UNIVERSITY OF BERGEN
DEPARTMENT OF INFORMATICS

# Speech Command Recognition on Encrypted Data

*Author:* Liv Ingunn Dornfest
*Supervisors:* Martha Norberg Hovd & Håvard Raddum


UNIVERSITY OF BERGEN
*Faculty of Science and Technology*

18th December 2024

Year: 2024
Title: Speech Command Recognition on Encrypted Data
Author: Liv Ingunn Dornfest

# Acknowledgements

First and foremost, I would like to thank my supervisors Martha Norberg Hovd and Håvard Raddum for their support and guidance through this writing process. Your feedback, input and expertise has been truly invaluable. I would also like to thank Simula UiB for facilitating this master thesis, and providing not only an office space, but also a great community.

I would also like to express my gratitude to my wonderful family and friends for all the support I have received. Thank you for your patience, encouraging words and welcome distractions. A special thanks to my mother and sister for helpful inputs. My gratitude also goes out to my work colleagues who have offered and provided feedback.

This thesis would have been difficult to complete without the unconditional support from my partner Didrik. Thank you for stepping up in morning sickness and in health, I could not have done this without you.

# Abstract

This thesis examines the use of fully homomorphic encryption (FHE) schemes, specifically the CKKS and TFHE schemes, in combination with machine learning models for speech command recognition. Advancements in machine learning and artificial intelligence has put concerns over data protection at the forefront. This thesis aims to investigate FHE as a way to securely process speech data.

Speech recognition is approached as an image classification problem on spectrograms using Convolutional Neural Networks (CNN). CNN models using TFHE or CKKS are implemented using leading cryptographic libraries, and then benchmarked and compared with each other.

The results show that currently available libraries for CKKS and TFHE can be used for encrypted inference, but that runtime and memory usage remains too high to be practical. The model with the highest accuracy used the TFHE scheme, with an accuracy of 89.6%, an average inference time of 470 seconds for one sample, and a memory usage of 5.9 GB. A model using CKKS with a similar accuracy of 87.6% had an average inference time of 155 seconds, but also a memory usage of 22.7 GB.

The CKKS scheme seems to be more suitable due to its faster inference time. Conversely, the TFHE library offer better machine learning functionality and compatibility with existing machine learning frameworks. These factors might come to play a more important role in the coming years. Additionally, the experiments demonstrate that the FHE schemes can be used on consumer hardware by those without cryptographic expertise, and is therefore available for exploration on other machine learning problems.

# Contents

# Chapter 1

# Introduction

## 1.1 Background

In today's information age, data is exchanged continuously through digital platforms. This includes sensitive data, such as health, financial or other personal information that has a need for protection. Cryptography is used to maintain the confidentiality and integrity of this data, and encryption algorithms are a part of almost every modern communication system. The need to protect sensitive data, and the development of cryptographic tools for this purpose, remains crucial in a fast-changing technological landscape where data is referred to as the new currency. Using data efficiently presents many opportunities for growth, insights and technological advancements, but is juxtaposed with rising challenges in securing the data.

Machine learning and generative artificial intelligence have in recent years become commonplace words, as research in the field has showcased advancements and the wide area of use for such methods. Machine learning algorithms are prevalent in the information sphere, especially as increasingly sophisticated technology digitalize many tasks and data is collected en masse. An ecosystem has developed of data collection, use of collected data to train machine learning models, and introducing new functionality that again collects new data when used. This leads to concerns over privacy, data protection and the right to own the information concerning oneself.

Many problem areas where machine learning models are applied involves handling of sensitive data, for example financial fraud detection, health diagnostics and biometric identification. The internet of things and smart home appliances introduce voice control mechanisms, which inadvertently requires microphones to be active and present in the most private spheres. How can data be protected and privacy ensured, while also allowing the data to be used for beneficial functionality and future advancements? One of the approaches to such privacy preserving machine learning is to use a special type of cryptographic system called *fully homomorphic encryption* (FHE).

FHE is a type of cryptography where one can calculate on encrypted data. This means that sensitive data can be encrypted before being sent to a machine learning model, and only decrypted once the final result has been returned to the data owner. FHE research has been a field of rapid developments following a major breakthrough in 2009, and the use of FHE for machine learning applications is a continued area of interest. This has great relevance, especially due to the advent of cloud computing and remote

models placed in cloud environments, as FHE systems would allow data to be sent to and from remote sources without risking confidentiality breaches.

## 1.2 Problem Statement and Objectives

Speech recognition technologies are used in private spheres, for instance through voice assistants, live translation services and control of home appliances. With data as an increasingly valuable currency in data-driven technology, the question on how to protect this personal data and prevent unintentional or unauthorized data collection is ever relevant. Exploring the use of FHE on speech data can present a part of the solution in how to handle security and privacy for applications using voice as the main control interface.

Can FHE systems be used with a model that recognizes spoken word commands to ensure that the speech data is handled in a secure manner? This is a question that the thesis will seek to answer. In conjunction with this, the thesis will investigate to what degree existing FHE systems can be used in a practical application on such a problem, and which FHE system is the most suitable.

Research in FHE is generally focused on a specific FHE system, and for machine learning problems often applied to well-known problems and datasets. This thesis will aim to illustrate the state of FHE systems and existing libraries that implement them from a practical point of view. The speech command recognition problem is also mainly unexplored in FHE research, and the thesis will therefore give insight into how FHE libraries can be used for a problem that falls beyond the scope of most existing demonstrations, examples and guides. Finally, even though different FHE systems are intended for and applied to the same kind of problems, they are rarely directly compared. An attempt at a fair comparison of the two different FHE systems CKKS and TFHE, on the same machine learning problem, using the same hardware and available resources, will hopefully give an intuition on how to choose the most suitable FHE system for the problem at hand.

## 1.3 Thesis Structure

This thesis examines a problem that falls within the fields of cryptography and machine learning. The background in Chapter 2 introduces both fields, focusing on FHE systems for cryptography and convolutional neural networks for machine learning. These networks are the primary type of model explored in the thesis, and more details on this and its application to the speech command recognition problem is also presented in the background. In Chapter 3, privacy preserving machine learning is introduced, which details how FHE and convolutional neural networks can be combined to achieve model inference on encrypted data.

Chapter 4 describes the thesis experiment, where machine learning models using different FHE systems are designed, trained and tested on a speech command recognition dataset. Here, the dataset, experiment framework and methodology are described, before the results of models using two main FHE systems are presented individually. The chapter concludes with a comparison of the two schemes. The overall findings are presented and discussed in Chapter 5.

## 1.4 Thesis Scope and Limitations

The main focus of this thesis is the use of FHE systems and the cryptographic libraries used for this. The machine learning problem is thus approached from an FHE perspective; the models are limited to types and architectures for which FHE can be practically applied using existing cryptographic libraries, as opposed to necessarily being the optimal models to solve the problem.

To limit the scope of the thesis, only the current two main FHE systems CKKS and TFHE are studied. They are explored using two libraries that each represent one system. The chosen libraries are well-known within the field, cited in relevant research, and open source.

The experiment is structured by testing a simpler and a more complex model for each system, as well as some models being tested for the sake of comparison. Naturally, there are endless configurations of libraries, models, machine learning parameters and FHE parameters that could be explored and tested, but an exhaustive search is not feasible within the time and resource

constraints for this thesis. The current scope aims to give a realistic picture of what someone without cryptographic expertise can hope to achieve using currently available technology to implement FHE for speech command recognition. An additional aim is to give some intuition on what FHE system to choose, and why.

# Chapter 2

# Background

## 2.1 Overview

This chapter will introduce both the fields of cryptography and machine learning, focusing on the parts relevant for the thesis experiment in Chapter 4.

The first section will give a brief explanation of cryptography, how cryptographic systems are fundamentally designed, and the security of such systems. This is followed by a section delving deeper into homomorphic encryption. Here, the two schemes that are explored in the thesis, namely TFHE and CKKS, are explained in greater detail.

Machine learning is the subject of the next section. After an overview of machine learning and the main approaches in the field, the attention is turned towards deep learning and neural networks. Convolutional neural networks, the model type which is used in the experiment, are then introduced.

The final section presents the speech command recognition problem, and how speech data is processed so that it can be solved using convolutional neural networks.

## 2.2 Cryptography

Cryptography is the study of how mathematical techniques can be used to maintain information security. In the digital domain, cryptographic tools are used to maintain confidentiality and privacy of information, prevent manipulation of data and authenticate entities and data origins [58].

Typically, a cryptographic system involves transforming information into a form that is unintelligible to unauthorized users, while the intended recipients hold a key that allows them to transform the information back into its original, readable form. An unencrypted message is referred to as a plaintext and an encrypted message as a ciphertext. The encryption and decryption processes are functions applied to the plaintext and ciphertext respectively. These processes are illustrated in Fig. 2.1.

Cryptography systems can be divided into two main categories: symmetric and asymmetric schemes. In symmetric cryptography, a shared secret key is used for encryption and decryption. In an asymmetric cryptography scheme, often also referred to as public key cryptography, the secret key is replaced with a pair of keys, where one key is public and one key is private.

Information is encrypted using the public key of a key pair and the

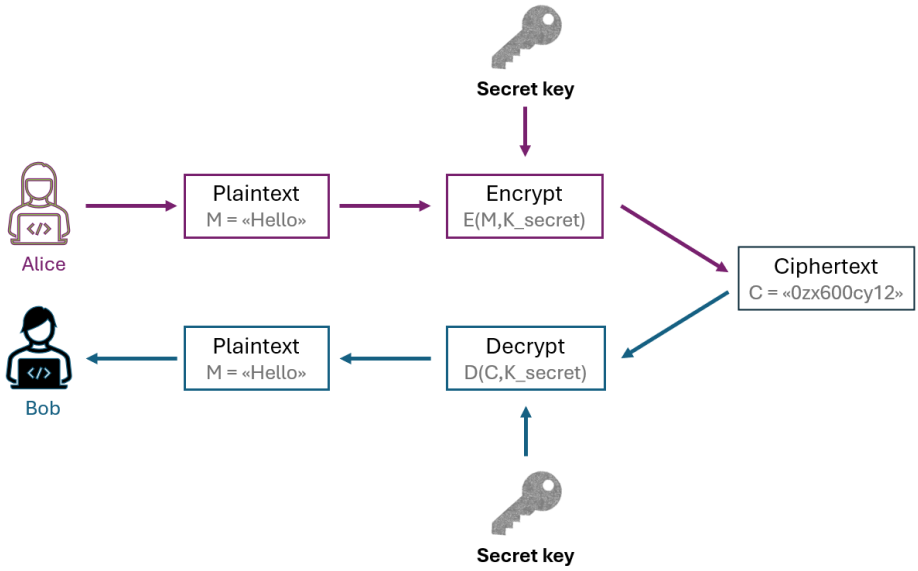


Figure 2.1: Encryption and decryption with a symmetric key

private key is for decryption. For two entities, Alice and Bob, Alice will encrypt a message for Bob using Bob's public key. Only Bob will be able to decrypt the message, using his private key that is kept secret. Fig. 2.2 depicts a simple message delivery from Alice to Bob. If Bob wants to encrypt a message for Alice, he will in turn use Alice's public key for encryption.

### 2.2.1 Security of Cryptographic Systems

The security level of a modern cryptographic system is a measure of how costly it would be to retrieve the secret information without access to the secret key. Essentially, "guessing" the secret key by exhausting all possible alternatives should involve a number of guesses (or operations) so high as to make it an infeasible approach with regards to time and resources.

The security of cryptographic systems are measured by this number of operations, denoted in bits. A scheme of $n$-bit security would take $2^n$ operations to recover the secret key and thus "break" the system. Widely used systems today often have security levels of 128 or 256 bits, which per current technology would take billions of years to break [7]. Attacks against crypto-

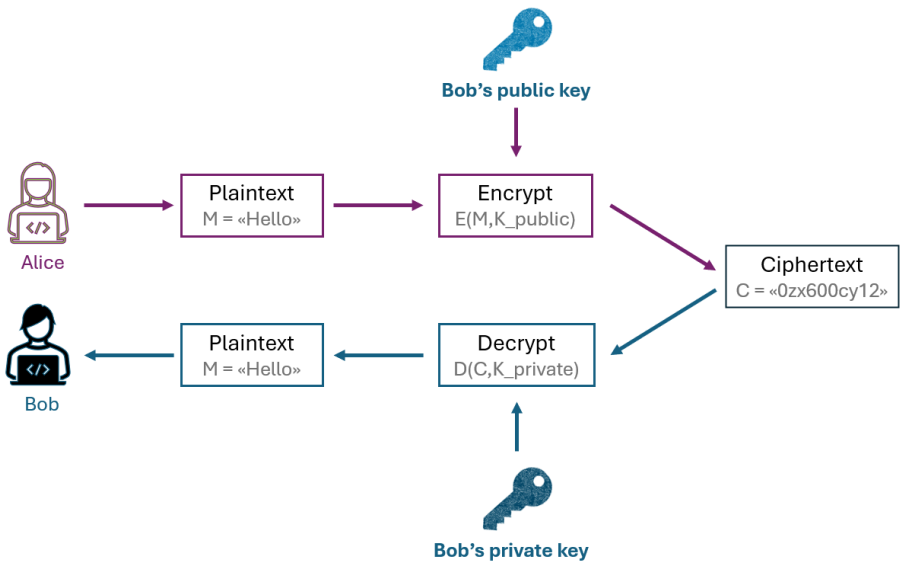


Figure 2.2: Encryption and decryption with Bob's key pair

graphic systems therefore involve circumventing this exhaustive search by looking for ways to deduce the secret key more efficiently.

In a symmetric system, the scheme should be designed in such a way that breaking a key of $n$ bits would take $2^n$ operations and therefore have $n$-bit security. In asymmetric systems where more information is necessarily made public, security level does not correspond to key size in the same straightforward way.

An asymmetric cryptographic system bases its security on hard mathematical problems. To be able to decrypt a ciphertext without possession of the correct key should entail solving a thoroughly studied mathematical problem of which there are currently no known efficient solution.

For example, the asymmetric RSA cryptosystem [75] is based on the *integer factorization problem*: given a composite number $N$, find the two prime integers $x$ and $y$ such that $x * y = N$. In RSA, the number $N$ is part of the public key and is used to encrypt information. The value of $x$ and $y$ is used to determine the private key used for decryption, and is kept hidden. Without knowing $x$ and $y$, it is almost impossible for an attacker to guess the secret key and decrypt the ciphertext, thus breaking RSA must also somehow entail solving the integer factorization problem.

The number of operations needed to find $x * y = N$ in RSA is necessarily dependent on the size of $N$, a parameter that is chosen when configuring an RSA system. An $N$ of 2048 bits provides a security level of 112 bits, given the algorithms available today, while an $N$ of 4096 bits provides 152-bit security [10].

