ConvNetSharp

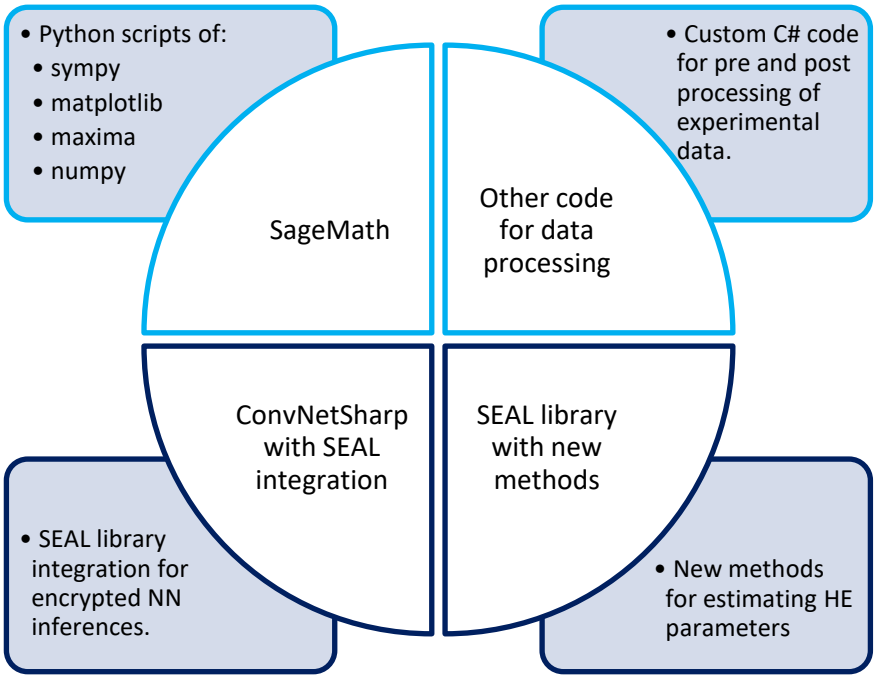


Figure 5-3 Block diagram of the tools and libraries used in this research

The ConvNetSharp library helped in training of our CNN in plaintext. All the major training algorithms are already implemented withing the ConvNetSharp. For our CNN, the trainings were performed like CryptoNets by first training a full form of the network and then merging some layers to make a reduced form as in CryptoNets. The detail of such merging is provided in 5.2.1.2. The training was performed in mini batches of 100 random samples from MNIST images. Overall, the training took around 20 minutes on an NVIDIA GTX 1060 laptop GPU resulting in a 98% classification accuracy rate which is good but common for the MNIST dataset. However, a single inference task took around 1.4 hours on average. It should be considered that this time is for the inference of a single input image.

5.3 Applying our method with CryptoNets

For selecting the best encryption parameters for any convolutional neural network, we have to calculate the total number of operations that any single ciphertext of the convolutional neural network goes through. Once we have calculated for the

value of a single ciphertext, then the same parameters are valid for all of the ciphertext inputs individually. When considering only a single ciphertext throughout the network, we observed the following phenomena:

- In a convolution layer, estimating parameters for single filter represents the whole layer irrespective of the number of filters in the layer.
- In a fully connected layer, estimating for a single neuron represents the whole layer irrespective of the number of neurons in the layer.

From our noise calculation formulas, as shown in the Table 4-6, we calculate the encryption parameters for the full as well as the reduced forms of CryptoNets. We show the systematic calculation of the reduced as well as the full form of the CryptoNets networks for complete coverage as follows.

5.3.1 Estimating the initial noise from the encryption

The first step in the usage of an encryption algorithm is to convert our original numerical message in the plaintext form to the encrypted form of a ciphertext. For such a conversion to happen, the original numerical message plaintext has to be encoded in a form usable by the encryption algorithm.

However, in the estimation of encryption parameters, we do not have a fixed number as our original numeric message. Therefore, the estimation formula uses the maximum numerical message that can be provided as input. By using the maximum, the estimation process will certainly accommodate all the smaller numbers between 0 and the maximum.

For understanding CryptoNets as a case, it is necessary to understand the inputs being provided to it. The inputs to the CryptoNets network are grayscale images having a single channel. For each image, we have a matrix of pixels each having a

value from 0 to 255. However, these pixel values are all scaled between 0 and 1. Each pixel of this grayscale image is encrypted separately, therefore, in this example the maximum value that we can have as an input will be the value of 1. This 1 represents the scaling down of 255. Hence, in the estimation formula for the initial encryption noise, $\|m\|$ will get the value of 1.

For this step, the noise is estimated using the formula given in SEAL documentation as [22]:

$$\underline{v_i = \frac{q \bmod t}{q} \cdot \|m\| \cdot N_m + \frac{7nt}{q} \cdot \min\ \text{noiseMaxDeviation}, 6 \times \text{noiseStandardDeviation}}$$

Here, the t, q, and n are the encryption parameters to be estimated, m represents our input message, $\|m\|$ is the maximum of the absolute values of m, and N_m is the highest degree of the polynomial representing m. In our case, the initial values of $\|m\|$ and N_m were set as 1 and 10, respectively. The variables *noiseMaxDeviation* and the *noiseStandardDeviation* controls the randomness of the noise value itself and is normally fixed for the SEAL library.

5.3.2 Estimating for the first convolution layer

After calculating the noise for the initial encryption, we calculate for the first layer of CryptoNets. The first layer is a convolution layer; therefore, we estimate for a single filter as: $\underline{v_1 = N_m \times \|m\| \cdot \left(\frac{q \bmod t}{q} + i \cdot v_k\right)}$. This will eventually represent the whole layer. Here, we have v_0 from step 1, and the value of i is the total number of inputs to a single filter. In the first layer, we have filters of size 5×5, therefore, the value of i is 25.

5.3.3 Estimating for the square layer

After the first convolution layer, we get a ciphertext of noise value v_1. The second layer is the square activation layer, so we calculate the noise for this layer by using

the formula: $\underline{v_2 = \mathrm{t} \cdot \sqrt{3n}\ (144n^2 + 24nv_1) + v_{relin}}$. Here we have 2 summands in which the first one is for the square of the ciphertext, and the second summand of v_{relin} is for relinearization to keep the ciphertext size to 2. v_{relin} is calculated by using the relinearization formula given in the SEAL documentation [22]. Moreover, the value of v_1 is taken as the initial noise calculated in step number 1.

5.3.4 Estimating for the sum-pooling layer

After calculating the noise v_2 for the second layer, we calculate the noise for the sum-pooling layer next. Again, we estimate for a single output of the sum pooling that represents the whole layer as: $\underline{v_3 = iv_2}$. Here, the value of i is 9 because the sum-pooling window size is 3×3.

5.3.5 Estimating for the convolution layer

Next, we calculate for the fourth layer that is a convolution layer. This layer has filters having the window size of 5×5. Similar to the previous convolution layer in step 2, the noise for this layer is calculated as: $\underline{v_4 = N_m \times \|m\| \cdot \left(\frac{q \bmod t}{q} + i \cdot v_3\right)}$. The value of i is 25 because the filter window size is 5×5.

5.3.6 Estimating noise for the sum-pooling layer

After calculating the noise v_4 for the fourth layer, we calculate the noise for the sum-pooling layer. Again, we estimate for a single output of the sum pooling which represents the whole layer as: $\underline{v_5 = iv_4}$. Here, the value of i is 9 because the sum-pooling window size is 3×3.

5.3.7 Estimating for the fully connected layer

The sixth layer of CryptoNets is a fully connected layer having 100 neurons. Here, we calculate the noise for a single output of the layer, that is a single neuron, and this will represent the whole layer. Moreover, a neuron takes all of the output values from the previous layer as its input. Therefore, the noise for this layer is calculated as: $\underline{v_6 = N_m \times \|m\| \cdot \left(\frac{q \bmod t}{q} + i \cdot v_5\right)}$ where the value of i is 1250 because every neuron will take i inputs from the previous layer.

5.3.8 Estimating noise for the square activation layer

The next layer is a square activation layer as in step 3. We calculate the noise for this layer by using the formula: $\underline{v_7 = \mathrm{t} \cdot \sqrt{3n}\ (144n^2 + 24nv_6) + v_{relin}}$. Here we have two summands in which the first one is for the square of the ciphertext, and the second summand of v_{relin} is for relinearization to keep the ciphertext size to 2. v_{relin} is calculated by using the relinearization formula given in the SEAL documentation [22].

5.3.9 Estimating noise for the fully connecting layer

Next, we calculate for the final fully connected layer. The noise for this layer is calculated as: $\underline{v_8 = N_m \times \|m\| \cdot \left(\frac{q \bmod t}{q} + i \cdot v_7\right)}$ where the value of i is 100.

5.3.10 Estimating the encryption parameters for the network

When finished with calculating the noise for all the layers of the network, we get a single ciphertext c_8 having some noise represented by v_8. This ciphertext c_8 noise value is then checked for proper decryption against the encryption parameters (t, q, and n) selected in step 1. If the selected encryption parameters are able to decrypt c_8, then they are returned as the final output of the estimation process. Otherwise, the next set of encryption parameters are selected for another round of noise

estimation and once again, the process is repeated from step 1 until the end. The repetition of noise estimation for all the layers is required because the change of encryption parameters will result in a different amount of noise throughout the layers.

By following the above steps, we are able to estimate the encryption parameters required for the execution of the full form of the CryptoNets. The estimation for the reduced form of CryptoNets work in the same manner with a few differences. The differences are the removal of steps 4, 5, and 6; and changing the value of i from 1250 to 845 in step 7.

The final estimated encryption parameters for the CryptoNets are in the following table.

Table 5-8 Our estimated encryption parameters for CryptoNets

	CryptoNets Full-Network	**CryptoNets Reduced network**
Polynomial modulus (n)	$1x^{8192} + 1$	$1x^{8192} + 1$
Coefficient modulus (q)	218 bits	218 bits
Plaintext modulus (t)	65,536	32,768

After estimating the encryption parameters for both of the network variations of CryptoNets, we get the values for the set of encryption parameters given in Table 5-8. These values for encryption parameters (t, q, and n) tell us that, irrespective of the amount of noise produced from the homomorphic operations of the network, the output ciphertext can be decrypted correctly by using the given t, q, and n. Interestingly, the q and n variables have the same values, for both the networks. This serves as a good example to reiterate that the number of operations that a

single ciphertext goes through in a network dictates the encryption parameters and not the total number of operations in the layers. The total number of operations, however, affects the overall processing time and required memory for a full network.

The estimation of encryption parameters for CryptoNets has been an important case to study. This case has worked as a benchmark for comparing our method of estimating the parameters for the execution of an encrypted convolutional neural network. The benchmarking is done using comparisons with the original encryption parameters reported with our estimated ones. The parameters are given in the following table.

Table 5-9 Comparison with the encryption parameters of CryptoNets (reduced form)

	CryptoNets Parameters [1]	**Our estimated Parameters**
Polynomial modulus (n)	$1x^{8192} + 1$	$1x^{8192} + 1$
Coefficient modulus (q)	384 bits	218 bits
Plaintext modulus (t)	$t_1 = 1099511922689$, $t_2 = 1099512004609$	32768

In Table 5-9, we compare the original encryption parameters used by the CryptoNets researchers with our estimated encryption parameters. Since they did not perform encrypted classification on their full-network, this table shows the comparison of only the reduced-form of the networks. The polynomial modulus is same for both, however, the coefficient and plaintext moduli used by the CryptoNets are larger than actually required. For a fair comparison, it is imperative to share that our work has focused on the classification of a single image. Whereas CryptoNets focuses on packing several images into a single ciphertext to process all in one execution. Therefore, CryptoNets uses two values for t in the CRT method.

This could be one of the reasons that the CryptoNets researchers were required to use the above encryption parameters.

In our estimation of the two CryptoNets network (reduced and full) models, a difference of plaintext modulus exists. The plaintext modulus in the reduced architecture was estimated to be half of the one estimated for the full form. The reason for the difference seems to be the huge range of numbers found in the calculation of the full form of the network. The full-form of the CryptoNets network has several square and sum-pooling layers, which make all the numbers larger than the reduced-form of CryptoNets. Therefore, to cope with this huge range of numbers, the plaintext modulus is double than that in the reduced form of the CryptoNets network.

5.4 LeNet-5 Case Study

LeCun et al. [47] provides the pioneering work inventing the convolutional neural networks. In their work, they share a model of how we can perform image classifications using neural networks. LeCun et al. showed how can a convolutional neural network be used on multi-dimensional data, like images, in an effective and efficient way. Since their work, numerous researchers have improved their network model to achieve classification accuracies above 99%.

5.4.1 Detail of CNN operations in LeNet-5

The total number of encrypted operations for LeNet-5 are listed in Table 5-10:

Table 5-10 Total encrypted operations for the LeNet-5

Legend: C (Ciphertext), P (Plaintext)					
Layer Number	**Layer Type**	**Total $C \times C$**	**Total $C \times P$**	**Total $C + C$**	**Total $C + P$**
1	Convolution	0	117600	112896	4704
2	Average	0	1176	3528	1176
3	Convolution	0	240000	238400	1600
4	Average	0	400	1200	400
5	FCL	0	48000	47880	120
6	FCL	0	0	9996	84
Grand Total		**0**	**407176**	**413900**	**8084**

The table presents the total number of encrypted operations, for all the filters in the convolution layers and the neurons in the fully connected layers as calculated by using the formulas given in Table 4-5.

The total number of operations can only tell us how fast the execution performance can be, but not the exact number of operations that an individual ciphertext goes through. The number of operations for a single ciphertext is essential for estimating the encryption parameters, rather than the total number of operations in a layer. When a convolution layer is provided with an input, then each of the input pixels affects only some of the output values in that layer and not all of them.

Table 5-11 Individual encrypted operations for the LeNet-5

Layer Number	**Layer Type**	**Individual $C \times C$**	**Individual $C \times P$**	**Individual $C + C$**	**Individual $C + P$**
1	Convolution	0	1	0	1
2	Average	0	1	3	0
3	Convolution	0	1	0	1
4	Average	0	1	3	0
5	FCL	0	1	399	1
6	FCL	0	1	119	1
Grand Total		**0**	**0**	**6**	**524**

The number of individual encrypted operations in the LeNet-5 are shown in Table 5-11. These were calculated by using our formulas shown in Table 4-5.

