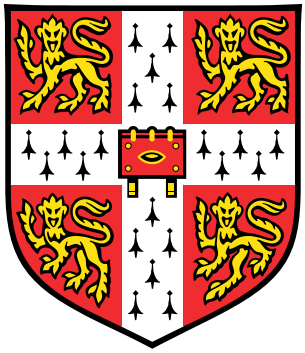
***Han Xuanyuan***

# Privacy-Preserving Deep Learning Inference using Homomorphic Encryption

Computer Science Tripos – Part II



Churchill College

May 13, 2021

# Declaration of originality

I, Han Xuanyuan of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed *Han Xuanyuan*

Date *13/05/2021*

# Acknowledgments

I would like to thank my supervisors, Dr Stephen Cummins and Francisco Vargas, for their support and advice throughout the project. I would also like to thank Charles Chen and Joe Mayford for proof-reading my dissertation and their helpful comments, as well as JC Tang for helping me understand ring theory. Finally, I would like to thank Dr John Fawcett for being my Director of Studies over the past three years and motivating me to be the best that I can.

# Proforma

**Candidate number**: 2398E
**Title**: Privacy-preserving deep learning inference using homomorphic encryption
**Examination and year**: Computer Science Tripos – Part II, 2021
**Word count**: 11954[1]
**Lines of code**: 5139[2]
**Project originator**: Dr Stephen Cummins and the author
**Project supervisors**: Dr Stephen Cummins, Mr Francisco Vargas Palomo

## Original aims of the project

The original aim was to create a library for building privacy-preserving neural networks and performing secure inference using homomorphic encryption, and to apply the library to the MNIST and CIFAR-10 datasets. Neural network operations would be implemented using a homomorphic encryption library, and the project would be evaluated in terms of the latency, accuracy and security of the implemented models via a comparison with works from the literature. Suggested extensions included application to a real-world problem, or a hardware-focused route of accelerating underlying operations using GPUs.

## Work completed

All base requirements and selected extensions were fully met. I devised a revised set of software-focused extensions, and implemented multiple techniques from the literature to achieve efficient dense and convolutional layer computation. I applied two different model compression techniques, demonstrating that both are highly effective in reducing model latency. The performance of my MNIST and CIFAR-10 models closely resemble and occasionally outperform those from the literature. I also constructed a novel encrypted facial recognition model using transfer learning, and implemented a Python version of the recent CKKS scheme with the aim of introducing homomorphic encryption to a wider audience.

## Special difficulties

This project and the dissertation writing were completed entirely within the COVID-19 pandemic. Deadlines, though met, were tighter than they would have usually been due to the disruption to progress caused by the enforcement of multiple national lockdowns.

[1] Computed using `texcount` with the command `texcount -merge -sum thesis.tex` to include captions, footnotes and equations. Instructions `%TC:group tabular 1 1` and `%TC:group table 0 1` are inserted into the LaTeX file to include table contents. Instructions `%TC:ignore` and `%TC:endignore` are inserted to only count the five main chapters.

[2] Computed using `cloc`. Includes Python code in Jupyter notebooks.

# Contents


**1 Introduction** **1**
- 1.1 Introduction . . . 1
- 1.2 Aims and contributions . . . 2
- 1.3 Related work . . . 2

**2 Preparation** **4**
- 2.1 Preliminaries . . . 4
  - 2.1.1 Threat model . . . 4
  - 2.1.2 Homomorphic encryption . . . 4
  - 2.1.3 Neural network inference . . . 7
- 2.2 Project strategy . . . 8
  - 2.2.1 Requirements analysis . . . 8
  - 2.2.2 Methodology . . . 9
- 2.3 Starting point . . . 10
  - 2.3.1 Knowledge and experience . . . 10
  - 2.3.2 Use of libraries and datasets . . . 11
  - 2.3.3 Pre-implementation decisions . . . 11
  - 2.3.4 Computer resources . . . 12

**3 Implementation** **13**
- 3.1 Architecture . . . 13
  - 3.1.1 High-level framework . . . 13
  - 3.1.2 Software interface . . . 14
- 3.2 Encryption primitives . . . 15
  - 3.2.1 Scheme overview . . . 15
- 3.3 Network layers . . . 18
  - 3.3.1 SIMD Packing . . . 18
  - 3.3.2 Efficient packing . . . 20
- 3.4 Model pruning . . . 23
  - 3.4.1 Lottery ticket pruning . . . 23
  - 3.4.2 Singular value decomposition . . . 24
- 3.5 Application to datasets . . . 24
  - 3.5.1 MNIST . . . 24
  - 3.5.2 CIFAR-10 . . . 26
  - 3.5.3 Encrypted facial recognition . . . 28

**4 Evaluation** **30**
- 4.1 Requirements analysis . . . 30
- 4.2 Comparison of performance with existing work . . . 31
  - 4.2.1 Latency and throughput . . . 31
  - 4.2.2 Accuracy . . . 33
  - 4.2.3 Message size . . . 34

4.3 Practicality . . . 34
4.3.1 Supported architectures . . . 34
4.3.2 Security and homomorphic depth . . . 36
4.3.3 Encrypted facial recognition . . . 36
4.4 Summary . . . 37

**5 Conclusions** **38**
5.1 Summary of project . . . 38
5.2 Lessons learnt . . . 38
5.3 Future directions . . . 38

**A Supplementary materials for cryptography** **42**
A.1 A brief detour into abstract algebra . . . 42
A.1.1 Ring learning with errors . . . 43
A.2 A more in-depth explanation of CKKS . . . 43
A.2.1 Encoding and decoding . . . 43
A.2.2 CKKS operations . . . 44
A.3 Encryption parameters . . . 45

**B Additional information** **46**
B.1 Example use of $\Psi$-lib . . . 46
B.2 Extracting the weight matrix of a convolution . . . 47
B.3 Experimental setup . . . 48
B.4 Full model architectures . . . 49
B.5 Additional results . . . 50
B.6 Tools and resources used to produce diagrams . . . 50
B.7 Changes from original proposal . . . 50

**C Proposal** **52**

# List of Figures


2.1 The assumed threat model. . . . . 5
2.2 Homomorphic encryption . . . . . 5
2.3 Diagram of a neural network. . . . . 7
2.4 Project timeline. . . . . 10

3.1 High-level overview of the implementation. . . . . 13
3.2 Repository overview. . . . . 15
3.3 The basic operations in CKKS. . . . . 17
3.4 The SIMD packing scheme introduced by CryptoNets. . . . . 18
3.5 Depiction of SIMD Network operations. . . . . 19
3.6 Ciphertext dot product. . . . . 19
3.7 A depiction of convolution packing. . . . . 21
3.8 Two approaches for encrypted matrix-vector multiplication. . . . . 22
3.9 Weight matrix rotation for different dimensions $m \times n$. . . . . 23
3.10 Applying iterative lottery pruning on the modified CryptoNets architecture. 25
3.11 The original CryptoNets architecture and our optimised design. . . . . 26
3.12 Obtaining the CF-*fast* architecture from CF-*base*. . . . . 27
3.13 Matrix-vector multiplication splitting. . . . . 27
3.14 The CryptoFace model. . . . . 28

4.1 Break-down of low latency model operations. . . . . 33
4.2 The ReLU activation and its approximation. . . . . 34
4.3 Comparison of convergence using the two activation functions. . . . . 35
4.4 The ROC curve of the CryptoFace model. . . . . 36

# List of Tables

4.1 Project requirements analysis. . . . . 30
4.2 Comparison of MNIST models with previous works. . . . . 31
4.3 Effect of compression techniques applied. . . . . 32

B.1 Experimental setup details. . . . . 48
B.2 The original MNIST architecture and our optimised design. . . . . 49
B.3 Comparison of CIFAR-10 architectures. . . . . 49
B.4 Comparison of CIFAR models with previous works. . . . . 50

# Chapter 1

# Introduction

## 1.1 Introduction

Ever since the inception of computing devices and algorithms, it has become possible to automatically solve problems that once required human intervention. These problems range from performing integer calculations to complicated tasks such as fully autonomous driving. The use of *machine learning* techniques has been quite substantial in allowing practitioners to solve problems such as image recognition, which are particularly challenging for traditional algorithms.

Deep learning is the theory and application of machine learning algorithms based on deep architectures – those consisting of artificial neural networks with multiple layers. An artificial neural network is a collection of neurons in a manner that loosely resembles neurons in the brain. Each neuron is connected in some way to other neurons – often with specific neurons designated as inputs and outputs of the network. Deep learning as a paradigm has proven to be very effective in solving problems such as image classification and speech recognition.

Over the last decade, deep neural networks have had a major effect in improving the accuracy of machine learning systems. They have been deployed in a range of sectors, powering a wide variety of applications such as recommendation systems, medical diagnosis, and content filtering. Machine Learning as a Service (MLaaS) is a framework in which cloud services apply machine learning algorithms on user-supplied data to produce an inference result which is then returned to the user [1]. Cloud systems are an attractive platform for deploying pretrained models due to the relatively low cost and high availability of computation. However, one privacy issue is that data has to be decrypted before inference, thereby potentially allowing a server-side adversary to have access to the user's (possibly sensitive) information.

Homomorphic encryption (HE) is the art of devising encryption schemes that enable computations to be performed on encrypted data, or *ciphertext*. For example, consider the calculation $3 \times 4$. Is it possible to first encrypt 3 and 4 separately, then compute their product whilst they are encrypted, before decrypting the result? Another more complicated example would be the problem of performing facial recognition on an encrypted image of a person's face. In recent years, many works have focused on the problem of applying HE to the inference phase of deep learning [2]. As this area is still relatively new, there are few publicly available libraries to aid the implementation of secure inference models.

## 1.2 Aims and contributions

This dissertation documents the design and implementation of a Python library for the construction and application of neural networks that support private inference, which I shall refer to as $\Psi$-lib[1] through the rest of this dissertation. In particular, I contribute the following:

- I implement algorithms for enabling private inference using the single instruction multiple data (SIMD) ciphertext packing method by Dowlin et al [3] and the CKKS encryption scheme [4] from Microsoft's Secure Encrypted Arithmetic Library (SEAL) [5]. As an extension I show that the *lottery ticket hypothesis* [6] can be used in conjunction with the SIMD packing approach to reduce inference latency.
- I package the implementation in a library format that can be used to build and evaluate arbitrary secure inference models. The structure of $\Psi$-lib is designed with a focus on separating the underlying cryptographic primitives from the interface for constructing models.
- As extensions, I implement low-latency packing techniques introduced by Lu et al [7] to support private inference using fewer operations than the SIMD packing scheme. Additionally, I combine these techniques with an encrypted matrix-vector product method proposed by Juvekar et al [8] to construct lower latency models without sacrificing accuracy.
- I demonstrate $\Psi$-lib by constructing both low-latency MNIST and CIFAR-10 models that achieve lower inference latency than works from the literature that do not utilise multiparty computation. As an extension, I also use $\Psi$-lib to solve the problem of encrypted facial recognition, achieving a low comparison latency of 4.8 ms, when comparing a single face against a gallery of 4096.
- Finally, to help demystify the workings of the CKKS scheme to a wider audience, I provide a toy implementation of the original CKKS scheme by Cheon et al [4].

## 1.3 Related work

The concept of securing machine learning inference using HE was introduced in 2012 by Graepel et al [9], who used a HE scheme to achieve secure inference using polynomial classifiers. In 2016, HE was introduced to the field of deep learning by Dowlin et al [3] who devised the CryptoNets model. They used the YASHE scheme and applied a model on the MNIST handwritten digits dataset, achieving 99% accuracy and a throughput of 59000 classifications an hour. Following CryptoNets, Badawi et al [10] achieved improvements in the direction of greater throughput. They implemented a GPU-accelerated scheme and were able to make predictions with the MNIST model at a higher throughput of 5.7 million classifications an hour. Another notable milestone was Faster CryptoNets [11], where the authors used model compression techniques to improve the throughput of the original CryptoNets construction without relying on hardware acceleration. Moreover, other works such as GAZELLE [8] and FALCON [12] have employed techniques from secure multiparty computation in combination with HE to achieve greater inference speedups.

[1] This stands for 'Private (and) Secure Inference library'.

As of today, HE is not yet practical for application to very deep models such as ResNet50 and DenseNet121. This is largely due to time-complexity limitations of currently available HE schemes and the lack of hardware acceleration for HE. Much of the recent work in the field has been focused on the hardware perspective – for example, Boemer et al [13] have added an HE extension to the Intel nGraph compiler, which compiles a deep learning model into a graph representation on which it can perform hardware-dependent optimisations. Riazi et al [14] introduced HEAX, a hardware architecture focused on accelerating the expensive number theoretic transform operations found in contemporary HE schemes.

# Chapter 2

# Preparation

This chapter discusses the work done in preparation for the project's implementation. Section 2.1 explains the preliminary details and background, Section 2.2 discusses the methodologies used throughout the implementation, and Section 2.3 gives an overview of the project's starting point.

## 2.1 Preliminaries

This section is dedicated to introducing readers to the underlying concepts as well as the terminologies and notations used in the implementation. The project stretches across the two domains of deep learning and cryptography, and therefore for brevity I focus on explaining the aspects that are the most challenging. Links to resources are provided wherever there is insufficient space for a thorough explanation.

### 2.1.1 Threat model

$\Psi$-lib is designed in consideration of the MLaaS framework, in which the user first sends data to a server, which then performs machine learning inference on the received data using a model that is typically pretrained. The inference result is then delivered back to the user. For example, consider an online machine learning service which claims to detect the probability of a person having COVID-19 from an audio recording of their cough. Suppose that Alice decides to send a recording of her cough to this service, in the hopes of receiving a diagnosis. There are two key threats in this scenario: (i) the risk of an adversary eavesdropping on the data transmission, and (ii) the risk of the MLaaS provider performing unauthorised access on the user's data – in this case, the recording produced by Alice. The first threat can be mitigated using cryptographic protocols such as TLS. However, the second risk is harder to address, especially if the user data is decrypted before inference [15].

The use of HE mitigates both risks. The data is encrypted using HE, which is sufficient to prevent an adversary from eavesdropping. In addition, the provider is only able to perform computations on the encrypted data and will output the inference result without being able to decrypt. Figure 2.1 illustrates this scenario with an example applied to image recognition.

### 2.1.2 Homomorphic encryption

#### Basic terminology

A secret key encryption scheme is a triple of functions $\Pi = (\text{KeyGen}, \text{Enc}, \text{Dec})$, where KeyGen is the key generation function and Enc and Dec are the encryption and decryption

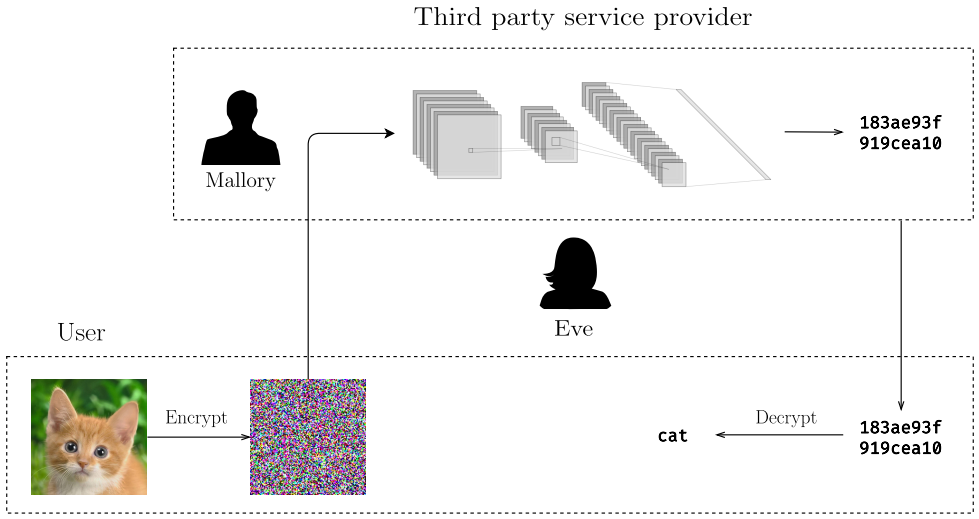



**Figure 2.1:** A depiction of the threat model. Mallory is a malicious individual within the third-party, and Eve is an eavesdropper listening to the communication. With HE, neither Mallory nor Eve can feasibly discover the contents of the image without the user's private key.

functions respectively. Such a scheme is initialised by generating a key $K \leftarrow \text{KeyGen}(1^l)$ where $l$, the security parameter, controls the number of bits of security offered by the scheme[1]. Denote the space of plaintext messages with $\mathcal{M}$. A message $m \in \mathcal{M}$ is encrypted using $\text{Enc}_K(m)$ and a ciphertext $c \in \mathcal{C}$ is decrypted using $\text{Dec}_K(c)$.

$\Pi$ can be extended into a homomorphic encryption scheme by introducing an evaluation function $\text{Eval}(f, c_1, \ldots, c_n)$ that takes a Boolean circuit $f$ and ciphertexts $c_1, \ldots, c_n$ such that for all $f$ and $c_1, \ldots, c_n$ it holds that

$$\text{Dec}_K(\text{Eval}(f, c_1, \ldots, c_n)) = f(x_1, \ldots, x_n), \tag{2.1}$$

where $x_1, \ldots, x_n$ are the respective decryptions of $c_1, \ldots, c_n$. This is illustrated in Figure 2.2. A similar construction can be done for a public key encryption scheme.

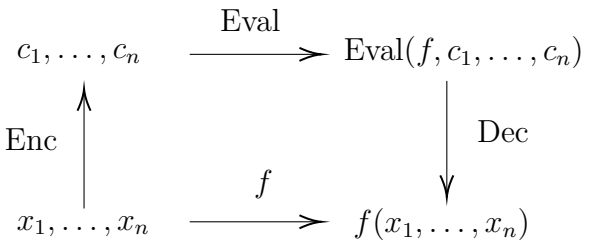



**Figure 2.2:** Homomorphic encryption

In other words, a *fully* homomorphic scheme enables evaluation of Boolean circuits up to an arbitrary depth. In practice, due to matters of time complexity, many schemes are *levelled* homomorphic schemes, which only support evaluation of circuits up to a *bounded depth*. In applications of HE to machine learning and especially deep neural inference, the supported depth is a crucial factor and places a limit on the scope of supported models. A model with several dozen layers, such as ResNet50, would not be supported by a levelled scheme with a maximum depth of only 10. In practice, many schemes support

[1] $n$-bits of security' means that an adversary would need to perform $2^n$ queries on average to break the scheme.

the arithmetic operations of addition and multiplication, rather than the functionally complete NAND gate.

### Ring learning with errors

Most readers are probably familiar with the notion of the hardness of a computational problem as a basis upon which cryptographic schemes are secure. For instance, RSA relies on the hardness of the integer factorisation problem. There exists no known algorithm for factorising a large composite integer into its prime factors in polynomial time on a classical computer. Recall from complexity theory that this problem is known to be in NP but it is unknown whether it is in P. Likewise, the security of the schemes used in the project's implementation are based on the hardness of the *ring learning with errors* (RLWE) problem, introduced by Lyubashevsky et al [16]. There exists a polynomial time reduction to RLWE from the *shortest vector problem*, which is NP-hard under certain choices of parameters.

A ring can be informally viewed as a set that satisfies certain properties, including being closed under the operations of addition and multiplication. One example of a ring is $\mathbb{Z}[X]$, the set of polynomials with integer coefficients. The RLWE problem is concerned with a more specific ring – namely the set of polynomials modulo $\Phi(X)$ that must also have coefficients in $\mathbb{Z}_q$[2]. This is a *quotient* ring denoted $\mathcal{R}_q = \mathbb{Z}_q[X]/(\Phi(X))$, where $\Phi(X)$ is an *irreducible* polynomial – one that cannot be factored into two non-constant polynomials. For instance, if $\Phi(X) = X^{1024} + 1$, then one can write

$$\mathcal{R}_q = \frac{\mathbb{Z}_q[X]}{(X^{1024}+1)} = \left\{a_0 + a_1X + a_2X^2 + \ldots + a_{1023}X^{1023} \mid a_0, \ldots, a_{1023} \in \mathbb{Z}_q\right\}. \quad (2.2)$$

Informally, RLWE is the problem of finding a secret $s \in \mathcal{R}_q$ given a vectors of polynomials computed using $s$ and some sampled errors. For interested readers, a detailed definition is given in Appendix A.1, along with the formal definition of a ring. The implication is that an encryption scheme can potentially encode a plaintext vector $\mathbf{v}$ as a list of polynomials, and then use a secret polynomial to turn this list into a ciphertext, from which it is infeasible to recover $\mathbf{v}$ in polynomial time.