## 2.3 Homomorphic Encryption

Generally, when a plaintext $X$ is encrypted into the ciphertext $C$, the primary quality of $C$ is that it can be decrypted back into $X$. In homomorphic encryption, the schemes are designed so that $C$ not only decrypts into $X$, but also retains the mathematical qualities of $X$ in the encrypted domain. With its mathematical structure preserved, any mathematical operation one would want to apply to $X$ can instead be applied to $C$, without decrypting. Any changes to the ciphertext will be reflected when it is decrypted back into plaintext.

Let $E$ and $D$ be the encryption and decryption functions in a homomorphic scheme, $x$ and $y$ two plaintexts, and $E(x), E(y)$ their respective ciphertexts. Fig. 2.3 demonstrates how the same mathematical operation (here addition) can be applied to the inputs in the plaintext and encrypted domain, and yield the same result.

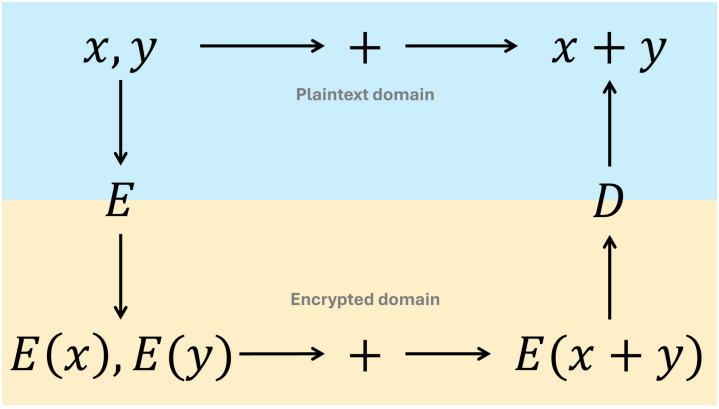


Figure 2.3: Addition operation evaluated homomorphically

There are many encryption schemes that preserve some mathematical qualities of ciphertexts, and thus allow for some operations to be evaluated homomorphically. However, they are often restricted in type and extent of operations, for instance only supporting either additions or multiplications, or a certain amount of operations [57]. An example of such a system is RSA.

In RSA, a plaintext $m$ is encrypted into a ciphertext by raising the value of $m$ to the $e$-th power, and ${m_1}^e * {m_2}^e = (m_1 * m_2)^e$. Since the encryption of the product of the two plaintexts is the same as the product of the ciphertexts, RSA is homomorphic with respect to multiplication. It is, however, not

homomorphic with respect to addition, as ${m_1}^e + {m_2}^e \neq (m_1 + m_2)^e$. RSA is therefore referred to as a *partially* homomorphic encryption scheme.

A scheme that can do both addition and multiplication could in theory evaluate any mathematical expression. Any computation can be modeled as an arithmetic circuit, and any arithmetic circuit can be described as a polynomial, consisting only of additions and multiplications. Fig. 2.4 shows an arithmetic circuit, with nodes representing variables that pass through a series of *gates*, each gate signifying either addition or multiplication of the nodes that pass through it. The circuit is described as the polynomial $x_2^3 x_1 + x_2^2 x_1$. The highest number of consecutive operations in a circuit is referred to as *depth*. Computations can also be modeled as a *boolean circuit*, where each gate represents a binary operation instead of multiplication or addition.

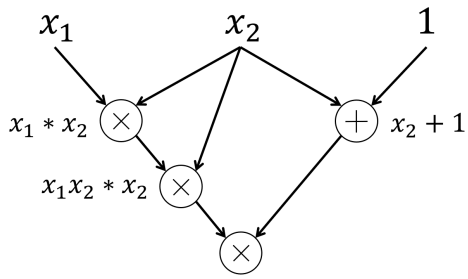


Figure 2.4: An arithmetic circuit of depth 3

In practice, a homomorphic scheme that supports both addition and multiplication might still be limited in the amount of operations it can realistically perform. This is because such systems also require the introduction of *noise* to be secure. Noise is small random errors that are added during encryption to mask the secret information. When ciphertexts are added or multiplied with each other, the noise of each ciphertext is also added or multiplied, and so the noise grows with every consecutive operation. Noise grows especially fast when two ciphertexts are multiplied, as the product of two noise elements will generally be larger than the sum of the same elements.

If the noise grows too large, the decryption will fail and the secret information can no longer be retrieved. Thus, the multiplicative depth of a

circuit generally indicates how challenging it will be to evaluate it homomorphically.

The management of noise is an ongoing challenge in homomorphic systems, and the strategy of handling it is a key component in the various encryption schemes. Some systems can only evaluate expressions of a certain complexity before the noise grows too large, some carefully set parameters ahead of time to accommodate the expression they want to evaluate, while others use the technique of *bootstrapping*.

Bootstrapping is a noise-reduction method that essentially ”refreshes” a ciphertext through re-encryption and then decryption. This technique enables a scheme to reduce noise whilst staying in the encrypted domain, but has the drawback of being very computationally demanding.

Homomorphic encryption (HE) schemes can be divided into *somewhat*, *leveled* and *fully* homomorphic encryption schemes, based on how they manage noise:

- A somewhat homomorphic encryption scheme only supports a bounded number of operations. If the bound is superseded, the noise grows too large for correct decryption of the ciphertext. The bound leaves the scheme unable to evaluate polynomials that require too many operations.

- A leveled homomorphic encryption scheme supports the evaluation of circuits up to a predefined depth. The level of a scheme generally refers to the multiplicative depth (the number of consecutive multiplications) of the circuits that can be evaluated and decrypted correctly. The level is set through parameter choices. In contrast to a somewhat homomorphic scheme, a leveled scheme is not limited by a fixed bound of operations, but by parameters that must be appropriately chosen beforehand to handle the evaluation of a circuit.

- A fully homomorphic encryption (FHE) scheme supports the evaluation of arbitrary circuits, typically through bootstrapping.

The idea of an encryption scheme that allowed unlimited and unrestricted calculations on the ciphertexts was long thought to be impossible, until Craig Gentry proposed such a scheme in his PhD thesis in 2009 [35]. Here, Gentry presented a scheme that would allow for homomorphic evaluation of any

arbitrary circuit. Furthermore, he introduced a way to transform a somewhat homomorphic scheme into a fully unrestricted one through the process of bootstrapping.

Following this breakthrough, several new homomorphic schemes have been introduced that expand on Gentry's initial contribution. Most notably among these are the BGV scheme introduced by Brakerski, Gentry and Vaikuntanathan in 2012 [16], the TFHE scheme by Chillotti, Gama, Georgieva and Izabachene in 2016 [27], and the CKKS scheme by Cheon, Kim, Kim and Song in 2016 [24, 25]. The two latter schemes are the main focus of this thesis.

While the modern schemes following Gentry's breakthrough are able to evaluate arbitrary circuits homomorphically, they still struggle with high computational costs which increase running time, and large ciphertexts that result in high memory usage. Much of the research in the field is aimed at overcoming these challenges, in order to make these schemes more practical for a wider range of applications.

### 2.3.1 Security of Homomorphic Encryption Schemes

Homomorphic encryption (HE) schemes belong to a family of schemes referred to as *lattice based cryptography*. Lattices are mathematical structures upon which hard mathematical problems can be defined, such as the *Shortest Vector Problem* (SVP) and *Learning With Errors* (LWE). HE schemes in use today base their security on LWE and variants of LWE, which can be used to construct asymmetric schemes conjectured to be quantum secure [70].

#### Lattices

A vector consists of direction and magnitude in a vector space, which can be expressed through *bases* and *scalars*. Fig. 2.5 depicts a vector $\vec{v}$ constructed of basis $(\vec{b_1}, \vec{b_2})$ and scalars $a_1, a_2 \in \mathbb{R}$. The basis spans the vector space $\mathbb{R}^2$: Every vector in $\mathbb{R}^2$ can be expressed with $(\vec{b_1}, \vec{b_2})$ and $a_1, a_2 \in \mathbb{R}$. This gives the following definition: $\mathbb{R}^2 = \{a_1\vec{b_1}, a_2\vec{b_2} \mid a_{\in}\mathbb{R}\}$. A given vector space can have many bases, as exemplified in Fig. 2.6, where the basis $(\vec{d_1}, \vec{d_2})$ is used instead.

A lattice is a vector space that is spanned by a basis where every scalar is an integer: $L = \{a_1\vec{b_1}, .., a_m\vec{b_m} \mid a_i \in \mathbb{Z}\}$. Constructing a lattice from pre-

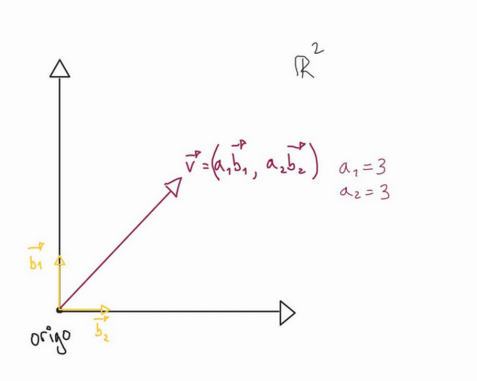


Figure 2.5: A vector in a two-dimensional vector space

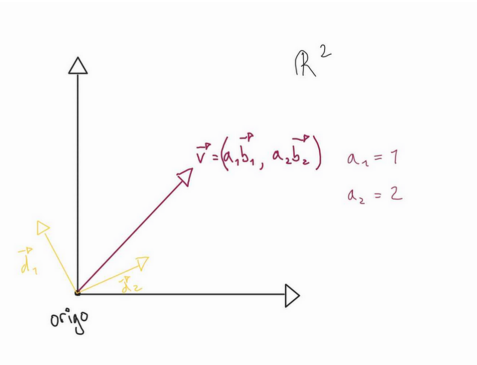


Figure 2.6: The same vector as in Fig. 2.5, constructed from a different basis

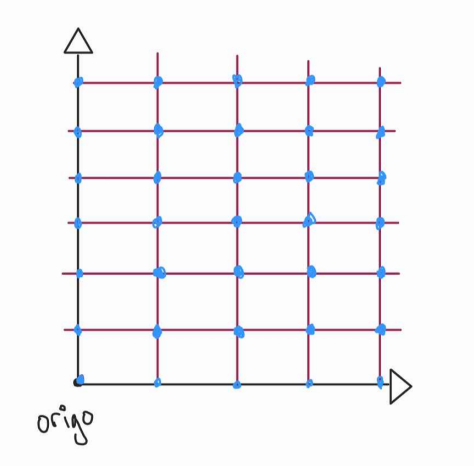


Figure 2.7: A lattice constructed from integer scalars in $\mathbb{R}^2$

viously shown $(\vec{b_1}, \vec{b_2})$ and instead using scalars $a_1, a_2 \in \mathbb{Z}$ would result in a lattice of points, as opposed to the whole continuous vector space $\mathbb{R}^2$. Fig. 2.7 depicts an example of a lattice.

### Shortest Vector Problem

In SVP, the problem is to find the shortest vector in a lattice $L$, measured from the origin. The difficulty of this problem depends on the basis at hand to construct $L$. Recall that a vector space (and therefore also lattices) can have many bases. Fig. 2.8 shows two such bases for a lattice. The first basis is considered a "good" basis: the vectors are short, and approximately at a $90^\circ$ angle to each other. The second basis is a "bad" basis: the vectors are long, and only deviates from each other by a small degree.

When the dimensions of the lattice grow, finding the shortest vector in the lattice only knowing a bad basis is almost impossible: There are no known efficient algorithms that can solve this problem in a feasible amount of time.

### Learning With Errors

LWE is a lattice-based problem that is conjectured to be at least as hard as SVP: Finding an efficient solution for LWE would also yield a solution to SVP [70].

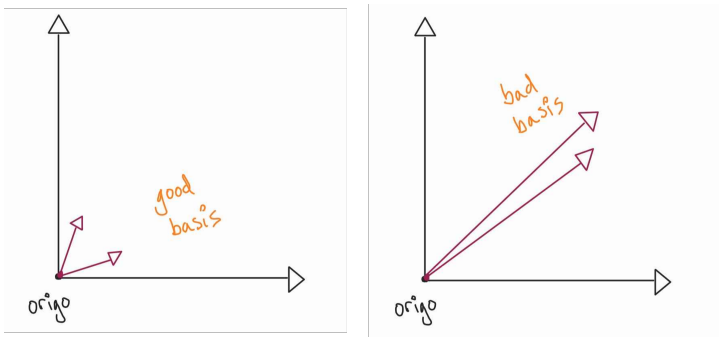


Figure 2.8: A good and a bad basis for SVP

The following building blocks are used in LWE: A matrix $A$ of integers sampled uniformly at random, a randomly sampled secret key $\vec{s}$, and a small error $\vec{e}$, sampled from a Gaussian distribution, generally referred to as noise. More specifically, these variables are defined as follows:

$$\underset{m \times n}{A} \in \mathbb{Z}_q, \quad m > n \tag{2.1}$$

$$\vec{s} = (s_0, \ldots, s_{n-1}), \quad s_i \in {\mathbb{Z}_q}^n \tag{2.2}$$

$$\vec{e} \in {\mathbb{Z}_q}^m, \quad \text{where } ||e|| \text{ is very short} \tag{2.3}$$

In the LWE problem, $(A, b)$ is known, where $A$ is the sampled matrix and $b = A\vec{s} + \vec{e}$. A potential attacker will try to find the secret $\vec{s}$. However, to find $\vec{s}$ one must also find $\vec{e}$, the second unknown in the equation. Note that there is a unique solution to the equation $A\vec{s} + \vec{e} = b$, meaning there is only one $\vec{e}$ that will give the solution. Finding the right $\vec{e}$ can be reformulated as finding the shortest vector in a lattice using a bad basis, where $\vec{e}$ will be the shortest vector.

### Ring Learning With Errors

*Ring Learning With Errors* (RLWE) is a variant of LWE where the vectors in $A$ and $\vec{s}$ are given as polynomials. The coefficients $a$ are sampled from $\mathbb{Z}_q$ and the polynomials are reduced modulo $x^n + 1$, such that $a, s \in \mathbb{Z}_q[x]/x^n + 1$. By only using the first row polynomial, the whole matrix can be generated through a series of algebraic steps. As a result, the first row provides sufficient information for encryption.

In public key systems, $a$ is used in the construction of the public key. Using the ring variant with $a$ as a polynomial thus reduces the size of the public key, which in turn increases efficiency.

### 2.3.2 TFHE

TFHE is an FHE system that supports homomorphic calculations on integers, and bases its security on LWE and variants of this such as RLWE. The main building blocks of any FHE scheme are the encoding of input messages into plaintexts, the encryption and decryption of plaintexts and ciphertexts, the homomorphic operations of addition and multiplication of ciphertexts and constants, and a bootstrapping procedure to manage noise.

Recall that any polynomial can be evaluated through a series of additions and multiplications, and as such an FHE scheme can evaluate any polynomial. A key property of TFHE is that its bootstrapping procedure *also* allows the evaluation of univariate non-polynomial functions on ciphertexts, for instance functions where a variable has a negative exponent, such as $f(x) = 1/x$.