5.4.2 Selecting encryption parameters for LeNet-5

By using our formulas of noise estimation shown in Table 4-6 and the steps as shown in section 5.3 above, we estimate the following encryption parameters for the LeNet-5 network presented in Table 5-12.

Table 5-12 Our estimated encryption parameters for LeNet-5 network

	LeNet-5 Network
Polynomial modulus (n)	$1x^{4096} + 1$
Coefficient modulus (q)	109 bits
Plaintext modulus (t)	8

The estimated encryption parameters shown allow us to perform encrypted classifications and still be able to decrypt the result properly.

If we compare the encryption parameters for LeNet-5 to those of CryptoNets, we see that the LeNet-5 requires comparatively smaller encryption parameters. Interestingly, the total number of encrypted operations for LeNet-5 and the full-form of CryptoNets are very similar. Several reasons exist for the smaller encryption parameters of LeNet-5. One reason is the use of an average-pooling layer instead of the sum-pooling layer. Unlike average pooling, sum pooling results in bigger and bigger numbers. Another reason is the absence of the square activation function that requires the heavy operation of multiplying a ciphertext with itself. Another rationale for the smaller encryption parameters is the comparison of operations that a single ciphertext has to go through by looking at the LeNet-5 Table 5-11 and the CryptoNets Table 5-6. LeNet-5 requires a much smaller number of operations over an individual ciphertext than the CryptoNets, therefore, smaller parameters arise.

5.5 Summary

The aim of this chapter was to observe the parameter selection approach for different CNN models. We consider the usage of our encryption parameter estimation method for two different CNN models: CryptoNets, and LeNet-5. For the CryptoNets case, we compare our estimated parameters with those parameters originally used by CryptoNets. Here the parameters estimated by our method were closer; but smaller than those used by CryptoNets. Hence, this provides the confidence that our estimation method is working as expected. From the LeNet-5 case, we observed that our method can be seen as applicable to a wide variety of convolutional neural network models. Our method can be applied to a convolutional neural network other than CryptoNets and still be able to provide proper encryption parameters.

With the estimation of parameters in the two cases above, we show the mechanism for a machine-learning researcher or an application developer to apply homomorphic

encryption within their convolutional neural network. This in turn will allow developers and researchers to create convolutional neural network models that are good at classification tasks while at the same time preserving the privacy of the users.

Along with the above observations, we also observe that there may be certain convolutional neural network models that are still not practical to perform in encrypted form. Even if we are able to estimate the encryption parameters for those networks, they require a huge number of encrypted operations and that makes them impractical.

Moreover, another important observation from the case studies was the selection of maximum absolute value and the maximum coefficient count. For the sake of the case studies, we fixed the value of the maximum absolute value to be 1 and the maximum coefficient count to be 10. This has so far given us good encryption parameters and provides a 3-digit fractional accuracy.

It is also important to note that the estimation of the encryption parameters is an upper bound and not necessarily the most optimal fixed value. This upper bound can often be an overestimation of the encryption parameters for the CNN model in practice. However, the parameter estimation is based on the latest knowledge currently available and that can be optimised for an improved estimation instead of the upper bound.

6 Probabilistic Optimization of Encryption Parameters

In our previous part of the study, we devised and tested a novel and generic method to determine the required encryption parameters for the evaluation of any feed-forward convolutional neural network. However, it is observed that the parameters are not optimal for every case because they are based on the upper boundary of the noise value in the ciphertext. Therefore, it is necessary to examine the determined parameters and to optimize them further for any specific case one may require.

For optimizing the determined parameters further, one avenue of investigation is the reduction of the parameters based on a probabilistic method. The notion of selecting parameters probabilistically is based on the likelihood of the most common cases for which it will work flawlessly. However, in some cases, the possibility exists that the encrypted operations will fail entirely because of a less likely noise value.

In this chapter, we explore a powerful method for selecting the initial encryption parameters of a feed-forward convolutional neural network. The new proposed method presents a statistical framework for a better approximation of the encryption noise for the neural network under consideration. As a result, the wastage of computing resources can be minimized by selecting tighter and more practical encryption parameters. Furthermore, the new proposed framework allows probabilities to be determined for decryption failures more accurately than the widely used existing methods.

We name such a method for selecting encryption parameters the probabilistic method of selecting encryption parameters. In contrast, the parameter selection

mechanism shown in the previous parts of our study will be called the deterministic method of parameter selection.

The motivation for the reduction of encryption parameters is that we can, first, gain performance boost. Second, we will close the gap of theoretical to practical noise growth.

6.1 Introduction

The solution of a convolutional neural network (CNN) to provide encrypted inference results for encrypted private inputs is not a insignificant and trivial one. This becomes further complicated when we consider the abundance of convolutional neural network architectural models overlapped with the complexity of encryptions. Nonetheless, in the previous parts of our study, we showed how one could utilize the available knowledge of convolutional neural networks and homomorphic encryptions to enable encrypted inferences.

Our work in this research primarily deals with the practical aspects of encrypted classifications and inferences. Consequently, our focus has been on the computational performance of evaluating an encrypted convolutional neural network correctly in the least amount of CPU time. Owing to this focus, we first addressed that how one can adopt homomorphic encryption within a vast array of neural networks. As a second step, we show how one can find the initial encryption parameters required for the correct evaluation of encrypted convolutional neural networks.

The step of setting the initial parameters for any encryption scheme is important and determines the most important characteristics of the schemes themselves. For traditional encryption schemes, that do not allow operations directly on the

ciphertexts, the important characteristics are the speediness of encryption or decryptions steps, and the security of the encrypted ciphertext and their security keys. Unlike the traditional schemes, there are additional characteristics to consider for the homomorphic encryption schemes. These additional characteristics include the speediness of the processing operations directly over the ciphertext; the maximum number of operations correctly performable using a freshly encrypted ciphertext; and the range of numbers permissible as input and output for the encrypted operations. Therefore, the setting of the correct initial parameters has become a more important step to take during the development of a system using a homomorphic encryption scheme.

6.2 Motivation for a statistical noise analysis approach

Fully homomorphic encryption has rightly been called the holy grail of cryptography for the wondrous applications that can benefit from it. In this regard, we have focused our research goals on the practical aspects of homomorphic encryption within neural networks. By practical aspects, we have focused our research to bridge the gap between the latest cryptographic theory and implementations, and their proper usage and execution in neural networks.

Murphy and Player, the authors of the latest noise estimates for the SEAL library have also published their recent work [64] giving a similar framework for Ring-LWE encryption schemes. In their paper [64], they analysed the symmetric key homomorphic cryptosystem given by Lyubashevsky, Peikert and Regev [65] using a statistical framework and compared this with the existing approaches for approximating encryption parameters. After their analysis, they concluded that their statistical approach determines probabilities more accurately and so gives rise to better bounds for decryption failure probabilities in Ring-LWE than the widely used sub-gaussian approach. Their research direction further encourages our

research direction to design a more efficient parameter approximation mechanism for evaluating a neural network using the SEAL library.

6.3 Problem Description

Owing to their importance, the initial parameters for evaluating an encrypted convolutional neural network are necessarily highly optimized. Our method to find the initial encryption parameters is very effective in suggesting encryption parameters for a wide variety of networks irrespective of their complexity. However, we saw room for further improvement to increase the computational performance.

To elucidate the problem statement, the deterministic method for selecting the encryption parameters works by approximating for the maximum possible noise values. It, therefore, selects encryption parameters from a pre-determined list of secure parameters. In our implementation for a convolutional neural network, we count the total number of operations that a single ciphertext goes through. After this count, we approximate the encryption parameters for a single ciphertext to complete the CNN inference in an encrypted form. The approximated encryption parameters always result in selecting bigger parameters than required for the exact computation. The reason for this over-approximation was found to be the original estimations of noise growth for the encrypted operations. Therefore, there was a need to tighten the estimate of noise growth for better approximation of encryption parameters in CNN environments.

The method of calculating the total noise first and then suggesting the minimum encryption parameters to cope with the calculated amount of noise is an effective method. This method gives us the smallest encryption parameters required for the upper-maximum boundary of the possible noise in the encrypted ciphertexts. The handling of the upper noise boundary is effective; however, we do not see it as an

efficient method. From our observations, we see that the determined parameters sometimes have room for a few more operations to be performed on the ciphertexts. Therefore, there is a wastage of computational and memory resources which can be avoided.

The selection of encryption parameters for the most frequent amount of noise instead of the maximum noise has associated benefits as well as penalties. The most prominent benefit will be the improved computational performance due to smaller parameters. In this way, the wastage of computational and memory resources can be reduced. Conversely, the reduction in parameters can cause some homomorphic operations to produce incorrect results. In cases where the noise within a ciphertext is bigger than the parameters have been optimized for, the decryption step will yield an incorrect result.

6.3.1 Main Contributions

This chapter examines a novel method of probabilistically optimising the parameters for homomorphic encryption in the inference stage of neural networks. To define the scope for practical reasons, this chapter explores the theoretical and empirical feasibility of using a probabilistic method for optimising the encryption parameters for the FV encryption scheme as implemented in SEAL. The optimisation is based on the likelihood of the maximum noise threshold in the final output(s) of a neural network.

6.4 Related Work

It is common to use analytic methods to set a very tight boundary of the random noise with the required probability. Lyubashevsky et al. [65] gave a toolkit of algorithms and analytical techniques that can be used in a variety of ring-based cryptographic applications. Their analytic method for the noise bounds shows how

one can adopt a statistical approach to find very tight boundary of noise with a required probability. Their algorithms, for all the main cryptographic operations, are able to execute faster relative to the best-known techniques. Among their algorithms, they provided specialized techniques for the generation of noise terms under probability distributions by guaranteeing the cryptographic hardness as well as decoding of the noise terms of a ciphertext for decryption and related operations. Besides the algorithms, they also provide analytical methods of bounding the noise growth for operations like addition, multiplication, and discretization. Their analytical methods give high level and generic analysis of noise terms for ring-LWE based encryption schemes. In their proofs for the noise bounds in homomorphic operations, they provided the statistical tail bound on sub-gaussian random variables that can be used within applications.

In the same context, Murphy and Player [66] improved and corrected the statistical analysis framework of Lyubashevsky et al. [65] for using the sub-gaussian random variables to analyse noise growth of homomorphic operations. In comparison, these derive the noise bounds by applying the central limit theorem to random variables and show that this is a more correct approach.

In a separate work, Bos et al. [67] show that the calculation of probability for a successful decryption is used commonly in the cryptographic community. In their work, they provide several post-quantum cryptographic primitives, for their encryption scheme, that are also secure against CPA (chosen-plaintext attacks). More importantly, they also show the correctness of their scheme in terms of the probability of successful decryption. Bos et al. state their definition of the probability which has been implemented in their code as well Bos et al. Theorem 1 [67].

6.5 Noise Analysis

We begin by first finding the probability density of the noise terms in any SEAL ciphertext. For this, we will analyse and follow the lifecycle of the noise terms from their generation step to the final decryption step.

We know the details of encryption and decryption from the existing literature as detailed in section 2.3. The next step after understanding the decryption of a ciphertext is the analysis of the noise polynomial v embedded in ciphertexts. From section 2.3.9 above, we know that a SEAL ciphertext ct is represented as:

$$ct = \left(\left[pk_0 \cdot u + e_1 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m \right]_q, [pk_1 \cdot u + e_2]_q \right)$$

From a decryption perspective, the expansion of the public key in ct and removal of the masking will provide the following polynomial to decrypt.

$$\left[e_2 s + e_0 u + e_1 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m \right]_q = \left[v + \left\lfloor \frac{q}{t} \right\rfloor \cdot m \right]_q .$$

In this way, we can mathematically follow and extract the terms for the noise polynomial v. Let us consider only the noise terms without our message m in the ciphertext as in equation 2.2 section 2.3.9 above,

$$v = [e_2 s + e_0 u + e_1]_q$$

After extracting the exact random terms from a ciphertext, we are able to calculate the probability distribution from where all of the random coefficients for the polynomial v are taken. The method to find the final probability distribution of v will be to use the algebra for random variables in order to calculate the combined mean and variance for all the terms in v (i.e., e, s, and u). These noise terms for v are taken from known probability distributions as discussed subsequently. Moreover, they are all encoded as positive values in the ciphertext, therefore, we

will also take the absolute of the random values for our analysis. These calculations are as follows.

6.5.1 Solving for e_1

To find the mean and standard deviation for e_1 in v, we will have to consider how it is generated as well as encoded into a ciphertext. The distribution of random variables represented by e is known from the SEAL literature [22]. Therefore, we start by taking a normal random variable r_1 having a mean μ_r of 0 and a standard deviation σ_r of $\frac{8}{\sqrt{2\pi}}$.

$$r_i = N\left(0, \left(\frac{8}{\sqrt{2\pi}}\right)^2\right)$$

We know from the SEAL implementation as well as theory that all the values are integers, and they are only positive. In SEAL code, normal random values are generated, their absolute is taken and then this is rounded off to the nearest integer. Therefore, we will convert our normal random variables to folded normal random variables [68]. This is done by taking the absolute of each of the r_1 which gives us a folded normal variable e_1 without any negative values so that $e_i = |r_i|$. This conversion will affect the new mean μ_e and standard deviation σ_e as follows [68].

$$\mu_e = \sigma\sqrt{\frac{2}{\pi}}\ \exp\left(\frac{-\mu^2}{2\sigma^2}\right) + \mu\ \mathrm{erf}\left(\frac{\mu}{\sqrt{2\sigma^2}}\right)$$

After putting in the starting values of μ and σ, we get the following expression for the mean μ_e for any single random variable e_i:

$$\mu_e = \frac{8}{\sqrt{2\pi}}\sqrt{\frac{2}{\pi}}\ \exp\left(\frac{0}{2\left(\frac{8}{\sqrt{2\pi}}\right)^2}\right) + 0 \cdot \mathrm{erf}\left(\frac{0}{\sqrt{2\left(\frac{8}{\sqrt{2\pi}}\right)^2}}\right)$$

$$\mu_e = \frac{8}{\pi} \tag{6.1}$$

$$\mu_e = 2.546479089470325372302140213960229792551354331847303179962\ldots$$

After finding the mean value for one of the folded normal variable, we find the variance [68] as:

$$\begin{aligned}\sigma_e^2 &= \mu_r^2 + \sigma_r^2 - \mu_e^2 \\ &= 0^{\,2} + \left(\frac{8}{\sqrt{2\pi}}\right)^2 - \left(\frac{8}{\pi}\right)^2 \\ &= \frac{32}{\pi} - \frac{64}{\pi^2}\end{aligned} \tag{6.2}$$

$$\sigma_e^2 = 3.701360604771684116800275210418350280326455748365433593731\ldots$$

In other terms, the standard deviation of a single e random variable will be:

$$\sigma_e = \sqrt{\frac{32}{\pi} - \frac{64}{\pi^2}} = 1.923892046028488604916105466 81\ldots \tag{6.3}$$

This outcome was confirmed experimentally as well, resulting in almost exact values for the mean as well as the variance.