Modern HE schemes rely on the RLWE assumption, which is that the RLWE problem is computationally infeasible, in order to assert IND-CPA security. On a high-level, a scheme is said to have *indistinguishable encryptions under a chosen-plaintext attack*, or IND-CPA security, if for all probabilistic, polynomial-time adversaries[3], the probability of correctly deciding if a given ciphertext is an encryption of one of two distinct plaintext messages is no better than 50%. The practical implication is that, even if granted access to an *oracle* that encrypts data, an adversary's success rate in finding the plaintext of a given ciphertext is no better than randomly guessing.

In practice $\Phi(X)$ is chosen as a *cyclotomic polynomial*. The $n$-th cyclotomic polynomial, $\Phi_n(X)$, is defined as

$$\Phi_n(X) = \prod_{k\in[1,n],\ \gcd(k,n)=1} \left(X - e^{2i\pi k/n}\right). \quad (2.3)$$

[2] $\mathbb{Z}_q$ is the set of integers modulo $q$.

[3] Such an adversary is one that can 'toss coins' in its computations, and must operate in polynomial time.

If $n$ is an even power of two, then $\Phi_n(X) = X^{n/2} + 1$. Polynomials of this form are suitable candidates because they allow multiplication to be implemented using the *number theoretic transform* (NTT)[4] which can be accelerated in hardware [17].

Since values are encoded as polynomials with coefficients modulo $q$, it is natural to expect the coefficients to grow large after a few multiplications. If the coefficients wrap around the modulus, then the decryption will be incorrect. This is the premise behind the limited depth property of leveled schemes. The modulus $q$ can be increased as a parameter, but there is a caveat: given a polynomial degree $n/2$, for a sufficiently small coefficient modulus $q$ there exists efficient attacks against the RLWE problem. This imposes a vital trade-off between the supported depth of multiplication and the security level.

### 2.1.3 Neural network inference

The problem of producing a result from a given feature vector is a common task in data science. A well-known method is linear regression, which produces a single output as a linear combination of the inputs. This can be expressed as

$$y = \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n = \mathbf{w}^T \mathbf{x}, \tag{2.4}$$

where $\mathbf{w}^T = [\beta_1, \ldots, \beta_n]$. This can be generalised to a multi-output regression by making $\mathbf{w}$ a matrix rather than a vector. A *neural network* can be seen as a further generalisation, where the result of one multi-output regression is fed as input to a subsequent regression until a set of outputs is produced. Such a network can be visualised as a series of nodes connected with edges representing weights, as shown in Figure 2.3. For the sake of brevity, I recommend the first seven lectures of *Deep Neural Networks*[5] given by Lawrence, Huszár and Lane for a comprehensive introduction to concepts including gradient descent, convolutional neural networks, pooling, and loss functions.

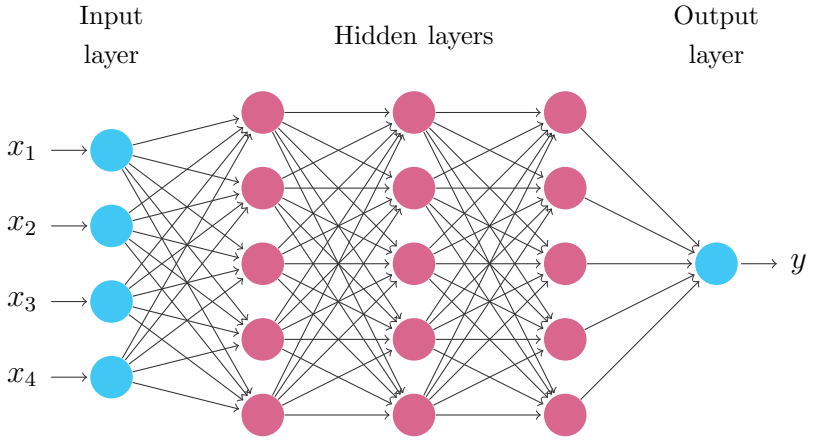



**Figure 2.3:** A neural network with four inputs $x_1, \ldots, x_4$ and a single output.

It is possible to express the components of many neural networks as arithmetic circuits comprised solely of addition and multiplication. For example, this is true for fully connected or *dense* layers, as they can be expressed as matrix-vector products.

[4] This is a variant of the discrete Fourier transform that operates over finite fields.
[5] Available online at `https://mlatcl.github.io/deepnn/`.

The same can be done for convolutional layers, which are equivalent to dense layers with particular connections removed and others made to share weights. There are, however, some components that cannot be directly treated as a sequence of additions and multiplications. In particular, this makes some pooling operations such as max-pooling and min-pooling infeasible to implement using RLWE-based HE operations. However, average pooling is still possible. In addition, the ReLU activation function $\mathrm{ReLU}(z) = \max(0, z)$ cannot be directly implemented using HE and must be approximated with a polynomial.

## 2.2 Project strategy

### 2.2.1 Requirements analysis

The project proposal identifies a set of requirements, which are shown below. The nature of the project required theoretical knowledge to be gained first before implementation, and as such the initial requirements of the proposal were subject to minor changes, as list of which is given in Appendix B.7. There are two categories of requirements (A and B): those in A are deemed essential and constitute the minimum baseline of the project. Those in B are more advanced and are treated as 'extensions' from the perspective of project management.

#### Essential requirements

- A1: *implement a method of defining an arbitrary neural network from its component layers.*

  This component is vital and is the one upon which all other components are built. Although conceptually simple, caution must be taken in ensuring that the resulting framework is usable from the perspective of human-computer interaction (HCI). Code clarity is especially important here.

- A2: *implement basic neural network components using HE semantics and integrate them with the SEAL library.*

  This involves designing and implementing a solution for using HE to perform neural network operations such as the matrix-vector product, and 2D convolutions.

- A3: *train and evaluate MNIST and CIFAR-10 models and convert them into HE models using Ψ-lib. Evaluate this model in terms of its efficiency, accuracy and security.*

  This involves training models using a machine learning library and packaging the weights of the model into a format that can be read by Ψ-lib.

#### Extensions

- B1: *apply optimisations to the implementations of the neural network architecture components using HE.*

  Both CryptoNets [3] and Faster CryptoNets [11] use the SIMD packing method, which sacrifices latency for high throughput. There are alternative packing methods such as the ones discussed by Juvekar et al [8] that focus on optimising latency.

- B2: *apply pruning techniques on models to make them more feasible for secure inference.*

  Pruning and model compression techniques can make secure models more practical. For instance, Chou et al [11] use network surgery and weight quantisation to prune network operations. More recent pruning techniques, such as the lottery ticket hypothesis, have yet to be applied to private inference.

- B3: *train a facial recognition model and achieve privacy-preserving facial recognition using Ψ-lib.*

  This is perhaps the ultimate goal of the project and would require a combination of many of the previous requirements to be realised.

- B4: *provide a Python implementation of CKKS and compare with the implementation of SEAL.*

  An implementation in Python is not strictly necessary and will likely not outperform existing C++ implementations such as SEAL – but doing so would reveal greater insight into the inner workings of the encryption scheme, which is quite sparsely documented and not well known in the data science community.

### 2.2.2 Methodology

An iterative development cycle was employed as a means of completing the project within the proposed schedule. The absence of an existing framework to develop from meant that it was difficult to evaluate the precise amount of effort each implementable component would take. Therefore, it was clear that the flexibility of an iterative process would be more suitable than the rigidity of a waterfall approach. Rather than performing the entire literature review in one large stage separated from the start of implementation, the project timeline was further enhanced into a model which takes advantage of the modular nature of the project. It consists of the following iterative process:

1. *Perform literature review and background reading into component to be implemented.*
2. *Divide project component into modular subparts. Identify the order of priority of the subparts.*
3. *Implement subparts.*
4. *Identify next component to be implemented.*

This methodology's main advantage is its flexibility and ability to mitigate uncertainty – a factor especially important in the context of the COVID-19 pandemic, which was present throughout the whole duration of the project. A Gantt chart of the project's timeline is shown in Figure 2.4.

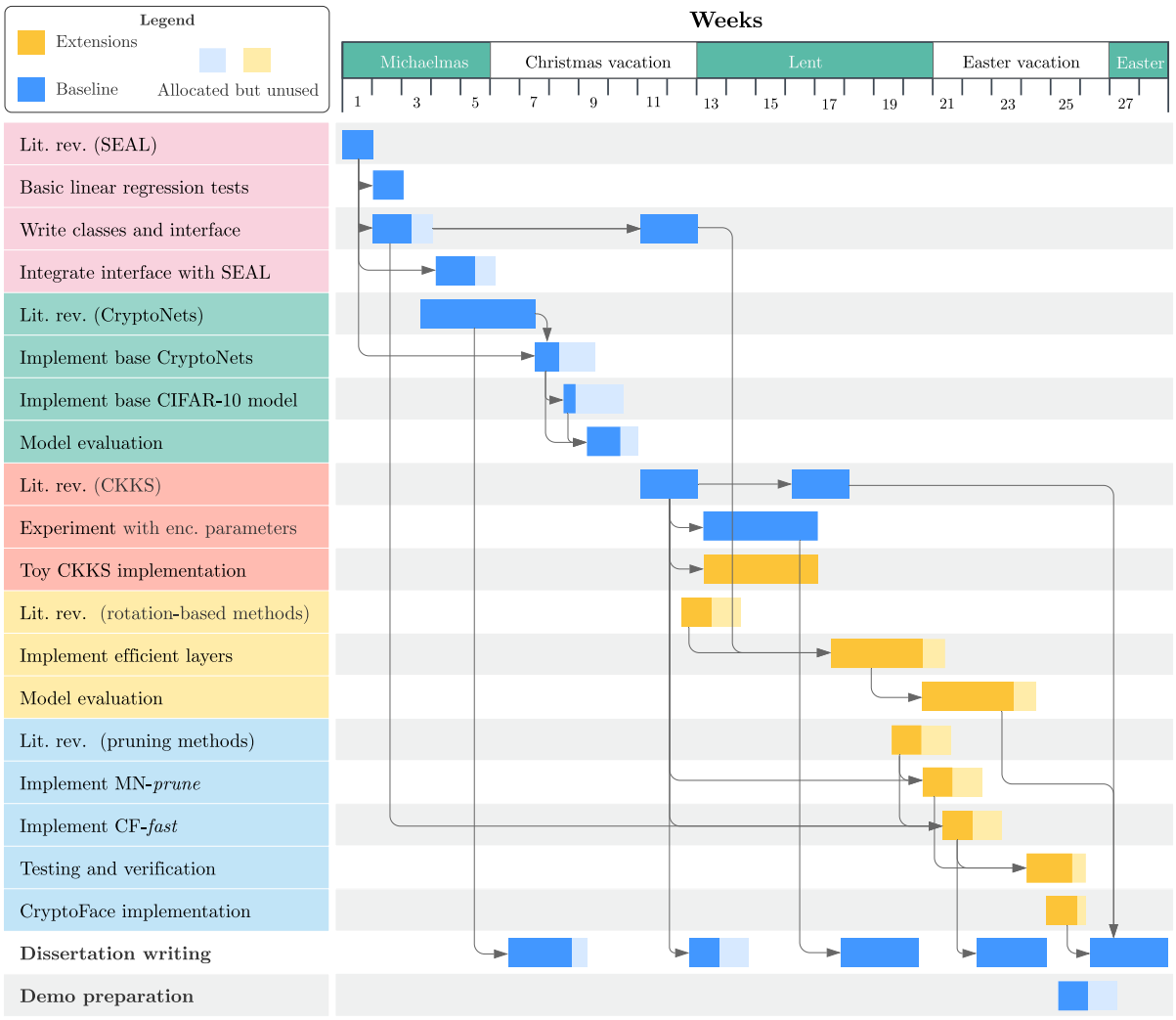



**Figure 2.4:** Project timeline.

## 2.3 Starting point

### 2.3.1 Knowledge and experience

Prior to the start of the project, I had taken or was in the process of taking the following courses relevant to the themes of the project: *Machine Learning and Real World Data*, *Foundations of Data Science* and *Security*. I was also familiar with Python and applied machine learning. The Part II *Deep Neural Networks* course was also useful in understanding the theoretical underpinnings of contemporary deep learning.

It should be mentioned that my understanding of HE was very limited, as the subject is beyond the scope of the Part II Cryptography course, which meant it was necessary to learn much of the theory independently during the project. The subject of applied HE, and especially the details of recent schemes, is quite sparsely documented. I credit much of my understanding of the subject to academic papers; notably [4] and [18].

### 2.3.2 Use of libraries and datasets

#### Cryptography

The project makes use of Microsoft's SEAL library [5], which provides C++ implementations of optimised versions of the BFV and CKKS schemes. SEAL was chosen as it implements several optimisations that alleviate memory and compute time requirements. In particular, it implements residue-number-system variants of both schemes to support large plaintext moduli. The SEAL API is made available through the use of a Python wrapper library [19].

#### Machine learning training

Tensorflow [20] was chosen for model training and evaluation due to the availability of resources and documentation on its usage. Tensorflow conveniently allows the extraction of model weights into a `numpy` array form, which is used as the standard input format to $\Psi$-lib.

#### Datasets

My experiments revolve around three publicly available datasets:

- The MNIST handwritten digits dataset is a standard 'toy' dataset primarily used to benchmark machine learning models. The dataset contains 60,000 training examples and 10,000 test examples, each of which is a $28 \times 28$ grayscale image. There are 10 classes, each corresponding to a digit from 0 to 9.
- The CIFAR-10 dataset is a comparatively harder dataset containing 50,000 training images and 10,000 test images of various objects. Each image is a $32 \times 32$ RGB image belonging to one of ten classes. This is a more difficult dataset to train and one deemed quite challenging from the perspective of achieving private inference.
- The third dataset is the CelebA dataset [21], which contains 202,599 images and 10,177 identities of celebrity images collected from the Internet. The use of CelebA is primarily for showing the applicability and feasibility of $\Psi$-lib in domains outside of the standard benchmark datasets used numerously in the literature. Facial recognition is one area that highly warrants the use of encryption due to the associated privacy issues of storing large quantities of biometric data on a server.

### 2.3.3 Pre-implementation decisions

#### Choice of language

A motivation of the project is to create a library usable for machine learning practitioners. Python was a natural choice due to the machine learning community's familiarity towards it. From the perspective of project management, Python is a highly suitable choice since it is usually faster for implementing machine learning algorithms than a lower level language. It must, however, be remarked that cryptographic applications are often written in a lower level language such as C++ for additional speed. I leave a possible reimplementation of my work in C++ for speedup purposes as an extension for any reader who is interested.

#### Choice of encryption scheme

I chose to support the class of schemes based on the RLWE problem, as they are frequently studied in the literature. In particular, the $\Psi$-lib is designed to support the CKKS scheme due to its affinity for representing real numbers. The written interface can also be easily extended to other RLWE-based schemes supporting the same set of operations, such as BFV.

### 2.3.4 Computer resources

In the original project proposal it was mentioned that a cloud service for machine learning training such as Azure or AWS may be necessary. However, this was not required in the implementation of the project. A desktop machine with a 4.7 GHz Intel Skylake processor and a GPU with 10 GB of VRAM was sufficient for training all models.

# Chapter 3

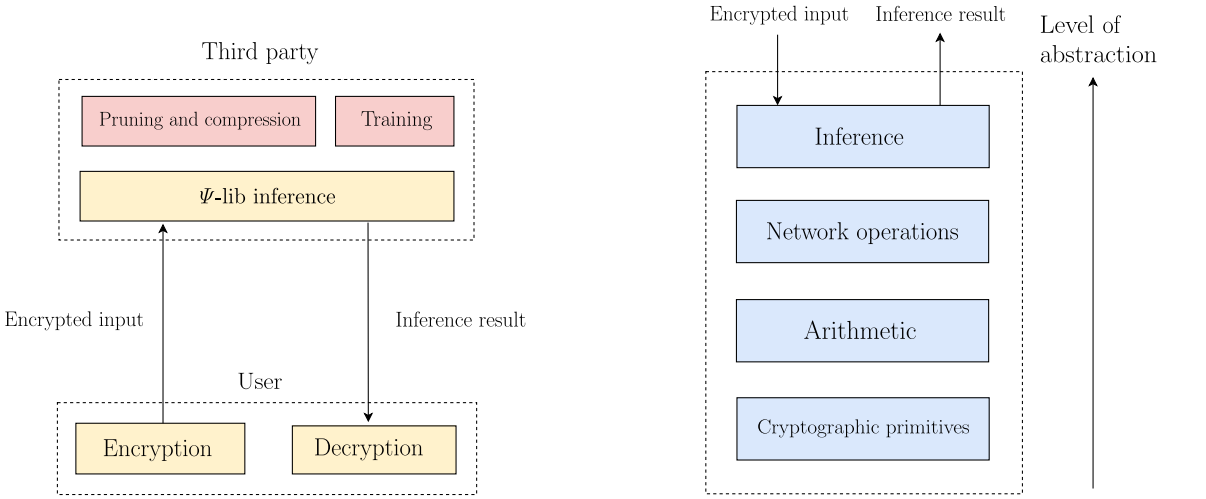
# Implementation

## 3.1 Architecture

This section details the architecture and design of the system from a high-level perspective, and discusses the roles of each implementation component, the design choices made, and the software engineering principles applied.

### 3.1.1 High-level framework

I first introduce a framework around which the core components of the system are based. This framework is a solution to the threat model outlined in Section 2.1.1. The system's




**(a)** The core components of the project.

**(b)** Levels of abstraction in performing secure inference.

**Figure 3.1:** High-level overview of the implementation.

core components are shown in Figure 3.1a. There are components which operate on-line as part of inference – these have been highlighted in yellow, as well as off-line components that are highlighted in red.

- The on-line components rely on the use of HE and is where most of my discussion occurs. In the *encryption* component, input data is encrypted using a selected scheme, and the encrypted counterpart is outputted. SEAL's CKKS implementation is used as the underlying scheme to produce the benchmark results in later sections. I also provide my own Python implementation of CKKS. In the *inference* component the encrypted input is received and private network inference is performed to produce the *inference result* in encrypted form. The *decryption* component is responsible

for decrypting the inference result and producing a human-readable version of the network's output.

- The off-line components are intended for use before model deployment by the service provider. Compute time is a smaller concern as a result. The framework's *training* component refers to the process of the provider training a model, or selecting a pretrained one. I perform this component when applying $\Psi$-lib to benchmark datasets and the facial recognition problem. The *pruning and compression* component is an additional step that the provider can apply to make large models with high parameter counts practical in the private inference scenario.

In Figure 3.1b, a more detailed overview is given for the framework's *inference* component, which spans multiple layers of abstraction that all sit above the encryption primitives at the very bottom. An abstraction layer which is beyond the scope of this project but very much relevant, is that between the cryptographic primitives and hardware. This layer has the potential to produce considerable speedups – for example, one can use an accelerator such as a GPU for performing cryptographic operations, such as in [10].

### 3.1.2 Software interface

A repository overview is shown in Figure 3.2. $\Psi$-lib is written in a fashion that allows separation of the different abstraction layers highlighted in 3.1.1. This approach was chosen to allow use of $\Psi$-lib with minimal interaction with the underlying cryptographic operations. I now give an overview of the most important components of my design, from the high-level interface to the low-level encryption.

- The `model` package contains the high-level interface used to define secure models. It provides functions allowing the user to build arbitrary neural networks. In particular, I opt for a modular approach, in which layers are treated as if they were modular components designed to be inserted and removed. This approach was chosen due to its intuitive nature and the ability to increase code reuse.

- The level below is the `real` package, which uses the underlying cryptographic primitives to provide a notion of a real number that the above level can use as part of computation. I offer several representations of real numbers using cryptographic primitives, each of which is placed into a separate file within the package.

- The lowest level is the `crypto` package, which contains the cryptographic primitives and the operations which are performed on them. The form of primitives supported are polynomials that encode *real vectors*, and thus there are multiple options for the mapping between these primitives and the actual real numbers used in inference.