Another special property of TFHE is its use of several types of ciphertexts, using different representations of the encrypted values for different types of calculations. The following sections are based on a series of blog posts by Ilaria Chillotti [26].

#### Encoding and Decoding of Messages

Any input message $m$ is encoded using a scalar $\Delta$, such that the plaintext of the message is $\Delta m$. The scalar is calculated using the ciphertext modulus $q$ and the plaintext modulus $p$, so that $\Delta = q/p$. A message is decoded by dividing the sum of the scaled message and the noise element by the scalar and then rounding to the nearest integer, $\lfloor \frac{\Delta m+e}{\Delta} \rceil$.

When the ciphertext and plaintext moduli are expressed as powers of 2, the exponents denote how many bits of information can be stored in the ciphertext and plaintext respectively. Let $q = 64 = 2^6$ and $p = 4 = 2^2$ in a toy example. Then the ciphertext will have a length of 6 bits total. The plaintext will have 2 bits available, meaning it can hold any message that can be expressed using 2 bits, that is $m \in \{0,1,2,3\}$ (or, in binary, $m \in 00,01,10,11$). The scalar is $\Delta = q/p = 16 = 2^4$. To correctly retrieve the message upon decoding, the error must be less than half of $\Delta$. Here, this would mean that $|e| < \Delta/2 = 8 = 2^3$, i.e., the error cannot exceed 3 bits.

Fig. 2.9 depicts how this encoding would look as a sequence of bits, with the message stored in the most significant bit(s) (MSB) to the left and the error in the least significant bit(s) (LSB) to the right. Another way to

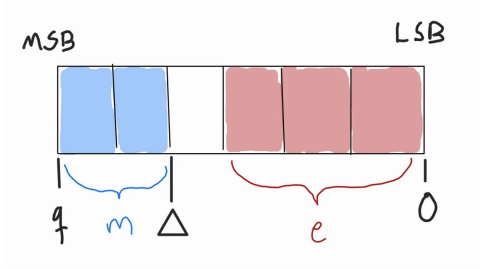


Figure 2.9: Encoded message as a bit sequence

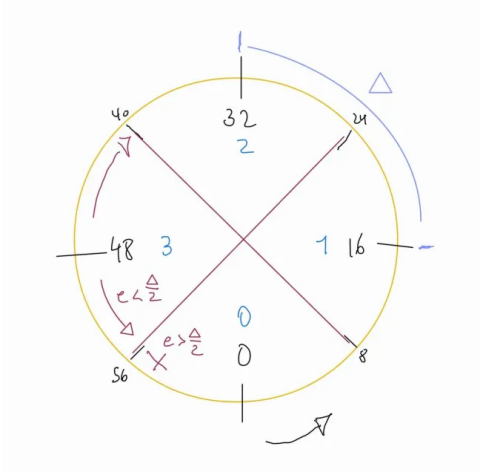


Figure 2.10: Encoded message over the torus

visualize this encoding is shown in Fig. 2.10. The torus depicts $\mathbb{Z}_q$, in which all the coefficients of the ciphertext belong. The messages $\{0,1,2,3\}$ are the four possible messages, and the scalar is seen as the distance between each message in $\mathbb{Z}_q$. The red cross that divides the four message sections marks the error margins. If the error exceeds the margin, the message will be decoded incorrectly as either the next or the previous value in the torus.

A staple of many FHE schemes is *modulus switching*, where the modulus $q$ is reduced in order to remove some of the accumulated noise from the ciphertext. Using the sequence of bits in Fig. 2.9 as an example, a modulus switching operation in TFHE entails ”compressing” the bit sequence; the total sum of error bits is reduced, but the distance between the plaintext bits and the error bits in the sequence is also reduced. In TFHE, modulus switching therefore entails reducing the overall noise, but at the same time

moving the noise elements closer towards the limit, beyond which decryption is no longer possible.

### Encryption and Decryption

TFHE bases its security on both LWE and its variants. The secret key $\vec{s}$ consists of $n$ randomly sampled coefficients from $\{0,1\}$, the vector $\vec{a}$ is $n$ values sampled uniformly at random from $Z_q$, and $b$ is calculated as the inner product of $\vec{a}$, $\vec{s}$ and a small error $e$ taken from a Gaussian distribution. To encrypt a message $m$, such as a bit or a modular integer, $m$ is first encoded and then added to $b$. To decrypt, the encoded message and the error is separated on one side of the equation. The message $m$ is retrieved by decoding the message, essentially cutting off the part of the encoded message containing the noise. This is also described in equations Eqs. (2.4) to (2.10).

$$\vec{s} = (s_0, ..., s_{n-1}), s_i \in \{0,1\}^n \tag{2.4}$$

$$\vec{a} = (a_0, ..., a_{n-1}), a_i \in \mathbb{Z}_q \tag{2.5}$$

$$b = \langle \vec{a}, \vec{s} \rangle + e + \Delta m \tag{2.6}$$

$$e = \text{small random noise} \tag{2.7}$$

$$Enc(m) = (\vec{a}, b) \tag{2.8}$$

$$Dec(m) = b - \langle \vec{a}, \vec{s} \rangle = \Delta m + e \tag{2.9}$$

$$\lfloor \frac{\Delta m + e}{\Delta} \rceil = m \tag{2.10}$$

### Ciphertext Representations

In TFHE, and in FHE schemes in general, there is an operation called *key switching*, that essentially allows one to switch the secret key of an encrypted message, while the plaintext stays the same. In TFHE, the key switching also allows for the changing of parameters, and as such gives the option to change ciphertext representation.

Both encoding/decoding and encryption/decryption has so far been explained in terms of what is, in TFHE, called LWE ciphertexts. This is only one type of ciphertext used in TFHE. For instance, where LWE ciphertexts

encrypts a single $m$, which is a bit or a modular integer, another type called an RLWE ciphertext will encrypt a polynomial. The purpose of changing how the encrypted data is represented is to be able to match the ciphertext structure with a mathematical operation. It is for this reason that the TFHE scheme switches between several types of ciphertexts.

### Programmable Bootstrapping

The only thing that removes noise completely from an FHE ciphertext is decryption. Bootstrapping is therefore, simplified, to evaluate the decryption of a ciphertext, while still keeping the ciphertext encrypted. It is a form of ”re-encryption”, where the result is a new ciphertext, encrypting the same plaintext as the old one, but with a reduced level of noise. Bootstrapping procedures generally involve several homomorphic operations and are computationally demanding, and this is the case also in TFHE. Furthermore, the homomorphic operations in themselves generate noise, and the parameters must therefore be carefully chosen so that the bootstrapping procedure does not end up generating more noise than it removes.

In TFHE, bootstrapping is done on LWE ciphertexts, using a bootstrapping key that consists of a different kind of ciphertext. The modulus of the LWE is switched, and the output and the bootstrapping key is used in a blind rotation together with a *lookup table* (LUT) encoded as a trivial general LWE polynomial. The operation is complex, but essentially the LUT polynomial is rotated in a specific manner, resulting in a ciphertext where the constant coefficient can be extracted as an LWE, which is the encryption of $f(m)$. If $f$ is the decryption circuit, then the LWE will be a new ”bootstrapped” ciphertext of $m$. However, the LUT can be any univariate function, which will be evaluated without any additional cost to the original bootstrapping. This programmable bootstrapping is what allows the TFHE scheme to evaluate non-polynomial functions while bootstrapping. In Zama’s TFHE implementation in Rust [97], a LUT can be represented as a list of predefined values. This list then takes in the encrypted value, $x$, and uses it as an index to ”look up” the correct value in the LUT.

### State Today

TFHE has some unique properties, especially with regards to its support for non-polynomial functions in the programmable bootstrapping. It can

provide exact homomorphic calculations, but works on integers and therefore does not readily support floating point arithmetic. In contrast to other FHE schemes like BGV and CKKS, TFHE lacks the option to pack a large amount of values into the same ciphertext and evaluate the values in parallel.

TFHE implementations of high quality exists, with Zama providing industry-level implementations in both Rust [97] and Python [96, 95], together with extensive documentation [94]. There are still implementation-specific restrictions on TFHE, for instance in the size of integers supported in an encrypted circuit, consequently also limiting how large LUTs can be.

### 2.3.3 CKKS

CKKS is an FHE scheme that takes complex numbers as input messages. Like TFHE, it bases its security on LWE and its variants, more specifically on RLWE. A CKKS ciphertext is structured as a vector of a given number of slots, where each slot holds a value and all slots can be evaluated in parallel. This possibility of *packing* a ciphertext full of different values is defining for CKKS: It can be used to evaluate potentially thousands of data points at the same time, albeit limited by the size of the ciphertexts and the memory available on the system that runs it.

CKKS performs encoding and decoding between the domain of complex numbers and real numbers, which results in small discrepancies between the original message and the final decoded one. Rounding and scaling methods are used when encoding messages and to manage the influence of noise from encrypted operations. This makes CKKS an approximate scheme where the ciphertexts are decrypted and decoded into an approximation of the messages with a defined level of precision.

CKKS can be implemented as a leveled scheme or with bootstrapping. The latter is not necessarily supported in well-known CKKS libraries such as Microsoft's SEAL [82]. The following sections are based on CKKS as a leveled scheme, as described in a blog post by the non-profit project OpenMined [41], which is behind the SEAL extension library TenSEAL [11].

#### Encoding and Decoding of Messages

In CKKS, the noise and message in a ciphertext is not separated, as illustrated in Fig. 2.11. When a message is decrypted, the noise element is still present and affect the accuracy of the result. If the noise is small enough in proportion to the message, the difference in accuracy is negligible. A scaling factor $\Delta$ is used in the encoding of a message. By scaling the message, the parts of the encoded plaintext containing the message is moved towards the MSB. Furthermore, this increases the accuracy of the coding, as encoding involves rounding to the nearest integer and decoding divides the plaintext by the scale: By scaling, the rounding operation can retain accuracy of real numbers to several decimal points, depending on the size of $\Delta$, as opposed to in practice only rounding to integers and losing further precision.

The encoding of a message into a CKKS plaintext thus takes as input a message $m$, which is a vector of complex numbers. This is encoded using the

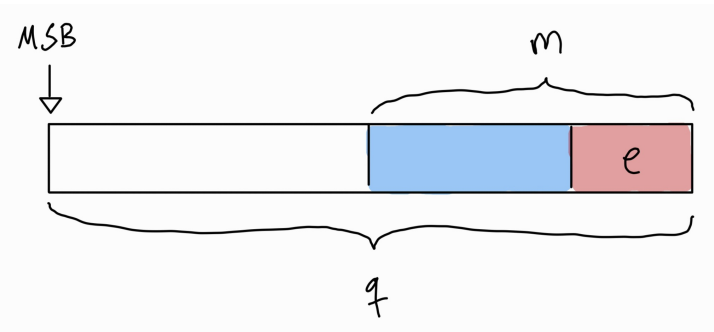


Figure 2.11: CKKS ciphertext structure, adapted from [24]

scale factor, a rounding operation and a transformation called a canonical embedding $\pi$ to gain a polynomial plaintext in the space of real numbers, as shown in Eqs. (2.11) and (2.12).

$$Encode(m,\Delta) = \lfloor \Delta \cdot \pi^{-1}(m) \rceil \tag{2.11}$$

$$Decode(m',\Delta) = \pi(\frac{1}{\Delta} \cdot m') \tag{2.12}$$

**Encryption and Decryption**

CKKS encryption and decryption works on the same base principles as shown for TFHE in the LWE ciphertext example. CKKS uses RLWE ciphertexts only, and so the secret key $\vec{s}$, the uniformly sampled matrix $A$ and the small random noise $\vec{e}$ are polynomials $s$, $a$ and $e$ instead of vectors, sampled from $\mathbb{Z}_q[X]/(X^N+1)$.

Using polynomials instead of vectors reduces the size of $b$ and $a$, which are used as the public key $pk = (b,a)$ in asymmetric implementations. The message $m$ is appended to $b$ for encryption. Thus, we have encryption and decryption as follows in Eqs. (2.13) and (2.14).

$$Encrypt(m) = (m+b,a) = (c_0,c_1) \tag{2.13}$$

$$Decrypt(c) = c_0 + c_1 \cdot s = m + e \approx m \pmod{q} \tag{2.14}$$

When two ciphertexts are multiplied by each other, the terms in the ciphertext polynomials grow exponentially with subsequent multiplication,

resulting in inflated ciphertext growth. To avoid this, a *relinearization* procedure is used to keep the ciphertext size constant. Relinearization uses a specially designed evaluation key that reshapes the ciphertext and avoids exponential growth, and is done after every such multiplication.

### Levels and Rescaling

CKKS schemes use a predefined set of levels, where each level is defined by the size of the current ciphertext modulus $q$, ranging from the size for a fresh ciphertext, $q_L$, down to the last level of an applicable modulus size $q_1$. The number of levels bounds the number of subsequent multiplications that can be applied to a ciphertext. When CKKS is implemented without bootstrapping, it is a leveled scheme. A CKKS bootstrapping procedure is a computationally expensive procedure to "refresh" the levels of a ciphertext, and thus a leveled scheme that avoids this operation is generally faster.

Recall that every message $m$ is scaled by scaling factor $\Delta$ during encoding. When ciphertext polynomials are multiplied, both the noise and the scaling factor grow, with $\Delta$ multiplied first to $\Delta^2$, then $\Delta^4$ and so forth in exponential growth. To keep the scaling factor constant and manage noise, a *rescaling* operation is conducted. Rescaling is simply stated a way to gradually reduce the noise by reducing the ciphertext modulo $q$, the same idea as the modulus switching technique described for TFHE. Let $q_L$ be the ciphertext modulus of a fresh ciphertext, $q_i$ be the ciphertext modulus of a given level and $\Delta$ be the scaling factor. Then, a naive interpretation of rescaling would be $q_{L-1} = \frac{q_L}{\Delta}$. Each new ciphertext will have a reduced modulus, reducing noise, and $\Delta$ remains constant. In practice, the scaling factors and polynomials involved can be quite large, and so the different modulus values of the ladder are chosen carefully so that the Chinese Remainder Theorem can be used to divide up the calculations into smaller pieces.

Rescaling is described in Fig. 2.12: A ciphertext is rescaled from $q_l$ to $q_{l-1}$. Note that in rescaling, it is the LSB side (right hand side) of the ciphertext that is reduced. Since the noise $e$ and the message $m$ is not separated, $m$ is changed while the noise is also reduced. When multiplying two ciphertexts, they must be of the same scale. This can be achieved either by rescaling, or by a more straightforward modulus reduction where the scale is changed from $q_l$ to $q_{l-1}$ by reducing on the MSB side and leaving $m$ unchanged.