6.5.2 Solving for $e_0 u$

To find the mean and standard deviation of $e_0 \times u$, we will calculate for the product of two random distributions. The e_0 is a folded normal distribution for which we have already found the mean and variance in equation 6.1 and 6.2 respectively. The second term of the product is a discrete uniform distribution u from the set $\{-1,0,1\}$. However, we will take the absolute of this distribution resulting in a distribution with the values of 0 and 1. After taking the absolute of the distribution, probability distribution of -1 will be added onto the probability distribution of 1. After understanding this, we will now calculate the mean and variance for $e_0 u$.

6.5.2.1 Mean $\mu_{e_0 u}$

After taking the absolute of the discrete uniform distribution, we will get a probability distribution of $P\ 0\ = \frac{1}{3}$, and $P\ 1\ = \frac{2}{3}$. Therefore, the mean will be

$$\mu_u = 0 \cdot \frac{1}{3} + 1 \cdot \frac{2}{3} = \frac{2}{3} \approx 0.6666666667$$

The mean of e_0 is the same as e_1 in equation 6.1, therefore

$$\begin{aligned} \mu_{e_0 u} &= \mu_{e_0} \cdot \mu_u \\ &= \frac{8}{\pi} \cdot \frac{2}{3} = \frac{16}{3\pi} \approx 1.697652726 \end{aligned} \tag{6.4}$$

6.5.2.2 Variance $\sigma^2_{eu_c}$

Similar to above, we find the variance

$$\begin{aligned} \sigma_u^2 &= \sum \quad x - \mu\ ^2 \cdot P\ x \\ &= \left(0 - \frac{2}{3}\right)^2 \cdot \frac{1}{3} + \left(1 - \frac{2}{3}\right)^2 \cdot \frac{2}{3} \\ &= \frac{2}{9} \approx 0.2222222222 \end{aligned}$$

Hence, the standard deviation of u will be

$$\sigma_u = \sqrt{\frac{2}{9}} = \frac{\sqrt{2}}{3} \approx 0.4714045208$$

The variance of e_0 is the same as e_1 in equation 6.2, therefore

$$\begin{aligned} \sigma^2\ e_0 u\ &= \sigma^2\ X\ \sigma^2\ Y\ + \sigma^2\ X\ \left(\mu\ Y\ \right)^2 + \sigma^2\ Y\ \left(\mu\ X\ \right)^2 \\ &= \left(\frac{32}{\pi} - \frac{64}{\pi^2}\right) \cdot \frac{2}{9} + \left(\frac{32}{\pi} - \frac{64}{\pi^2}\right) \left(\frac{2}{3}\right)^2 + \frac{2}{9} \cdot \left(\frac{8}{\pi}\right)^2 \\ &= \frac{192 \cdot \pi - 256}{9 \cdot \pi^2} \approx 3.908586126 \end{aligned} \tag{6.5}$$

Hence, the final standard deviation of σ_{e_0u} will be

$$\sigma_{e_0u} = \sqrt{\frac{192 \cdot \pi - 256}{9 \cdot \pi^2}}$$

$$= \frac{\sqrt{192 \cdot \pi - 256}}{3 \cdot \pi} \approx 1.977014448 \tag{6.6}$$

This outcome was confirmed experimentally as well, resulting in almost exact values for the mean as well as the variance.

6.5.3 Solving for e_2s

The mean and variance for e_2s is the same as e_0u. This is because the nature of the random variables involved is the same. The e_2 and e_0 random variables have the same mean and variance as shown in equation 6.1 and 6.2 respectively. Similarly, the s and the u random variables have the same mean and variance. Therefore, the mean and variance for e_2s will be as follows.

The mean μ_{e_2s} of the product of e_2s will be

$$\mu_{e_2s} = \mu_{e_2} \cdot \mu_s$$

$$= \frac{8}{\pi} \cdot \frac{2}{3} = \frac{16}{3\pi} \approx 1.697652726 \tag{6.7}$$

The variance $\sigma^2_{e_2s}$ of e_2s similar to equation 6.5 will be

$$\sigma^2_{e_2s} = \frac{192 \cdot \pi - 256}{9 \cdot \pi^2} \approx 3.908586126 \tag{6.8}$$

6.5.4 Calculating for v

We summarise the means and variances of all the random variables for v in Table 6-1.

Table 6-1 Mean and variance of the terms of v

Term	**Mean**	**Variance**
$\boldsymbol{e_1}$	$\frac{8}{\pi} \approx 2.546479089$	$\frac{32}{\pi} - \frac{64}{\pi^2} \approx 3.701360605$
$\boldsymbol{e_0 u}$	$\frac{16}{3\pi} \approx 1.697652726$	$\frac{192 \cdot \pi - 256}{9 \cdot \pi^2} \approx 3.908586126$
$\boldsymbol{e_2 s}$	$\frac{16}{3\pi} \approx 1.697652726$	$\frac{192 \cdot \pi - 256}{9 \cdot \pi^2} \approx 3.908586126$

The next step is to sum the three random variables as calculated above for v. Recall that the v polynomial is represented as:

$$v = [e_2 s + e_0 u + e_1]_q$$

The final mean value of the random variable v is the sum of all the mean values of its terms. This will be as follows.

$$\mu_v = \frac{8}{\pi} + \frac{16}{3\pi} + \frac{16}{3\pi} = \frac{56}{3 \cdot \pi} \approx 5.941784542 \tag{6.9}$$

Moreover, the sum of the variances of the terms of v will be as follows.

$$\sigma_v^2 = \left(\frac{32}{\pi} - \frac{64}{\pi^2}\right) + \left(\frac{192 \cdot \pi - 256}{9 \cdot \pi^2}\right) + \left(\frac{192 \cdot \pi - 256}{9 \cdot \pi^2}\right)$$

$$= \frac{672 \cdot \pi - 1088}{9 \cdot \pi^2} \approx 11.51853286 \tag{6.10}$$

Hence, the final standard deviation for v will be

$$\sigma_v = \sqrt{\frac{672 \cdot \pi - 1088}{9 \cdot \pi^2}} = \frac{\sqrt{672 \cdot \pi - 1088}}{3 \cdot \pi} \approx 3.393896412 \tag{6.11}$$

6.5.5 Experimental analysis

The mean and variance values were independently confirmed with a repeated Monte Carlo simulation of our theoretical analysis. The simulation was performed through python code in SageMath[7] (explained in 5.2.4 above) by generating huge arrays of random numbers and combining them through same arithmetic operations to create a final array of values representing v. The simulation results yielded same or extremely close values. The density histograms from the simulations are shown in Figure 6-1 to Figure 6-5 and display as below:

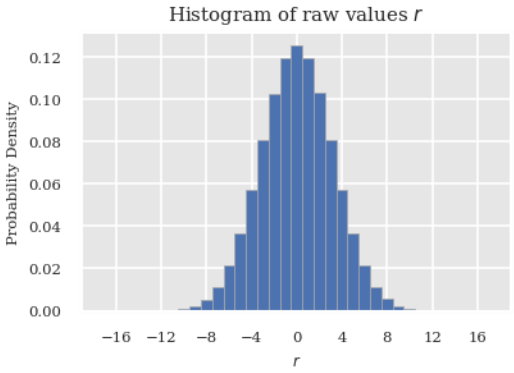


Figure 6-1 Histogram of the simulation of normal random variables r in section 6.5.1

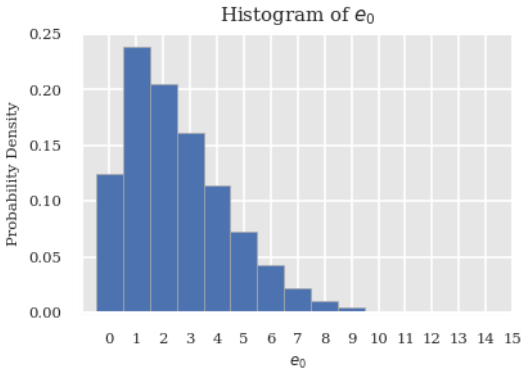
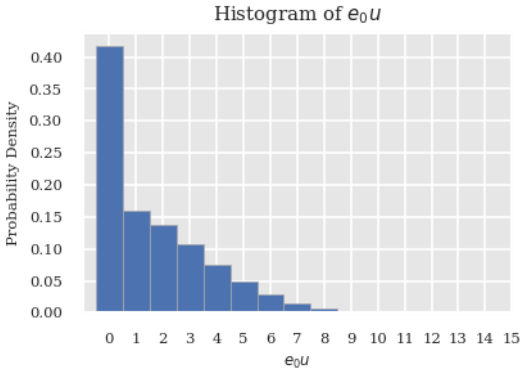
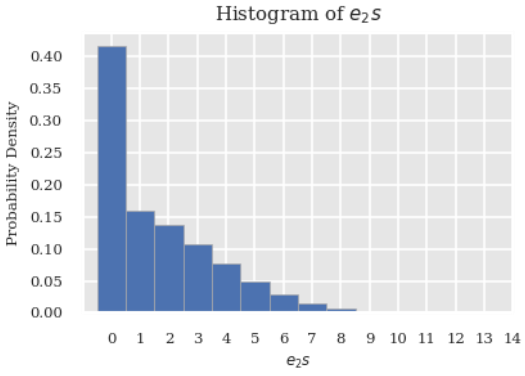


Figure 6-2 Histogram of the simulation of folded normal variables e in section 6.5.1





[7] https://www.sagemath.org/

Figure 6-3 Histogram of the simulation of random variables in section 6.5.2

Figure 6-4 Histogram of the simulation of random variables in section 6.5.3

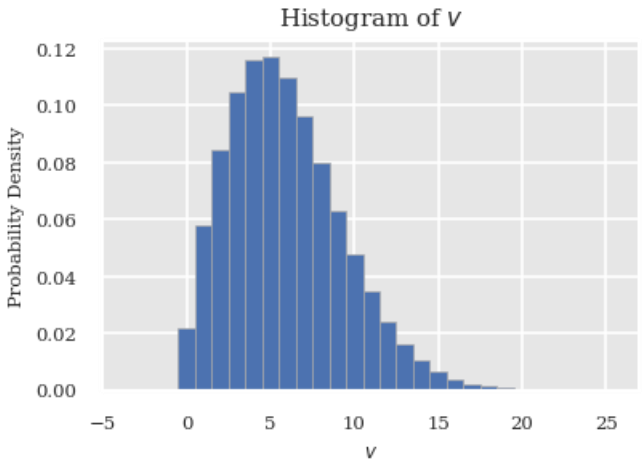


Figure 6-5 Histogram of the simulation of random variable in section 6.5.4

Interestingly, the simulated v in Figure 6-5 shows that the final random distribution of v looks like a log-normal distribution. For this distribution, we know the mean μ_v and the variance σ_v^2. However, we must be cautious and consider that this is an independent analysis of only the noise coefficients apart from the encryption scheme. In the encryption scheme, these coefficients are part of the encryption ciphertexts with the coefficients modulo q. The negative values, for example $-v_i$, in SEAL are encoded as $q - \ -v_i \bmod q$ for each of the negative value coefficients.

In this section until now, we have calculated the mean, variance, and standard deviation of noise within a ciphertext. Next, we will see how this analysis can help us to find the change in the distribution of noise coefficients after homomorphic operations.

6.6 Noise after homomorphic operations

In the FV scheme, we are able to perform addition and multiplication of numbers despite the fact that they are still encrypted. Any homomorphic operation

performed over the ciphertexts will give us another ciphertext as a result. The resultant ciphertext will hold the outcome of the operation, say addition or multiplication, along with the noise v. The noise value in the resultant ciphertext increases with the number of consecutive homomorphic operations when the resultant ciphertext of a previous homomorphic operation is used. For a better understanding of this increase of noise, we provide a similar analysis to that of Hardy [69]. This is given below.

6.6.1 Homomorphic addition of two ciphertexts

Consider two ciphertexts ct_a and ct_b, both encrypted under the same encryption keys.

$$ct_a = \left(\left[pk_0 \cdot u_a + e_1 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m_a \right]_q , [pk_1 \cdot u_a + e_2]_q \right)$$

$$ct_b = \left(\left[pk_0 \cdot u_b + e_3 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m_b \right]_q , [pk_1 \cdot u_b + e_4]_q \right)$$

The two ciphertexts ct_a and ct_b, use the same public key polynomial pairs of $pk_0 = [-as + e]_q$ and $pk_1 = a$ generated under the same n, t, q initialization parameters. However, the noise polynomials of the ciphertexts are different. These are the u_a and u_b, and the e_1, e_2, e_3, e_4 noisy error polynomials. The addition of the two ciphertexts can be stated as below:

$$\begin{aligned}
ct_c &= ct_a + ct_b \\
&= \left(\left[pk_0 \cdot u_a + e_1 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m_a \right]_q , [pk_1 \cdot u_a + e_2]_q \right) \\
&\qquad + \left(\left[pk_0 \cdot u_b + e_3 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m_b \right]_q , [pk_1 \cdot u_b + e_4]_q \right) \\
&= \Big([pk_0 \cdot \ u_a + u_b \ + \ e_1 + e_3 \ + \left\lfloor \frac{q}{t} \right\rfloor \cdot \ m_a + m_b \ ]_q \ , \\
&\qquad\qquad [pk_1 \cdot \ u_a + u_b \ + (e_2 + e_4)]_q \Big) \\
&= \Big([pk_0 \cdot \ u_c \ + \ e_5 \ + \left\lfloor \frac{q}{t} \right\rfloor \cdot \ m_c \ ]_q \ , \qquad [pk_1 \cdot \ u_c \ + (e_6)]_q \Big)
\end{aligned}$$

From the above simplification, we see that the noisy error terms are fundamentally combined into new error terms by a direct addition. This is shown in the noise estimate for adding the two ciphertexts having noise v_a and v_b in the SEAL documentation (see Appendix A) as:

$$v_o = v_a + v_b$$

Irrespective of the scaling by $\lfloor \frac{q}{t} \rfloor$, the resultant message m_c is ultimately a direct addition within the ciphertext. The final ciphertext ct_c is represented as below with new noise terms.

$$ct_c = \left([pk_0 \cdot u_c + e_5 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m_c]_q, [pk_1 \cdot u_c + e_6]_q \right)$$

From a decryption perspective, the expansion of the public key in ct_c and removal of the masking in the similar way will provide the following polynomial to decrypt.

$$\begin{aligned} &= \left[e_6 s + e u_c + e_5 + \left\lfloor \frac{q}{t} \right\rfloor \cdot m_c \right]_q \\ &= \left[v + \left\lfloor \frac{q}{t} \right\rfloor \cdot m_c \right]_q \end{aligned}$$

From this polynomial, we can extract the same terms for the noise polynomial v as:

$$v = [e_6 s + e_0 u_c + e_5]_q$$

Here, as long as the new noise values are not too large, then we can decrypt the result successfully. To be more specific, if any of the coefficient in the noise polynomials is bigger than $\lfloor \frac{q}{2t} \rfloor - \frac{t}{2}$, then the decryption will fail. This is based on the proof given by Player in Lemma 35 [57].