Furthermore, implementations of the models are given in `.ipynb` notebooks inside the `notebooks` directory, which contains the source code behind the implementation of Section 3.5. An example of the usage of the interface is given in B.1.

```
.
├── notebooks
│   ├── Inference.ipynb          #Application of psi-lib to trained models
│   ├── Models.ipynb             #Implementations of MN-fast, MN-pruned and CF-fast
│   ├── CryptoFace.ipynb         #Implementation of the CryptoFace architecture
│   └── Plots.ipynb              #Additional plots used in the dissertation
├── model
│   ├── creator.py               #Methods for encrypting data
│   ├── revealer.py              #Methods for decrypting data
│   ├── lin_algebra.py           #Helper functions for linear algebra on ciphertext
│   ├── low_lat.py               #Implementation of low-latency operations
│   ├── model_funcs.py           #Helper functions for extracting model weights
│   └── model_builder.py         #Model constructor
├── real
│   ├── batched_real_double.py   #A representation of real numbers using CKKS
│   ├── batched_real_integer.py  #A representation of integers using BFV
│   └── batched_real.py          #An interface class
├── crypto
│   ├── ckks.py                  #My CKKS implementation
│   ├── polynomial.py            #Polynomial operations
│   └── schemes.py               #Interface with SEAL
├── utils
│   ├── data_loader.py           #Functions for loading weights
│   ├── logger.py                #Functions for debugging
│   └── utils.py                 #Additional miscellaneous functions
└── data                         #Contains weights and parameters of models
```

**Figure 3.2:** Repository overview.

## 3.2 Encryption primitives

In this section, I briefly discuss the fundamentals of the CKKS scheme, as described in the original paper by Cheon et al [4], and its usage in my implementation. To unveil some of the complexities behind the original CKKS scheme to a wider audience, I provide a toy Python implementation in `crypto/ckks.py`. The SEAL version of CKKS relies on the same principles but applies advanced optimisations such as a residue number system [22], the details of which are beyond the scope of this dissertation.

### 3.2.1 Scheme overview

The CKKS scheme, on a high level, encrypts plaintext polynomials into ciphertext polynomials using a *public key*, on which the operations of addition and multiplication are performed using an *evaluation key*. Decryption uses the *private key* to retrieve the plaintext with a certain amount of approximation error.

To apply CKKS to real vectors, a vector must first be encoded as a polynomial in the ring $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$ where $N$ is a power of 2. The encoding procedure involves a rounding stage, meaning that the vector must be multiplied by a *scaling factor* $\Delta$ in order to preserve precision. Once encoded, a plaintext polynomial in $\mathcal{R}$ can be encrypted into a ciphertext, represented by a *pair* of polynomials.

Suppose we have two vectors $\mathbf{v}$ and $\mathbf{w}$. We encode scaled versions $\Delta\mathbf{v}$ and $\Delta\mathbf{w}$ to obtain plaintext polynomials, which we encrypt to obtain ciphertexts $\mathsf{ct}_1$ and $\mathsf{ct}_2$. Multiplying these will produce the encryption of $\Delta^2\mathbf{v} \odot \mathbf{w}$. The key intuition is that subsequent chaining of multiplications will result in an ever-increasing scaling factor, which will in turn reduce the precision of the decryption result. Hence, CKKS introduces a *rescaling* procedure, which effectively divides the ciphertext by $\Delta$ to change the scale from $\Delta^2$ back to $\Delta$.

Furthermore, each ciphertext belongs to a certain level. Each level has a *coefficient modulus* $q$ – that is, the ciphertext polynomial's coefficients must be in modulo $q$. When a plaintext is first encrypted, the resulting ciphertext belongs to level $L$, the maximum level, and has coefficient modulus $Q = q_0 \cdot \Delta^L$ where $q_0$ is the *base modulus*. When rescaling is applied to a ciphertext, the level decrements, and the coefficient modulus is divided by $\Delta$. Thus, a ciphertext at level $l$ has coefficient modulus $q_l = q_0 \cdot \Delta^l$. The lowest level is $l = 0$, at which further multiplication cannot be performed. For correct decryption, the coefficients of the plaintext polynomial must not exceed the base modulus $q_0$. Hence in practice, we set $q_0$ to be larger than $\Delta$.

### Encoding and decoding

The plaintext message space in CKKS is the polynomial ring $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$. In particular, this is a *cyclotomic polynomial ring* $\mathcal{R} = \mathbb{Z}[X]/(\Phi_M(X))$ where $\Phi_M(X)$ is the $M$-th cyclotomic polynomial and $M = 2N$ is a power of two. The objective of *decoding* is to map an element in $\mathcal{R}$ to complex vector, and that of *encoding* is to do the reverse.

Decoding can be seen as mapping an element $a \in \mathcal{R}$ to a complex vector in $\mathbb{C}^N$ via the embedding $\sigma : \frac{\mathbb{R}[X]}{(X^N+1)} \to \mathbb{C}^N$, which is applied via evaluation of $a$ on each root of $\Phi_M(X)$. This is expressed as

$$\sigma(a) = (a(\xi), a(\xi^3), \ldots, a(\xi^{M-1})) = \left(a(\xi^k) \mid k \in \mathbb{Z}_N^*\right) \in \mathbb{C}^N, \tag{3.1}$$

where $\xi = e^{2\pi i/M}$ is a primitive $M$-th root of unity. However, to establish a one-to-one mapping, half the resulting complex vector must be *discarded* during decoding, and the input must be *scaled and rounded* during encoding. The resulting form of the encoding function is

$$\mathsf{Encode}(\mathbf{z}, \Delta) : \mathbb{C}^{N/2} \times \mathbb{Z}^+ \to \mathcal{R}, \tag{3.2}$$

where $\Delta$ denotes a scaling factor. Supplementary section A.2 provides a more thorough explanation as to why.

### Operations

The list of operations in the base CKKS scheme is shown in Figure 3.3. In particular, note that ciphertext multiplication requires polynomials to be multiplied – which is significantly costlier than addition. Formal definitions and explanations for each operation are given in supplementary section A.2.2.

### Rescaling

The previously mentioned rescaling operation is defined as

$$\mathsf{Rescale}(\mathbf{c}, \Delta^{l_{\text{new}}}) : \mathcal{R}_{q_l}^2 \times \mathbb{Z}^+ \to \mathcal{R}_{q_l}^2 = \lceil \Delta^{l_{\text{new}}-l} \cdot \mathbf{c} \rfloor \bmod (q_{l_{\text{new}}}). \tag{3.3}$$

Given a ciphertext $\mathbf{c}$, the rescaling procedure enables some of the least significant bits of the ciphertext to be truncated via division by a power of $\Delta$. In practice, we perform this procedure after every multiplication, since multiplication grows the scaling factor to $\Delta^2$ and rescaling brings it down to $\Delta$.

- $\mathsf{KeyGen}$: sample $\mathbf{s} \leftarrow \chi_{\text{key}}$, $\mathbf{a} \leftarrow \mathcal{R}_{q_L}$, $\mathbf{a}' \leftarrow \mathcal{R}_{P \cdot q_L}$, $\mathbf{e} \leftarrow \chi_{\text{err}}$ and $\mathbf{e}' \leftarrow \chi_{\text{err}}$.
  1. Compute $\mathbf{b} := -\mathbf{a}\mathbf{s} + \mathbf{e} \bmod q_L$ and $\mathbf{b}' := -\mathbf{a}'\mathbf{s} + \mathbf{e}' + P\mathbf{s}^2 \bmod P \cdot q_L$.
  2. Return $\mathsf{sk} := (1, \mathbf{s})$, $\mathsf{pk} := (\mathbf{b}, \mathbf{a})$, $\mathsf{ek} := (\mathbf{b}', \mathbf{a}')$.
- $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{m})$: sample $\mathbf{v} \leftarrow \chi_{\text{enc}}$ and $\mathbf{e}_0, \mathbf{e}_1 \leftarrow \chi_{\text{err}}$. Let $\mathsf{pk} \equiv (\mathbf{b}, \mathbf{a})$.
  1. Set $\mathbf{c}_0 := \mathbf{v} \cdot \mathbf{b} + \mathbf{m} + \mathbf{e}_1$ and $\mathbf{c}_1 := \mathbf{v} \cdot \mathbf{a} + \mathbf{e}_2$.
  2. Return $(\mathbf{c}_0, \mathbf{c}_1) \bmod q_L$.
- $\mathsf{Dec}_{\mathsf{sk}}((\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s})) = (\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s}) \bmod q_l$
- $\mathsf{Add}\,((\mathbf{c}_0, \mathbf{c}_1), (\mathbf{d}_0, \mathbf{d}_1)) = (\mathbf{c}_0 + \mathbf{d}_0, \mathbf{c}_1 + \mathbf{d}_1) \bmod q_L$
- $\mathsf{AddPC}\,((\mathbf{c}_0, \mathbf{c}_1), x) = (\mathbf{c}_0 + \mathbf{m}, \mathbf{c}_1) \bmod q_L$
- $\mathsf{MultPC}\,((\mathbf{c}_0, \mathbf{c}_1), x) = (\mathbf{c}_0 \cdot \mathbf{m}, \mathbf{c}_1 \cdot \mathbf{m}) \bmod q_l$
- $\mathsf{Mult}((\mathbf{c}_0, \mathbf{c}_1), (\mathbf{d}_0, \mathbf{d}_1)) = (\mathbf{a}_0, \mathbf{a}_1) + \left(\lceil P^{-1} \cdot \mathbf{a}_2 \cdot \mathsf{ek}[0] \rfloor, \lceil P^{-1} \cdot \mathbf{a}_2 \cdot \mathsf{ek}[1] \rfloor\right) \bmod q_l$

**Figure 3.3:** The basic operations in CKKS. $P$ is a large scaling factor used during ciphertext-ciphertext multiplication, and $\lfloor \cdot \rceil$ denotes rounding to the nearest integer.

### Rotations

A useful feature of RLWE based schemes is the ability to perform rotations on ciphertext slots. Given an encryption of $\mathbf{v}$ as input, a rotation produces an encryption of $\mathbf{v}'$, such that $\mathbf{v}'$ is the vector obtained by rotating the elements in $\mathbf{v}$ by some offset $d$. This is achieved by homomorphically performing a *Galois automorphism* on the ciphertext, which uses results from *Galois theory*.

### SEAL's RNS implementation and rescaling

The polynomial coefficients can grow very large in practice and would require arbitrary-precision arithmetic, which is slower than normal arithmetic over 64-bit integers. SEAL uses a *residue number system* (RNS) which exploits the Chinese Remainder Theorem to decompose a large integer coefficient in $\mathbb{Z}_q$ into $n$ smaller ones $\mathbb{Z}_{p_1}, \ldots, \mathbb{Z}_{p_n}$ using a *moduli chain* $p_1, \ldots, p_n$ such that $\prod_i p_i = q$ and each $p_i$ is a prime representable with fewer than 64 bits. In hardware it is faster to compute $n$ separate multiplications using $p_1, \ldots, p_n$ than a single multiplication using a large $q$. The separate results are combined together using the isomorphism between $\mathbb{Z}_q$ and $\mathbb{Z}_{p_1} \times \ldots \times \mathbb{Z}_{p_n}$.

The implication is that $q_l = q_0 \cdot \Delta^l$ cannot be maintained, because it is not always possible to find every $q_l$ as the product of $n$ primes. SEAL instead defines $q_l = q_0 \cdot \prod_{i=1}^{l} p_i$ where $q_0$ must be prime. Whereas the original scheme divides the ciphertext by $\Delta$ during rescaling, SEAL divides by $p_l$. The resulting scaling factor is only approximately equal to $\Delta$. Hence, in practice, we always multiply by a constant close to 1 after rescaling to set the scale back to $\Delta$, and choose primes for the modulus chain of approximately the same size as $\Delta$.

## 3.3 Network layers

### 3.3.1 SIMD Packing

I implement the SIMD packing scheme first introduced by CryptoNets. The idea is to represent a batch of $m$ inputs each with $n$ values using $n$ ciphertexts. Each ciphertext has $N > m$ slots. In Figure 3.4 we show this packing scheme for image inputs. This scheme is most effective when the batch size $m$ is the same as the slot count $N$ to maximise the exploitable parallelism. For high resolution inputs, a large number of ciphertext are required, resulting in high memory usage. Precisely, the size of the ciphertext stored in memory is roughly $N$ times that of the total batch size. Despite this, the inference throughput is increased by the fact that $m$ inferences are performed in parallel.

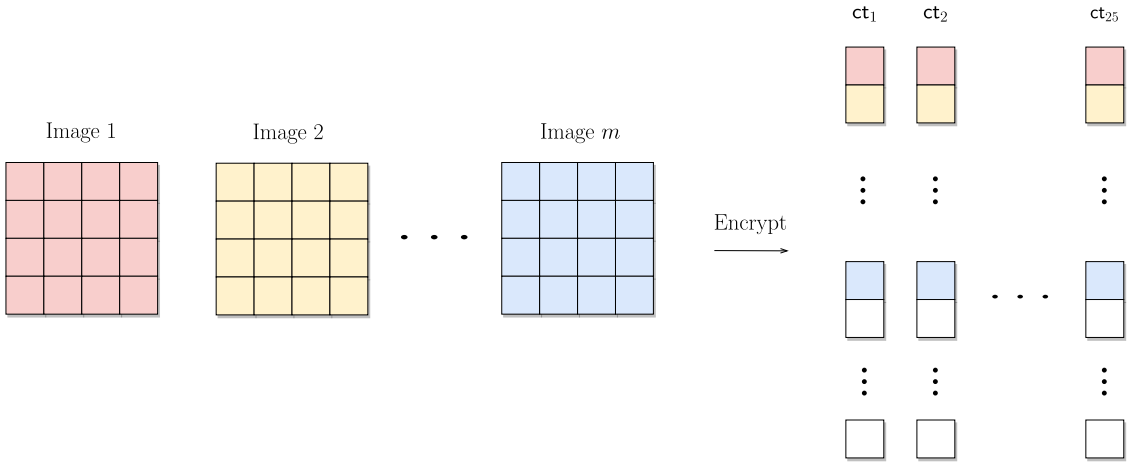



**Figure 3.4:** The SIMD packing scheme introduced by CryptoNets.

#### Network operations

Layers under the SIMD packing scheme are implemented by treating each ciphertext as a different value within the input, allowing the entire notion of ciphertext to be abstracted away, with the only limitation being the computation depth. I thus apply the traditional algorithms for dense and convolutional layers: SIMD-Dense and SIMD-Conv, shown in Figure 3.5. I utilise two functions: EncodeWeights, which takes a tensor of weights and produces a tensor of the same dimension but containing a corresponding ciphertext in each position, and DotProd, which takes a weight tensor, a ciphertext tensor, and performs element-wise multiplication and calculates the sum.

#### Activations

I implement the square activation function which is a simple application of the multiplication circuit:

$$\text{Square}(\mathsf{ct}) = \mathsf{Mult}(\mathsf{ct}, \mathsf{ct}) \tag{3.4}$$

The popular activation functions $\text{ReLU}(z) = \max(0, z)$, tanh and $\sigma(z) = 1/(1 + e^{-z})$ are not polynomials and therefore cannot be implemented using the HE operations in CKKS. Previous works have discussed the trade-offs associated with using polynomial approximations for avoiding exploding gradients during training [11].

**Algorithm 3.1:** SIMD-Dense

**input :** $\mathbf{v} \in \mathcal{C}^M, \mathbf{W} \in \mathbb{R}^{N \times M}$
**output:** $\mathbf{v} \in \mathcal{C}^N$
$\mathbf{u} \leftarrow$ empty array
$\mathbf{W}' = \text{EncodeWeights}(\mathbf{W})$
**for** $i \in \{0, 1, \ldots, N-1\}$ **do**
  $\mathbf{u}[i] \leftarrow \text{DotProd}(\mathbf{v}, \mathbf{W}'[i])$
**end**
**return u**

**Algorithm 3.2:** SIMD-Conv

**input :** $\mathbf{v} \in \mathcal{C}^{c_{\text{in}} \times d_{\text{in}} \times d_{\text{in}}}, \mathbf{W} \in \mathbb{R}^{k \times k \times c_{\text{in}} \times c_{\text{out}}}, k, s$
**output:** $\mathbf{u} \in \mathcal{C}^{c_{\text{out}} \times d_{\text{out}} \times d_{\text{out}}}$
$d_{\text{out}} \leftarrow \lfloor \frac{d_{\text{in}} - k}{s} \rfloor + 1, \mathbf{W}' \leftarrow \text{EncodeWeights}(\mathbf{W})$
$\mathbf{u} \leftarrow$ empty array of shape $c_{\text{out}} \times d_{\text{out}} \times d_{\text{out}}$
**for** $i \in \{0, 1, \ldots, c_{out} - 1\}$ **do**
  $\mathbf{w} \leftarrow \mathbf{W}'[:, :, :, i]$
  **for** $r, c \in \{0, 1, \ldots, d_{out}\}$ **do**
    $\mathbf{v}' \leftarrow \mathbf{v}[:, rsk : (r+1) \cdot sk, csk : (c+1) \cdot sk]$
    $\mathbf{u}[i, r, c] \leftarrow \text{DotProd}(\mathbf{v}', \mathbf{w})$
  **end**
**end**
**return u**

**Figure 3.5:** SIMD Network operations. The notation $A[\ldots, i : j, \ldots]$ denotes splicing $A$ at the specified dimension using bounds $i$ and $j$. $\mathcal{C}$ indicates the ciphertext space.

**Algorithm 3.3:** DotProd

**input :** $\mathbf{A} \in \mathcal{C}^{d_1 \times \ldots \times d_n}, \mathbf{B} \in \mathbb{R}^{d_1 \times \ldots \times d_n}$
**output:** $c \in \mathcal{C}$
$c \leftarrow$ null
**for** $0 \le d'_1 < d_1, \ldots, 0 \le d'_n < d_n$ **do**
  $p \leftarrow \mathsf{MultPC}(\mathbf{A}[d'_1, \ldots, d'_n], \mathbf{B}[d'_1, \ldots, d'_n])$
  $c \leftarrow \begin{cases} c + p, & \text{if } c \neq \text{null} \\ p & \text{otherwise} \end{cases}$
**end**
**return** $c$

**Figure 3.6:** Ciphertext dot product.

### Pooling and layer composition

Pooling is typically performed after convolution as a way to downsample the intermediate tensor or to extract features. Average pooling is computed by taking the sum of elements in each pooling window and dividing by the window size. This is a linear operation and is implemented as a special case of convolution where each kernel weight is $1/k^2$ such that $k$ is both the kernel side length and the stride. If padding is used, kernels that reach outside the image boundaries will hold different weights, depending on the area that extends into the padding. Since average pooling is linear, I implement a pooling layer that follows a convolution by *combining* them into a single linear layer. This uses a simple yet effective method – in which we reduce the multiplicative depth of a network by composing adjacent

linear sections into a single matrix-vector product. A network of $d$ layers can be expressed as a function $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^{n'}$ such that

$$f(\mathbf{x}) = \Phi_d \left(\mathbf{W}_d(\dots \Phi_2 \left(\mathbf{W}_2 \left(\Phi_1 \left(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1\right) + \mathbf{b}_2\right)\dots\right) + \mathbf{b}_d\right), \tag{3.5}$$

where $\mathbf{W}_i$, $\mathbf{b}_i$ and $\Phi_i$ are the weight matrix, bias vector and activation function of the $i$-th layer, respectively. Note that if all $\Phi_i$ are linear then $f(\mathbf{x})$ is linear itself, and can be expressed as the form $f(\mathbf{x}) = \mathbf{W}x + \mathbf{b}$. In my case, we use the identity activation $\Phi(x) = x$ in which case the expression above reduces to

$$f(\mathbf{x}) = \mathbf{W}_d \dots \mathbf{W}_2\mathbf{W}_1\mathbf{x} + \sum_{i=1}^{d} \mathbf{W}_d \dots \mathbf{W}_{i+1}\mathbf{b}_i. \tag{3.6}$$

This technique of composing layers was used in CryptoNets to compute multiple layers using only a single multiplication level. I apply this idea by converting the weights of convolution and average pooling layers to matrix-vector products, and by composing the weight matrices of consecutive convolution and pooling layers. It must be emphasised that this method, although effective in reducing the multiplicative depth required to evaluate the network, does not benefit model training, because the composition of linear operations is yet another linear operation and provides no additional representational power. One cannot reliably expect to gain better training accuracy by adding extra linear layers. However, it *can* be beneficial to add linear average pooling layers to reduce the working set size of intermediate layers.

### 3.3.2 Efficient packing

Networks evaluated using the SIMD packing scheme achieve high throughput, but will suffer from high message size and inference latency. The original CryptoNets model requires an inference time of 249.6 seconds and has a message size of 367.5 MB.

I utilise the ability to perform rotations on ciphertext to implement efficient convolution and dense layers, substantially reducing the amount of ciphertext required to perform a forward pass of the network. I adopt the convolutional layer packing scheme from [24], and a variation of the dense layer packing scheme from [8].