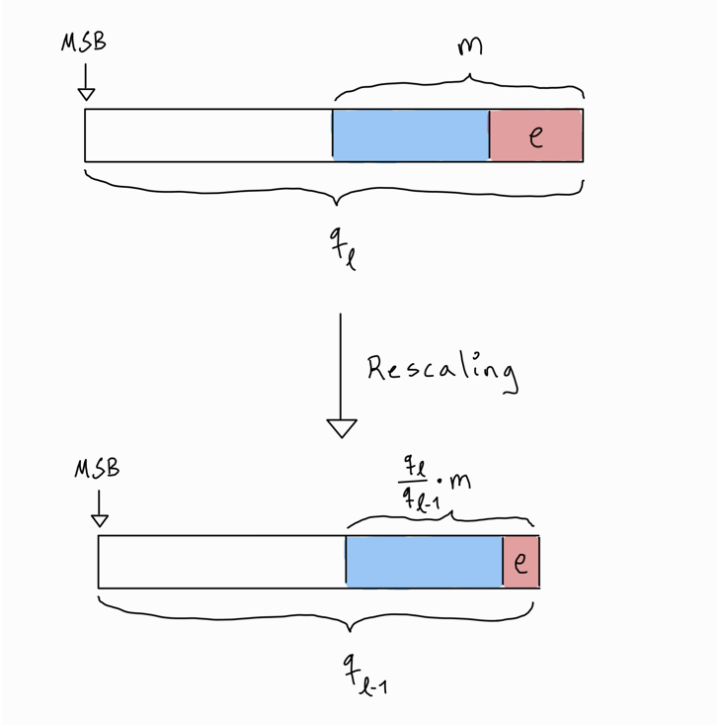


Figure 2.12: Rescaling operation, adapted from [24]

### Parameter Choices, Security and Ciphertext Size

There are some important parameters that must be considered when implementing a CKKS scheme. The scaling factor, for instance, denotes what precision of integer and decimal points will be used in the calculations, and is set as a power of 2.

Other parameters are the degree of the polynomials used, $N$, and the ciphertext modulus $q$ together with the number of levels. The security of a CKKS scheme depends on the ratio $N/q$. A problem that requires many multiplications (and therefore many levels) will need to set $q$ to a high value. Consequently, the value of $N$ must also be raised to maintain security, resulting in higher-degree polynomials which demand more memory and are heavier to compute. There is thus a trade-off between circuit complexity and computational efficiency.

The value of $N$ decides the size of the ciphertexts. A ciphertext consists of $N/2$ number of slots, where each slot can be filled with a value. If for instance $N = 2^{14} = 16384$, then the number of slots in the ciphertexts will

be $N/2 = 8192$. This results in a ciphertext that can take as input a vector of up to 8192 values, and store all values in separate slots. This capacity to take a vector of several values and pack them into a single ciphertext is called *batching*. Every operation applied to a CKKS ciphertext is applied in parallel to every slot, and so batching allows encrypted operations on thousands of values at the same time. The different values of the ciphertext cannot, however, be retrieved individually. Also, operations cannot be selectively applied to certain slots; once the ciphertext has been constructed, its integrity must be maintained.

### State Today

CKKS is an approximate scheme suited for problems with real numbers, which is the case for many real-world applications, but is for the same reason not suited in situations where exact precision is needed. The batching property makes CKKS efficient for encrypted evaluations of thousands of values in parallel, which can be especially useful to achieve good average calculation times. However, unlike TFHE, it does not support non-polynomial functions. Such operations must therefore be approximated using polynomials.

There are many available implementations of CKKS, such as the previously mentioned library SEAL [82] and OpenFHE [9], both in C++, and the closed-source library HEaaN [68]. There exists wrappers for the more data-science oriented programming language Python, such as PySEAL [88] and TenSEAL [11]. The latter is an example of libraries that also provide rudimentary additional functionality for more specific tasks, such as machine learning. Specific implementation limitations vary between libraries, for instance whether there exists a bootstrapping procedure, what sizes of ciphertexts are supported, and which parameters (if any) are automatically calculated and which must be manually set by the user.

## 2.4 Machine Learning

Machine learning (ML) is a field within artificial intelligence that studies statistical algorithms and how they can be used to find patterns in data and make predictions. Where the field of statistics use algorithms to find patterns in a sample of known data, machine learning concerns itself with generalizing these patterns so that they can be used to make predictions on so far *unseen* data [20]: Statistics is used to explain the data at hand, machine learning is used to explain the data one might get in the future.

A key property of a machine learning model is that it *learns* patterns from input data in an iterative process, continually getting a better *fit* to the data without explicit instructions. A widely used definition of what constitutes a machine learning algorithm by Tom M. Mitchell is as follows: ”A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at task T, as measured by P, improves with experience E.”[59, p. 2].

A model is first trained on a set of input data, before it is then tested on a separate, previously unseen, dataset to evaluate performance at the given task. Performance metrics might be accuracy of predictions, prevalence of false positives, similarity between group members and so forth. The learning approach of the model and algorithms used is chosen to best suit the task. Broadly, machine learning encompasses the methods of *supervised*, *unsupervised* and *reinforcement learning* [14].

Supervised learning is a learning approach where the model is given training data, $x$, with some label $y$ attached. Based on the values of properties of $x$ (called *features*) the task of the model is to predict the correct label, $y$. In training, the model iteratively improves itself by making predictions, comparing the prediction to the actual label, and updating itself based on its performance. Thus, the training data of a supervised learning model must be labeled with the correct ”answer”, and the model is tasked to find general patterns within the training data that can be used to predict the label of some new unseen input. A model that trains on a large amount of spam emails, identifies what separates a spam email from a non-spam email, and then predicts whether a new email is spam or not will engage in supervised learning [37]. The input features of a training sample would be properties of the email, such as length, inclusion of certain words or pictures, and the label indicating whether the email in question was spam or not.

Unsupervised learning is an approach where the data is given without any labels attached. The task is not to find patterns to predict the label of some input data, but rather to discover the inherent structure of the data. Unsupervised learning can for example be used to cluster data based on some similarity between samples or condense data while keeping the most meaningful properties [14, 72].

Reinforcement learning is the concept of training a model by giving the model feedback on the choices it makes, thus reinforcing behavior that maximizes the performance in accordance with the chosen metrics. A use case for reinforcement learning could be teaching an autonomous vehicle how to drive [28].

The experiment in this thesis uses a supervised learning approach with a type of deep learning model called a convolutional neural network, and subsequent sections will therefore focus on this method of learning and type of model.

### Supervised Learning

The supervised learning tasks that machine learning is used for can generally be divided into *classification* and *regression* tasks, with the models referred to as *classifiers* and *regressors* respectively. A classification task typically involves labeling some data according to a predefined set of classes; based on some properties of an object, what class does it belong to? Some such problems can be predicting the quality level of a red or white wine, based on properties such as sourness, sweetness, and alcohol level [33], or identifying bicycles in a picture [8]. A regression task concerns the prediction of numerical values; how tall will a sunflower grow given certain soil conditions [42]?

There are many different algorithms to employ for supervised learning, each with its specific design and implementation details. The general idea of a model can, however, be illustrated with a very simple linear regression model; $y = ax + b$. The label or value, $y$, is what the model aims to predict, based on the value of the input variable, $x$. Given a set of input samples to train on, the model will learn the general patterns of how $x$ relates to $y$. More specifically, the model will try to find the optimal values for $a$ and $b$, so that it best fits the data and can be used to predict the label of some new input $x$.

The $a$ variable is referred to as a *weight*, since it determines how much

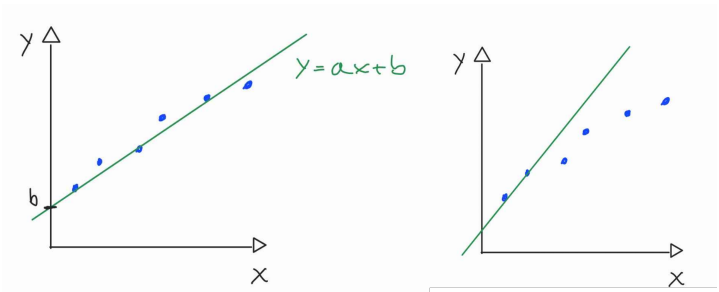


Figure 2.13: Two linear regression lines with different fit

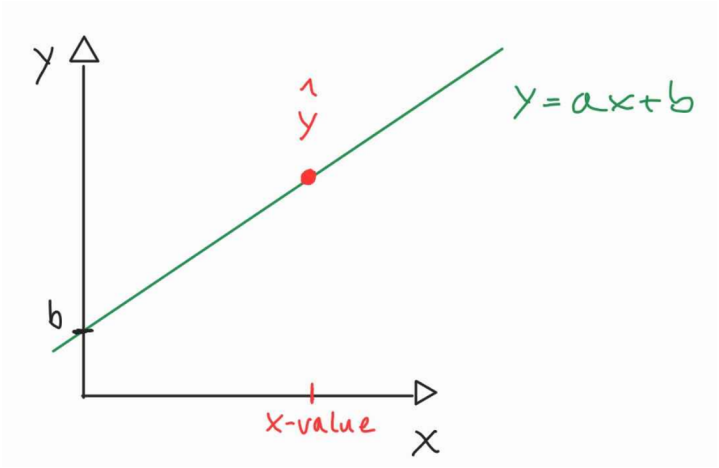


Figure 2.14: Prediction of $\hat{y}$ based on regression line

influence the value of $x$ should have. If $a$ is less than 1, then the influence of $x$ is diminished, else it is increased. The $b$ variable is a constant independent of $x$, referred to as a *bias*. A learned model is in essence a set of weights and biases that is fine-tuned for high performance in predicting $y$. Fig. 2.13 depicts two different regression models, where the bias is shown as the intersection between the regression line and the y-axis, and the weight as the slope of the line. The models fit the data differently. The first model fits the overall pattern of the blue data points well. The second model fits exactly to the first two data points, but fails to accurately model the remaining data points. In Fig. 2.14, a prediction $\hat{y}$ is made for a new $x$-value, based on the weight and bias of the regression line.

In the learning process, the model will have some initial values for the weights and biases, and use these values to make a prediction $\hat{y}$. A *loss*

*function* is used to calculate the difference between the predicted and the true label, and the goal is to alter the weights in a way to minimize the loss function.

The standard way of updating the weights is through the use of the *gradient descent* method. The core concept of gradient descent is to gradually alter the weights of the model, so that the model outputs move in the right direction to minimize the loss function and thus achieve better performance (see Fig. 2.15). The weights are altered by a learning rate, $\lambda$, essentially representing the size of the step in the given direction. Too high a learning rate might overstep the desired destination (i.e., the correct prediction) while too low a learning rate will take too long a time to arrive. Fig. 2.16 shows how a too high learning rate fail to reach meaningful loss reductions, while a too low rate improves little with each step.

Gradient descent is most commonly used in its optimized versions of *stochastic gradient descent* (SGD) [78]. In SGD, the gradient (the direction) is replaced by an estimate of the gradient, which speeds up computation. Today, the most common optimizers are extensions of SGD, notably SGD with momentum [78] and *Adam* [45].

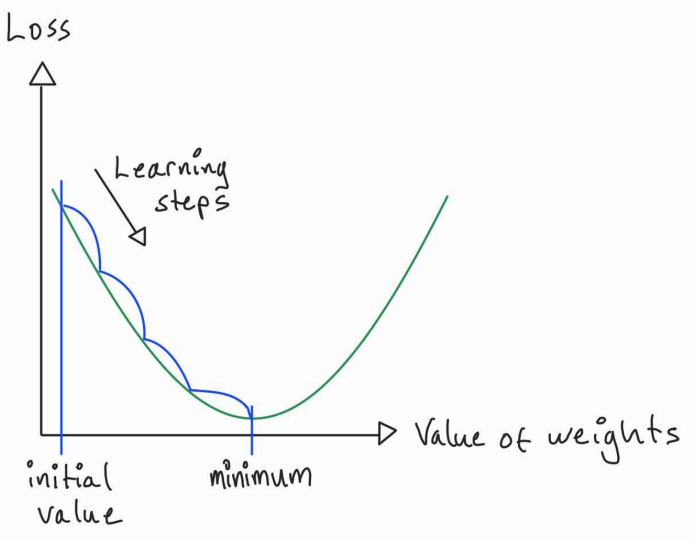


Figure 2.15: Gradient descent towards minimizing the loss function

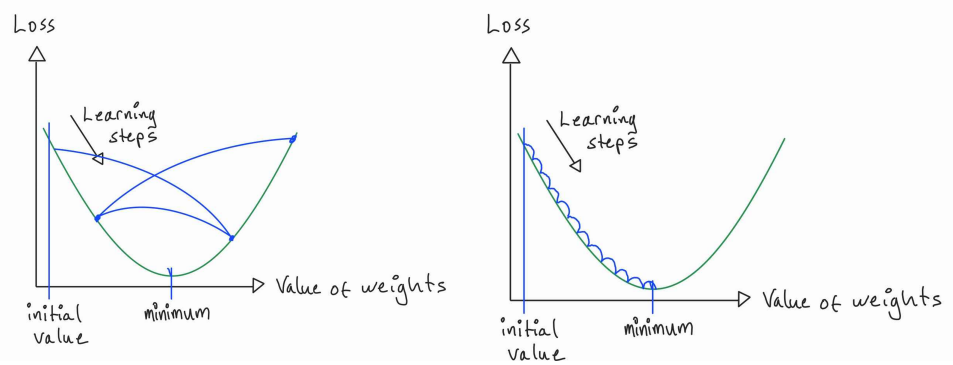


Figure 2.16: Very high and small learning rate, respectively

### 2.4.1 Deep Learning and Neural Networks

Deep learning is a subfield within machine learning dedicated to the study of a specific type of model, called neural networks [40]. Consider the earlier example of a simple linear regression model, $y = ax + b$. In the training process, the feature of the input sample, $x$, is passed into the equation with the current weight and bias to make a prediction, and then this prediction is used to update the learned variables. There is only one set of weights, and the prediction is calculated directly; it is a model of one *layer*. In deep learning, the neural networks are models of several layers, thus ”deep” models, where the output of each layer is passed on to subsequent layers.

The basic structure of a neural network is visualized in Fig. 2.17: An input layer that takes in the $n$ features of the input sample, $x_1, x_2, ..., x_n$, a hidden layer with its own set of weights that is applied to the inputs, and at the end an output layer that conducts the final evaluation to give a prediction. Any layer between the input and output layer is referred to as a hidden layer. Each node in the network is referred to as a neuron. Since each layer in itself can be construed as a linear model, only able to model linear data, a neural network would not be able to model non-linear relationships between variables were it not for the activation function. The activation function is a non-linear function applied to the output of a layer that introduces non-linearity into the model.