6.6.2 Homomorphic multiplication of two ciphertexts

The noise estimate for multiplying two ciphertexts as given in the SEAL documentation (see Appendix A) is:

$$v_o = t \cdot \sqrt{3n} \cdot \left[\; 12n^{\;\frac{j_1}{2}} \cdot v_2 + \; 12n^{\;\frac{j_2}{2}} \cdot v_1 + \; 12n^{\;\frac{j_1+j_2}{2}} \right]$$

We treat the noise v as randomly distributed and find the mean and standard deviation of the output noise v_o. Both the input ciphertexts have the mean μ_i and standard deviation σ_i. These are given in Table 6-2 and Table 6-3.

Table 6-2 Defining the μ and σ of v in the operands for a ciphertext product.

Ciphertext ct_1		**Ciphertext ct_2**	
Mean μ_1	**SD σ_1**	**Mean μ_2**	**SD σ_2**
$12n^{\;\frac{j_1}{2}} \cdot \mu_1$	$12n^{\;\frac{j_1}{2}} \cdot \sigma_1$	$12n^{\;\frac{j_2}{2}} \cdot \mu_2$	$12n^{\;\frac{j_2}{2}} \cdot \sigma_2$

We calculate the Mean μ_v and Standard Deviation σ_v of the output noise as:

Table 6-3 Calculation steps for finding μ and σ of v in a ciphertext product.

Mean μ_v	**Standard Deviation σ_v**
$= \left[\; 12n^{\;\frac{j_1}{2}} \cdot \mu_1 \right] + \left[\; 12n^{\;\frac{j_2}{2}} \cdot \mu_2 \right]$	$= \left[\; 12n^{\;\frac{j_1}{2}} \cdot \sigma_1 \right]^2 + \left[\; 12n^{\;\frac{j_2}{2}} \cdot \sigma_2 \right]^2$
$= \left[\; 12n^{\;\frac{j_1}{2}} \cdot \mu_1 \right] + \left[\; 12n^{\;\frac{j_2}{2}} \cdot \mu_2 \right] + \; 12n^{\;\frac{j_1+j_2}{2}}$	
$= t \cdot \sqrt{3n} \cdot \left\{ \left[\; 12n^{\;\frac{j_1}{2}} \cdot \mu_1 \right] + \left[\; 12n^{\;\frac{j_2}{2}} \cdot \mu_2 \right] + \; 12n^{\;\frac{j_1+j_2}{2}} \right\}$	$= t \cdot \sqrt{3n} \cdot \sqrt{\left[\; 12n^{\;\frac{j_1}{2}} \cdot \sigma_1 \right]^2 + \left[\; 12n^{\;\frac{j_2}{2}} \cdot \sigma_2 \right]^2}$
	Simplifying: $= t \cdot \sqrt{3n} \cdot \sqrt{\; 12n^{\;j_1} \cdot \sigma_1^2 + \; 12n^{\;j_2} \cdot \sigma_2^2}$

From the calculation and simplifications shown in Table 6-3, we get the following mean and standard deviation for the noise in the product of two ciphertexts.

$$\mu_v = t \cdot \sqrt{3n} \cdot \left\{ \left[\ 12n^{\ \frac{j_1}{2}} \cdot \mu_1 \right] + \left[\ 12n^{\ \frac{j_2}{2}} \cdot \mu_2 \right] + \ 12n^{\ \frac{j_1+j_2}{2}} \right\} \tag{6.12}$$

$$\sigma_v = t \cdot \sqrt{3n} \cdot \sqrt{\ 12n^{\ j_1} \cdot \sigma_1^2 + \ 12n^{\ j_2} \cdot \sigma_2^2} \tag{6.13}$$

6.6.3 Squaring a ciphertext

When we multiply a ciphertext ct_1 with another ciphertext ct_2 that have noises v_1 and v_2 respectively, and have polynomial array sizes of j_1 and j_2, respectively, then the maximum noise v_o in the output will be calculated as:

$$v_o = t \cdot \sqrt{3n} \cdot \left[\ 12n^{\ \frac{j_1}{2}} \cdot v_2 + \ 12n^{\ \frac{j_2}{2}} \cdot v_1 + \ 12n^{\ \frac{j_1+j_2}{2}} \right]$$

As both the ciphertexts are the same in the case of square, we set the polynomial array sizes j_1 and j_2 to be j, and similarly use v instead of v_1 and v_2. each with the same value because both the ciphertexts are the same. If we are always keeping the array size of the ciphertext polynomial to its smallest value, $j = 2$, then this formula can be further simplified as to get our un-relinearized ciphertext maximum noise v_o as:

$$\begin{aligned} v_o &= t \cdot \sqrt{3n} \cdot \left[\ 12n^{\ \frac{j}{2}} \cdot v_i + \ 12n^{\ \frac{j}{2}} \cdot v_i + \ 12n^{\ \frac{j+j}{2}} \right] \\ &= t \cdot \sqrt{3n} \cdot \left[2^{2\ +\ 1} \cdot 3^{2/2} \cdot v_i \cdot n^{2/2} \ + \ 12^2 \cdot n^2 \right] \\ &= \mathrm{t} \cdot \sqrt{3n} \ (144n^2 + 24nv_i) \end{aligned} \tag{6.14}$$

So, from the equation above, if we have input noise v_i, then we get v_o as our output noise by following these steps:

1. Multiply $24 \cdot n$ with v_i.
2. Add $144n^2$ to the noise in step 1.
3. Multiply $\mathrm{t} \cdot \sqrt{3n}$ to the noise in step 2.

4. Add the number r_{relin} for the noise of relinearization performed through the equation 2.3 on page 36.

$$v_o = v_i + \frac{2t}{q} \cdot \text{minimum}\{B, 6\sigma\} \cdot \ J - K \ \cdot n \cdot \ l + 1 \ \cdot z$$

$$v_o = v_i + r_{relin} \ \left[\text{where } r_{relin} = \frac{2t}{q} \cdot \min\{B, 6\sigma\} \cdot \ J - K \ \cdot n \cdot \ l + 1 \ \cdot z\right]$$

Here, minimum$\{B, 6\sigma\}$ is taken as the minimum of the maximum noise deviation value B, and the noise standard deviation times six as 6σ. Then the J is the current polynomial array size of the ciphertext (which in our case will be 3 after $C \times C$) and K is the target polynomial array size of the ciphertext (which in our case is 2). The $(l + 1)$ corresponds to the evaluation key size. Here, z is the number base for the relinearization.

Now that we have established the steps to find the maximum noise boundary after a square function, we can calculate the mean and standard deviation for the output random noise. Let us consider that the ciphertext had mean μ_v and standard deviation σ_v before the square function. Then by following the four steps of calculation from above, we will get:

1. The mean as $24n \cdot \mu_v$, and the standard deviation as $24n \cdot \sigma_v$
2. The mean as $144n^2 + \ 24n \cdot \mu_v$, and the standard deviation as $24n \cdot \sigma_v$
3. The mean as $\text{t} \cdot \sqrt{3n} \cdot [144n^2 + \ 24n \cdot \mu_v \ ]$, and the standard deviation as $\text{t} \cdot \sqrt{3n} \cdot \ 24n \cdot \sigma_v$
4. The mean as $\{\text{t} \cdot \sqrt{3n} \cdot [144n^2 + \ 24n \cdot \mu_v \ ]\} + r_{relin}$, and the standard deviation as $\text{t} \cdot \sqrt{3n} \cdot \ 24n \cdot \sigma_v$.

The distribution of the random noise in a ciphertext having an initial mean μ_v and the standard deviation σ_v as follows:

$$\mu_{\text{square}} = \{\mathrm{t} \cdot \sqrt{3n} \cdot [144n^2 + \ 24n \cdot \mu_v \ ]\} + r_{relin} \tag{6.15}$$

$$\sigma_{\text{square}} = \mathrm{t} \cdot \sqrt{3n} \cdot \ 24n \cdot \sigma_v \tag{6.16}$$

6.6.4 Concrete effect of homomorphic operations

Putting the values in the maximum noise estimates provided by SEAL (as shown in Appendix A) gives us the values for mean and standard deviation of the changed distribution of the noise. Table 6-4 summarises the effect on the noise distribution.

Table 6-4 Effect of Ciphertext operations on the noise

Step	**Mean** μ_v	**SD** σ_v
Fresh Encryption	$\frac{56}{3 \cdot \pi}$ ≈ 5.941784542	$\frac{\sqrt{672 \cdot \pi - 1088}}{3 \cdot \pi}$ ≈ 3.393896412
Plaintext Multiplication	$\mu_v \times \ N_m \cdot \|m\|$	$\sigma_v \times \ N_m \cdot \|m\|$
Plaintext Addition	$\mu_v + \left(\frac{q \bmod t}{q} \cdot N_m \cdot \|m\|\right)$	σ_v
Ciphertext Addition	$\mu_v + \mu_v$	$\sqrt{\sigma_v^2 + \sigma_v^2}$
Square + Relinearize	$\{\mathrm{t} \cdot \sqrt{3n} \cdot [144n^2 + \ 24n \cdot \mu_v \ ]\} + r_{relin}$	$\mathrm{t} \cdot \sqrt{3n} \cdot \ 24n \cdot \sigma_v$
Ciphertext Multiplication	$t \cdot \sqrt{3n} \cdot \left\{ \left[\ 12n^{\ \frac{j_1}{2}} \cdot \mu_1 \right] + \left[\ 12n^{\ \frac{j_2}{2}} \cdot \mu_2 \right] + \ 12n^{\ \frac{j_1+j_2}{2}} \right\}$	$t \cdot \sqrt{3n} \cdot \sqrt{\ 12n^{\ j_1} \cdot \sigma_1^2 + \ 12n^{\ j_2} \cdot \sigma_2^2}$

6.7 Noise after processing a single neuron

In order to calculate for a full network, we will have to start from a single neuron. The arithmetic operations after a single neuron will yield a single output ciphertext, say ct_o, as the output of the neuron. Whereas, there can be i ciphertext inputs to the neuron. The precise sequence of operations performed to produce the output from the inputs will be as follows:

- Input multiplied with weight value:

$$\text{ciphertext} \times \text{plaintext (incurs } v_{pmult})$$

- Summation of all the weighted values:

$$\text{ciphertext} + \text{ciphertext (incurs } v_{cadd})$$

- Adding of the bias value:

$$\text{ciphertext} + \text{plaintext (incurs } v_{padd})$$

Recall that each operation will incur a certain random noise represented by corresponding v. As we have already found the random distribution for the noise terms, we will now consolidate them to find the precise random distribution of the noise in the output ciphertext ct_o.

In a neuron, first each input ciphertext ct_i will be multiplied by its corresponding plaintext weight value. Since these operations are performed independently of each other, therefore the noise distribution will be the same for each of the weighted products. Consider that the plaintext weight value has a maximum range of $N_m \cdot \|m\|$. Then the noise distribution of each of the weighted encrypted products will have

$$\mu_{pmult} = \frac{56}{3 \cdot \pi} \times \ N_m \cdot \|m\|$$
$$\sigma_{pmult} = \ 3.393896412 \times \ N_m \cdot \|m\|^{\ 2}$$

Next, these weighted products are added together to produce a single encrypted weighted sum. The encrypted sum will incur a noise, say v_{cadd}, and this depends on the total number of inputs i to be added together. Therefore, the weighted sum will have a noise distribution as

$$\mu_{cadd} = \mu_{pmult} \times i$$
$$\sigma_{cadd} = \sigma_{pmult} \times i^2.$$

Next, we add a bias value to the sum to produce the final noise distribution after a neuron, say v_{neuron}. For this, we will have to add a plaintext constant, say P_{max}, to v_{cadd} to represent the addition of the bias value with the weighted sum in a neuron. The maximum value of the bias value is represented by

$$P_{max} = \frac{q \bmod t}{q} \cdot N_m \cdot \|m\|$$

Therefore, putting the values in v_{cadd}, we get the distribution of v_{neuron} as follows.

For the mean based on Table 6-4,

$$\begin{aligned}
\mu_{neuron} &= \mu_{cadd} + \left(\frac{q \bmod t}{q} \times N_m \times \|m\|\right) \\
&= (\mu_{pmult} \times i) + \left(\frac{q \bmod t}{q} \times N_m \times \|m\|\right) \\
&= \left[\left(\frac{56}{3 \cdot \pi} \times N_m \times \|m\|\right) \times i\right] + \frac{q \bmod t}{q} \cdot N_m \times \|m\| \\
&= \left(\frac{q \bmod t}{q} + \frac{56 \times i}{3\pi}\right) \cdot N_m \times \|m\| \qquad 6.17
\end{aligned}$$

For the standard deviation based on Table 6-4, we put in the values and get

$$\begin{aligned}
\sigma_{neuron} &= \sigma_{pmult} \times i^2 \\
&= 3.393896412 \times N_m \cdot \|m\|^{\,2} \times i^2 \qquad 6.18
\end{aligned}$$

Based on equations 6.17 and 6.18, Table 6-5 summarizes the distribution of the noise v_{neuron} in the output from a single neuron.