#### Efficient convolution

The discrete convolution of an image $f$ with a filter of width $w$, height $h$ and depth $d$ is

$$(f * g)[i, j, k] = \sum_{x=0}^{w-1} \sum_{y=0}^{h-1} \sum_{z=0}^{d-1} g[x, y, z] f[i + x, j + y, k + z]. \tag{3.7}$$

Observe that the data-level parallelism inherent in this computation enables it to be vectorised as

$$f * g = \sum_{x=0}^{w-1} \sum_{y=0}^{h-1} \sum_{z=0}^{d-1} g[x, y, z] \cdot \mathbf{F}^{(x,y,z)}, \tag{3.8}$$

where $\mathbf{F}^{(x,y,z)}$ is a matrix such that $\mathbf{F}_{ij}^{(x,y,z)} = f[i + x, j + y, k + z]$. This is the pretext behind the convolutional-packing method I apply to accelerate convolution using the SIMD ciphertext operations.

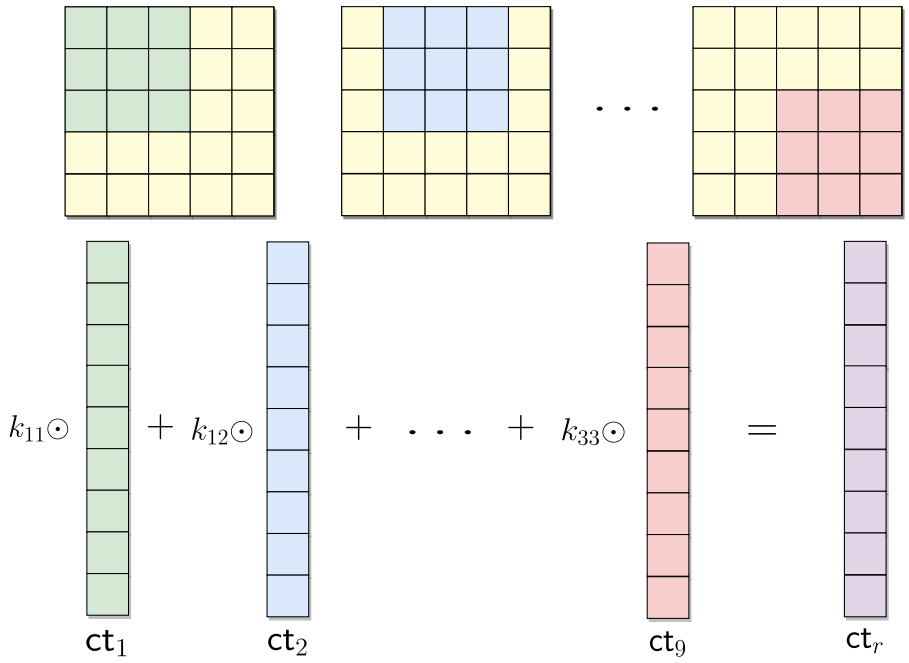



**Figure 3.7:** A depiction of the convolution packing method for $c_{\text{in}} = 1$, $d_{\text{in}} = 5$, $k = 3$ and $s = 1$.

The input channels of the convolution are arranged in a representation referred as *convolution packing*. For an input image $I \in \mathbb{R}^{c_{\text{in}} \times d_{\text{in}} \times d_{\text{in}}}$ feature maps and kernel of window size $k \times k$, the input image is represented as $k^2 \cdot c_{\text{in}}$ vectors $\mathbf{v}_1, \ldots, \mathbf{v}_{k^2 \cdot c_{\text{in}}}$, where $\mathbf{v}_i$ contains all elements convolved with the $i$-th value in the filter. Denote corresponding ciphertexts as $\mathsf{ct}_1, \ldots, \mathsf{ct}_{k^2 \cdot c_{\text{in}}}$. The process of producing the $j$-th output feature map is now reduced to ciphertext-plaintext multiplication of each $\mathsf{ct}_i$ with the $i$-th value in the $j$-th filter. An example is shown in Figure 3.7. In total, the process requires $k^2 \cdot c_{\text{in}}$ ciphertext-scalar multiplications per output feature map, leading to a total of $k^2 \cdot c_{\text{in}} \cdot c_{\text{out}}$ multiplications. Rotations, which are more expensive than multiplications, are not needed. Note that the input to the convolutional layer has to be arranged in the convolution packing format, which is cheap to compute on plaintext, but expensive on ciphertext. The primary reason is that $\mathbf{v}_i$ contains values which are not adjacent in the flattened representation of an image channel. I therefore use this method for the *first* convolutional layer of any model. Subsequent convolutional layers are implemented using the efficient matrix-vector product method detailed next.

### Rotate-and-sum

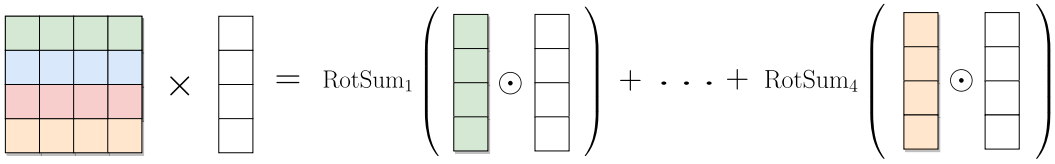
Here I discuss the important procedure of summing the elements in a ciphertext. The naive method of summing elements in a ciphertext with $2^k$ slots would require $2^k - 1$ rotations. It is not hard to devise an improved method where the ciphertext is first rotated $2^{k-1}$ times, added onto itself, rotated $2^{k-2}$ times and so on, until a final rotation by one slot. In total, this would take $k - 1$ rotations, logarithmic to the total slot count. In Algorithm 3.4 I give a definition that generalises this approach to groups of consecutive elements.

**Algorithm 3.4:** RotSum

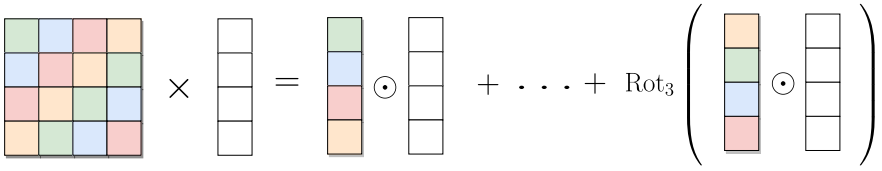
**input :** $\mathbf{v} \in \mathcal{C}, n \in \mathbb{Z}, k \in \mathbb{Z}$
**output:** $c \in \mathcal{C}$
$n \leftarrow k \cdot 2^{\lceil \log_2 \frac{n+k-1}{k} \rceil}$
$\mathbf{c} \leftarrow \mathbf{v}$.copy()
**while** $n \geq k$ **do**
  $n \leftarrow \frac{n}{2}$
  $\mathbf{c} \leftarrow \mathbf{c} + \text{Rot}(\mathbf{v}, -n)$
**end**
**return** $c$

### Efficient matrix-vector products

For the implementation of dense layers, we can first consider the inefficiency of the naive method whereby one takes a weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$, and performs element-wise multiplication of each row of $\mathbf{W}$ with input ciphertext $\mathbf{v}$, before summing each of the products. The sum of the first $N$ elements in a ciphertext is obtained using the *rotate-and-sum* algorithm (Algorithm 3.4) requiring $O(\log_2 N)$ rotations. In total, this naive method requires $O(m)$ element-wise ciphertext-scalar multiplications, $O(m \log_2 n)$ ciphertext rotations and $O(m)$ additions. The runtime is therefore dominated by the $O(m \log_2 n)$ rotations.




**(a)** The naive approach.



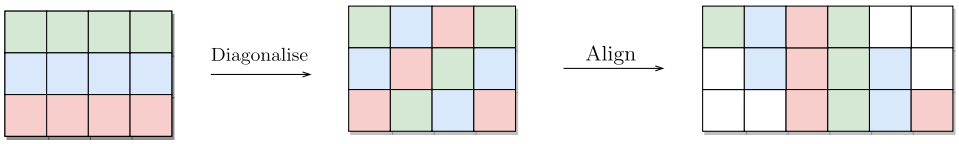

**(b)** The diagonals approach.

**Figure 3.8:** Two approaches for encrypted matrix-vector multiplication.

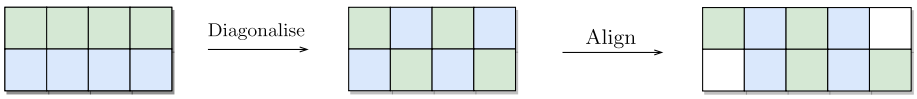
Ideally, we desire a number of rotations that grows linearly with $m$ or $n$. This is achieved using a procedure in which we obtain $m$ diagonals of $\mathbf{W}$, denoted $\{d_1, d_2, \ldots, d_m\}$ such that $d_i = [\mathbf{W}_{i,0}, \mathbf{W}_{i+1,1}, \ldots, \mathbf{W}_{i+n-1,n-1}]$. Note that all row positions are in modulo $m$. For each $d_i$, we then left-pad with $i$ zeros to align values belonging to the same row of $\mathbf{W}$, and finally perform ciphertext-plaintext multiplication of each padded diagonal with rotations of $\mathbf{v}$. The last stage is to apply the rotate-and-sum algorithm on the resulting product ciphertexts. Overall, this procedure requires $O(m)$ multiplications, $O(m + \log_2 n)$ additions and $O(m + \log_2 n)$ rotations, which yields a significant improvement over the $O(m \log_2 n)$ rotations of the naive approach when $n$ is large. Figure 3.8 shows this method alongside the naive method.

This type of method was introduced by [25] but on the set of diagonals along the columns rather than the rows. [8] gave a method equivalent to the one I implemented, but performed left-rotations instead. I chose to use right-rotations as they are easier to visualise.

I implement the diagonal method for dense layers, and also extend this idea to implement pooling and convolution layers by first converting them into a dense representation. The process of producing a weight matrix from a convolutional layer's kernels is nontrivial, especially if padding is accounted for, and I include my algorithm for this in Appendix B.2. Furthermore, one can realise that the alignment procedure depicted in 3.9 requires at least $m+n-1$ slots. It can also be shown that the rotate-and-sum algorithm requires the number of groups to be a power of two. Therefore, it is important to note that the diagonal method for dense layers requires at least $m \cdot 2^{\lceil \log_2 \frac{n+m-1}{m} \rceil}$ slots. For instance, if $m = 128$ and $n = 2000$ then 4096 slots are required.




**(a)** $(m+n-1)/m$ is a power of 2.




**(b)** $(m+n-1)/m$ is not a power of 2.

**Figure 3.9:** Weight matrix rotation for different dimensions $m \times n$.

## 3.4 Model pruning

In this section, I discuss my application of methods that reduce the size of models by pruning their parameters, which in turn reduces the number of operations during inference. Techniques from the domain of model pruning have traditionally been employed to reduce the duration and power consumption of inference on mobile devices with limited on-chip memory. The same techniques are also applicable to reducing the inference time of secure models.

### 3.4.1 Lottery ticket pruning

The *lottery ticket hypothesis* by Frankle and Carbin [6] states that for any dense, randomly initialised feed-forward network $M$, there exists a subnetwork which when trained, will reach similar test accuracy to the original network. This allows us to utilise a procedure which has been shown to effectively compress models by an order of magnitude without sacrificing significant levels of test accuracy. The use of pruning techniques is highly effective in reducing inference latency when the SIMD packing approach is used, because the pruning of a weight will always cause a reduction in the number of multiplications.

To apply lottery pruning to a model $M$ with weights $\mathbf{w}$, I first train $M$ to achieve a satisfactory performance threshold. Then the following procedure is taken:

1. Create a mask of the same shape as $\mathbf{w}$, based on a mask criterion.
2. Use this mask to prune $M$ by setting the corresponding weights to zero.
3. Set all non-pruned weights in $\mathbf{w}$ to their original initialisations. Train the pruned version of $M$ until an early stopping criterion is met.

I choose the masking criterion of eliminating the weights of each layer which are below the $p$-th lowest percentile in magnitude globally across all layers. In 3.5.1 I use the iterative version of the procedure [23] to prune 84.9% of the CryptoNets MNIST model, providing a much faster model. Frankle and Carbin mention that better initialisations are found when iteratively pruning, compared to a one-shot process.

### 3.4.2 Singular value decomposition

As mentioned in 3.3.2, the drawback of the convolutional packing method is that it is only effective for the first convolution that receives the input, due to the high cost of transforming the ciphertext output into the necessary input format. Recent work by Lu et al [7] introduced the notion of factoring a convolutional layer into two smaller sub-convolutions, such that the first sub-convolution produces fewer feature maps whilst the second one uses a $1 \times 1$ kernel to upscale the tensor to the original number of output channels.

Consider a discrete convolution of an image with $c_{\text{in}}$ channels with a filter of width $f_w$, height $f_h$ and depth $f_d$, and an output feature map with width $d_w$ and height $d_h$. The output feature map can be expressed as a matrix-vector product of the form $\mathbf{Mw}$ such that $\mathbf{M} \in \mathbb{R}^{d_w \cdot d_h \times f_w \cdot f_h \cdot f_d}$. An entire convolutional layer produces $c_{\text{out}}$ feature maps and therefore can be expressed as a matrix product of $\mathbf{M}$ and $\mathbf{W}$ where each column of $\mathbf{W}$ is a different filter. Lu et al suggest factoring $\mathbf{W}$ into matrices $\mathbf{A} \in \mathbb{R}^{f_w \cdot f_h \cdot c_{\text{in}} \times C'}$ and $\mathbf{B} \in \mathbb{R}^{C' \times C_{\text{out}}}$ such that $C' < C_{\text{out}}$. The key insight is that the second convolutional layer of a model can be factored into an initial sub-convolution that is feasible to compute as a dense layer, and a subsequent $1 \times 1$ sub-convolution that is quick to compute using the convolutional packing method. In my implementation, I perform the factoring of $\mathbf{W}$ using singular value decomposition (SVD). Via SVD, a matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$ can be factored into three matrices $\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}^*$, where $\mathbf{\Sigma}$ is a diagonal matrix containing the singular values of $\mathbf{W}$.

$$\underset{\mathbf{U},\mathbf{\Sigma},\mathbf{V}^*}{\operatorname{argmin}}(||\mathbf{W} - \mathbf{U\Sigma V}^*||^2)$$

SVD is performed using numpy's `numpy.linalg.svd` function. The matrices $\mathbf{U}$ and $\mathbf{\Sigma}$ are multiplied to form $\mathbf{A}$ and $\mathbf{V}^*$ is chosen as $\mathbf{B}$.

## 3.5 Application to datasets

### 3.5.1 MNIST

In this section, I explain my implementations of two MNIST models: MN-*prune*, a pruned version of the original CryptoNets model focussed on achieving high throughput, and MN-*fast* which uses the low-latency algorithms discussed in 3.3.2 to reduce inference latency and message size.

#### High throughput inference using pruning

I implement and train a modified version of the CryptoNets architecture using Tensorflow. The original architecture, shown in Figure 3.11a, contains two convolutional pooling blocks followed by two dense layers. I adjust this network by removing the second convolutional-pooling block. In the original CryptoNets paper, this block is composed together with

the subsequent dense layer using the approach described in 3.3.1 so that there is effectively only one layer present during inference. The main reason for this modification is that pruning techniques are less effective for reducing inference latency if layers are later composed. The composition of several layers, although each having a high proportion of pruned weights, will not necessarily result in a composed layer of the same sparsity. Note that the final softmax activation is omitted from the inference pipeline, as softmax

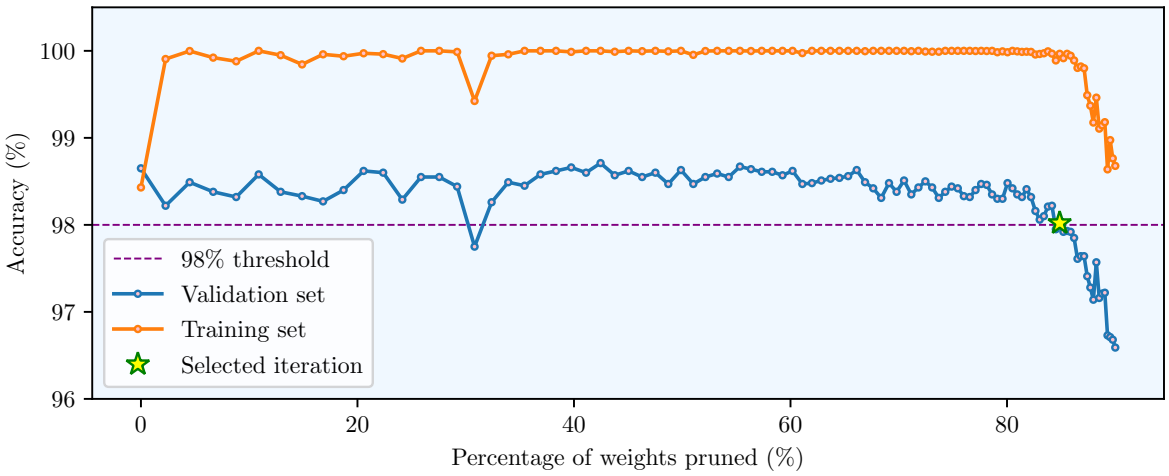


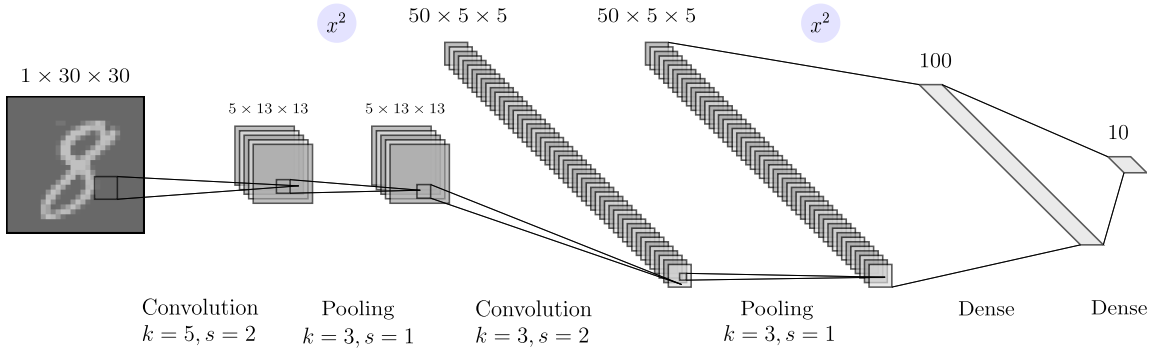

**Figure 3.10:** Applying iterative lottery pruning on the modified CryptoNets architecture.

preserves the input size-order which allows us to extract the top prediction by comparing the sizes of the decrypted output.

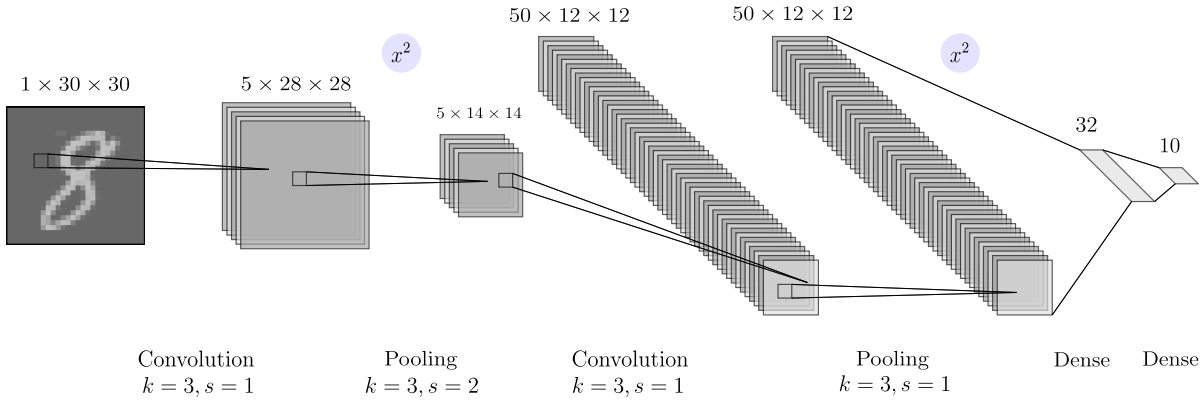
I apply the lottery pruning method 3.4.1 to this architecture, producing a model MN-*prune*. An iterative process is used with 100 pruning iterations such that $(1-\sqrt[100]{0.1})\cdot 100\%$ of the weights are pruned each iteration. The Python `lottery-ticket-pruner` package [32] is used to create masks and automatically prune after every training iteration. I employ an early stopping criterion in which training is ended once the validation loss is not seen to decrease after 30 epochs. A large drop in accuracy is observed after 85% of the weights are pruned, as shown in Figure 3.10. I select the iteration with the highest percentage of pruned weights such that the validation accuracy is above 98%, which gives a test accuracy of 97.9% and 84.9% of weights pruned.