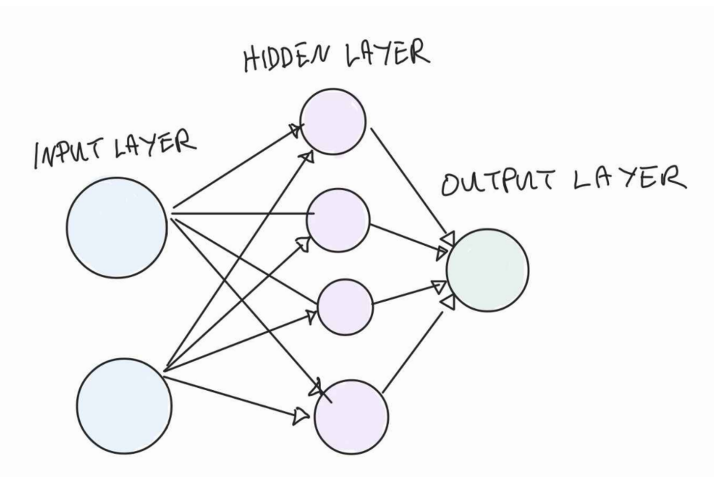


Figure 2.17: Basic neural network with one hidden layer

### Activation Functions

The most important attribute of an activation function is that it introduces non-linearity to the model. The different functions have, however, other properties that might make them more or less suited for a given task. The widely used *logistic sigmoid* function $f(x) = \frac{1}{1+e^{-x}}$ has, for instance, the property that the output range is bounded between 0 and 1, but a drawback is its inability to deal with vanishing small gradients and thus inefficient weight updates for very deep networks [32].

Another commonly used function, the *Rectified Linear Unit* (ReLU) [62] function $f(x) = max(0,x)$ addresses this issue, but has its own drawbacks by for example the possibility of neurons not contributing as desired to the learning process.

There are also activation functions typically applied to the final output layer of a model, such as the *softmax* [17] function

$$f(x_i) = e^{x_i} / \sum_{j \in \text{classes}} e^{x_j}.$$

This function takes a vector of real numbers and translates it into a probability distribution, which is useful when the goal of the model is, for example, to output a prediction of what is the most probable class for a given input sample.

### Feedforward and Recurrent Neural Networks

Neural networks can be divided into *feedforward* and *recurrent* neural networks, based on how information is passed through the network. In a feedforward neural network, the values are passed once through each layer, the neurons connecting to the neurons in the next layer, forward towards the output layer. In training, the weights are updated through the process of *backpropagation* [55, 51], a gradient estimation method. The network estimates the gradient based on a *loss function*, which measures the distance between the prediction and the actual value. To avoid unnecessary calculations and improve computation speed, the gradient is calculated one layer at a time, starting from the final layer and moving backwards. Fig. 2.18 shows how inputs $x_1$ and $x_2$ are fed forward through a hidden layer to the output layer, and how the output layer loss is then passed backwards through the network to update the weights.

In a recurrent neural network, information is not passed a single time through the network, but instead processed through multiple steps. Neurons can connect to other neurons in the same or previous layers. The networks have recurrent units that remember previous states and uses these to create a feedback loop that enables learning.

## 2.4.2 Convolutional Neural Networks

A *convolutional neural network* (CNN) is a type of feedforward network. Often in such networks, every neuron in one layer is connected to every neuron in the next, so called *fully connected* networks. They can be prone to overfitting to the training data, that is adapting so well to the training data that the model cannot generalize properly to new unseen data.

In CNNs, the input is passed through convolutional layers that act as a filter on the inputs. This allows the model to filter out unnecessary details from the data and find broader and more general patterns. In CNNs, the spatial patterns of the data, pertaining to how data is arranged and distributed across a two-dimensional space, are taken into account in the learning process. This makes such models especially suitable for image classification tasks and other computer vision problems.

A CNN is built from an input layer, hidden layers and the final output layer. The hidden layers that are typically used are convolutional layers

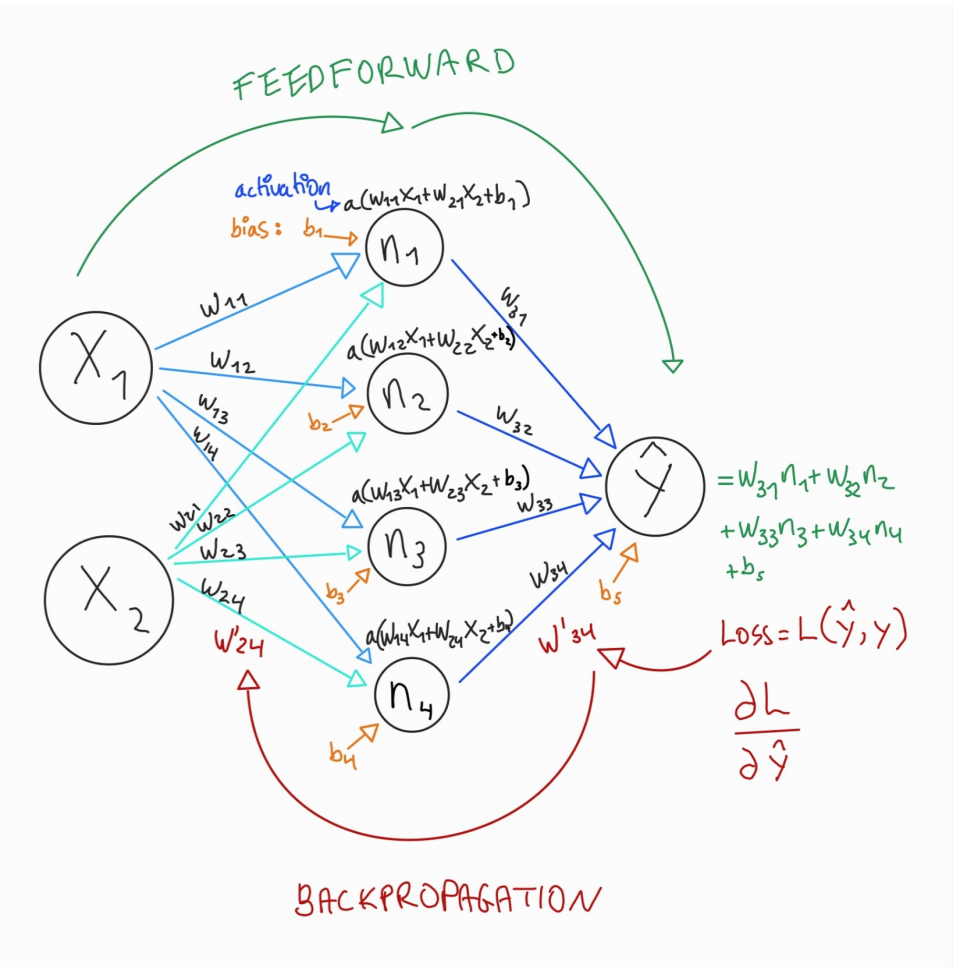


Figure 2.18: A feedforward network with backpropagation, where red weights are examples of updated weights.

where the input is filtered, pooling layers to condense data, and dense layers of fully connected neurons [64]. Activation functions are generally applied on the intermediate outputs of a convolutional or dense layer to introduce nonlinearity, with ReLU (or variants of ReLU) being widely used [53]. For classification problems, softmax activation is often applied to the final output to interpret the class probabilities.

Generally, a CNN extracts features from the data through a convolutional layer, introduces nonlinearity with an activation function, and downsample (selecting only parts or aggregated parts of the data) through pooling layers.

A simple CNN architecture is depicted in Fig. 2.19.

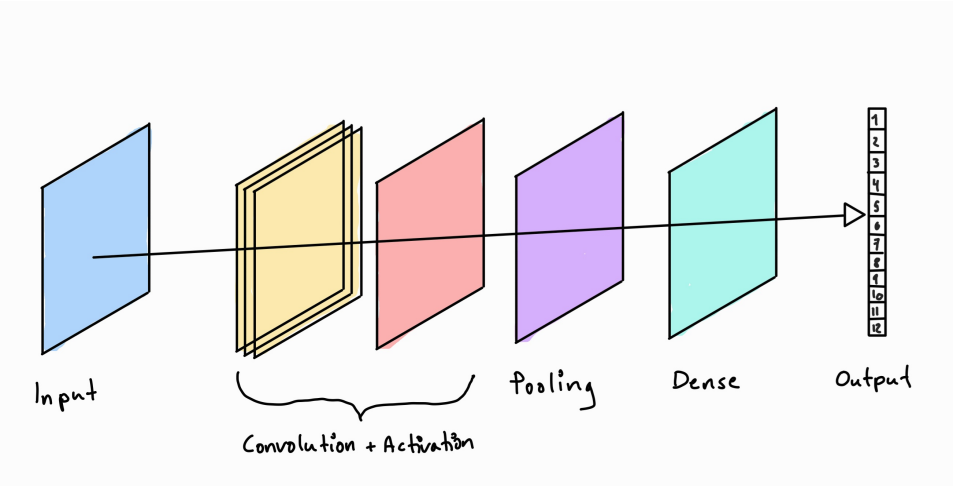


Figure 2.19: Different kinds of layers in a CNN

**Convolution** A convolutional layer applies a filter, or *kernel*, of weights to the input data, and store the output in a feature map. CNNs are often used on images, with an image being represented as a matrix of values. For a grayscale image, each value denotes the scale of gray for the given pixel, ranging from 0 (white) to 255 (black). Color images are generally represented using three such matrices, with each matrix representing the scale of red, blue or green. The matrices are then ”stacked” on top of each other to render a color image. A matrix used to represent an image is called a channel, with grayscale images being one-channel and color images 3-channel images.

Fig. 2.20 shows an example of a convolutional layer, where the input vector is processed as a one-channel grayscale image; a 4x4 matrix with a total of 16 values. In the example, the kernel is a 2x2 matrix of weights. The kernel passes over the first 2x2 square of the input matrix - the green square - and calculates the sum of the element-wise multiplication of the green square and the filter. Conversely, this would be the same as finding the dot product between the green square and the filter, if they were processed as one-dimensional vectors. This value is then stored in a feature map.

The kernel will then continue to slide along the input vector, moving a predetermined amount of steps, or *stride*, to the right for each iteration.

The example feature map is the result of a kernel moving two steps to the left for each iteration, thus covering the four colored windows in turn, and calculating four values to put in the feature map. For each kernel in a convolutional layer, there will be a resulting feature map, and a convolutional layer can have many kernels.

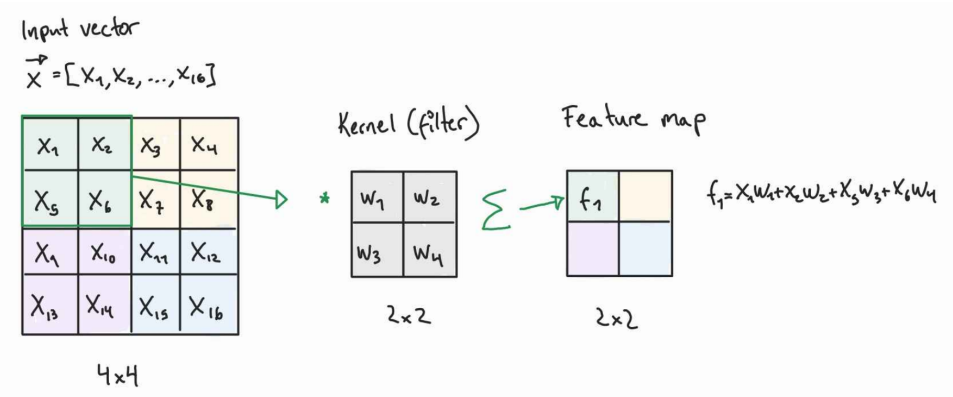


Figure 2.20: A simple convolutional layer

**Pooling** The pooling layer is a layer that downsamples the input by reducing several values to one. Common approaches are maximum pooling [30] and average pooling [49]. In a convolutional layer, different ”windows” of the input are multiplied with the kernel of weights and placed in a feature map. In maximum pooling, the maximum value of the window is placed in the feature map. Average pooling, on the other hand, places the average of the window values in the feature map; essentially performing a convolution with a kernel where the weights are replaced with a replicated decimal value (see Fig. 2.21).

**Dense layers** A dense layer is a fully connected layer where each neuron connects with each neuron in the next layer, such as the previously depicted hidden layer in Fig. 2.17. They are often used in CNNs to ultimately gather the various local patterns of the convolutional and pooling layers into a layer suitable for representing the whole of the input - and from that make a prediction. A dense layer at the end of a CNN classifier will have the same number of neurons as the number of classes the classifier is operating with.

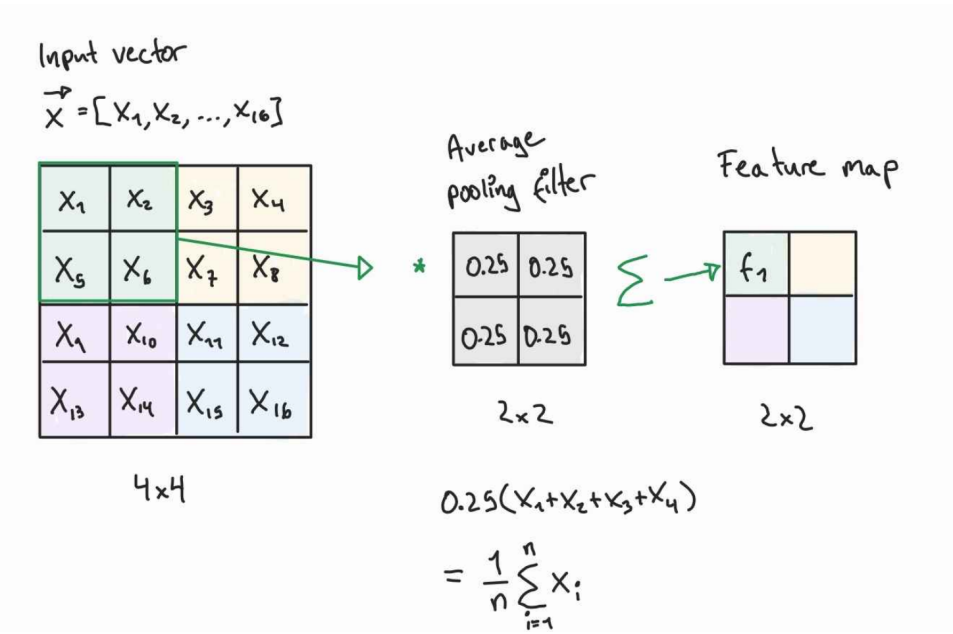


Figure 2.21: Illustration of an average pooling layer

### CNNs and Image Classification

CNN models are top performers on image classification benchmark datasets and challenges, such as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [80] where images from the internet are sorted into a thousand different classes. Here, the CNN AlexNet [47] won in 2012 in a breakthrough of computer vision performance. Other notable CNN models used for the challenge are the 2014 winner GoogLeNet [86] and runner-up of the same year VGG [83], as well as Microsoft's ResNet [38] model which won in 2015.