Table 6-5 Distribution of noise after a single neuron

μ_{neuron}	σ_{neuron}
$\left(\frac{q \bmod t}{q} + \frac{56 \times i}{3\pi}\right) \cdot N_m \times \|m\|$	$3.393896412 \times \ N_m \cdot \|m\|^{\ 2} \times i^2$

Here, for any single neuron of the first layer in a neural network, v_{neuron} represents the random distribution from which each of the coefficients of the noise in ct_o is sampled. This finding will help us find the noise for a complete neural network.

From the perspective of the first layer in a network, all the ciphertext outputs will have the same distribution as characterized in equations 6.17 and 6.18. A summary of the procedure for calculating the equations is presented in Table 6-6 below.

Table 6-6 Effect of homomorphic operations on the noise in a weighted layer of a neural network

Step	**Description**	**Change in Mean μ_v**	**Change in SD σ_v**
1	Generate random noise variates and perform encryption of input	$\frac{56}{3 \cdot \pi} \approx 5.941784542$	$\frac{\sqrt{672 \cdot \pi - 1088}}{3 \cdot \pi} \approx 3.393896412$
2	Multiply encrypted ciphertext with a plaintext weight value	$\mu_v \times \ N_m \cdot \|m\|$	$\sigma_v \times \ N_m \cdot \|m\|$
3	Sum all the weighted products	$\mu_v + \mu_v + \cdots$	$\sqrt{\sigma_v^2 + \sigma_v^2 + \cdots}$

4	Add the plaintext bias value to the weighted sum.	$\mu_v + \left(\frac{q \bmod t}{q} \cdot N_m \cdot \|m\|\right)$	σ_v

A similar procedure will be adopted for further layers by calculating the mean and standard deviation until the final layer.

Next, we will see how this knowledge can be used for optimizing encryption parameters.

6.8 Probabilistic Optimization Method

Once we find the noise distribution of our final layer then we can find the probability of our maximum acceptable noise within the final distribution. According to the proof given in Lemma 35 of Player [57], the maximum noise value for any particular combination of q and t will be $\lfloor \frac{q}{2t} \rfloor - \frac{t}{2}$. An algorithm is devised to find the exact probability for the successful decryption of a ciphertext.

Algorithm 1: Finding probability of successful decryption.

Algorithm 1: Finding the probability of a successful decryption

Input: Encryption parameters: q,n,t, Ciphertext: ct
Output: probability of decrypting ct successfully.
1 **Calculate the mean μ_{ct} and the standard deviation σ_{ct} for the noise in ct;**
2 **Calculate the maximum noise value that can be decrypted using the encryption parameters using** $v_{max} = \frac{q}{2t} - \frac{t}{2}$;
3 **Find and return** $1-$**(the Probability of v_{max} or greater) in the noise distribution having μ_{ct} and σ_{ct} ;**

6.9 Validation and implementation of the statistical noise analysis approach

The algorithm as stated above has 3 main independent steps to find the exact probability for a successful decryption. Step 1 deals with the calculation of the mean and standard deviation. This has been given in detail in sections 6.5, 6.6, and 6.7.

Step 2 is trivial and only requires putting the values of the encryption parameters q and t in the equation to find the maximum decryptable noise value.

However, step 3 requires us to know the shape and nature of the probability distribution for which we have calculated the mean and standard deviation. From our simulation as shown in section 6.5.5 above, we saw that the final distribution of the noise was shaped as a normal distribution, as shown in Figure 6-5. However, the histogram of the simulated noise only shows the random distribution independent of the encoding mechanism used in the SEAL encryption. Therefore, we performed two independent validation tests to find more precise information about the final probability distribution. These tests will also serve as a validation for the working of the probabilistic parameter optimization method.

First, we tried checking the probability of the maximum noise coefficient value in known distributions like the normal distribution and folded normal distribution. For this, a SageMath[8] module (explained in 5.2.4 above) was developed so that there is not a limitation on how huge the numbers can get. This module takes a convolutional neural network and SEAL encryption parameters as inputs. The input of a convolutional neural network can have any number of layers or hyperparameters. Similarly, the encryption parameters can be any value regardless of the limitations set in the SEAL library. The outputs of our SageMath module are the means and standard deviations of the noise after each layer; and the probability of the maximum noise coefficient if it is within known random distributions like the normal distribution. That probability, however, is already known to be incorrect because the noise distribution is not a normal distribution as seen in the Monte Carlo simulation in 6.5.5 above. Moreover, the SEAL encoding

[8] https://www.sagemath.org/

mechanism maps into positive integers only. This method of validation needs to have a concrete probability density function that is beyond the scope of this study. The final noise in a ciphertext shows to be a mixed probability distribution.

For a second and more direct type of validation, the SEAL library code was modified, and a method was created to return the complete noise polynomial of a ciphertext. The noise polynomials have $n + 1$ coefficients as per the set polynomial modulus n of the ciphertext.

For direct validating using the extracted noise polynomials, the best candidate was the CryptoNets inference network. Therefore, the CryptoNets was run repeatedly for a large number of times and the noise polynomials of the final outputs were extracted. The execution of CryptoNets was done carefully and the exact same set of inputs were given to produce the same outputs. The network outputs 10 ciphertexts representing the 10 MNIST classes. For each of the 10 outputs, there were 8193 noise coefficients for the set encryption parameter $n = 8192$. All of these coefficients for each of the output were averaged over the repeated executions. The findings are as follows.

6.10 Findings

Our findings, from analysis of the extracted noise from the encrypted outputs of CryptoNets, are shown in Figure 6-6, Figure 6-7, and Figure 6-8. The noise coefficients are very big numbers; therefore, they were min-max normalized to lie between zero and one. In Figure 6-6, we see the histograms of the noise polynomials from all of the outputs of CryptoNets. All of the outputs show a mixed distribution, which is certainly not a normal distribution. These histograms are also shown in a 3-dimensional chart in Figure 6-8 for a better view. For these outputs, a probability density diagram is shown in Figure 6-7. The conclusion from these diagrams is that

the final noise distribution in a SEAL ciphertext is a mixed distribution. On a side note, the value of 0 has a very high frequency count. Recall that this was evident in the histograms of our Monte Carlo simulation shown in Figure 6-1 to Figure 6-5 as well.

The Algorithm 1 on page 171 gives a new approach of approximating and optimising the SEAL encryption parameters for a convolutional neural network. Although, the current investigation was unable to find an exact probability value for the step 3 of the algorithm, this was only because the final probability distribution has been found as a mixture distribution for which the statistical probability density and cumulative density functions are not yet known.

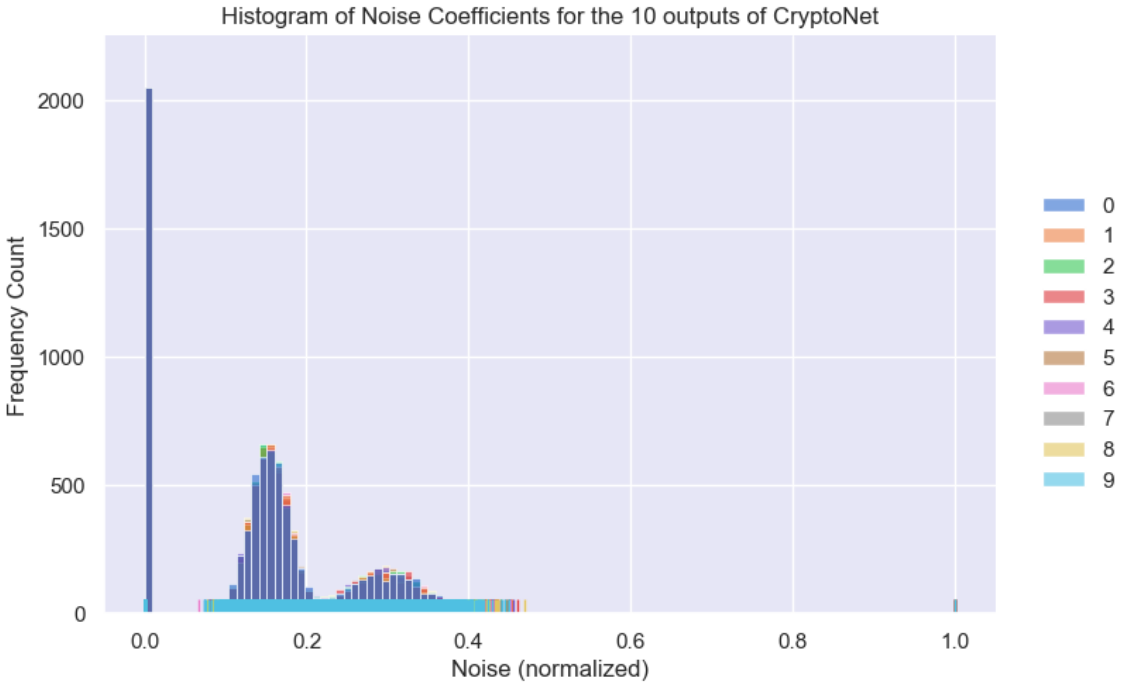


Figure 6-6 Histograms of noise coefficients for the 10 outputs of CryptoNets

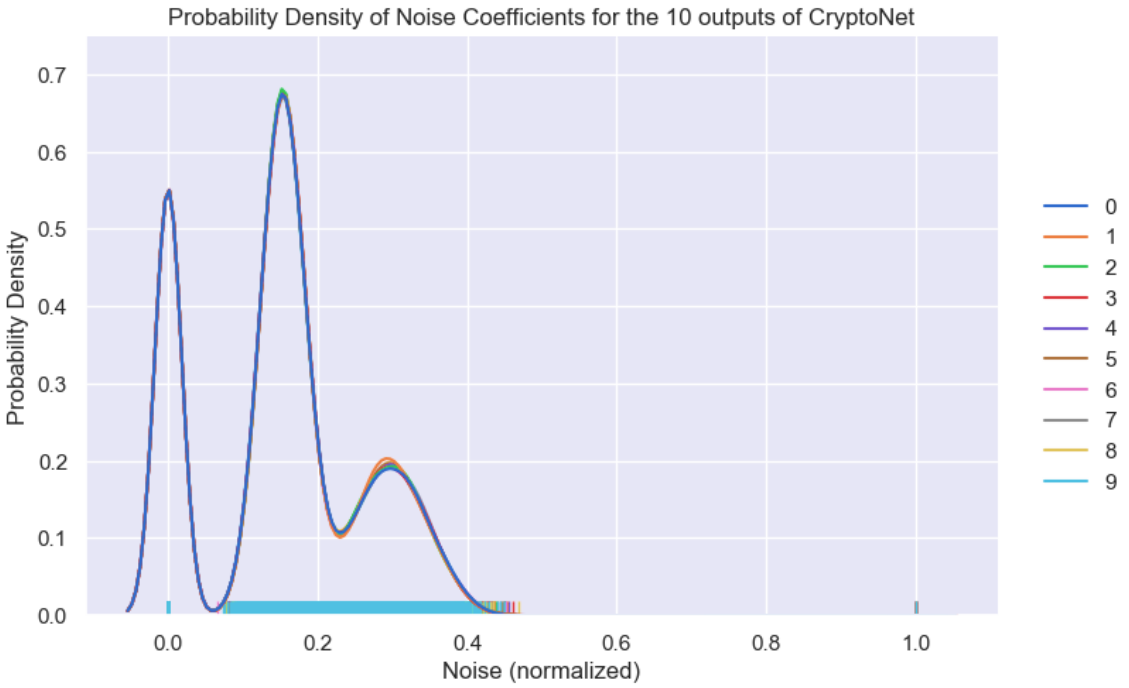


Figure 6-7 Probability density of noise coefficients for the 10 outputs of CryptoNets

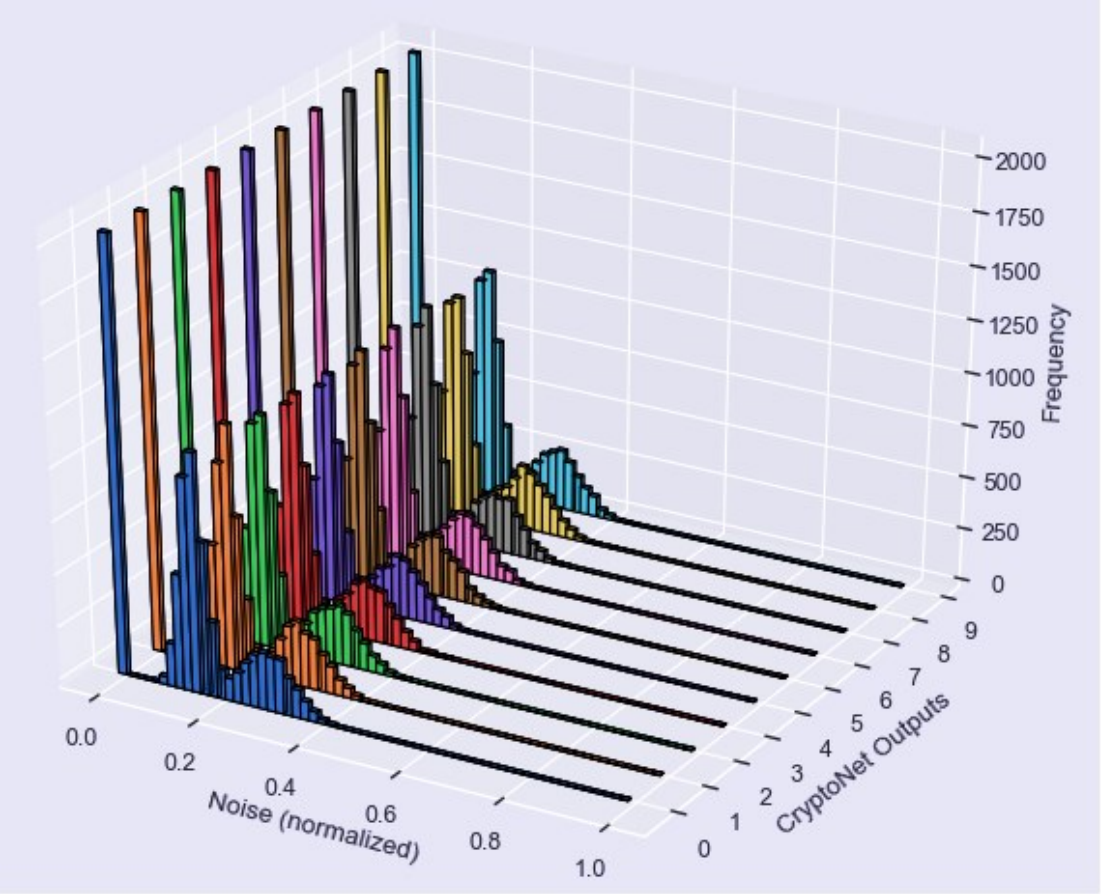


Figure 6-8 3-D Histograms of noise coefficients for the 10 outputs of CryptoNets

6.11 Summary

Despite its exploratory nature, this study offers several noteworthy contributions to the existing knowledge of encrypted operations in general, and encrypted convolutional neural networks in particular.