### Low latency MNIST

I now present an optimised architecture MN-*fast* targeted for my implementation of the low-latency network operations discussed in 3.3.2. I reduce the kernel size from $5\times5$ to $3\times3$, as the convolution packing method I implemented requires a number of multiplications proportional to $k^2$, the size of the kernel. Furthermore, smaller stride does not increase the number of operations required for convolution packing, given that there are enough ciphertext slots. Therefore, the stride is decreased to $s = 1$ for the first convolution, allowing greater kernel overlap without increasing the computation cost. This reduces the downsampling effect of the convolution, allowing a higher-resolution set of features to be extracted. Since dense layers use rotations proportional to the number of output nodes, I change the final dense layer to have only 32 outputs. 98.98% validation accuracy is achieved using this architecture. The first convolution layer is implemented using the convolution packing method and all matrix-vector products are implemented using the




**(a)** CryptoNets.




**(b)** MN-*fast*.

**Figure 3.11:** Comparison of MNIST architectures. $k$ and $s$ indicate the kernel width and stride respectively. Note that the input is padded.

diagonal matrix multiplication method. The updated architecture is shown in Figure 3.11*b*.

### Encryption parameters

Both MN-*prune* and MN-*fast* consume five levels of multiplication. I therefore choose the polynomial modulus degree to be 8192, allowing for a maximum coefficient modulus size of 218 bits to attain 128-bit security. The chosen bit-lengths for each prime in the modulus chain are given in Appendix A.3. The base modulus and scaling factor lengths are chosen as 32 and 25 bits, respectively, since I observed that virtually all outputs of the final layer lie within the range $[-2^7, 2^7]$.

## 3.5.2 CIFAR-10

Training on CIFAR-10 requires substantially more network parameters than on MNIST, due to the input being almost four times larger. The SIMD packing scheme becomes almost impractical here, as vastly more ciphertexts are required, meaning that the memory required for intermediate layers will be large. Thus, I focus on producing a low-latency model using the algorithms implemented in 3.3.2.

I first train a network CF-*base* shown in Figure 3.12, achieving 82% training and 72.2% test accuracy after 300 epochs, with dropout applied before the final dense layer to prevent overfitting. This architecture has three convolutional layers, the first of which is computed using the convolutional packing method. The second convolution is combined with the

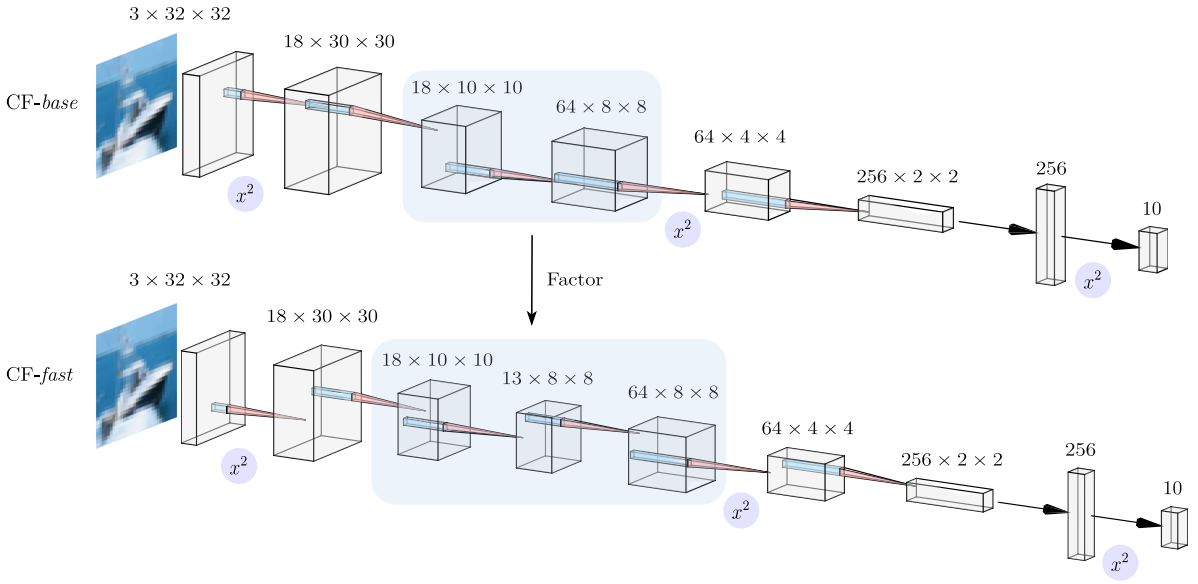



**Figure 3.12:** Obtaining the CF-*fast* architecture from CF-*base*.

subsequent pooling layer to form a linear block. The SVD method from 3.4.2 is applied to split the second convolutional layer into two smaller convolutions, the first of which produces 13 feature maps instead of 18, which reduces the rotation count and enables the dimensions required by the diagonal method to fit within 8192 slots. I distribute the first smaller convolution over three ciphertext, using an intuitive splitting method shown in Figure 3.13. The resulting model is denoted CF-*fast*. To mitigate the loss in model accuracy caused by factoring the convolutional layer, I retrain CF-*fast* using the weight values obtained from SVD to initialise the second convolution, resulting in a final test accuracy of 73.1%.

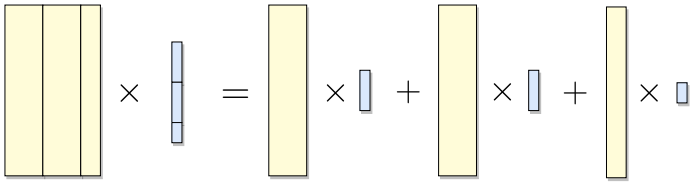


**Figure 3.13:** Matrix-vector multiplication splitting.

### Encryption parameters

CF-*fast* consumes eight multiplication levels. A polynomial modulus degree of $2^{13}$ is insufficient because it enforces a maximum coefficient modulus size of 218 bits, which highly limits the coefficient modulus bits allocated to each level of the modulus chain. Hence, a degree of $2^{14}$ is chosen. Moreover, if the SVD method were not applied, the architecture would be severely limited unless the much slower $2^{15}$ is chosen as the degree. 30 bits are given to each intermediate level of the modulus chain, because CIFAR-10 is more sensitive to precision than grayscale MNIST.

### 3.5.3 Encrypted facial recognition

I demonstrate the capabilities of encrypted inference by solving the task of encrypted facial recognition: matching an encrypted face image against a gallery of faces, without revealing the faces to the server. I first solve 1:1 recognition – the binary classification problem of determining whether two faces have the same identity, and then extend this notion to the context of matching.

Highly accurate face-recognition networks such as FaceNet typically have more than 10 layers and prove to be unsuitable for HE. I propose the use of transfer learning in the context of encrypted facial recognition to circumvent this: use a pretrained network as a feature extractor, and train a triplet-loss network receiving the extracted features as input. The overall architecture is that of a Siamese network, shown in Figure 3.14, consisting of a non-encrypted *encoder* half intended to be user-computed, and a secure *discriminator* half intended to be computed on the server for the more time-consuming matching procedure. The dataset used is CelebA dataset which contains 202,599 images and 10,177 identities. The first 14 layers of a pretrained VGG-16 network are used as a

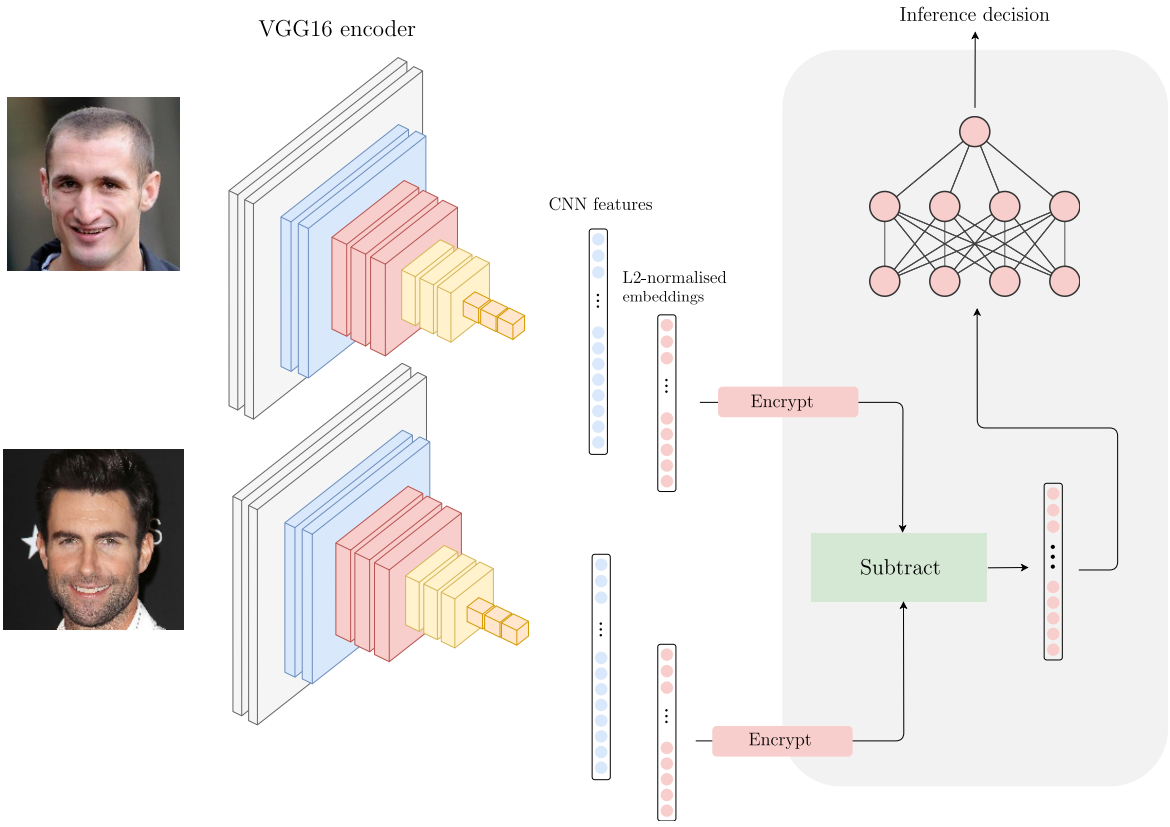



**Figure 3.14:** The CryptoFace model.

feature extractor to produce 512-length feature vectors, which are fed into a single dense layer to produce an embedding vector of length 64. This layer is trained using the triplet loss function

$$\mathcal{L}(A, P, N) = \max\left(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0\right), \tag{3.9}$$

introduced by Schroff et al [27]. The triplet $(A, P, N)$ is always chosen such that $A$ (the anchor) and $P$ (the positive example) have the same class while $A$ and $N$ (the negative example) have opposite classes. $\|\cdot\|$ denotes the Euclidean distance and $\alpha$ is a margin

parameter which I set to be 0.5. The convergence of the triplet loss encourages pairs from the same class to have similar embeddings, and pairs from opposite classes to have distant embeddings. I use an implementation of semihard online learning provided by Tensorflow to select suitable triplets. The image embeddings are encrypted, subtracted, and squared, before being inputted into two dense layers with output sizes of 64 and 1, respectively. The use of dense embeddings couples well with the SIMD packing approach, the high message size of which is mitigated by the compactness of the embeddings. In total, only 128 ciphertexts are required to perform a comparison between two face embeddings. I propose a method of extending this to perform a match against $N$ face embeddings by a simple yet effective use of the SIMD packing method – the key idea is to pack *all* the gallery faces into merely 64 ciphertexts such that the $i$-th slot of the $j$-th ciphertext contains the $j$-th embedding value of the $i$-th face. The encrypted portion of the network is computed relatively cheaply with fewer than 10,000 operations. For the encryption parameters, a polynomial modulus degree of 8192 is used, which yields 4096 comparisons per inference. I discuss the results and effectiveness of this approach in Section 4.3.3.

# Chapter 4

# Evaluation

In this chapter, I evaluate the implementation in terms of three main evaluation criteria: the extent to which it meets the requirements analysis, its performance relative to other works in the field, and its practicality in being applied to real-world problems. This chapter is divided into three sections for each evaluation criterion, respectively. The section on performance is the most quantitative and presents a detailed comparison of my models with those from the literature, and an analysis on the aspects in which my solutions improve over previous authors' works. The section on practicality focuses on qualitatively evaluating and comprehending the scope to which the implementation can be applied to real-world machine learning problems.

## 4.1 Requirements analysis

| Requirement | Achieved? | Means of completion |
|---|---|---|
| A.1 | ✓ | I provide an interface for building secure deep learning models. Our software engineering approach is discussed in 3.1.2. Examples of usage can be found in the notebook `inference.py` in the repository. |
| A.2 | ✓ | I implement the SIMD packing operations from [3], as detailed in 3.3.1. |
| A.3 | ✓ | I construct privacy-preserving models for MNIST and CIFAR-10, as detailed in 3.5. |
| B.1 | ✓ | In 3.3.2, I discuss my implementation low-latency operations from [24] and [8]. |
| B.2 | ✓ | I utilise compression techniques to further optimise the MNIST and CIFAR-10 models, as detailed in 3.4, using both lottery pruning and SV decomposition. |
| B.3 | ✓ | I demonstrate a practical privacy-preserving face recognition model in 3.5.3. |
| B.4 | ✓ | I provide an implementation of the original CKKS scheme, which can be found in the repository under `crypto/ckks.py`. |

**Table 4.1:** Project requirements analysis.

In section 2.2.1 I identified a set of base success criteria as well as extensions to the project. All base requirements have been implemented, and all four extensions have been fully completed. The results of applying the low-latency operations on MNIST and CIFAR-10 datasets and applying the SIMD packing method to the facial recognition problem have been especially promising and suggest that $\Psi$-lib contains methods sophisticated enough to attain performance similar to the state-of-the-art. As such, I believe that the success criteria have been met.

## 4.2 Comparison of performance with existing work

In this work, I implement several techniques to build a total of four models targeting both the cases of low inference latency and high inference throughput. I provide three models focusing on low latency inference: MN-*fast*, which targets a relatively simple dataset, CF-*fast* which targets a more challenging dataset, and finally CryptoFace, which solves the problem of encrypted facial recognition. I present a model MN-*prune* to achieve high throughput. All models are evaluated in terms of their latency, throughput, message size and inference accuracy. The results show that the use of efficient rotation-based algorithms implemented in $\Psi$-lib can significantly improve the practicality of inference, and that the application of model compression techniques can make both high-inference and high-throughput models more feasible. I shall use comparisons of performance with existing works as a primary basis for evaluation. In Table 4.2 I provide a detailed comparison of my secure MNIST models with those from the literature, and in Appendix B.4 I provide an equivalent table for my secure CIFAR model.

| **Metric** | **CrypNts** | **CrypDL2**‡ | **FCrypNts**‡ | **MN-p** | **LoLa**‡‡ | **FFConv**‡‡ | **MN-f** |
|---|---|---|---|---|---|---|---|
| Mul CC† | **945** | 64512 | **945** | **945** | 2 | **1** | 2 |
| Mul PC | 106625 | $4.62 \times 10^7$ | 23952 | **18382** | 136 | 1544 | **87** |
| Add CC | 104715 | $4.61 \times 10^7$ | 38302 | **16472** | **84** | 1650 | 94 |
| Add PC | **847** | 161546 | 3995 | **847** | N/A | N/A | **7** |
| Rot | - | - | - | - | 94 | 183 | **56** |
| Total | 213132 | $9.27 \times 10^7$ | 67194 | **36646** | 331 | 3378 | **246** |
| Enc+Dec lat. (s) | 47.5 | 16.7 | **6.7** | 8.1 | N/A | N/A | **2.4** |
| Inference lat. (s) | 249.6 | 320.0 | **39.1** | 92.8 | 2.1 | **0.37** | 0.98 |
| Test accuracy | 98.95% | **99.52%** | 98.71% | 97.9% | 98.95% | 98.40% | **98.98%** |
| Msg size (MB) | 367.5 | **336.7** | 411.1 | 719.3 | N/A | N/A | **9.44** |
| Scheme | YASHE | BGV | FV | CKKS | BFV | BFV | CKKS |

**Table 4.2: Comparison of MNIST models with previous works**. † - CC indicates a ciphertext-ciphertext operation and PC indicates a plaintext-ciphertext operation. ‡ - the details of these models are borrowed from [11]. ‡‡ - the details of these models are calculated using the results provided in [24],[7]. N/A indicates insufficient data to deduce from.

### 4.2.1 Latency and throughput

For each model, I perform an architectural analysis, calculating the number and type of operations computed at each layer. I also perform a timing analysis for each model, measuring the time that each component requires. These are both done using a logging and debugging system integrated into $\Psi$-lib for the purpose of model evaluation. All benchmarks are run on an Intel Skylake i7-8700k processor with a clock frequency of 4.7GHz. Precise details of our experimental setup for all models is given in B.3. Since previous works were evaluated on various setups, I use the number of homomorphic operations (HOPs) as a more meaningful measure of latency. Note that rotations are by far the most expensive operation to perform, with the worst case time complexity being equivalent to performing both an NTT and an inverse NTT transform on the ciphertext [8]. In Figure 4.1 I provide a visualisation of the breakdown of the models.

As shown in 4.2, my MN-*fast* model completes a single inference in 246 HOPs and 0.98 seconds, which is significantly less than previous approaches that do not use rotations.

MN-*fast* requires fewer than 0.0012% as many HOPs as CryptoNets[1] and has an inference time more than 200 times faster. In addition, MN-*fast* achieves a lower inference latency than rotation-based methods such as LoLa whilst maintaining a high prediction accuracy. Fewer rotations are required due to the use of the diagonal matrix-vector method, which was not implemented in either LoLa [24] or FFConv [7]. LoLa uses an efficient stacked representation for their first fully-connected layer, which results in a permuted set of inputs for the final layer. I did not implement this, although it would reduce the rotation count of MN-*fast* even further.

Scaling up to the CIFAR-10 dataset suggests that even greater speedup is attained with the use of rotation-based methods I implemented. CF-*fast* requires 11,134 HOPs, which is far less than the 92,000,000 HOPs used by the CNN-8 model from [11] whilst achieving higher accuracy. The SIMD packing scheme scales poorly with increased dataset complexity, since the HOP count for a convolution grows super-linearly when adding output filters. However, low latency packing methods are able to reduce this growth closer to a linear one as long as the computation fits within a small number of ciphertexts.

Overall, I conclude that the techniques implemented in $\Psi$-lib are sufficient for creating models that perform similarly to the state-of-the-art for secure models with regard to the number of HOPs that a model inference consumes.

### Effectiveness of model compression

My results show that application lottery pruning and SV decomposition are effective for increasing throughput and reducing prediction latency, respectively. The MN-*prune* model, which has 15.1% of the weights of the original CryptoNets model, has only 18% the multiplication count and 16.4% the addition count of the original model, as shown in Table 4.3. The reduction in the number of operations, and by extension the inference time, is approximately the same as the pruning percentage.