The architecture of CNNs, particularly the use of convolutional and pooling layers, provide the models with some distinct advantages in computer vision tasks. In convolution, the input data is processed as an image, which preserves the spatial structure of the data. The placement of each pixel is taken into account. When the kernel extracts features, it extracts features of certain regions of the image, processing the input values together with other values in its spatial proximity. This enables the kernels to learn local spatial patterns. Furthermore, since these layers are not fully connected, this spatial proximity is also carried over between layers: Each neuron is not

connected to every other neuron, but only to certain neurons, dependent on its spatial position. This local connectivity enables a CNN to learn the features of certain areas of the image, such as edges, corners and shapes. As described in the convolutional layer example, the same kernel is passed over the whole input matrix; this means that the weights are shared across the image. Not only does this reduce the amount of weights needed for the model, but it also ensures that the same features can be learned different places in the image: Edges, circles and bicycles can be identified regardless of where they are placed in an image.

The pooling layers downsample to extract the most prominent features, capturing patterns of different parts of the image. This also mitigates the model fitting too closely to the training data, and thus failing to generalize well. This is called *overfitting*.

CNNs reduce the number of neurons and weights in the network through downsampling, shared weights and the absence of many dense layers. This allows CNNs to build in depth while keeping within boundaries of memory capacity and computational power. Through many layers, the CNNs then learn progressively advanced spatial features, which is finally put together to a global image in the final dense output layer.

## 2.5 Machine Learning for Speech Recognition

Speech is a fundamental mode of communication for humans, and speech signals have been a focus of research for deep learning applications for the last two decades [63]. In a literary review of this research, Nassif et al. defines speech recognition as giving ”[...] information about the content of speech signals”, and consequently automatic speech recognition as the capability of a machine to recognize speech content and transform it to a machine-understandable format [63, p. 3].

Typical applications for automatic speech recognition are speech-to-text applications, for instance used in transcription software and automatic generation of subtitles, and voice assistants such as Apple’s Siri, Amazon’s Alexa and more recently also ChatGPT and other large language models that provide an audio interface.

### 2.5.1 Speech Command Recognition

Automatic speech recognition is also used for more specific use cases, such as speech command recognition, which will be the focus of the following sections. Speech command recognition, also referred to as keyword spotting (KWS), is the task of recognizing certain keywords or commands from speech, according to a set of predefined classes. An example is how Google’s voice assistant is activated by the phrase ”Hey Google”, which is a keyword designed to activate the voice assistant. Use cases are not limited to recognizing a single activation word; with several classes, the keywords can for instance be used for speech control of applications.

Often, KWS systems are required to run on devices with less computational power, such as mobile devices, and thus a small memory footprint and low computational demands are often desirable in a KWS model. Small-footprint KWS models have been proposed for both deep neural networks [23], recurrent neural networks [87] and convolutional neural networks [81].

### 2.5.2 Preprocessing of Speech Data

Audio is recorded by measuring the air pressure registered by the input sensor - the microphone. Thus, speech data will in its most raw form be a

series of amplitudes, measured over time. This characteristic waveform is shown in Fig. 2.22.

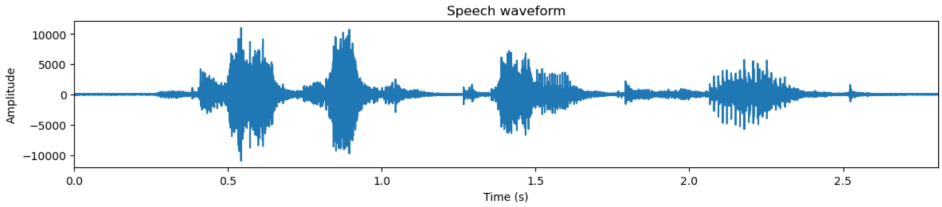


Figure 2.22: Amplitude over time [21]

This raw data is often preprocessed to be better suited for model training, such as dividing the data into frames of specific lengths and extracting the most informative features from the data. The most commonly used form of preprocessing of speech data in deep learning research is to extract the features using the *mel-frequency cepstral coefficient* (MFCC) [63, 21]. This entails processing the audio into a specific form of spectrogram called mel-spectrograms, before extracting some key spectral features from those spectrograms. Other models might use spectrograms or mel-spectrograms directly, and some the raw audio data.

### From Raw Audio to MFCC

When raw speech data is divided into time frames, such as 1-second, 3-second or 30-second utterances, it can be visualized as in Fig. 2.22, but the time axis will be of the specified length. By using the Short Time Fourier Transformation, this can be transformed into a spectrogram [21], shown in Fig. 2.23. A spectrogram is a 2D representation of audio, represented as frequency over time. Often, the spectrograms are shown in color, with the color intensity denoting the amplitude.

The human ear does not process audio information similarly across the frequency range. The *mel-scale* is a scale that aims to better represent the way humans perceive sound. By converting the frequency range to the mel-scale, the resulting *mel-spectrogram* will better represent the inherent information in the audio signal that is discernible to humans. These spectrograms are also referred to as *log-mel spectrograms*, due to the use of a logarithmic operation in their creation. The MFCC can be extracted by applying the Discrete Cosine Transformation to the mel-spectrogram. A waveform of

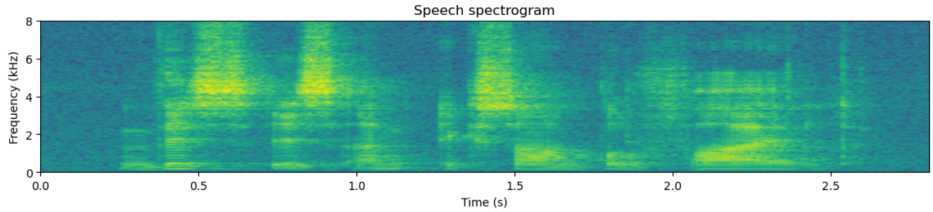


Figure 2.23: Spectrogram - frequency over time [21]

audio data that is processed into a mel-spectrogram and MFCCs is presented in Fig. 2.24

MFCCs are extracted features that capture the characteristics of the audio signals, while the mel-spectrogram is a more easily interpretable visual representation of audio data. As CNNs are powerful image classifiers, the processing of speech data into visual representations gives a straightforward way to approach speech command recognition as an image classification problem.

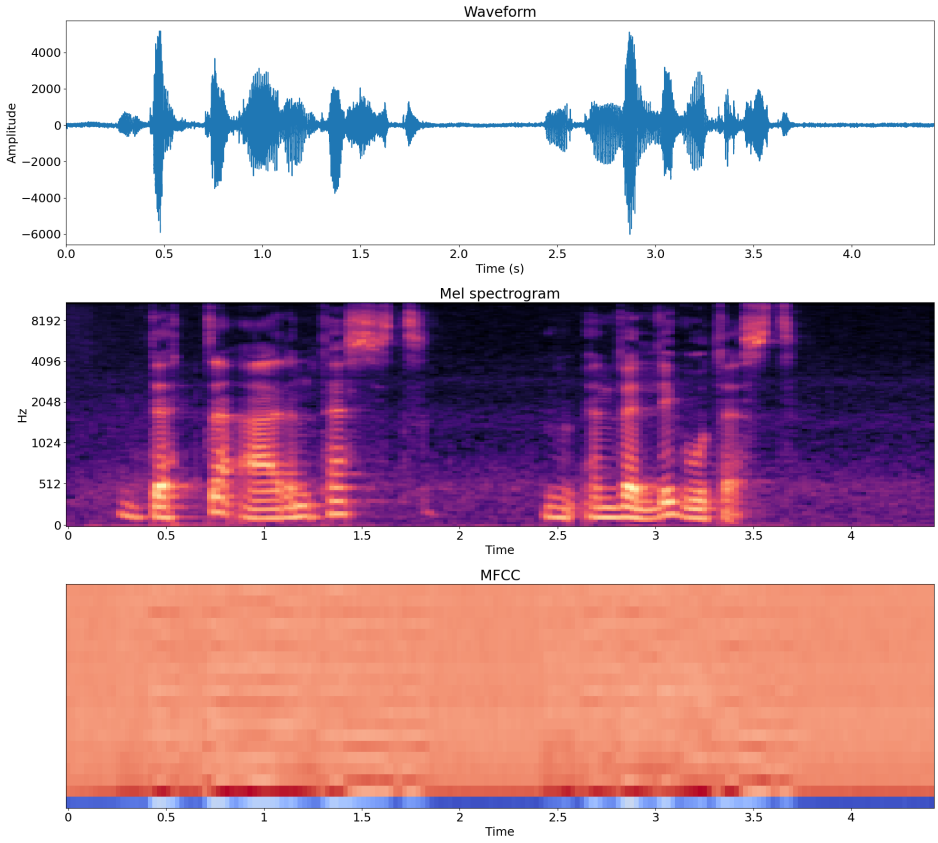


Figure 2.24: Waveform into mel spectrogram and MFCC

# Chapter 3

# Homomorphic Encryption in Machine Learning

## 3.1 Overview

This chapter will look at the specific uses of homomorphic encryption with machine learning. This includes an overview of how homomorphic encryption fits into a wider context of data protection measures in machine learning, both with regards to training and inference. The latter is then looked at in more detail, with a focus on the aspects that are most relevant to the thesis experiments; the use of homomorphic encryption for image classification using CNNs with either CKKS or TFHE. This part will also look at the specific strengths and weaknesses of each scheme, as related to their application for machine learning.

## 3.2 Privacy Preserving Machine Learning

The techniques used to ensure the privacy and confidentiality of sensitive data in machine learning training and inference are referred to as *privacy preserving machine learning* (PPML) [79]. Machine learning models consume data both in training and deployment; in training they require training samples from which to learn patterns, in deployment they are fed inputs on which to make predictions. Where a small model for a simple problem might be sufficiently trained on a dataset of a few hundred samples, a large and complex model can require vast amounts of training data. GPT-3, the precursor to GPT-3.5 (”ChatGPT”) from OpenAI, was trained on 400 billion data pieces, so called ”tokens”, distilled from petabytes of raw data [18].

Training data is the currency on which the models run, and the need for sufficiently sized training sets can conflict with security concerns over what data is used and how it is handled. In Europe in 2024, the technology company Meta sparked controversy over its plans to harvest public personal user data from Facebook and Instagram to train their AI models, with allegations that this was in conflict with the user’s right to data protection under the European Union’s General Data Protection Regulation (GDPR) [22].

Some models are designed for problems which involve the handling of sensitive information both in training and inference, such as medical imaging and diagnosis [13], DNA sequencing [98], or biometric authentication [90]. With widespread use of cloud computing and advanced models running on sophisticated and expensive hardware, models might be both trained

and deployed in the cloud, which could require sending highly private information to a remote service. Despite data being securely encrypted in transit, the data must be decrypted at some point: a model computes on input data, and therefore needs the mathematical qualities of its input intact. One of the challenges within PPML is the ensuring of a secure and confidential computing environment. One of the potential solutions to that challenge is to leverage the qualities of FHE schemes in machine learning, such that sensitive data never needs to be decrypted for use in the model at all.

### 3.2.1 FHE in Model Training

For most ML models, the heavy lifting of computations occur in the training phase. Training a model requires many repeated computations and the processing of many parameters, and can therefore be a computationally demanding operation with a large memory footprint. The biggest models, such as large language models, are often running on specialized hardware which supports the processing of billions of tokens and billions of parameters [89].

Due to FHE schemes' challenges with both computation speed and memory overhead, it is rarely proposed as a solution on its own to protect private data in a training process. There are some examples of FHE being used for encrypted training on simpler model architectures, such as Zama announcing the successful encrypted training of a logistic regressor with SGD [74, 85] in March 2024, and algorithms have been proposed for encrypted training of Support Vector Machines (SVM) [48, 67]. For complex models such as neural networks, FHE can be applied in combination with a decentralized learning approach called *Federated Learning* (FL) [76].

### 3.2.2 Federated Learning

FL [52, 73] is an approach for training models that lets a data contributor retain their data on their own local devices. Instead of sending local data to a remote global model, the global model is instead sent to the local device. There, the model is trained locally on the data, and afterwards a model update is sent back to the server where the global model originated. Thus, the data never leaves the local device, allowing users to maintain control of private data while still contributing data to improve the performance of the model.

One of the challenges of FL is, however, that the model updates that are sent from the local devices can still potentially leak private information [52]. An approach to mitigate this risk is to first encrypt the model update with FHE, then send the update back to the server. Fig. 3.1 depicts the flow of such an approach: The client receives encrypted model weights, decrypts them with their secret key, trains the model on local data, and then encrypts the updated weights and returns them to the server. The server aggregates the model updates in their encrypted form, and never sees the model updates in plaintext. As an example, NVIDIA's federated learning approach Clara Train [2, 76, 1] uses FL in combination with the CKKS scheme, as implemented by the TenSEAL library [11], a Python wrapper for Microsoft's SEAL.

### 3.2.3 FHE in Model Inference

FL is primarily a learning approach, and not suited for ensuring PPML in inference. Using FHE is a realistic alternative for privacy preserving inference on its own. This is especially due to inference being significantly less computationally demanding than training: Only a single input is processed to compute a prediction, and no weights or biases are updated.

There have been several research efforts demonstrating that model inference can be done using encrypted data: schemes and systems have been proposed for a wide variety of models, such as Support Vector Machines (SVM) [56], Hyperplane Decision, Naive Bayes and Decision trees [15], linear and logistic regression and neural networks [60]. While research has demonstrated FHE as a viable approach, challenges of computational complexity, memory overheads and low efficiencies keep it from being a feasible alternative for widespread practical application at this current stage.

With that said, a number of implementations in recent years have shown successful encrypted inference on neural networks, highlighting both developments in the field and specific challenges and limitations connected to different approaches. The following section will delve deeper into this, with a focus on CNNs.

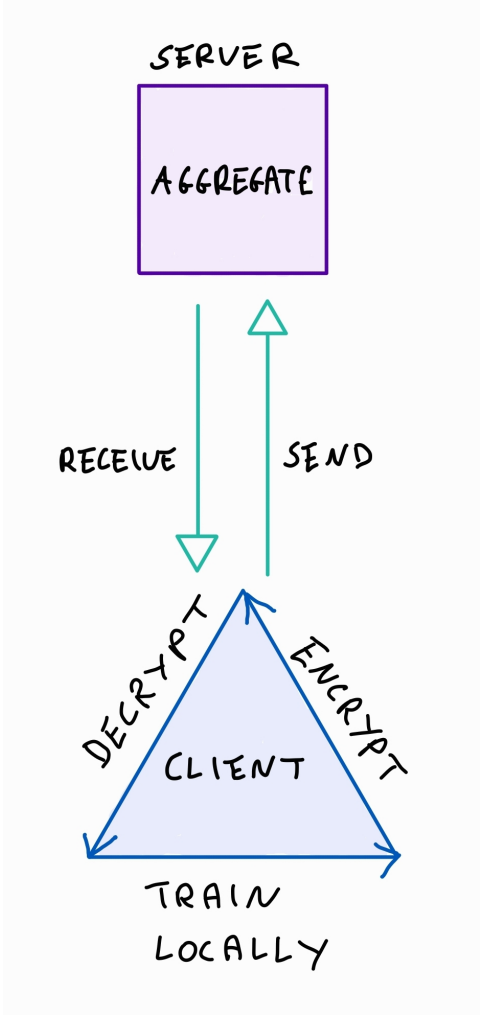


Figure 3.1: Federated Learning with FHE

## 3.3 Encrypted Inference in CNNs

Neural networks are increasingly common, and plaintext models are easily available and accessible through open-source models and libraries for training and inference, such as PyTorch [69] and TensorFlow [5]. Publicly available datasets are used for benchmarking model performances. CNN models are often benchmarked using, among others, the MNIST [31] dataset of 28x28 grayscale images of handwritten digits and the CIFAR-10 [46] dataset of 32x32 color images.