In this chapter, we have seen a novel view of the noise and its growth in a ciphertext. The knowledge of the noise as a random variable in a ciphertext holds significant importance for making practical improvements for a convolutional neural network especially, and other applications in general.

7 Conclusion

Performing computation on encrypted data is termed as the holy grail of cryptography. In today's service-oriented nature of the internet, we have very mature neural networks that can help us to understand our data by arranging it to known classes. This study has sought the integration of these two technologies.

This dissertation discusses the current issues and gaps for privacy-preserving classification networks; and suggests possible solutions in terms of implementation and practical adoption of homomorphic encryption.

This thesis begins by setting the research goals and setting the scope. This is followed by the introduction and of the two technologies, i.e., convolutional neural network and homomorphic encryption. In this regard, the latest code libraries of homomorphic encryption are reviewed and SEAL [22] is selected for practical adoption and implementation. This is followed by reviewing related works and their shortcomings to position this research. Subsequently, the technical challenges in adopting state-of-the-art homomorphic encryption schemes within convolutional neural network are expounded.

Following the background knowledge, a detailed design and the architecture of our proposed dual-cloud system is presented. The main motivation for developing the system was to overcome all the technical challenges highlighted earlier while performing privacy-preserving classification. Our dual-cloud system uses a partially homomorphic encryption scheme over two cloud systems. In our dual-cloud system, there is a trade-off in performance due to the networked entities involved.

Next, we show the usage of fully homomorphic encryption in a convolutional neural network with a focus on the encryption parameter selection. We show how one can select very fine-tuned encryption parameters for any feed-forward convolutional neural network. This results in a comprehensive method for automating the selection of encryption parameters of a convolutional neural network. Based on this, we then show the effects of varying the parameters of convolutional neural network and homomorphic encryption on each other.

Following this, insights and practical optimisation techniques of using homomorphic encryption are discussed using the case studies of well-known convolutional neural networks.

Afterwards, a novel method of optimising the encryption parameters is proposed using a probabilistic approach. Our probabilistic method of selecting and optimising encryption parameters can predict the exact decryption probability for a ciphertext. This holds significant value for both the designers of a convolutional neural network as well as homomorphic encryption schemes. Our probabilistic method is optimised for a convolutional neural network; however, it is generic enough to be adopted in other application areas as well. A comprehensive mathematical model and computer algorithm is developed to find out the probability of a successful decryption. During the research work for the probabilistic method, the SEAL library was modified to return the actual noise polynomials from a ciphertext. This was used to check the validity of our mathematical model with that of the practical implementation and it held extremely accurate. The final random distribution of the noise polynomial in a ciphertext is found to be a mixed distribution. Unhappily, this finding makes it beyond the scope of this study to find the density function for the particular distribution.

7.1 Major works in our research

In our research, we aimed to improve the state of the privacy in online computations. For this purpose, we have presented several works that all contribute towards this goal. Here, we will try to summarise all our presented works from a broader perspective.

Primarily, there are three interconnected works of significance. The first of these works (presented in Chapter 3) is about a novel implementation of a partially homomorphic encryption scheme to evaluate a neural network. As a self-criticism, we saw that the partially homomorphic encryption scheme supports only a single direct arithmetic operation i.e., addition of ciphertexts. Moreover, this system requires non-colluding dual cloud architecture. Although this system is very effective in privacy-preserving classification, we wanted to improve it further and explore in the direction of using a fully homomorphic scheme as a single cloud solution instead of dual-cloud.

Consequently, this exploration led us to produce our second major work as presented in Chapter 4 and 5. In the second major work, we overcame our requirement of a dual-cloud architecture and presented the details of using a single cloud architecture. Moreover, this work uses a fully homomorphic encryption scheme. For this purpose, we presented the novel method of selecting precise encryption parameters for the inference of any convolutional neural network.

Although our second major work emphasised the use of precise encryption parameters using deterministic mathematical formulas, from our experiments we found that there is still some room for improvement. The method of selecting encryption parameters deterministically were always correct but were bigger than required and not optimal in practical cases. Therefore, in our third major work (as

presented in chapter 6), we explored a novel approach of optimising the encryption parameters for the most probable cases used in practice. We name this approach as the probabilistic optimization of encryption parameters.

All of the above three major works are connected with the aim to perform optimised privacy-preserving classifications.

7.2 Social Impact of our work

From the start, this research has aimed to achieve a more practical and privacy-preserving setup in which neural networks are used and deployed across services. These services can be untrusted third-party organisations that can be equipped with our proposed techniques to provide classification results on our private data. We believe that we have achieved this by presenting implementation and optimisations techniques for practical cases.

In section 2.1, we presented some of the obvious areas where we require privacy-preserving classification technologies. For example, consider the image classification system we mentioned for forensic purposes. In such a setup we have a central database of criminals across the globe, and we try to match new images with the ones already in the database to find similarity of faces and objects. Here, not every country will have legislations to upload the images of their citizens to an online personnel identification service. Therefore, in order not to share the new images in a form that is understandable to third-party services, we can use an encrypt classification system of images. This way, our input image will be encrypted and not understandable to the third-party service.

We can extend the same example to a medical classification service where the input images are private medical imagery of patients. An encrypted medical imagery classification service will not only retain the privacy of the patients, but also

continue to improve their system for accurate classifications by using other plaintext datasets, and state-of-the-art machine learning scientific advancements. This way, the hospitals will be receiving scientifically accurate and up to date classification results within a privacy-preserving manner.

7.3 Future Research Directions

The current study has opened several pathways for future research. Some of them are summarized below.

The proposed dual-cloud model can be converted to a fully homomorphic scheme instead of a partially homomorphic scheme. The benefit could be that a wider variety of operations required by applications will be supported. However, the downside can be the loss of performance due to a computationally heavy encryption scheme. Nonetheless, as mentioned in the related works section 2.5, researchers have been proposing faster homomorphic encryption schemes that support a wide range of numerical operations that can be explored.

Presently, part of our research has focused on the use of the FV homomorphic encryption scheme as implemented in SEAL. Parameter estimation and implementation of other types of homomorphic encryption schemes like the CKKS [23] can be explored.

In our current research, we have been working only on feed-forward neural networks. The most prominent example is the convolutional neural networks. However, there are several other architectures of neural networks, like the recurrent neural networks, can be explored for evaluation using homomorphic encryption.

In one of our works in this research, we focused on the statistical approximation of encryption parameters. This involves the estimation of the random noise polynomial

within a given ciphertext after passing it through certain homomorphic operations. However, during our work, we faced hurdles in finding the exact statistical functions for probability and cumulative densities of the random noise polynomial. We used the Monte Carlo simulations to fulfil our needs. However, these functions can be explored for not only the optimization of encryption parameters, but for evaluating the security characteristics of an encryption scheme.

References and Bibliography

[1] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "CryptoNets: Applying neural networks to Encrypted data with high throughput and accuracy - Microsoft research," in *International Conference on Machine Learning*, 2016, pp. 201–210, [Online]. Available: http://research.microsoft.com/apps/pubs/?id=260989.

[2] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "Gazelle: A Low Latency Framework for Secure Neural Network Inference," in *Proceedings of the 27th USENIX Conference on Security Symposium*, Aug. 2018, pp. 1651–1668, doi: 10.5555/3277203.3277326.

[3] E. Hesamifard, H. Takabi, and M. Ghasemi, "CryptoDL: Deep Neural Networks over Encrypted Data," *arXiv Prepr.*, pp. 1–21, Nov. 2017, Accessed: Sep. 19, 2020. [Online]. Available: http://arxiv.org/abs/1711.05189.

[4] F. Armknecht *et al.*, "A Guide to Fully Homomorphic Encryption," *IACR Cryptol. ePrint Arch.*, 2015, doi: 1192.

[5] L. Stevens, "Big Read: What does Google DeepMind want with the NHS?," *Digital Health*, 2017. https://www.digitalhealth.net/2017/03/deepmind-mustafa-suleyman-interview/ (accessed Jun. 24, 2017).

[6] Y. C. Chang and C. J. Lu, "Oblivious polynomial evaluation and oblivious neural learning," *Theor. Comput. Sci.*, vol. 341, pp. 39–54, Sep. 2005, doi: 10.1016/j.tcs.2005.03.049.

[7] Y. LeCun, C. Cortes, and C. J. C. Burges, "The MNIST database of

handwritten digits," 1998. http://yann.lecun.com/exdb/mnist/.

[8] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On Data Banks and Privacy Homomorphisms," *Found. Secur. Comput.*, vol. 4, no. 11, pp. 169–180, 1978, [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.500.3989&rep=rep1&type=pdf.

[9] T. Elgamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Trans. Inf. Theory*, vol. 31, no. 4, pp. 469–472, 1985, doi: 10.1109/TIT.1985.1057074.

[10] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1999, vol. 1592, pp. 223–238, doi: 10.1007/3-540-48910-X_16.

[11] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978, doi: 10.1145/359340.359342.

[12] K. Peng, C. Boyd, and E. Dawson, "A Multiplicative Homomorphic Sealed-Bid Auction Based on Goldwasser-Micali Encryption," in *Information Security: 8th International Conference, ISC 2005, Singapore, September 20-23, 2005. Proceedings*, J. Zhou, J. Lopez, R. H. Deng, and F. Bao, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 374–388.

[13] D. Boneh, E. Goh, and K. Nissim, "Evaluating 2-DNF formulas on ciphertexts," *Theory Cryptogr.*, pp. 325–341, 2005, doi: 10.1007/978-3-540-30576-7_18.

[14] M. Baryalai, J. Jang-Jaccard, and D. Liu, "Towards privacy-preserving

classification in neural networks," in *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, Dec. 2016, pp. 392–399, doi: 10.1109/PST.2016.7906962.

[15] I. Damgård and M. Jurik, "A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System," in *PKC 2001: Public Key Cryptography*, no. December, Springer, 2001, pp. 119–136.

[16] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully Homomorphic Encryption without Bootstrapping," *IACR Cryptol. ePrint Arch.*, 2011, [Online]. Available: http://eprint.iacr.org/2011/277.

[17] F. Armknecht *et al.*, "A Guide to Fully Homomorphic Encryption," *IACR Cryptol. ePrint Arch.*, 2015.

[18] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?," *Proc. 3rd ACM Work. Cloud Comput. Secur. Work. - CCSW '11*, pp. 113–124, 2011, doi: 10.1145/2046660.2046682.

[19] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," in *STOC'09 Proceedings of the forty-first annual ACM symposium on theory of computing*, 2009, pp. 169–178.

[20] C. Gentry, "A Fully Homomorphic Encryption Scheme," PhD Thesis, Stanford University, 2009.

[21] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," *Proc. 15th Int. Conf. Pract. Theory Public Key Cryptogr.*, pp. 1–16, 2012, [Online]. Available: https://eprint.iacr.org/2012/144.

[22] K. Laine, *Microsoft SEAL (release 2.3.1)*. Microsoft Research, 2018.

[23] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for

Arithmetic of Approximate Numbers," *Adv. Cryptol. – ASIACRYPT 2017*, pp. 409–437, 2017, doi: 10.1007/978-3-319-70694-8_15.

[24] S. Halevi, "HElib - an implementation of homomorphic encryption." http://shaih.github.io/HElib/.

[25] K. Laine, H. Chen, and R. Player, *Microsoft SEAL (release 2.1)*. Microsoft Research, 2016.

[26] N. P. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," *Cryptol. ePrint Arch.*, 2011, [Online]. Available: http://eprint.iacr.org/2011/133.

[27] S. Halevi and V. Shoup, "Bootstrapping for HElib," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9056, 2015, pp. 641–670.

[28] W. Dai and B. Sunar, "cuHE: A homomorphic encryption accelerator library," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016, vol. 9540, pp. 169–186, doi: 10.1007/978-3-319-29172-7_11.

[29] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," *Proc. 44th Symp. Theory Comput. - STOC '12*, pp. 1219–1234, 2012, doi: 10.1145/2213977.2214086.

[30] Y. Doröz, Y. Hu, and B. Sunar, "Homomorphic AES Evaluation using NTRU.," *Cryptol. ePrint Arch.*, pp. 1–16, 2014, [Online]. Available: https://eprint.iacr.org/2014/039.pdf.

[31] L. Ducas and D. Micciancio, "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second," in *Lecture Notes in Computer Science*

(including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 9056, Springer, 2015, pp. 617–640.

[32] C. Dwork, "Differential Privacy," in *Automata, Languages and Programming: 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*, M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–12.

[33] S. U. Bazai, J. Jang-Jaccard, and X. Zhang, "A Privacy Preserving Platform for MapReduce," in *Communications in Computer and Information Science*, vol. 490, Springer, 2017, pp. 88–99.

[34] T. van Elsloo, G. Patrini, and H. Ivey-Law, "SEALion: a Framework for Neural Network Inference on Encrypted Data," *arXiv Prepr.*, 2019, [Online]. Available: http://arxiv.org/abs/1904.12840.

[35] R. Dathathri *et al.*, "Chet: An optimizing compiler for fully-homomorphic neural-network inferencing," *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implement.*, pp. 142–156, Jun. 2019, doi: 10.1145/3314221.3314628.

[36] J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig, "Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8308 LNCS, Springer, 2013, pp. 45–64.

[37] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "NGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data," *ACM Int. Conf. Comput. Front. 2019, CF 2019 - Proc.*, pp. 3–15, 2019, doi: 10.1145/3310273.3323047.

[38] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "CrypTFlow: Secure TensorFlow Inference," in *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020, pp. 336–353, doi: 10.1109/SP40000.2020.00092.