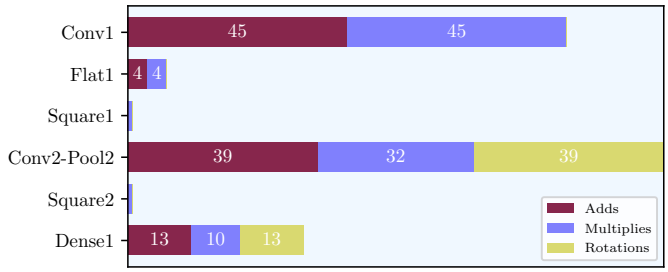
The SV decomposition method is also effective in reducing the rotation count required in the computation of large convolutional blocks, without sacrificing significant amounts of accuracy. The application of SVD for CF-*fast* is crucial in enabling the output activations of the second convolution to fit within the ciphertext slot count without increasing the polynomial degree. One could draw a parallel to the importance of fitting a working set of computation within on-chip memory, which imposes a performance 'cliff' after which the performance will deteriorate dramatically.

| | *MN-prune* | | | *CF-fast* | | |
|---|---|---|---|---|---|---|
| | Original | Compressed | Reduction | Original | Compressed | Reduction |
| Multiplications | 107570 | 19327 | 5.6× | 21239 | 4085 | 5.2× |
| Additions | 105562 | 17319 | 6.1× | 21260 | 4177 | 5.1× |
| Rotations | - | - | - | 20774 | 2872 | 7.2× |
| Test accuracy | 98.5 | 97.9 | - | 71.5 | 73.06 | - |
| Inference time | 447 s | 92.4 s | 4.8× | 2310 s | 273 s | 8.5× |

**Table 4.3:** Effect of compression techniques applied.

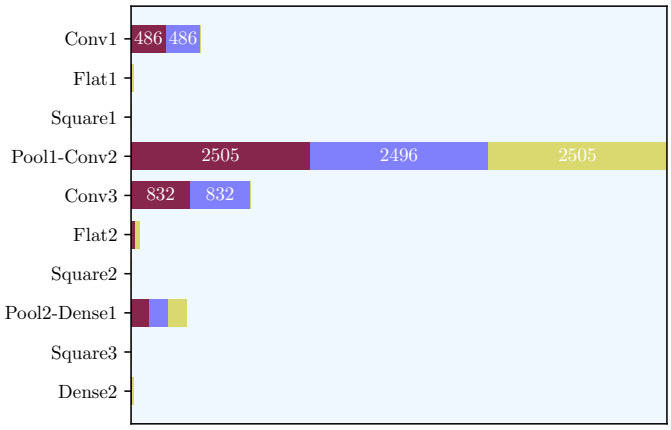
[1] [11] gave a similar breakdown in their paper, but they did not consider the reduction in HOPs due to linear layer composition.

| Layer | HOPs | Add PC | Add CC | Mul PC | Mul CC | Rot | Time |
|---|---|---|---|---|---|---|---|
| Conv1 | 90 | 5 | 40 | 45 | - | - | 0.28 |
| Flat1 | 8 | - | 4 | - | - | 4 | 0.076 |
| Square1 | 1 | - | - | - | 1 | - | 0.009 |
| Conv2-Dense1 | 110 | 1 | 38 | 32 | - | 39 | 0.529 |
| Square2 | 1 | - | - | - | 1 | - | 0.005 |
| Dense2 | 36 | 1 | 12 | 10 | - | 13 | 0.008 |
| Total | 246 | 7 | 94 | 87 | 2 | 56 | 0.983 |



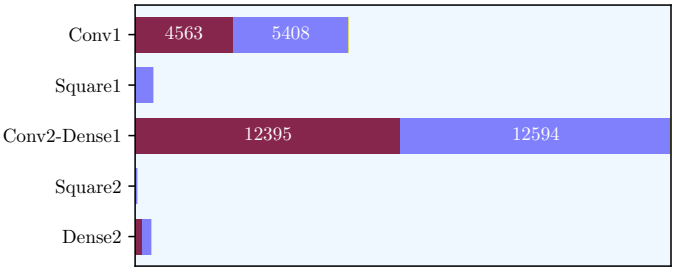

**(a)** MN-*fast* timing analysis.

| Layer | HOPs | Add PC | Add CC | Mul PC | Mul CC | Rot | Time |
|---|---|---|---|---|---|---|---|
| Conv1 | 972 | 18 | 468 | 486 | - | - | 7.95 |
| Flat1 | 30 | - | 15 | - | - | 15 | 1.57 |
| Square1 | 3 | - | - | - | 3 | - | 0.12 |
| Pool1-Conv2 | 7506 | 1 | 2504 | 2496 | - | 2505 | 240 |
| Conv3 | 1677 | 64 | 768 | 832 | - | 13 | 10.6 |
| Flat2 | 126 | - | 63 | - | - | 63 | 2.04 |
| Square2 | 1 | - | - | - | 1 | - | 0.02 |
| Pool2-Dense1 | 778 | 1 | 260 | 256 | - | 261 | 10.48 |
| Square | 1 | - | - | - | 1 | - | 0.01 |
| Dense2 | 40 | 1 | 14 | 10 | - | 15 | 0.18 |
| Total | 11134 | 85 | 4092 | 4080 | 5 | 2872 | 272.9 |




**(b)** CF-*fast* timing analysis (after SVD).

| Layer | HOPs | Add PC | Add CC | Mul PC | Mul CC | Rot | Time |
|---|---|---|---|---|---|---|---|
| Conv1 | 9971 | 845 | 3718 | 5408 | - | - | 35.78 |
| Square1 | 845 | - | - | - | 845 | - | 7.39 |
| Conv2-Dense1 | 24,989 | 1 | 12K | 13K | - | - | 48.06 |
| Square2 | 100 | - | - | - | 100 | - | 0.49 |
| Dense2 | 741 | 1 | 360 | 380 | - | - | 1.09 |
| Total | 36646 | 847 | 16K | 18K | 945 | - | 92.8 |




**(c)** MN-*prune* timing analysis (after lottery pruning).

**Figure 4.1:** Break-down of low latency model operations.

### 4.2.2 Accuracy

I evaluate the accuracy of my MNIST models by performing an inference on every example in the test set in both the encrypted and non-encrypted domains. We define accuracy as the true positive count, added to the true negative count, divided by the total inference count. Both MNIST models achieve close to 99% test accuracy using the trained model on plaintext, and suffer no prediction changes when evaluated on ciphertext with my choice of encryption parameters. This is on par with other works on encrypted MNIST.

For my CIFAR-10 model, secure predictions are evaluated on a randomly chosen subset of 200 images within the test set, such that the class frequencies are equally distributed. This is due to the long time required in evaluating the entire test set of 10000 images. The secure predictions match the nonsecure ones exactly. On the non-encrypted test set, the model achieves 73.1% accuracy which is slightly lower than the 76.5% reported by FFConv.

### 4.2.3 Message size

A practical issue is the size of the message that the user has to send to the server. In previous works using SIMD packing, this message size was especially large – CryptoNets has a message size of 367.5 MB due to the need of sending a separate ciphertext for each value in the input. For larger input sizes, this becomes infeasible if the sender does not have enough data to make full use of the ciphertext slots. My low latency models inherently reduce this message size by packing different values of the same input inside the same ciphertext. For MN-*fast*, this message size is reduced to nine CKKS ciphertexts. Since we use an RNS version of the scheme, each ciphertext is represented using 8 pairs of polynomials of degree 8192. Hence the total message size is $9 \times 8192 \times 16 \times 8\text{B} \approx 9.44$ MB.

## 4.3 Practicality

### 4.3.1 Supported architectures

My implementation on a fundamental level supports all architectures with operations that can be represented as a fully connected layer or a squaring of the same layer. A key limitation, however, is the inability to compute other nonlinear operations such as L2-normalisation and activation functions such as ReLU and Swish. This is a limitation in RLWE-based schemes. The inability to perform ReLU is perhaps a limiting factor for the accuracy of models that are supported. In this section, I investigate the extent to which being limited to the square activation function affects the convergence of models, and discuss possible remedies.

#### Model convergence

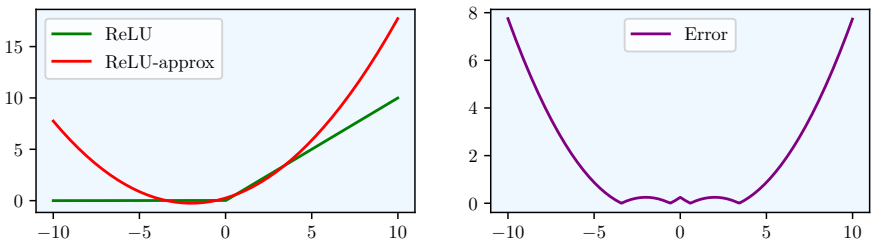



**Figure 4.2:** The ReLU activation and its approximation.

During the implementation of my models, I found that convergence is more unstable when using the square activation compared to ReLU. This is likely due to the problem of exploding gradients. The derivative of the square function is $2x$, implying that the gradient of the loss function increases exponentially with the number of activations. Large gradient magnitudes cause gradient descent to make coarser steps during training, leading to poorer convergence. This issue has been extensively studied. Existing techniques include the use of *gradient clipping*, which prevents gradients from exceeding a threshold [28]. I investigate an alternative solution of using a quadratic approximation to ReLU, namely $f(x) = 8^{-1}x^2 + 4^{-1}x + 2^{-1}$. The use of this approximation was introduced by Chou et al [11], who suggest the use of batch normalisation to localise the activation input

within the region where the approximation holds well. During training, for a mini-batch size $B$, let $\mathbf{x}_1, \ldots, \mathbf{x}_B$ be the contents of the batch, where $\mathbf{x}_i = \left\{x_i^{(1)}, \ldots, x_i^{(k)}\right\}$. Then for each $x_i^{(k)}$, compute its normalised value,

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mathbb{E}\left(x_i^{(k)}\right)}{\sqrt{\text{Var}\left(x_i^{(k)}\right)}}, \tag{4.1}$$

where $\mathbb{E}\left(x_i^{(k)}\right)$ and $\text{Var}\left(x_i^{(k)}\right)$ are computed using a moving average over mini-batches. During inference, the normalisation is performed using the moving average mean and variance computed over all mini-batches in training. This constitutes a linear operation which consumes one multiplication level. I perform a comparison of the square activation with

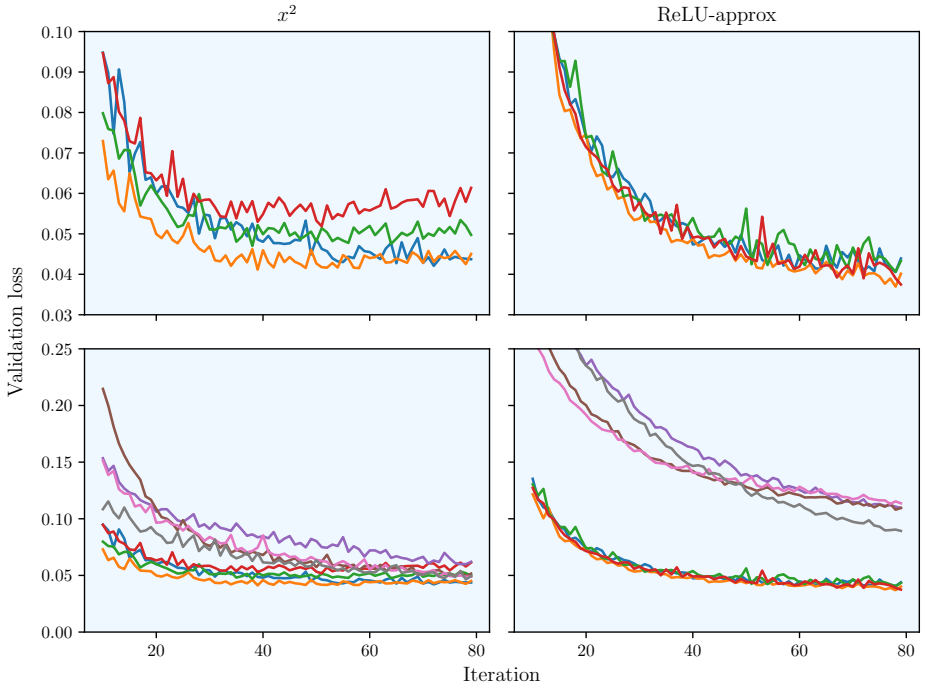



**Figure 4.3:** Comparison of convergence using the two activation functions. In the top row, only the validation loss for the batch-normalised models are used. In the bottom row, the original model without batch-normalisation is also shown.

ReLU-approx, as shown in Figure 4.3. I train 4 models using the MN-*fast* architecture as a baseline, on four variants of the model, two of which use $x^2$ for all activations and the other two of which use ReLU-approx. For each activation I train one variant using batch normalisation and another without. Plotting the convergence of the validation loss function suggests that ReLU-approx converges more stably, but only finds good minima consistently under the use of batch normalisation.

This suggests that using a quadratic approximation to ReLU can aid convergence and training stability. Furthermore, I find that the use of ReLU-approx with batch normalisation produces similar test accuracy compared to ReLU on both the MN-*fast* and CF-*fast* models.

### 4.3.2 Security and homomorphic depth

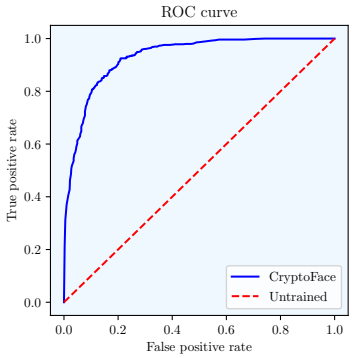
All my models provide 128-bit IND-CPA security, in accordance with the HE security standard [26]. Using the CKKS scheme, I can confidently achieve up to ten levels of multiplication using a polynomial modulus degree of $N = 2^{14}$ and a scaling factor of $\log_2 \Delta = 30$. All demonstrated models use no more than eight levels. Increasing the modulus degree to $N = 2^{15}$ yields a maximum of $\log_2 Q = 881$ which provides more than twenty-five levels for this scaling factor. However, current hardware is much slower when using $N = 2^{15}$ – a single rotation takes as much as 0.08 seconds on my setup. Hence, the maximum practical depth supported by $\Psi$-lib using CKKS is around 8 to 10, depending on the scaling factor used. This is feasible for benchmark datasets, and application to harder datasets can benefit from using pretrained networks to compress the input, as I have demonstrated in the CryptoFace model.

### 4.3.3 Encrypted facial recognition

In this section, I evaluate the feasibility of CryptoFace, which I proposed in Section 3.5.3. The performance of CryptoFace in comparison to an untrained model is shown in 4.4 in the form of a receiver operating characteristic (ROC) curve. I calculate the true positive rate TPR and false positive rate FPR as

$$\text{TPR} = \frac{\text{True positives}}{\text{True positives} + \text{False negatives}}, \qquad \text{FPR} = \frac{\text{False positives}}{\text{True negatives} + \text{False positives}}. \tag{4.2}$$

Different pairs $(\text{TPR}, \text{FPR})$ are obtained whilst varying the classification threshold from 0 to 1, producing the ROC curve. CryptoFace has three main advantages: firstly, assuming




| | |
|---|---|
| Enc+dec time | 0.73 s |
| Inference time | 19.78 s |
| Time per comparison | 4.83 ms |
| HOPs | 8384 |
| Message size | 50.33 MB |

**Figure 4.4:** The ROC curve of the CryptoFace model.

the maximum of 4096 gallery faces is used, the time each comparison takes is only 4.8 ms on average when the embeddings are available. The time taken to produce and encrypt the embeddings is amortised by the SIMD packing method if the same gallery is used for a large number of comparisons. Secondly, my solution is flexible in allowing the method of obtaining the embeddings to be switched without affecting the performance of the secure part of the computation – this is due to the separation between the encoder and discriminator parts of the architecture. Thirdly, the solution allows fine-tuning of the faces to be compared with the gallery, by adjusting the packing accordingly. For example, comparing one face against a gallery of 4096 requires the same number of operations as comparing 16 faces against a gallery of 256 faces.

## 4.4 Summary

This evaluation gives evidence that the techniques implemented in $\Psi$-lib, together with the application of model pruning, are able to achieve promising results comparable to previous works in terms of model latency, accuracy, and security. It also reinforces the feasibility of a privacy-preserving facial recognition network that utilises dense embeddings. Overall, I believe that with further improvements in the underlying hardware, such techniques will become increasingly practical for real-life deployment.

# Chapter 5

# Conclusions

## 5.1 Summary of project

The project was an overall success, having met the original success criteria and extensions. I implement several techniques for performing encrypted neural network inference using HE and apply those techniques to construct practical models for private inference on the MNIST and CIFAR-10 datasets. As an extension, I apply these concepts to create a model for encrypted facial recognition, achieving both high accuracy and throughput. I implement algorithms for fast inference using ciphertext rotations, and investigate the effect of methods such as pruning and compression for making models faster and more practical. Furthermore, I discuss the tradeoffs associated with using HE for deep neural networks, including the effect of activation function on model convergence, and the balance between security and computation depth.

## 5.2 Lessons learnt

Throughout the course of this project, many challenges were faced. A critical challenge was cultivating the background knowledge in both deep neural networks and cryptography, upon which our work is built. Neither of these areas are covered in detail before Part II. The sparsely documented nature of the techniques and practices used in the area of encrypted machine learning also warranted several days of writing literature reviews and deciphering research papers. Furthermore, many of the computations took a long time to run, which made debugging a time-consuming process. These challenges encouraged me to allocate my time strategically and overall made the project a very productive learning experience, from both the the perspective of exploring an emerging area of computer science and that of becoming better at managing long-term projects.

## 5.3 Future directions

In this dissertation, I focus on the application of public-key HE schemes. Other works have shown that secure multiparty computation between two parties can be used to aid performance, which would be an interesting extension to explore.

An interesting yet relatively unexplored area is the application of HE to deep polynomial networks, also known as Π-nets, first proposed by Chrysos et al [29]. These are a class of networks where the output is a high-order polynomial of the input, and can produce state-of-the-art results in image classification tasks using only Hadamard products and addition. An investigation into the practicality of Π-nets for secure inference would be an interesting path to take.

Finally, the main limitation of the current line of work in HE inference is that of computation depth. One can argue that this is perhaps as much of a hardware issue as it is a software one – deep neural networks have benefited hugely from the growth of GPUs, and one could hope that the practicality of HE will similarly benefit from the growth of dedicated accelerators.

# Bibliography


[1] T Hunt, C Song, R Shokri, V Shmatikov, and E Witchel. *Chiron: Privacy-preserving Machine Learning as a Service*. arXiv:1803.05961, 2018.

[2] F Mireshghallah, M Taram, P Vepakomma, A Singh, R Raskar, and H Esmaeilzadeh. *Privacy in Deep Learning: A Survey*. arXiv:2004.12254, 2020.

[3] N Dowlin, R Gilad-Bachrach, K Lauter, M Naehrig, and J Wernsing. *CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy*. ICML, 2016.

[4] J H Cheon, A Kim, M Kim, and Y Song. *Homomorphic Encryption for Arithmetic of Approximate Numbers*. Cryptology ePrint Archive: Report 2016/421, 2016.

[5] K Laine. *Simple Encrypted Arithmetic Library 2.3.1*. Microsoft Research TechReport, 2017.

[6] J Frankle and M Carbin. *The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks* ICLR, 2019.

[7] Y Lu, J Lin, C Jin, Z Wang, K M M Aung, X Li. *FFConv: Fast Factorized Neural Network Inference on Encrypted Data*. arXiv:2102.03494, 2021.

[8] C Juvekar, V Vaikuntanathan, and A Chandrakasan. *GAZELLE: A Low Latency Framework for Secure Neural Network Inference*. Proceedings of the 27th USENIX Conference on Security Symposium, 2018.

[9] T Graepel, K Lauter, and M Naehrig. *ML confidential: machine learning on encrypted data*. International Conference on Information Security and Cryptology, 2012.

[10] A A Badawi, C Jin, J Lin, C F Mun, S J Jie, B H M Tan, X Nan, and K M M Aung. *Towards the AlexNet Moment for Homomorphic Encryption: HCNN, the First Homomorphic CNN on Encrypted Data with GPUs*. arXiv:1811.00778, 2020.

[11] E Chou, J Beal, D Levy, S Yeung, A Haque, and F Li. *Faster CryptoNets: Leveraging Sparsity for Real-World Encrypted Inference*. arXiv:1811.09953, 2018.

[12] S Li, K Xue, C Ding, X Gao, D S L Wei, T Wan, and F Wu. *FALCON: A Fourier Transform Based Approach for Fast and Secure Convolutional Neural Network Predictions*. arXiv:1811.08257, 2018.

[13] F Boemer, A Costache, R Cammarota, and C Wierzynski. *nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data* Workshop on Encrypted Computing and Applied Homomorphic Cryptography, 2019.

[14] M S Riazi, K Laine, B Pelton, and W Dai. *HEAX: An Architecture for Computing on Encrypted Data*. Architectural Support for Programming Languages and Operating Systems, 2020.

[15] H Bae, J Jang, D Jung, H Jang, H Ha, H Lee, and S Yoon. *Security and Privacy Issues in Deep Learning*. arXiv:1807.11655, 2018.

[16] V Lyubashevsky, C Peikert, O Regev. *On Ideal Lattices and Learning with Errors Over Rings*. Journal of the ACM, 2013.

[17] S Kim, W Jung, J Park, J H Ahn *Accelerating Number Theoretic Transformations for Bootstrappable Homomorphic Encryption on GPUs* IISWC, 2020.

[18] S Halev, Y Polyakov, and V Shoup. *An Improved RNS Variant of the BFV Homomorphic Encryption Scheme*. Cryptology ePrint Archive: Report 2018/117, 2018.

[19] `https://github.com/Huelse/SEAL-Python`. Retrieved December 2020.

[20] M Abadi, P Barham, J Chen, Z Chen, A Davis, J Dean, M Devin, S Ghemawat, G Irving, M Isard, M Kudlur, J Levenberg, R Monga, S Moore, D G Murray, B Steiner, P Tucker, V Vasudevan, P Warden, M Wicke, Y Yu, and X Zheng, Google Brain *TensorFlow: A System for Large-Scale Machine Learning*, 2015. http://tensorflow.org/. Available from tensorflow.org.

[21] Z Liu, P Luo, X, Wang, and X Tang. *Deep Learning Face Attributes in the Wild*. ICCV, 2015.