Popular frameworks for machine learning do not readily support encrypted inference, while HE libraries only rarely provide any ML support. Implementations of encrypted CNNs vary in the schemes they use, the HE libraries they are based on, the architecture of the networks, and the approach to handle challenges of encrypted inference. CKKS is the most used encryption scheme for encrypted CNNs.

CNNs are known as powerful image classifiers, and research on CNNs using FHE schemes generally uses well-known datasets of images as input to the models. This thesis' experiments also use CNNs on images, that is, on speech data represented as spectrograms. The following overview is therefore centered on encrypted CNNs for image recognition, and refers to the specific data points in the input as pixels.

### 3.3.1 Encrypted Inference using CKKS

While approaches, focus areas and results differ in implementations of CKKS-encrypted inference over CNNs, the fundamental properties of CKKS present some strengths to be taken advantage of, and some weaknesses that must be addressed.

#### Single Instruction Multiple Data

Because of batching, a CKKS scheme can fill up a ciphertext with thousands of values and evaluate them all in parallel. This has lead to the *Single Instruction Multiple Data* (SIMD) approach, where thousands of images can be evaluated homomorphically at the same time. The first pixel of the first image is placed in the first slot of a ciphertext, the first pixel of the second image is placed in the second slot, and so on, as shown in Figure 3.2. Any

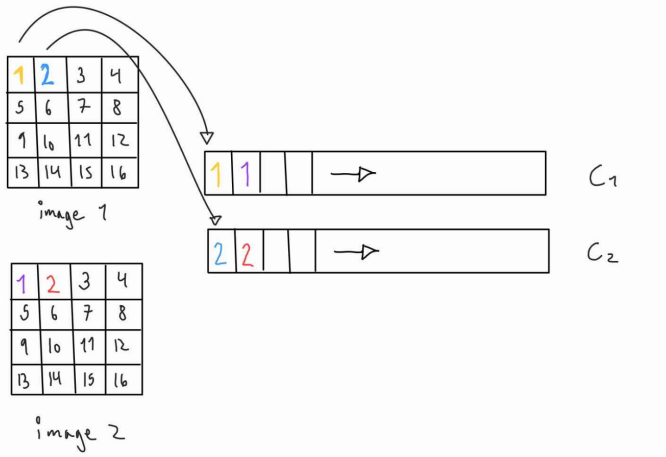


Figure 3.2: SIMD packing of two images

operation on the ciphertexts is done element-wise, thus while the pixels from the second image is in the same ciphertext, this does not affect the results of the first image.

Ishiyama, Suzuki and Yamana presented a CNN in 2020 for encrypted inference using CKKS, based on Microsoft SEAL [43]. Using a SIMD approach, they achieved an average inference time of 190 ms for a CIFAR-10 color image, with a top accuracy of 81%. While the average inference time per image was low, the total inference time was over 25 minutes (1555 seconds). This means that if the model was only to make a prediction on *one* image, it would still take 25 minutes to get a result.

The SIMD approach also requires a lot of memory, as every pixel must be stored in a separate ciphertext: A 32x32 pixel image would require 1024 ciphertexts, even before any calculations in the network were completed. They ran the model on a CPU with 72 cores (or 144 threads) in total, and a total memory available of 3 TB. The model had a memory usage of 1.41 TB.

### Single Input Single Output

Batching can also be used in a *Single Input Single Output* (SISO) packing approach, where a single image is packed into a ciphertext such that the first pixel of the image is in the first slot, the second pixel of the image is in the second slot, and so forth.

In 2022, Lee et al. demonstrated a highly accurate ResNet CNN with SISO packing, reaching 92% accuracy [50]. The inference time for one CIFAR-10 image was 3 hours with 112 cores and 512 GB available memory. The model simulation required 172 GB of memory; significantly lower than 1.41 TB, but still far more than personal computers usually are equipped with. In 2024, Rovida and Leporati published a ResNet CNN using the OpenFHE [9] open-source library with 91% accuracy that could do inference of one CIFAR-10 image in less than 5 minutes on a MacBook with 16 GB of working memory [77].

Where SIMD has a high memory usage, especially for larger images, SISO face other limitations on image size: The image size is restricted by the amount of slots available in the CKKS ciphertext that is used. For instance, if a library supports ciphertexts of up to 16384 slots, image size for a SISO approach will be limited to 128x128 pixels.

Speech command recognition is generally concerned with classification of a single voice command at a time. Therefore, some further details of how element-wise operations on SISO-packed ciphertexts work are presented below.

**Operations on SISO-packed ciphertexts** When packing a whole image into a single ciphertext, operations such as convolutions must be carefully managed, as every operation is done element-wise on a vector. Note that a ciphertext cannot be manipulated like a plaintext, and individual slots and slices cannot be extracted directly.

Figure 3.4 depicts a simple average pooling operation using element-wise operations on a ciphertext. A 4x4 image is flattened into a vector and packed into a single ciphertext, as shown in Fig. 3.3. The colored squares represent the positions in the vector that would make up the pooling windows in a matrix. Through a series of copies and rotations, these elements can be added together. Redundant values are removed by multiplying a vector with a mask. Masking and rotations are used to pack the sum of each pooling window into the first 4 slots of the ciphertext. The average is taken by multiplying the end result with the appropriate factor (here 0.25).

Any operation in a SISO model that requires access to specific pixels - such as when sliding a kernel over an image in convolution or pooling - must find a way to achieve this with element-wise operations. In the Python wrapper for SEAL, TenSEAL, convolution is made more efficient by

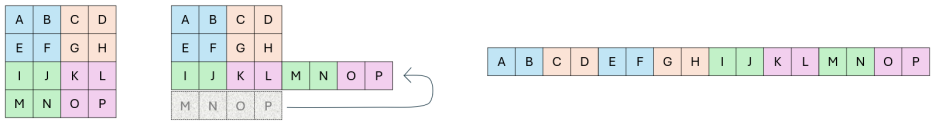


Figure 3.3: A 4x4 image is flattened into a vector

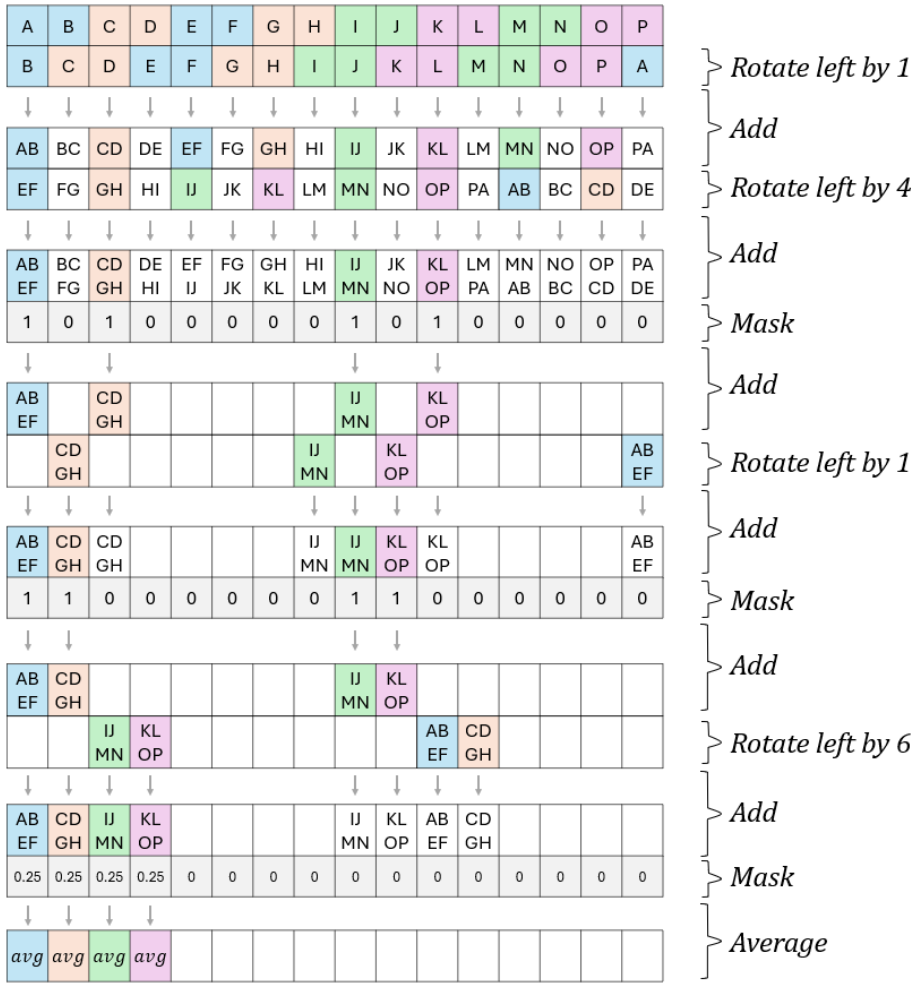


Figure 3.4: Average pooling with element-wise operations on a vector

preprocessing the image data before encryption, ensuring a column-wise orientation through the *im2col* operation [12]. This leverages smart prepro-

cessing to carry out a convolution as a single matrix multiplication operation, but requires handling of the image data in plaintext; thus, this approach can only be used for the first convolution in a CNN. TenSEAL therefore does not currently support models with more than one convolutional layer, placing limits on the model architecture.

UniHENN [29], a model based on SEAL and using the Python wrapper PySEAL [88], avoids the *im2col*-preprocessing and instead conducts convolutions using incremental rotations and structured multiplications on the flattened image. UniHENN supports processing of up to 10 samples at a time (for the MNIST dataset), and outperforms the average inference time of TenSEAL on MNIST when more than 3 images are processed and average time is calculated. For CIFAR-10 images, UniHENN reports an inference time of 21 minutes, with an accuracy slightly above 70%, running a 16-core CPU and 64 GB of memory. Note that TenSEAL does not have any data on its performance on CIFAR-10, as that would require a more complex model architecture than is currently supported.

CryptoNets, a model using the BFV scheme (a variant of BGV, which allows batching) over SEAL, launched a low-latency version [19] in 2019 where a multitude of different data structures are implemented to represent the ciphertext data in ways that enable more efficient layer operations. Running on 8 cores and 32 GB memory, it achieved inference on a single CIFAR-10 image in 730 seconds, with an accuracy of 74% and memory usage of 12 GB for a single prediction.

Ultimately, any implementation that packs an image into a single ciphertext will need a way to overcome the restriction of only element-wise operations, through methods like preprocessing, data representations and carefully structured rotations. Rotations require rotation keys, and switching these keys can be computationally intensive. Rovida and Leporati reduced the number of rotation keys needed in their implementation, by expressing rotations as aggregates of other rotations.

### Non-polynomial Functions

CKKS supports the arithmetic operations of addition and multiplication, and thus any polynomial function can be evaluated homomorphically. However, CKKS can also *only* do addition and multiplication, and therefore does not support the evaluation of non-polynomial functions. Specifically, this means

that models running with CKKS cannot do the non-polynomial activation functions ReLU and Softmax, which are commonly used functions in CNNs. These functions are therefore approximated, exploiting that CKKS is able to closely approximate a function through polynomials and real-number coefficients.

Lee et al. [50] used high-degree polynomials to approximate both the ReLU and Softmax function. As this strained the limit of the multiplicative depth, they also implemented a bootstrapping compatible with CKKS in SEAL, which otherwise provides only a leveled scheme. Due to the evaluation of many composite polynomials, the model required more than a thousand bootstrapping procedures.

Rovida and Leporati [77] used *Chebyshev polynomials* to approximate the ReLU function and a modular reduction function used in bootstrapping, consuming 5-7 levels for the former (depending on the degree) and 5 levels for the latter. They achieved results that were 98% similar to the output of their equivalent plaintext model.

#### Approximate Arithmetic

The need to approximate non-polynomial functions may result in encrypted models lacking the same accuracy as a corresponding plaintext model. The more closely the polynomial function approximates the non-polynomial, the higher the degree of the polynomial and the higher the latency cost. Conversely, a lower degree approximation generally results in a higher accuracy loss when compared to a plaintext model.

Apart from this type of approximation, CKKS is also a scheme of approximate arithmetics, and the noise can corrupt the values in the slots. Very small discrepancies will therefore be found between the message and the CKKS-encoded data. The extent of the inaccuracies depends on the polymodulus degree and the scale. With the many repeated operations that occur in the hidden layers of a neural network, even small accuracies can grow, and noise growth increases the chance of precision loss. That CKKS supports calculations with real numbers is an advantage when applied to neural networks, where floating points are generally used to represent numbers.

#### GPU Support

Neural networks consists of many linear algebra operations conducted in parallel, and GPUs, which are designed for such operations, are employed in deep learning to accelerate learning and inference. Plaintext machine learning with TensorFlow or PyTorch support the use of GPUs. Some work has demonstrated a promising speed acceleration through the use of GPUs when performing homomorphic operations.

Badawi et al. launched PrivFT in 2019, a closed-source text-classification model based on SEAL that achieved a performance speed increase by 1 to 2 orders of magnitude with their GPU implementation. Jung et al. [44] reported in 2021 a 257 times speed increase from a single-threaded CPU implementation of CKKS to their GPU-accelerated implementation and a bootstrapping procedure more than 100 times faster, and used this to train a logistic regression model with a 40x speed increase.

Running encrypted neural networks on GPU or other specialized hardware for machine learning has the potential to significantly speed up encrypted inference [36], as it also speeds up plaintext inference. The closed-source CKKS library HEaaN [68] announced support for GPU acceleration, but at the current stage, open-source libraries such as SEAL does not yet support GPU acceleration, and is far from providing such support for inference on neural networks.

### 3.3.2 Encrypted Inference using TFHE

TFHE is a less widespread approach for encrypted inference. Still, the open source TFHE library by Zama provides the most comprehensive and user friendly HE library for private inference, providing specific machine learning support in its Python library Concrete ML [95]. As Zama is spearheading the TFHE approach, the following observations are based primarily on their contributions of code (in libraries), documentation [94] and articles [84, 61].

#### Activation Functions during Bootstrapping

Recall that in TFHE, univariate functions can be evaluated through the use of a *lookup-table* (LUT) during bootstrapping, referred to as *programmable bootstrapping* (PBS). This allows TFHE schemes to evaluate activation functions such as sigmoid and ReLU homomorphically, while also reducing

noise. As such, when using TFHE to encrypt a CNN, the activation functions need not be approximated, and can therefore be expected to work just like the plaintext activations. Bootstrapping is a costly operation in TFHE, necessary to reduce noise and allow neural networks of unrestricted depths, but adding a function (such as ReLU) to evaluate while bootstrapping infers no extra cost. Thus, in TFHE, activation functions are evaluated as ”free” operations, piggybacking on the already costly bootstrapping procedure.