[39] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, "nGraph- HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data," *Cryptol. ePrint Arch. Rep. 2019/947*, 2019, [Online]. Available: https://eprint.iacr.org/2019/947.

[40] C. Hazay, A. Marcedone, Y. Ishai, and M. Venkitasubramaniam, "Leviosa: Lightweight secure arithmetic computation," *Proc. ACM Conf. Comput. Commun. Secur.*, pp. 327–344, 2019, doi: 10.1145/3319535.3354258.

[41] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. Ada, "DELPHI : A Cryptographic Inference Service for Neural Networks," *29th USENIX Secur. Symp. USENIX Secur. 20)*, 2020.

[42] R. Bost, R. Popa, S. Tu, and S. Goldwasser, "Machine Learning Classification over Encrypted Data," in *Network and Distributed System Security Symposium*, 2015, pp. 1–31, doi: 10.14722/ndss.2015.23241.

[43] C. Orlandi, A. Piva, and M. Barni, "Oblivious neural network computing via homomorphic encryption," *Eurasip J. Inf. Secur.*, vol. 2007, 2007, doi: 10.1155/2007/37343.

[44] F. Y. Rao, B. K. Samanthula, E. Bertino, X. Yi, and D. Liu, "Privacy-Preserving and Outsourced Multi-user K-Means Clustering," in *2015 IEEE Conference on Collaboration and Internet Computing (CIC)*, 2015, pp. 80–89, doi: 10.1109/CIC.2015.20.

[45] B. K. Samanthula, W. Jiang, and E. Bertino, "Privacy-preserving complex

query evaluation over semantically secure encrypted data," in *European Symposium on Research in Computer Security*, 2014, pp. 400–418.

[46] W. Diffie and M. E. Hellman, "New Directions in Cryptography," *IEEE Trans. Inf. Theory*, vol. 22, no. 6, pp. 644–654, 1976, doi: 10.1109/TIT.1976.1055638.

[47] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2323, 1998, doi: 10.1109/5.726791.

[48] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Manual for Using Homomorphic Encryption for Bioinformatics," in *Proceedings of the IEEE*, 2017, vol. 105, no. 3, pp. 552–567, doi: 10.1109/JPROC.2016.2622218.

[49] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious Neural Network Predictions via MiniONN Transformations," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2017, no. 2017/452, pp. 619–631, doi: 10.1145/3133956.3134056.

[50] P. Mohassel and Y. Zhang, "SecureML: A System for Scalable Privacy-Preserving Machine Learning," *Proc. - IEEE Symp. Secur. Priv.*, pp. 19–38, 2017, doi: 10.1109/SP.2017.12.

[51] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "DeepSecure: Scalable Provably-Secure Deep Learning," *arXiv Prepr.*, 2017, [Online]. Available: http://arxiv.org/abs/1705.08963.

[52] T. Lepoint and M. Naehrig, "A Comparison of the Homomorphic Encryption Schemes FV and YASHE," in *Lecture Notes in Computer Science (including*

subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 8469 LNCS, Springer, 2014, pp. 318–335.

[53] J. van de Pol and N. P. Smart, "Estimating Key Sizes for High Dimensional Lattice-Based Systems," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8308 LNCS, Springer, 2013, pp. 290–303.

[54] R. Lindner and C. Peikert, "Better Key Sizes (and Attacks) for LWE-Based Encryption," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6558 LNCS, Springer, 2011, pp. 319–339.

[55] M. S. Riazi and F. Koushanfar, "Privacy-preserving deep learning and inference," in *Proceedings of the International Conference on Computer-Aided Design*, Nov. 2018, pp. 1–4, doi: 10.1145/3240765.3274560.

[56] C. Moore, M. O'Neill, E. O'Sullivan, Y. Doroz, and B. Sunar, "Practical homomorphic encryption: A survey," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, Jun. 2014, pp. 2792–2795, doi: 10.1109/ISCAS.2014.6865753.

[57] R. Player, "Parameter selection in lattice-based cryptography," PhD Thesis, University of London, 2017.

[58] M. Chase *et al.*, "Security of Homomorphic Encryption," Jul. 2017. https://homomorphicencryption.org/introduction/ (accessed Dec. 20, 2018).

[59] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of Learning with Errors," *J. Math. Cryptol.*, vol. 9, no. 3, pp. 169–203, 2015, doi: 10.1515/jmc-2015-0016.

[60] A. Canziani, A. Paszke, and E. Culurciello, "An Analysis of Deep Neural

Network Models for Practical Applications," *arXiv Prepr.*, 2017, [Online]. Available: http://arxiv.org/abs/1605.07678.

[61] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Adv. Neural Inf. Process. Syst.*, pp. 1097–1105, 2012.

[62] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv Prepr. arXiv1409.1556*, Sep. 2015, [Online]. Available: http://arxiv.org/abs/1409.1556.

[63] C. Szegedy *et al.*, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2015, pp. 1–9, doi: 10.1109/CVPR.2015.7298594.

[64] S. Murphy and R. Player, "A Central Limit Framework for Ring-LWE Decryption," *IACR Cryptol. ePrint Arch. Rep. 2019/452*, pp. 1–24.

[65] V. Lyubashevsky, C. Peikert, and O. Regev, "A Toolkit for Ring-LWE Cryptography," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7881 LNCS, Springer, 2013, pp. 35–54.

[66] S. Murphy, R. Player, and R. Holloway, "Noise Distributions in Homomorphic Ring-LWE," *IACR Cryptol. ePrint Arch. Rep. 2017/698*, pp. 1–56, 2017, [Online]. Available: https://eprint.iacr.org/2017/698.pdf.

[67] J. Bos *et al.*, "CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, Apr. 2018, pp. 353–367, doi: 10.1109/EuroSP.2018.00032.

[68] M. Tsagris, C. Beneki, and H. Hassani, "On the folded normal distribution,"

Mathematics, vol. 2, no. 1, pp. 12–28, 2014, doi: 10.3390/math2010012.

[69] S. Hardy, "A Homomorphic Encryption Illustrated Primer," *N1 Analytics*, 2018. https://blog.n1analytics.com/homomorphic-encryption-illustrated-primer/ (accessed Oct. 19, 2019).

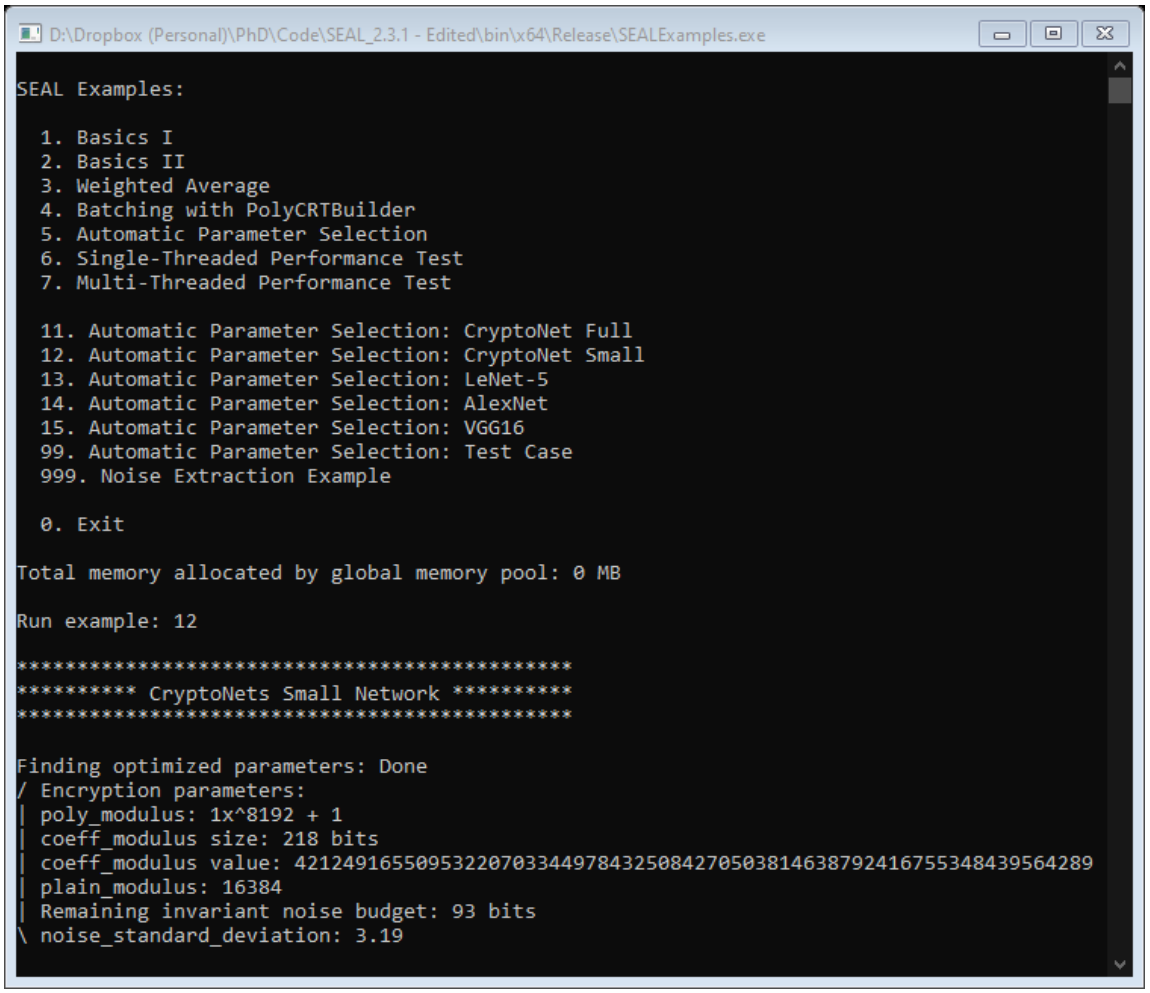
Appendix A. Noise estimates for operations in the SEAL Library

The following table reproduces the noise estimates for operations in the SEAL library as given in their documentation [22].

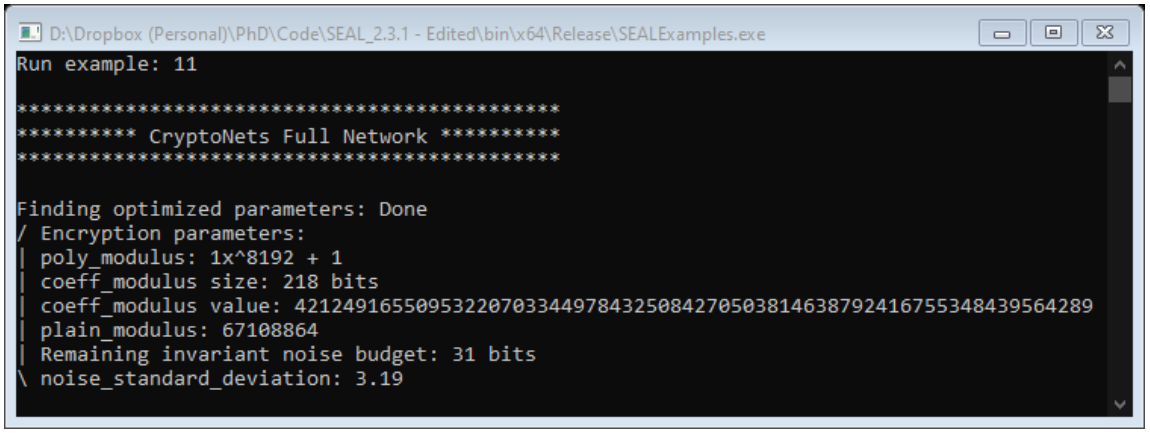
Operation	Input description	Noise bound of output
`Encrypt`	Plaintext m	$\frac{r_t(q)}{q}\|m\|N_m + \frac{7nt}{q}\min\{B, 6\sigma\}$
`Negate`	Ciphertext `ct`	ν
`Add/Sub`	Ciphertexts $\texttt{ct}_1$ and $\texttt{ct}_2$	$\nu_1 + \nu_2$
`AddPlain/SubPlain`	Ciphertext `ct` and plaintext m	$\nu + \frac{r_t(q)}{q}N_m\|m\|$
`MultiplyPlain`	Ciphertext `ct` and plaintext m	$N_m\|m\|\nu$
`Multiply`	Ciphertexts $\texttt{ct}_1$ and $\texttt{ct}_2$ of sizes j_1+1 and j_2+1	$t\sqrt{3n}\left[(12n)^{j_1/2}\nu_2 + (12n)^{j_2/2}\nu_1 + (12n)^{(j_1+j_2)/2}\right]$
`Square`	Ciphertext `ct` of size j	Same as `Multiply(ct, ct)`
`Relinearize`	Ciphertext `ct` of size K and target size L, such that $2 \leq L < K$	$\nu + \frac{2t}{q}\min\{B, 6\sigma\}(K-L)n(\ell+1)w$
`AddMany`	Ciphertexts $\texttt{ct}_1, \ldots, \texttt{ct}_k$	$\sum_i \nu_i$
`MultiplyMany`	Ciphertexts $\texttt{ct}_1, \ldots, \texttt{ct}_k$	Apply `Multiply` in a tree-like manner, and `Relinearize` down to size 2 after every multiplication
`Exponentiate`	Ciphertext `ct` and exponent k	Apply `MultiplyMany` to k copies of `ct`

Appendix B. Screenshots of our parameter estimation system

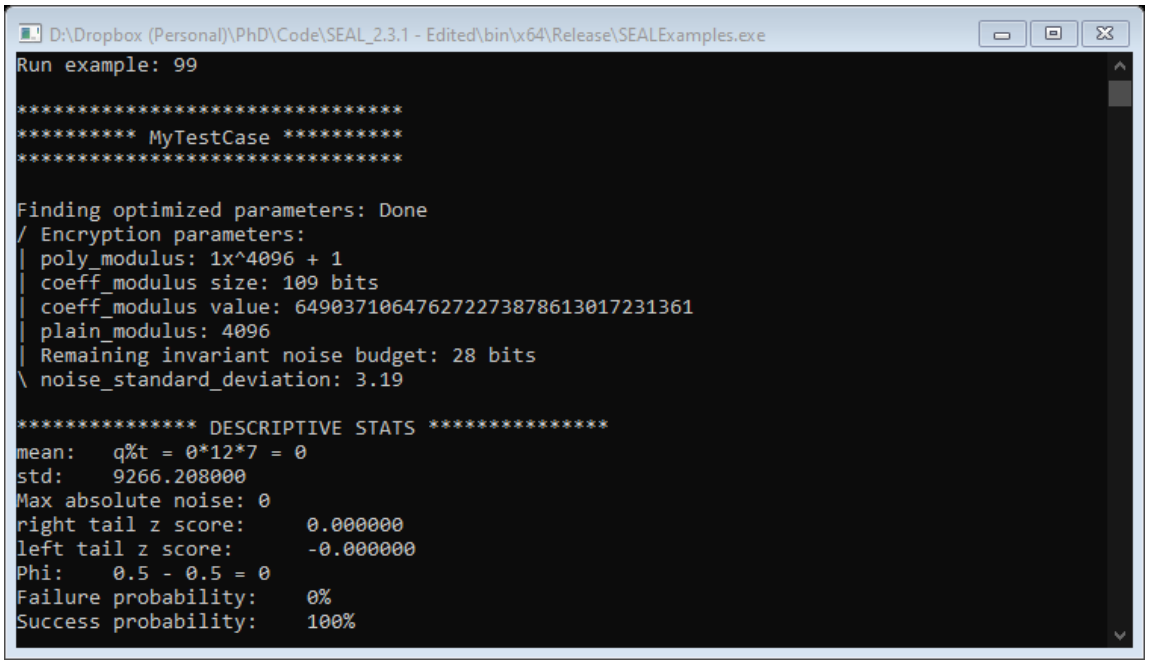
Here we share some screenshots of our parameter estimation system. For implementation and testing, we coded our derived formulas in the SEAL library. The exact details of the changes in the library are given in Appendix C. The changes done in the SEAL library equipped us to specify any convolutional neural network and get an estimation of the encryption parameters for it. This includes the specification of the exact number of inputs, and all the layers and their types. In the first screenshot, we see the parameter estimation performed for CryptoNets reduced (small) network.