[22] J H Cheon, K Han, A Kim, M Kim, and Y Song. *A Full RNS Variant ofApproximate Homomorphic Encryption* Cryptology ePrint Archive: Report 2018/931, 2018.

[23] `https://pypi.org/project/lottery-ticket-pruner` Retrieved March 2021.

[24] A Brutzkus, O Elisha, R Gilad-Bachrach. *Low Latency Privacy Preserving Inference*. ICML, 2019.

[25] S Halevi and V Shoup. *Algorithms in HElib*. CRYPTO, 2014.

[26] `https://homomorphicencryption.org/standard/`. Retrieved March 2021.

[27] F Schroff, D Kalenichenko, and J Philbin. *FaceNet: A Unified Embedding for Face Recognition and Clustering*. CVPR, 2015.

[28] R Pascanu, T Mikolov, and Y Bengio. *On the difficulty of training Recurrent Neural Networks*. ICML, 2013.

[29] G Chrysos, S Moschoglou, G Bouritsas, J Deng, Y Panagakis, and S Zafeiriou. *Deep Polynomial Neural Networks*. arXiv:2006.13026, 2020.

[30] J H Cheon, Y Son, and D Yhee. *Practical FHE parameters against lattice attacks* Cryptology ePrint Archive: Report 2021/039, 2021.

[31] `https://www.dpmms.cam.ac.uk/study/II/Galois/`. Retrieved May 2021.

[32] `https://pypi.org/project/lottery-ticket-pruner/`. Retrieved May 2021.

[33] `https://www.cl.cam.ac.uk/teaching/2021/Crypto/crypto-slides.pdf`. Retrieved May 2021.

# Appendix A

# Supplementary materials for cryptography

## A.1 A brief detour into abstract algebra

Much of modern cryptography is founded upon results from pure mathematics. In particular, the notions of groups and rings are useful to understand as a starting step. This appendix section aims to clarify these concepts in case the reader is new to the field of cryptography. Much of the notation is borrowed from [33].

**Definition 1.** *A* group *is a set $\mathbb{G}$ that has an operator $\bullet$. The operator $\bullet$ must have two operands and one output. Examples of a group operator are $+$, $\times$, and XOR. In addition, for $G$ to be the group, it must satisfy four properties:*

1. *Closure: for any $a, b \in \mathbb{G}$, we have that $a \bullet b \in \mathbb{G}$.*
2. *Associativity: for any $a, b, c \in \mathbb{G}$, it must hold that $a \bullet (b \bullet c) = (a \bullet b) \bullet c$.*
3. *Identity: there exists a unique identity element $e \in \mathbb{G}$ such that $e \bullet a = a \bullet e = a$ for all $a \in \mathbb{G}$.*
4. *Invertibility: for any $a \in \mathbb{G}$, there exists an element $b \in \mathbb{G}$ such that $a \bullet b = b \bullet a = e$.*

The *order* or a group $\mathbb{G}$ is $|\mathbb{G}|$, the number of elements it contains. Furthermore, a group $\mathbb{G}$ is an *abelian group* if for all $a, b \in \mathbb{G}$, it holds that $a \bullet b = b \bullet a$. The set of integers $\mathbb{Z}$ is an abelian group under the operator of addition. If $\mathbb{G}$ does not satisfy the invertibility property, then it is a *monoid* rather than a group. Commonly mentioned groups include:

- $\mathbb{Z}_n = \mathbb{Z}/n\mathbb{Z}$ is the set of all integers modulo $n$, which is also an abelian group under addition.
- $\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid \gcd(a, n) = 1\}$, is the set of all integers in $\{1, \ldots, n-1\}$ that are also relatively prime to $n$. This set forms an abelian group under multiplication. Notably, the order of this group is given by $\varphi(n)$, the Euler totient function of $n$.

**Definition 2.** *A ring $\mathcal{R}$ is a set with two operators (instead of one for a group) $+ : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$ and $\cdot : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$ satisfying the following properties:*

1. *$\mathcal{R}$ is an abelian group under $+$.*
2. *$\mathcal{R}$ is a monoid under $\cdot$.*
3. *For all $a, b, c \in \mathcal{R}$ it holds that $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and $(a \cdot b) + c = (a \cdot c) + (b \cdot c)$.*

Notable rings:

- $\mathbb{Z}[X]$ is the set of polynomials with coefficients in $\mathbb{Z}$, i.e.

$$\mathbb{Z}[X] = \left\{ a_N x^N + a_{N-1} x^{N-1} + \ldots + a_0 \mid a_N, \ldots, a_0 \in \mathbb{Z}, N \geq 0 \right\}.$$

  $\mathbb{Z}[X]$ with the operations of polynomial addition and multiplication, is a ring.

- $\mathbb{Z}_n[X]$ is commonly used to denote the set of polynomials in $\mathbb{Z}[X]$ that have coefficients in $\mathbb{Z}_n$. This also forms a ring, under the operations of polynomial addition and multiplication such that the coefficient is always computed modulo $n$.

- For polynomial $f(X) \in \mathbb{Z}[X]$, the notation $\mathbb{Z}_n[X]/\left(f(X)\right)$ denotes the set of polynomials in $\mathbb{Z}_n[X]$ with degree less than that of $f(X)$. This forms a *quotient ring* under polynomial addition and multiplication modulo $f(X)$ and coefficient modulus $n$.

### A.1.1 Ring learning with errors

**Definition 3.** *(Ring learning with errors [16]). Let $R_q = \mathbb{Z}_q[x]/\Phi(x)$. Uniformly sample an unknown polynomial $s \leftarrow R_q$. Then let $\boldsymbol{a} = [a_1, a_2, \ldots, a_m] \in R_q^m$ be a vector of $m$ entries where each entry is a polynomial in $R_q$. Uniformly sample errors $\boldsymbol{e} = [e_1, e_2, \ldots, e_m] \leftarrow R_q^m$, a vector of $m$ errors and output $\boldsymbol{b} = \boldsymbol{a} \cdot s + \boldsymbol{e}$ where $\cdot$ multiplies each element of $\boldsymbol{a}$ by $s$ over the ring $R_q$, and $+$ performs an element-wise addition over $R_q$. The RLWE problem is that of finding $s$ from knowledge of $\boldsymbol{a}$ and $\boldsymbol{b}$.*

## A.2 A more in-depth explanation of CKKS

### A.2.1 Encoding and decoding

Recall that the objective of encoding is to map a complex vector onto an element in $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, and that of decoding is to achieve the reverse. CKKS utilises the embedding $\sigma : \frac{\mathbb{R}[X]}{(X^N+1)} \to \mathbb{C}^N$, which is applied via evaluation of $a$ on each root of $\Phi_M(X)$:

$$\sigma(a) = (a(\xi), a(\xi^3), \ldots, a(\xi^{M-1})) = \left(a(\xi^k) \mid k \in \mathbb{Z}_*^N\right) \in \mathbb{C}^N. \tag{A.1}$$

However, as emphasised in the CKKS paper, this embedding produces elements of the form $(z_1, z_2, \ldots, z_N)$ such that $z_i = \overline{z_{-i}}$. Half of the values are the complex conjugates of the other half, due to the nature of evaluating a real polynomial on a set of complex roots. The implication is that when encoding a vector in $\mathbb{C}^{N/2}$, we must first expand it to a vector in $\mathbb{C}^N$. Hence, a projection $\pi$ is required to take only the first half of values in $\sigma(a)$. Thus the full decoding procedure is

$$\mathsf{Decode}(a) = \pi \circ \sigma(a). \tag{A.2}$$

The encoding process maps a complex vector $\mathbf{z} \in \mathbb{C}^{N/2}$ to an element in $\mathcal{R}$ via the inverse of the above, i.e.

$$\mathsf{Encode}(\mathbf{z}) = \sigma^{-1} \circ \pi^{-1}(\mathbf{z}), \tag{A.3}$$

where $\pi^{-1}$ expands $\mathbf{z}$ to include the other $N/2$ complex conjugates. However, the image of $\sigma$ is different from the image of $\pi^{-1}$, and therefore $\pi^{-1}(a)$ must be projected onto the image of $\sigma$. For any $b \in \frac{\mathbb{R}[X]}{(X^N+1)}$, it is possible to write $\sigma(b)$ as a linear combination of basis vectors $\{\sigma(1), \sigma(X), \ldots, \sigma(X^{(N-1)})\} = \{\mathbf{b}_1, \ldots, \mathbf{b}_N\}$, which means that $\mathbf{w} = \pi^{-1}(\mathbf{z})$ can be projected onto the image of $\sigma$ via the projection

$$\mathbf{w}' = \sum_{i=1}^{N} \frac{\langle \mathbf{w}, \mathbf{b}_i \rangle}{\|\mathbf{b}_i\|^2} \mathbf{b}_i, \tag{A.4}$$

where $\langle \cdot \rangle$ is the Hermitian inner product. However, recall that $\mathcal{R}$ contains only integer polynomials, and therefore for any $a \in \mathcal{R}$, $\sigma(a)$ contains complex numbers with integer coefficients. Hence $\mathbf{w}'$ should be rounded. CKKS suggest the use of 'coordinate-wise random rounding', which essentially rounds each number randomly to the higher or lower integer.

### A.2.2 CKKS operations

#### Key generation

Key generation is performed by the following procedures, in which addition and multiplication are computed over $\mathcal{R}$. The objective is to generate $\mathsf{sk}$, $\mathsf{pk}$ and $\mathsf{ek}$ which are the secret, public and evaluation keys respectively.

- Sample $\mathbf{s} \leftarrow \chi_{\text{key}}$, $\mathbf{a} \leftarrow \mathcal{R}_{q_L}$ and $\mathbf{e} \leftarrow \chi_{\text{err}}$. Compute $\mathbf{b} := -\mathbf{a}\mathbf{s} + \mathbf{e} \bmod q_L$. Set $\mathsf{sk} := (1, \mathbf{s})$, $\mathsf{pk} := (\mathbf{b}, \mathbf{a})$.
- Sample $\mathbf{a}' \leftarrow \mathcal{R}_{P \cdot q_L}$ and $\mathbf{e}' \leftarrow \chi_{\text{err}}$. Compute $\mathbf{b}' := -\mathbf{a}'\mathbf{s} + \mathbf{e}' + P\mathbf{s}^2 \bmod P \cdot q_L$. Set $\mathsf{ek} := (\mathbf{b}', \mathbf{a}')$.

#### Encryption and decryption

Let $R_q$ denote the ring modulo an integer $q$. The encryption function takes a plaintext polynomial in $\mathcal{R}$ and produces an element in $\mathcal{R}_{q_L}^2$ a pair of polynomials representing the ciphertext. The decryption function does the inverse.

- $\mathsf{Enc}_{\mathsf{pk}}(\mathbf{m}) : \mathcal{R} \to \mathcal{R}_{q_L}^2$: sample $\mathbf{v} \leftarrow \chi_{\text{enc}}$ and $\mathbf{e}_0, \mathbf{e}_1 \leftarrow \chi_{\text{err}}$. Let $\mathsf{pk} \equiv (\mathbf{b}, \mathbf{a})$. Set $\mathbf{c}_0 := \mathbf{v} \cdot \mathbf{b} + \mathbf{m} + \mathbf{e}_1$ and $\mathbf{c}_1 := \mathbf{v} \cdot \mathbf{a} + \mathbf{e}_2$. Return $(\mathbf{c}_0, \mathbf{c}_1) \bmod q_L$.
- $\mathsf{Dec}_{\mathsf{sk}}(\mathbf{c}) : \mathcal{R}_{q_l}^2 \to \mathcal{R}$: Let $\mathbf{c} \equiv (\mathbf{c}_0, \mathbf{c}_1)$ and $l$ be the current level of $\mathbf{c}$. Output $(\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s}) \bmod q_l$.

#### Addition and multiplication

Adding two ciphertext at the same level $l$ is a straightforward addition of the coefficients of the ciphertext polynomials over the modulus $q_L$. Adding a ciphertext to a plaintext requires the plaintext to be first encoded into $\mathcal{R}_{q_l}$. These are expressed as

$$\mathsf{Add}\left((\mathbf{c}_0, \mathbf{c}_1), (\mathbf{d}_0, \mathbf{d}_1)\right) : \mathcal{R}_{q_l}^2 \times \mathcal{R}_{q_l}^2 \to \mathcal{R}_{q_l}^2 = (\mathbf{c}_0 + \mathbf{d}_0, \mathbf{c}_1 + \mathbf{d}_1) \bmod q_L; \tag{A.5}$$

$$\mathsf{AddPC}\left((\mathbf{c}_0, \mathbf{c}_1), x\right) : \mathcal{R}_{q_l}^2 \times \mathbb{C}^{N/2} \to \mathcal{R}_{q_l}^2 = (\mathbf{c}_0 + \mathbf{m}, \mathbf{c}_1) \bmod q_L; \tag{A.6}$$

where $\mathbf{m} = \mathsf{Encode}(x, \Delta)$. Multiplication of a ciphertext with a plaintext is achieved by encoding the plaintext and then multiplying both ciphertext polynomials with the plaintext polynomial.

$$\mathsf{MultPC}\left((\mathbf{c}_0, \mathbf{c}_1), x\right) : \mathcal{R}_{q_l}^2 \times \mathbb{C}^{N/2} \to \mathcal{R}_{q_l}^2 = (\mathbf{c}_0 \cdot \mathbf{m}, \mathbf{c}_1 \cdot \mathbf{m}) \bmod q_l, \tag{A.7}$$

Ciphertext-ciphertext multiplication is less intuitive. Performing it naively by directly multiplying the two ciphertext will result in needing an additional polynomial to represent the result. This is not ideal as it will slow further multiplications. CKKS proposes a technique whereby the additional polynomial is scaled by a large constant $P$ to reduce its approximation error. Below we give a formulaic description; a deeper explanation of why the following works is given in the original paper. Let $(\mathbf{c}_0, \mathbf{c}_1)$ and $(\mathbf{d}_0, \mathbf{d}_1)$ be the two input ciphertext in $\mathcal{R}_{q_l}^2$, and let

$$(\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2) = (\mathbf{c}_0 \cdot \mathbf{d}_0, \mathbf{c}_0 \cdot \mathbf{d}_1 + \mathbf{c}_1 \cdot \mathbf{d}_0, \mathbf{c}_1 \cdot \mathbf{d}_1) \bmod q_l.$$

Then the multiplication of two ciphertexts is expressed as

$$\begin{aligned} \mathsf{Mult}((\mathbf{c}_0, \mathbf{c}_1),(\mathbf{d}_0, \mathbf{d}_1)) &: \mathcal{R}_{q_l}^2 \times \mathcal{R}_{q_l}^2 \to \mathcal{R}_{q_l}^2 \\ &= (\mathbf{a}_0, \mathbf{a}_1) + \left(\lceil P^{-1} \cdot \mathbf{a}_2 \cdot \mathbf{b}' \rfloor, \lceil P^{-1} \cdot \mathbf{a}_2 \cdot \mathbf{a}' \rfloor\right) \bmod q_l, \end{aligned} \tag{A.8}$$

where $(\mathbf{b}', \mathbf{a}') = \mathsf{ek}$ and $\lfloor \cdot \rceil$ denotes rounding to the nearest integer.

## A.3 Encryption parameters

| **Model** | **Multiplicative depth** | **Coefficient moduli sizes** | **Scaling factor** | **Polynomial modulus degree** |
|---|---|---|---|---|
| MN-*prune* | 5 | $[32, 25, 25, 25, 25, 25, 32]$ | $2^{25}$ | 8192 |
| MN-*fast* | 5 | $[32, 28, 28, 28, 28, 28, 32]$ | $2^{28}$ | 8192 |
| CF-*fast* | 8 | $[60, 30, 30, 30, 30, 30, 30, 30, 30, 60]$ | $2^{30}$ | 16384 |
| CryptoFace | 4 | $[32, 25, 25, 25, 25, 25, 32]$ | $2^{25}$ | 8192 |

# Appendix B

# Additional information

## B.1 Example use of $\Psi$-lib

```
from utils.data_loader import load_data
from utils.logger import msg, OutFlags
from crypto.schemes import *
from model.lin_algebra import *
from model.creator import Creator
from model.revealer import Revealer
import model.low_lat as low_lat

from keras.layers import ZeroPadding2D

N = 512 * 16 * 1 * 1
scheme, decryptor = init_scheme_ckks(N, primes=[32, 25, 25, 25, 25, 25, 32], scale_factor
    =25)
creator = Creator(scheme)

s = 2
k = 5

model = Model(creator)
model.add(InputLayer(input_shape=(1, 30, 30)))
model.add(ConvLayer(kernel_size=(k, k), stride=s, padding=0, num_maps=5, dense_mode=False
    ))
model.add(Flatten(groups=1))
model.add(ActivationLayer(mode='square'))
model.add(DenseLayer(output_length=100, prev_channels=5))
model.add(ActivationLayer(mode='square'))
model.add(DenseLayer(output_length=10))

weights, test_features, preds_base, outputs_base = load_data(name='MNIST-CRYPTONETS-
    ORIGINAL')
test_features = ZeroPadding2D()(test_features).numpy()

model.compile(data_mode=1)
model.load_weights(weights, scale=16)
model.summary()

image = test_features[0]
preds_base_i = preds_base[0]
outputs_base_i = outputs_base[0]

data = creator.encrypt_dense(mat=image)
img_groups = creator.obtain_image_groups(mat=image, k=k, s=s)
outputs = model.predict(input_data=img_groups)
outputs = Revealer(scheme, decryptor).reveal_outputs(outputs, length=10, data_mode=1)
```

## B.2 Extracting the weight matrix of a convolution

```
def conv_weights(d_in, W_c, s, padding='valid', pool=False):
    c_in = W_c.shape[1]
    c_out = W_c.shape[0]
    k = W_c.shape[-1]
    p = None
    if padding == 'valid':
        p = 0
        d_out = int(math.floor((d_in - k + p) / s)) + 1
    elif padding == 'same':
        d_out = math.ceil(d_in / s)

        if d_in % s == 0:
            p = max(k - s, 0)
        else:
            p = max(k - (d_in % s), 0)

    W = np.zeros((c_out * d_out ** 2, c_in * d_in ** 2))

    # iterate through each output
    for j in tqdm(range(c_out * d_out * d_out)):
        c_o = j // (d_out * d_out)
        y_o = j % (d_out * d_out) // d_out
        x_o = j % (d_out * d_out) % d_out

        # iterate through each weight of the filter for c_o
        for i in range(c_in * k * k):
            c_i = i // (k * k)
            k_r = (i % (k * k)) // k
            k_c = i % (k * k) % k
            w = W_c[c_o, c_i, k_r, k_c]

            # find the position of the input value that
            # contributes to output j, under weight w.
            pad_offset = p // 2
            y = y_o * s + k_r - pad_offset
            x = x_o * s + k_c - pad_offset

            # out-of-bounds, could happen due to padding
            if y >= d_in or x >= d_in or y < 0 or x < 0:
                continue

            # tune average pooling values
            if w != 0 and pool:
                t = k ** 2
                x_ = x - k_c
                y_ = y - k_r
                p_x = k * max(max(0, -x_), max(0, x_ + k - d_in))
                p_y = k * max(max(0, -y_), max(0, y_ + k - d_in))
                t -= p_x + p_y - (p_x // k) * (p_y // k)
                w = 1 / t
            W[j][c_i * d_in ** 2 + y * d_in + x] = w
    return W
```

## B.3 Experimental setup

| Detail | Description |
|---|---|
| Processor | i7 8700k @ 4.7GHz |
| Memory | 32GB DDR4 @ 2400MHz |
| GPU | NVIDIA RTX 3080 |
| Python version | 3.8.2 |
| SEAL version | 3.5 |
| Tensorflow version | 2.4.0 |
| CUDA version | 11.2 |

**(a)** Hardware and software versions.

| Model | Optimiser | Epochs | Learning rate | Batch size |
|---|---|---|---|---|
| MN-f | Adam | 500 | $4 \times 10^{-4}$ | 32 |
| MN-p | Adam | 200 | $7 \times 10^{-5}$ | 32 |
| CF-f | Adam | 300 | $8 \times 10^{-4}$ | 64 |
| CryptoFace encoder | Adam | 50 | $3 \times 10^{-3}$ | 256 |
| CryptoFace top portion | Adam | 3 | $3 \times 10^{-3}$ | 32 |

**(b)** Training parameters

**Table B.1:** Experimental setup details.

## B.4 Full model architectures

| Layer | Description | Parameters | Output |
|---|---|---|---|
| 1 | Convolution | $k = 5, s = 2$ | (5, 13, 13) |
| - | Square | - | (5, 13, 13) |
| 2 | Avg. Pool | $k = 3, s = 1$ | (5, 13, 13) |
| 3 | Convolution | $k = 3, s = 2$ | (50, 5, 5) |
| 4 | Avg. Pool | $k = 3, s = 1$ | (50, 5, 5) |
| 5 | Flatten | - | (1250) |
| 6 | Dense | - | (100) |
| - | Square | - | (100) |
| 7 | Dense | - | (10) |
| - | Sigmoid | - | (10) |