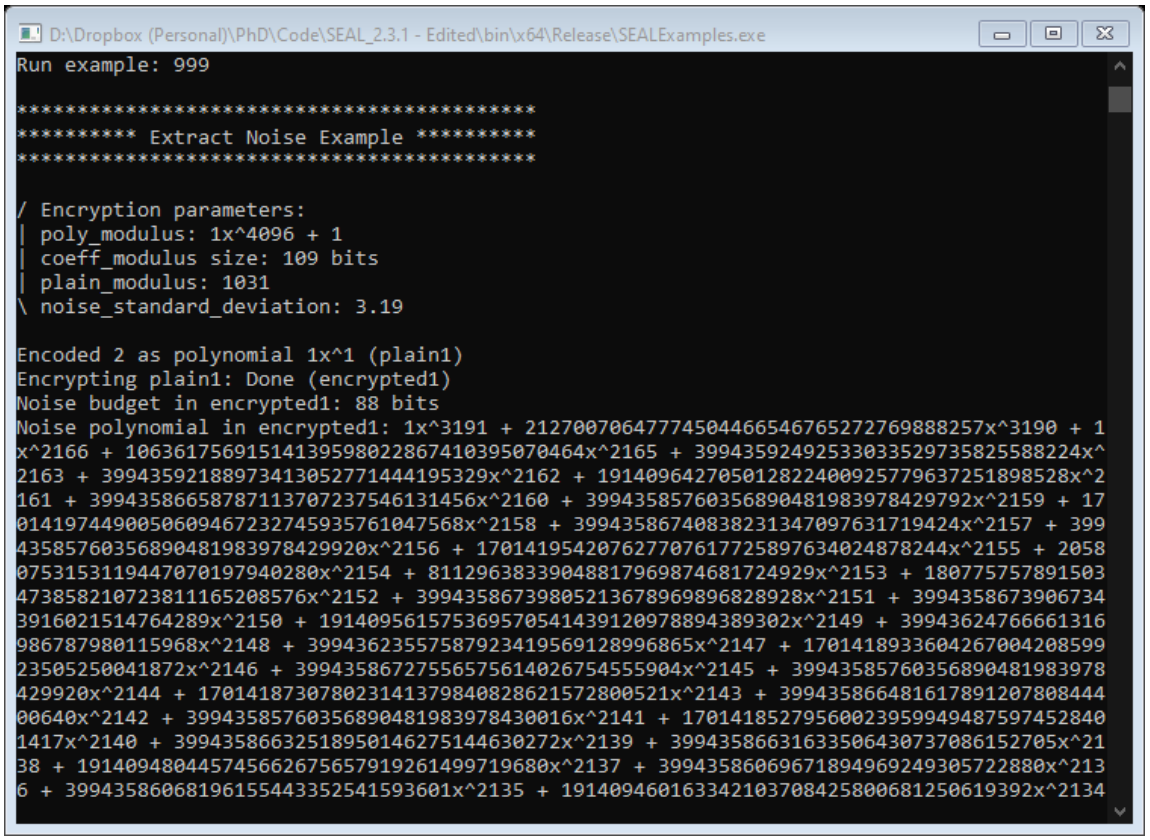
Similarly, for the full network, the estimation tool gave the following results.



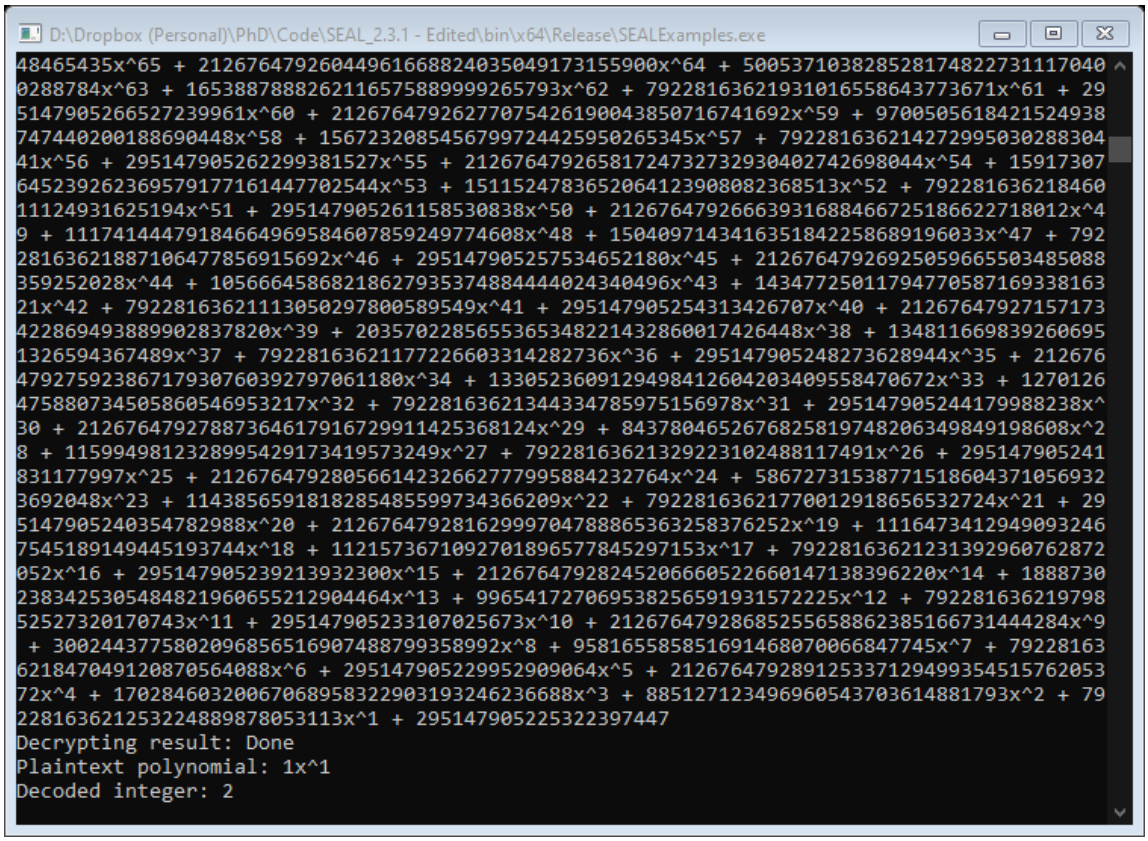
In our probabilistic estimation, we coded the probability calculation into SEAL which is seen in the following screenshot for testing parameters on the CryptoNets and checking the probability.



In a different test, the extraction of the exact noise polynomial from a ciphertext is performed and is shown in the following screenshot. Please note that since the noise polynomial has the length of 4096 coefficients, only the beginning and the end of the output is shown here.



The end of the output of noise polynomial from a single ciphertext.



Appendix C. Original Function Contributions

The following original functions were added into the SEAL library version 2.3.1. Here we enlist the function signatures within the respective code files and a brief description.

1. **/SEAL/seal/chooser.h**

In this file, five functions were declared for approximating encryption parameters of a single layer according to the specified parameters.

```
class ChooserEvaluator
{
public:
   ChooserPoly weighted_layer(
      const ChooserPoly & operand,
      int plain_max_coeff_count,
      std::uint64_t plain_max_abs_value,
      int inputs_to_a_neuron);
   ChooserPoly weighted_layer(
      const ChooserPoly &operand,
      const ChooserPoly &plain_chooser_poly,
      int inputs_to_a_neuron);
   ChooserPoly sum_pooling_layer(
      const ChooserPoly &operand,
      int inputs);
   ChooserPoly average_pooling_layer(
      const ChooserPoly& operand,
      int plain_max_coeff_count,
      std::uint64_t plain_max_abs_value,
      int inputs_to_a_neuron);
   ChooserPoly average_pooling_layer(
      const ChooserPoly& operand,
      const ChooserPoly& plain_chooser_poly,
      int inputs_to_a_neuron);
}
```

2. /SEAL/seal/chooser.cpp

In this file, the five functions declared in the above header file are implemented. These functions call the actual simulation functions specified in the `simulation.cpp` files specified next.

```
namespace seal
{
    ChooserPoly ChooserEvaluator::weighted_layer(
        const ChooserPoly & operand,
        int plain_max_coeff_count,
        std::uint64_t plain_max_abs_value,
        const int inputs_to_a_neuron) {...}

    ChooserPoly ChooserEvaluator::weighted_layer(
        const ChooserPoly & operand,
        const ChooserPoly & plain_chooser_poly,
        const int inputs_to_a_neuron) {...}

    ChooserPoly ChooserEvaluator::sum_pooling_layer(
        const ChooserPoly & operand,
        int inputs) {...}

    ChooserPoly ChooserEvaluator::average_pooling_layer(
        const ChooserPoly & operand,
        int plain_max_coeff_count,
        std::uint64_t plain_max_abs_value,
        const int inputs_to_a_neuron) {...}

    ChooserPoly ChooserEvaluator::average_pooling_layer(
        const ChooserPoly & operand,
        const ChooserPoly & plain_chooser_poly,
        const int inputs_to_a_neuron) {...}
}
```

3. /SEAL/seal/simulator.h

Three declarations are added into this file for functions that approximate encryption parameters for a specific kind of neural network layer. The weighted layer function is used to approximate for both the fully connected layer as well as the convolutional layer.

```
class SimulationEvaluator
    {
    public:
      Simulation weighted_layer(
         const Simulation &simulation,
         int plain_max_coeff_count,
         std::uint64_t plain_max_abs_value,
         int inputs_to_a_neuron);
      Simulation sum_pooling_layer(
         const Simulation & simulation,
         int number_of_inputs);
      Simulation average_pooling_layer(
         const Simulation& simulation,
         int plain_max_coeff_count,
         uint64_t plain_max_abs_value,
         int number_of_inputs);
    }
```

4. /SEAL/seal/simulator.cpp

In this file, the actual code of the three functions is implemented that were declared in the header file above. These three functions has the actual approximation codes for any particular neural network layer with some specific parameters.

```
namespace seal
{
   Simulation SimulationEvaluator::weighted_layer(
      const Simulation & simulation,
      int plain_max_coeff_count,
      std::uint64_t plain_max_abs_value,
      int inputs_to_a_neuron) {...}
   Simulation SimulationEvaluator::sum_pooling_layer(
      const Simulation &simulation,
      int number_of_inputs) {...}
   Simulation SimulationEvaluator::average_pooling_layer(
      const Simulation& simulation,
      int plain_max_coeff_count,
      uint64_t plain_max_abs_value,
      int number_of_inputs) {...}
}
```

5. **/SEAL/seal/util/computation.h**

Three classes were added in this file to represent the three different functions implemented for the estimation of encryption parameters of a neural network. These classes are used to hold the complete computation history for a neural network. In other words, all the computations performed prior to the current layer.

```
class WeightedLayerComputation : public Computation
{...};

class SumPoolingLayer : public Computation
{...};

class AveragePoolingLayer : public Computation
{...};
```

6. **/SEAL/seal/util/computation.cpp**

In this file, the constructors, destructors, and functions of the above declared three classes in the header file `computation.h` are implemented. The code is similar to the implementations of other similar classes in this file.

MASSEY UNIVERSITY
GRADUATE RESEARCH SCHOOL

STATEMENT OF CONTRIBUTION
DOCTORATE WITH PUBLICATIONS/MANUSCRIPTS

We, the candidate and the candidate's Primary Supervisor, certify that all co-authors have consented to their work being included in the thesis and they have accepted the candidate's contribution as indicated below in the *Statement of Originality*.

Name of candidate:	Mehmood Baryalai
Name/title of Primary Supervisor:	Professor Dianne Brunton
Name of Research Output and full reference:	
M. Baryalai, J. Jang-Jaccard, and D. Liu, "Towards privacy-preserving classification in neural networks," in 2016 14th Annual Conference on Privacy, Security and Trust (PST), IEEE, 2016, pp. 392–399.	
In which Chapter is the Manuscript /Published work:	2,3
Please indicate:	
• The percentage of the manuscript/Published Work that was contributed by the candidate:	70
and	
• Describe the contribution that the candidate has made to the Manuscript/Published Work:	
conceptualization, methodology, validation, analysis, investigation, original draft preparation, visualization, and responding to reviewers' comments.	
For manuscripts intended for publication please indicate target journal:	
Candidate's Signature:	
Date:	29.09.2020
Primary Supervisor's Signature:	
Date:	

(This form should appear at the end of each thesis chapter/section/appendix submitted as a manuscript/ publication or collected as an appendix at the end of the thesis)