**(a)** CryptoNets architecture

| Layer | Description | Parameters | Output |
|---|---|---|---|
| 1 | Convolution | $k = 3, s = 1$ | (5, 28, 28) |
| - | Square | - | (5, 28, 28) |
| 2 | Avg. Pool | $k = 3, s = 2$ | (5, 14, 14) |
| 3 | Convolution | $k = 3, s = 1$ | (50, 12, 12) |
| 4 | Avg. Pool | $k = 3, s = 2$ | (50, 12, 12) |
| 5 | Flatten | - | (7200) |
| 6 | Dense | - | (32) |
| - | Square | - | (32) |
| 7 | Dense | - | (10) |
| - | Softmax | - | (10) |

**(b)** Optimised architecture (MN-*fast*)

**Table B.2:** he original MNIST architecture and our optimised design. $k$ and $s$ indicate the kernel width and stride respectively. $m$ indicates the number of output nodes for a dense layer.

| Layer | Description | Parameters | Output | Activation |
|---|---|---|---|---|
| 1 | Convolution | $k = 3, s = 1$ | (18, 30, 30) | Square |
| 2 | Average Pooling | $k = 3, s = 3$ | (18, 10, 10) | Identity |
| 3 | Convolution | $k = 3, s = 1$ | (64, 8, 8) | Square |
| 4 | Average Pooling | $k = 2, s = 2$ | (64, 4, 4) | Identity |
| 5 | Convolution | $k = 3, s = 1$ | (256, 2, 2) | Identity |
| 6 | Flatten | - | (1024) | Identity |
| 7 | Dense | - | (256) | Square |
| 8 | Dense | - | (10) | Softmax |

**(a)** CF-*base*

| Layer | Description | Parameters | Output | Activation |
|---|---|---|---|---|
| 1 | Convolution | $k = 3, s = 1$ | (18, 30, 30) | Square |
| 2 | Average Pooling | $k = 3, s = 3$ | (18, 10, 10) | Identity |
| 3 | Convolution | $k = 3, s = 1$ | (13, 8, 8) | Square |
| 4 | Convolution | $k = 1, s = 1$ | (64, 8, 8) | Square |
| 5 | Average Pooling | $k = 2, s = 2$ | (64, 4, 4) | Identity |
| 6 | Convolution | $k = 3, s = 1$ | (256, 2, 2) | Identity |
| 7 | Flatten | - | (1024) | Identity |
| 8 | Dense | - | (256) | Square |
| 9 | Dense | - | (10) | Softmax |

**(b)** CF-*fast*

**Table B.3:** Comparison of CIFAR-10 architectures.

## B.5 Additional results

| Metric | FCrypNts‡ | LoLa‡‡ | FFConv‡‡ | CF-f |
|---|---|---|---|---|
| Mul CC | N/A | 2 | 2 | 5 |
| Mul PC | N/A | 4075 | 3575 | 4080 |
| Add CC | N/A | 52,975 | 9740 | 4092 |
| Add PC | N/A | N/A | N/A | 85 |
| Rot | - | 52,975 | 7162 | 2872 |
| Inference time | 22,372 s | 730 s | 96.4 s | 273 s |
| Test accuracy | 76.72 | 74.1 | 76.5 | 73.1 |
| Message size | 1,610.8 MB | N/A | N/A | 70.78 MB |
| Scheme | FV | BFV | BFV | CKKS |

**Table B.4:** Comparison of CIFAR models with previous works. CC indicates a ciphertext-ciphertext operation and PC indicates a plaintext-ciphertext operation. ‡ - details are borrowed from [11]. ‡‡ - details are calculated using the results from [24],[7]. N/A indicates insufficient data to deduce from.

## B.6 Tools and resources used to produce diagrams

The following tools and resources were used to produce the figures in this work:

- `https://app.diagrams.net/` was used to draw many of the diagrams and export them into `pdf` form.
- `http://alexlenail.me/NN-SVG/LeNet.html` was used to draw neural network diagrams.
- `matplotlib` was used to produce plots.
- The cat image from Figure 2.1 is from `https://www.itl.cat/wallview/ixmoh_cute-cat-wallpaper-iphone-cute-invasive-species-animals/`.
- The silhouettes from Figure 2.1 are from `https://svgsilh.com/image/303792.html` and `https://svgsilh.com/image/296989.html`.

All URLs were retrieved in May 2021.

## B.7 Changes from original proposal

The following is a list of deviations of the actual project from the original project proposal. Though mostly minor, they are still worth mentioning to the reader for the sake of clarification.

- The original proposal mentions a 95% validation accuracy threshold as a basis of evaluation for both MNIST and CIFAR-10 models. At the time of writing the proposal, I was not yet aware that 95% on CIFAR-10 is beyond the capabilities of even the state-of-the-art solutions in secure deep learning inference. Hence, I revised the evaluation criteria to only consider the performance *relative* to the work previously done, rather than set a fixed threshold.

- The original proposal mentions a hardware-focused route of extensions in applying GPU acceleration to reduce model latency. However, during the requirements analysis stage, it became apparent that implementing software-oriented extensions such as more efficient algorithms, and the use of model pruning, would be more suitable for the goal of producing a usable solution that is not limited by the user's hardware.
- The original proposal mentions the use of BFV as an initial step to take, and the use of CKKS as an extension. This was done as intended, but due to CKKS's recency and affinity for encoding real numbers, I chose to use CKKS to produce the results in this work.

# Appendix C

# Proposal

The original project proposal is shown starting from the next page.

Computer Science Tripos, Part II Project Proposal

# Privacy-Preserving Deep Learning Inference using Homomorphic Encryption

2398E

22/10/2020

## 1 Introduction and Description

Most deep learning applications today process data in their unencrypted form. This poses security and privacy concerns, especially for systems handling sensitive information, such as medical records. In recent years there has been much work on applying methods from homomorphic encryption to allow deep learning models to make inferences using encrypted input, without needing to decrypt it.

*Homomorphic encryption* (HE) is a form of encryption allowing computations to be performed on data without requiring access to the secret key. Computations are typically represented as arithmetic or Boolean circuits. Many HE algorithms are based on the *Ring Learning with Errors* problem, which is conjectured to be hard for quantum computers, making it suitable as a basis for post-quantum cryptography.

HE schemes are often categorised into types that include *partially*, *levelled fully* and *fully* homomorphic schemes. Their properties are summarised as follows:

- Partially homomorphic schemes allow only one type of operation (e.g. only multiplication) to be performed on encrypted data.
- Levelled fully homomorphic schemes support arbitrary computation on ciphertexts, but only up to a limited number of operations. This is because each application of an operation adds noise to the ciphertext. If too much noise is present, the plaintext will be unrecoverable.
- Fully homomorphic schemes support arbitrary computation up to an unlimited number of operations, thus allowing any polynomial function of the encrypted data to be computed.

It is possible to implement the inference of a deep neural network using HE. CryptoNets by Dowlin et al. [1] was one of the first examples of this, where the authors used a levelled homomorphic scheme to implement inference for a neural network trained on the MNIST dataset. Recently, Badawi et al. [2] implemented a Convolutional Neural Network (CNN) using a fully homomorphic scheme. Their network can classify an MNIST example in 5.16 seconds and a CIFAR-10 example in around 300 seconds. The main issues barring homomorphic image

models from being widely used today are the high inference time, memory requirements and the limitations of the types of models that can be implemented.

The aim of the project will be to create a Python library for training and applying privacy-preserving neural networks using HE with an emphasis on ease-of-use and inference performance. This will consist of the following:

- **Project baseline**: Create a Python library utilising homomorphic encryption to perform neural network inference on homomorphically encrypted data. A homomorphic encryption scheme will be used to implement the operations commonly found in neural networks (such as convolution, pooling and activation). The library will not only support basic neural networks but also convolutional neural networks. The library will focus on maintaining user-friendliness, and serve as an abstraction from the underlying homomorphic computations. It should support use of a scheme commonly found in recent studies in the field, such as BFV by Brakersk et al. [3] or THFE by Chillotti et al. [4]. An existing library, such as Microsoft SEAL [5] will be used to provide the encryption primitives.

- **Evaluation**: Apply the library by creating models for inference on standard datasets, such as MNIST and CIFAR-10. Perform evaluation of the library by recording metrics including model accuracy and inference latency, and comparing these metrics with results of frameworks from the literature, such as FHE-DINN by Bourse et al. [6]. The general goal is to be able to produce similar results to those from literature but with the use of our library. Methods of evaluation are discussed in further detail in section 4.

- **Extensions**: Possible extensions include a manual implementation of a HE scheme, an implementation of a GPU accelerated HE scheme, as well as applying my library to a more realistic problem, such as that of facial verification. These are discussed in further detail in section 5.

The motivation for creating such a library is the lack of current methods for conveniently creating homomorphically encrypted neural networks, as well as the fact that existing libraries are typically research-focussed and not aimed at typical machine learning practitioners. This project will also be an opportunity to apply recent ideas from the literature, such as the work by Bourse et al., in order to improve the inference performance of the library.

## 2 Substance and Structure of the Project

The project can be divided into multiple stages:

1. **Literature review and choice of encryption scheme**: review of the literature surrounding the various homomorphic encryption schemes in order to better understand the theory behind homomorphic neural networks. I will select an encryption scheme to use for the baseline of the project. The BFV encryption is a suitable choice due to the abundance of literature using it for machine learning applications. Another scheme is CKKS [7], which supports encryption of real numbers, making it a suitable choice for machine learning. I will consider implementing CKKS as an extension in the later stages of the project, as it is more recent and a greater challenge to implement.

2. **Design and implementation of homomorphic neural network library**: at this stage the high-level components of the homomorphic neural network library will be designed and implemented. I will also create an interface to separate the high-level components of the library from the underlying HE computations. The interface should support the fundamental neural network operations (such as convolution, pooling and activation), and allow combinations of them to create an arbitrary neural network. A pre-existing library implementing our choice of encryption scheme will be used to demonstrate the basic functionality of the library. PyTorch will likely be used to perform model training, as it is relatively high-level compared to other machine learning libraries.

3. **Application of library to benchmark datasets**: the CNN library will be used to train homomorphic models for the MNIST and CIFAR-10 datasets to a sufficient degree of accuracy (minimum 95% validation accuracy). The trained models will be used to apply inference to encrypted data from their respective datasets. *Performance metrics* such as model accuracy, inference latency and memory usage will be recorded.

4. **Evaluation of library with respect to benchmarks**: The metrics of the benchmark models will be evaluated against those from the literature. It is expected that our library should achieve similar performance metrics compared to existing models that use the same encryption schemes. The purpose of this evaluation is to assess whether our library's handling of the homomorphic operations is the limiting factor in terms of performance, and how its effect on performance can be optimised. In addition this will be an opportunity to reproduce the results from literature by using the same architectures and parameters.

5. **Working on extensions**: a list of possible extensions are detailed in section 5. These are:

   (a) **Add support for the CKKS scheme**

   (b) **Explore the potential of GPU acceleration for HE**

   (c) **Application of library to a problem such as facial-verification**

6. **Writing the dissertation**.

## 3 Success Criteria

The project in its most basic form will be deemed successful once the following tasks have been completed.

1. Implement an interface that allows the user to specify the architecture of a network, which may be a convolutional neural network. The interface should include basic architectural features such as convolutional layers, dense layers, pooling layers and activation functions.
2. Implement the ability train a model using a specified architecture and convert the model into a form that can make inferences on single examples encrypted with a homomorphic encryption scheme, as well as on batches of encrypted examples.
3. Utilise optimisations and design choices from literature to enhance the performance of the library. For example, Badawi et al. [2] discuss optimisations for using BFV to implement network convolutional, and techniques such as ciphertext packing.
4. Use the library to construct models trained on the MNIST and CIFAR-10 datasets.

## 4 Evaluation

The library will be primarily evaluated in terms of its ability to construct models with satisfactory performance. Since there is no fixed model, and different models may have differing architectures depending on the dataset being trained on, we will follow the convention of benchmarking on models trained on standard datasets inluding MNIST and CIFAR-10. The library will be used to train models for these datasets, and the performance metrics of those models will be compared with counterpart models from the literature.

The following metrics are likely choices:

- **Inference time**: this is the main barrier that HE faces in the domain of deep learning. The inference time will be compared with that of other models from the literature as well as non-HE models. If the inference time is too high (relatively), then I will investigate how to reduce it.
- **Validation accuracy**: the inclusion of HE means that there is typically a degree of noise added to the ciphertext forms of the data, which will affect the inference process compared to a non-HE model. It will be necessary to assess how the validation accuracy of the model compares with state-of-the-art models that do not utilise HE, as well as other HE models from the literature. The validation accuracy should exceed 95% as a minimum baseline.
- **Memory usage**: this will largely depend on how I utilise the underlying homomorphic operations.
- **Security level**: this is usually measured as $n$ bits corresponding to the fact that an attacker would have to perform $2^n$ guesses on average to break the encryption.

# 5 Extensions

## 5.1 Add support for the CKKS scheme

CKKS is a relatively recent scheme, with support for an approximate ciphertext representation for real numbers. This makes it a more ideal choice for machine learning applications than previous schemes such as BFV, which are limited to encrypting integers and require reals to be scaled and rounded before encryption. An interesting extension would be to manually implement CKKS, which would serve as an exercise of my understanding of the scheme, before integrating it into the back-end of the library.

## 5.2 Explore the potential of GPU acceleration for HE

The use of HE computations running on the CPU will limit performance and the practicality of using the library on higher resolution images. Integrating a GPU accelerated HE scheme into the back-end will increase performance. An interesting extension would be to create a GPU-implementation of the CKKS scheme for the library using a CUDA-based Python abstraction layer such as PyTorch or Numba. To the best of my knowledge there is no available open-source GPU implementation of CKKS at the time of writing this proposal.

## 5.3 Application of library to a real-world problem

The library could be used to create a model trained on a more realistic dataset than the standard datasets often used for benchmarking. An idea would be to create a model to solve the problem of facial-verification on encrypted face images. A face image dataset such as FaceNet [8] could be used. The effectiveness of different HE schemes integrated with the library up to this point in the project could be explored. Other possible applications could be medical image analysis and credit card fraud detection.

# 6 Starting Point

Prior to this project, I have had a moderate amount of experience in applying deep learning models to solving real-world problems. This mainly comes from a 3-month internship I did in machine learning over the summer.

I have a basic understanding of the theoretical workings of HE, which comes mainly from reading papers during the summers. I have also studied recent papers on applying HE schemes to machine learning. In addition I will use knowledge from the following courses, which I have either taken or will be taking:

- Artificial Intelligence, Security, Complexity Theory (Part IB)
- Cryptography, Deep Neural Networks, Machine Learning and Bayesian Inference (Part II)

In terms of the programming experience, I have a confident grasp of Python and a moderate amount of experience in using the Tensorflow framework. However, I have not used Pytorch before, and will need to familiarise myself with them in the research phase of the project.

# 7 Resources Required

I will require the use of Microsoft Azure GPUs for training models. I will request access to the service by asking the department.

I plan to use my own machine (Intel i7, Nvidia GPU, 32GB RAM, Windows 10 and Ubuntu 20.04) for prototyping the project as well as writing the dissertation. My machine is suitable because it has a discrete GPU, which is convenient for training models, although the cloud GPUs will have more VRAM. I accept full responsibility for my machine.

A private Github (github.com) repository will be used for storing source code and the LaTeX for the dissertation. I have prepared data contingency plans including weekly backups to a Google Drive (google.com/drive) account with 100GB of storage available, as well as to a 1TB external hard drive. In addition, in the case my machine fails, I will switch to my backup laptop which also has a discrete GPU.

# 8 Timetable and Milestones

### Michaelmas Term

- **23/10/2020 – 05/11/2020**

  **Literature review**: research the literature behind homomorphic encryption and its application to deep learning; consolidate my understanding of the suitable homomorphic encryption schemes including BFV and TFHE; learn the basics of Pytorch; select and justify the choice of encryption schemes I will use.

  **Milestone**: produce a document summarising the selected encryption schemes, as well as a detailed comparison of recent literature in the field.

- **06/11/2020 – 19/11/2020**

  Set up the development environment and install any required libraries and packages; design the structure of the inference library; begin implementing the library using an existing HE scheme as the back-end.

  **Milestone**: produce a document demonstrating the design of the library and the progress on the development of it.

- **20/11/2020 – 03/12/2020**

  Continuation of library implementation; carry out simple tests of my library on the MNIST and CIFAR-10 datasets. While doing this, fix any issues with the library that I find.

  **Milestone**: produce a document demonstrating the progress of the work on the library.

### Christmas Break

- **04/12/2020 – 17/12/2020**

  Finish implementing the library; write documentation for the library; train MNIST and CIFAR-10 models to perform intermediate evaluation of the library's performance.

  **Milestone**: complete the library and its documentation, and train the models to be used for evaluation.

- **18/12/2020 – 31/12/2020**

  **Slack period**: finish any milestones that have not yet been finished up to this point.

### Lent Term

- **01/01/2021 – 21/01/2021**

  Read papers from the literature on optimisations that can be applied to neural network operations built with HE; optimise the library, e.g. by tuning parameters of the encryption scheme or usage of the encryption scheme's operations.

  **Milestone**: add any necessary optimisations to the library.

- **22/01/2021 – 04/02/2021**

  *Note: progress report deadline on 05/02/2021*

  Evaluation of the library with respect to the MNIST and CIFAR-10 models; comparison of our models with those from literature; adjust the library to account for any issues met; write the progress report.

  **Milestone**: finish the evaluation of the first iteration of the library. Submit the progress report.

- **05/02/2021 – 18/02/2021**

  Read material on the theory behind the CKKS scheme; complete an implementation of CKKS. Perform tests to check the correctness of the implementation; integrate the implementation into the back-end of the library and benchmark with MNIST and CIFAR-10.

  **Milestone**: complete and benchmark the basic CKKS implementation.

- **19/02/2021 – 04/03/2021**

  Familarise myself with GPU programming in Python; start the GPU implementation of a HE scheme.

  **Milestone**: produce a document summarising the progress on the GPU implementation.

- **05/03/2021 – 18/03/2021**

  Complete the GPU implementation of a HE scheme and integrate it into the back-end of the library; perform tests to check the correctness of the implementation; benchmark with MNIST and CIFAR-10.

  **Milestone**: complete and benchmark the GPU implementation.

### Easter Break

- **19/03/2021 – 01/04/2021**

  **Slack period**: finish any milestones that have not yet been finished up to this point.

- **02/04/2021 – 15/04/2021**

  If time allows, implement the facial-verification model using the library; evaluate the feasibility of such a model in terms of its performance metrics.

  **Milestone**: implement the facial-verification model if time allows.

- **16/04/2021 – 29/04/2021**

  **Writing the dissertation**: focus on completing and submitting the dissertation; proof-reading and formatting.

  **Milestone**: complete and submit the dissertation.

### Easter Term

- **30/04/2021 – 13/05/2021**

  *Note: dissertation deadline on 14/05/2021*

  **Slack period**: finish any work that has not yet been finished up to this point.

## References


[1] Nathan Dowlin et al. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of the 33rd International Conference on Machine Learning*, pages 201–210, 2016.

[2] Ahmad Al Badawi, Jin Chao, Jie Lin, Chan Fook Mun, Jun Jie Sim, Benjamin Hong Meng Tan, Xiao Nan, Khin Mi Mi Aung, and Vijay Ramaseshan Chandrasekhar. Towards the AlexNet Moment for Homomorphic Encryption: HCNN, the First Homomorphic CNN on Encrypted Data with GPUs. In *IEEE Transactions on Emerging Topics in Computing*, 2020.

[3] Zvika Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Advances in Cryptology – CRYPTO 2012*, pages 868–886, 2012.

[4] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast Fully Homomorphic Encryption over the Torus. *Cryptology ePrint Archive*, Report 2018/421, 2018.

[5] Microsoft, *SEAL*, `github.com/microsoft/SEAL`. Accessed 22/10/2020.

[6] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. *Fast Homomorphic Evaluation of Deep Discretized Neural Networks.* In *Annual International Cryptology Conference, Springer*, pages 483–512, 2018.

[7] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. *Homomorphic Encryption for Arithmetic of Approximate Numbers.* In *ASIACRYPT 2017*, Springer, pages 409–437, 2017.

[8] Florian Schroff, Dmitry Kalenichenko and James Philbin. FaceNet: A Unified Embedding for Face Recognition and Clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.