In 2023, Stoian et al. from Zama [84] showed successful encrypted inference on the MNIST dataset with a 6-layer CNN in 5072 seconds, with an accuracy of 98.7%, while a 9-layer CNN, VGG-9, did inference on CIFAR-10 in 18000 seconds with 87.5% accuracy. The experiments were carried out using consumer hardware, an 11th generation i7 CPU from Intel with 16 threads.

#### Integer-based Scheme

TFHE only supports evaluation of integers, which, in the context of machine learning, requires TFHE-based implementations to make some adjustment to its input. Earlier versions of TFHE encrypted each bit in a separate ciphertext, but Zama’s TFHE libraries now encode integers up to 8 bits in a single ciphertext, while integers of size 9-16 bits are encoded in more than one ciphertext. Still, a neural network must be within the bounds of the supported *circuit bit width*, which is the number of bits required to encode the largest integer in the circuit. For Concrete [96] this limit is currently at 16 bits. This means that using available implementations, a TFHE-based neural network cannot have a *circuit bit width* exceeding 16 bits.

#### Quantization and Pruning

As neural networks generally use floating-point arithmetic, a TFHE approach also requires the use of quantization, which is a mapping of floating point numbers into discrete values. This allows the floating-point arithmetic of the neural networks to be combined with the integer-based TFHE operations: By quantizing the inputs, the look-up table (LUT) operations can be applied to encrypted integer values.

LUT is more costly if the bit width of the input is large. Thus, the accumulator size of the input to a LUT, for instance input to a convolutional or fully connected layer, are bounded. Restricting the bit width size of

the input can be done through *pruning*, such as setting some values to 0 to manage the increase in accumulator size during operations. Pruning is detrimental to the performance of a model, but necessary to manage the speed of the LUT operations. Furthermore, the speed of a LUT can be increased by lowering the accuracy, allowing the LUT to operate within a specified error margin. Note that LUTs can also be applied without this error margin, letting the LUTs evaluate functions with no loss of precision and with exact results.

Quantization of a neural network after the network is already trained can lead to loss in performance. Zama has found this performance loss to be lower for their TFHE models when using *quantization aware training* (QAT), where the quantization is introduced already in the training phase.

### GPU Support

There are some GPU implementations of TFHE schemes, such as REDcuFHE [34] which reported a speed-up on a so called *binary neural network* (BNN) on the CIFAR-10 dataset from 1081 seconds on CPU to 229 seconds on GPU, with an accuracy of 81.9%. The experiment used specialized machine-learning hardware through Amazon Web Services (AWS); eight NVIDIA T4 GPUs and 96 virtual CPUs. NVIDIA's ArctyrEx for accelerated encrypted execution uses the TFHE scheme, and claims 40x speed increase for encrypted operations from a 256-threaded CPU baseline [36].

Zama is gradually adding GPU support to their libraries, such as creating and updating the GPU back end in Concrete in June 2024, and communicating that this GPU support will be integrated in Concrete ML during the fall of 2024 [3].

# Chapter 4

# Experiment

## 4.1 Overview

In this experiment, neural networks are used to conduct speech command recognition homomorphically. The models are trained in plaintext, using open-source machine-learning software. Inference is done homomorphically using two different schemes: CKKS and TFHE. The schemes have different limitations and requirements, and inference is carried out as two separate experiments for each scheme. A third experiment attempts to give a fair comparison of the two schemes. This includes a comparison of the best performing models for CKKS and TFHE.

The chapter will first introduce the framework and methodology of the experiment. The first section explains the limitations of the project, pertaining to the choice of neural network type, available hardware and the cryptographic libraries that are used. Following this, a section will describe the dataset and how the data is preprocessed. Then, the specifics of how the different machine learning models are trained, tuned and selected is detailed.

The next part of this chapter is the model experiments. This is divided into three sections. The first section presents the CKKS models, describing the CKKS parameters, the model architectures and the results of each model. The second section presents the TFHE models in the same way. The third section explains how the two schemes have been compared, and presents the results of the models used for this comparison. A summary of the comparison concludes the chapter.

## 4.2 Framework and Methodology

The speech command recognition problem that is tackled in this experiment is carried out as an image classification problem using CNNs. While different approaches and network types have different merits, using CNNs is a valid approach that well fits the requirements of a homomorphic system.

Homomorphic inference using CNNs have been successfully tested for both CKKS and TFHE on image classification problems such as CIFAR-10 and MNIST, lending some instruction and library support in construction of such networks. Furthermore, CNNs are suited for low-footprint applications due to the smaller amount of parameters in such networks, which can be useful when computational efficiency and memory usage are known challenges

in homomorphic encryption.

Finally, speech command recognition problems can easily be transformed into an image classification problem through preprocessing of speech data into spectrograms. When Google published a dataset for speech command recognition in 2018 [93], it was accompanied by an official TensorFlow [5] tutorial where this approach was used.

The aim of this experiment is to investigate the use of CKKS and TFHE for speech command recognition, using available cryptographic libraries and consumer hardware. As part of this, a goal is also to see if one scheme is more suitable than the other for a practical application.

The model experiments for CKKS and TFHE are therefore structured as follows: Existing model implementations made for the MNIST dataset is used as an initial starting point. Both libraries used have presented such demonstrations. This is then followed up with applying the same model architecture on the SC dataset. This gives an intuition of the difficulty of the SC dataset, as compared to MNIST, and a starting point for designing a model that performs better for speech command recognition. An improved model for the SC dataset is then implemented and tested.

It can be challenging to conduct a fair comparison of two schemes operating within different restrictions, and using different libraries in their implementations. The comparison experiment explores additional models in an attempt to illustrate the differences between the schemes, starting with simple linear models and ending with a comparison of the improved models from the previous experiment sections.

### 4.2.1 Project Decisions and Limitations

As mentioned, the project has limited the choice of neural networks to straightforward CNN models. There already exists research on, and implementations of, CNNs performing encrypted inference, primarily on the MNIST and CIFAR-10 [46] datasets. There are other and more advanced approaches to a keyword spotting problem, either using convolution in combination with other techniques, or entirely different types of models, that could provide better results: These are not explored, as the focus is predominantly on the use of FHE schemes, and the experiment therefore has prioritized a model type that has been proven viable with these schemes using open-source libraries.

For plaintext models, two traditional low-footprint CNN models were reported by Sainath et al. to achieve accuracies of 84.6% and 90.7% on the speech command dataset [81]. The best model to date uses both convolution and representation learning techniques to achieve an accuracy of 98.55% [92].

### Hardware

In research on encrypted inference, the used hardware can vary from a simple MacBook to specialized high-end machine-learning hardware. The hardware for this project is limited to consumer hardware.

All experiments were run on a virtual machine (VM), using an 11th Gen Intel(R) Core(TM) i7-11700K 3.60 GHz processor with 5 physical cores allocated, resulting in 10 available CPU threads. The VM has 54 GB of allocated RAM.

### Use of Existing Libraries

The project uses existing libraries to support the implementation of CKKS and TFHE models. A goal has been to investigate the application of homomorphic schemes in inference using libraries that are available to those who are not experts in cryptography, and where the application is as compatible with plaintext machine learning frameworks as possible.

Thus, for CKKS, the library TenSEAL [11], which both provides specific machine learning support and can be run from the most widely used machine learning language Python, has been used. The TFHE models used Zama's Concrete ML [95], which is specifically designed to support machine learning applications using TFHE. Both libraries enforce 128-bit security.

### TenSEAL

TenSEAL is a Python wrapper for Microsoft's SEAL [82], and provides functionality for setting CKKS parameters, generating keys, encoding/decoding and encryption/decryption of data, homomorphic addition and multiplication. Rescaling and relinearization is done "under the hood", in accordance with the set parameters.

In addition, TenSEAL provides optimized functionality specific for machine learning, such as the tensor datatype, matrix multiplication, and a

convolutional layer. This functionality is, however, severely restricted when compared to plaintext frameworks such as PyTorch: For instance, a model is currently restricted to one convolutional layer, and there are no supported pooling operations. Activations are done through passing the tensors through a polynomial, such as $x^2$, as there are not provided any predefined or optimized activation approximations.

#### Concrete ML

Concrete ML [95] is a machine learning library in Python, and is a framework for privacy preserving machine learning. It is built on top of the Concrete [96] library, which is an open-source FHE compiler for the TFHE scheme. Using Concrete ML, a plaintext PyTorch model that meets certain requirements can be compiled into an FHE model that can evaluate a circuit homomorphically.

The model can only use layer types that are supported by Concrete ML, and the bit width of the largest value used in the model inference cannot exceed the bit width that Concrete can use in TFHE computations. For neural networks, a quantization of a model is often required, either during training or after training before compilation.

#### Inference on Single Samples

Finally, the speech command recognition problem itself is concerned with the recognition of *one* word at a time. Thus, for CKKS, a batching approach which leverages parallelization to achieve low average inference times instead of low inference time on a single sample is not viable.

### 4.2.2 Dataset

The data material used for the experiment is the *speech commands* (SC) dataset from Google [93], published in 2018 as an aid for the development of keyword spotting systems. The dataset consists of a limited vocabulary of spoken words in English, with each word being restricted to a one-second utterance. There are ten classes of typical command words in the dataset: ”Yes”, ”No”, ”Up”, ”Down”, ”Left”, ”Right”, ”On”, ”Off”, ”Stop”, and ”Go”. Later additions to the dataset add the command-type words ”Follow”, ”Learn” and ”Forward”. The dataset has utterances of the numbers ”Zero”

to ”Nine”. There are also ten classes of arbitrary words meant to represent non-command words that a keyword spotting model should learn to ignore. In total, the dataset has 105,289 utterances of 35 different words, distributed as shown in Fig. 4.1.

| Word | Number of Utterances | Word | Number of Utterances |
|---|---|---|---|
| Backward | 1,664 | No | 3,941 |
| Bed | 2,014 | Off | 3,745 |
| Bird | 2,064 | On | 3,845 |
| Cat | 2,031 | One | 3,890 |
| Dog | 2,128 | Right | 3,778 |
| Down | 3,917 | Seven | 3,998 |
| Eight | 3,787 | Sheila | 2,022 |
| Five | 4,052 | Six | 3,860 |
| Follow | 1,579 | Stop | 3,872 |
| Forward | 1,557 | Three | 3,727 |
| Four | 3,728 | Tree | 1,759 |
| Go | 3,880 | Two | 3,880 |
| Happy | 2,054 | Up | 3,723 |
| House | 2,113 | Visual | 1,592 |
| Learn | 1,575 | Wow | 2,123 |
| Left | 3,801 | Yes | 4,044 |
| Marvin | 2,100 | Zero | 4,052 |
| Nine | 3,934 | | |

Figure 4.1: Class distribution of dataset, from [93]

### Preparation of Dataset

The dataset consists of wave files, where each file is hashed and labeled. It is split into a training, validation and testing set in accordance with a file list that is provided with the dataset. The distribution is roughly 80/10/10 for training, validation and testing respectively.

Performance on encrypted inputs are measured using a randomly sampled subset of the test set, with 100 samples. The main purpose of this test set is to measure potential discrepancies between a model’s performance on unencrypted and encrypted inputs, as well as to measure memory usage and computational efficiency.

Each set is distributed across twelve classes. In addition to the ten command-type words, an unknown class is created by randomly sampling audio files from the remaining 25 classes and represents words that the keyword spotting system should learn to ignore. A silence class is constructed of background noise that is provided with the dataset, divided up into 1-second audio clips. This represents general noise without words that the

model should not react to. The unknown and silence class is scaled to match the mean size of the command classes, resulting in a balanced dataset with the class distribution of the training set as shown in figure Fig. 4.2. Each class represents approximately 8.3% of the total dataset.

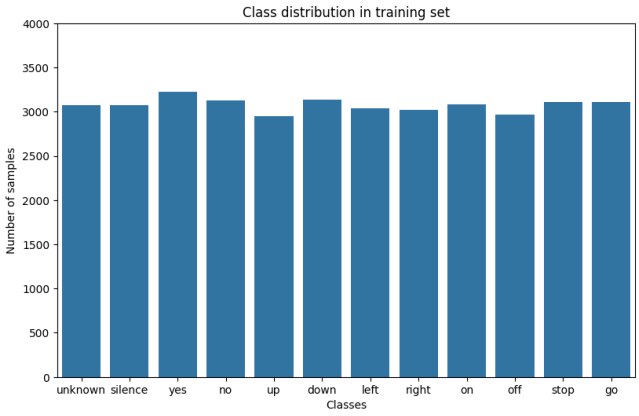


Figure 4.2: Class distribution of the training set

### Mel Spectrograms and MFCCs

Each wave file is loaded from the dataset and transformed into a mel spectrogram using audio processing functions provided by NVIDIA Data Loading Library (DALI) [4]. Audio samples are set to a fixed length of one second before transformation.

Some models process inputs as mel spectrograms, while others use mel-frequency cepstral coefficients (MFCC). MFCCs are a more abstract representation of the data, and is designed to be roughly uncorrelated [21]. While MFCCs are the most common form of preprocessing speech data for machine learning [63], there are examples of mel-spectrograms working better than MFCCs for some models [91]. The format of the input is chosen as part of the hyperparameter tuning process.

A 32x32 pixel mel-spectrogram of the word ”Off” from the training pipeline is depicted in Fig. 4.3. The length is fixed to one second, the decibel range is scaled from 0 db to -80 db. The MFCC of the same sample is shown in Fig. 4.4. A spectrogram of the silence class, consisting only of background noise, is shown in Fig. 4.5.

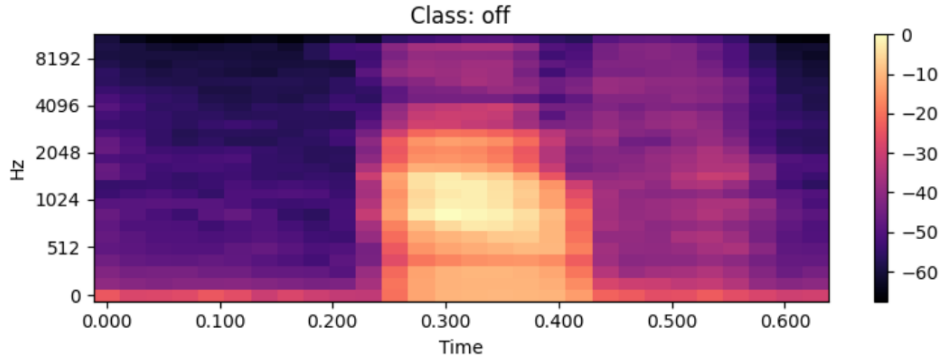


Figure 4.3: Mel spectrogram, 32x32, ”Off”

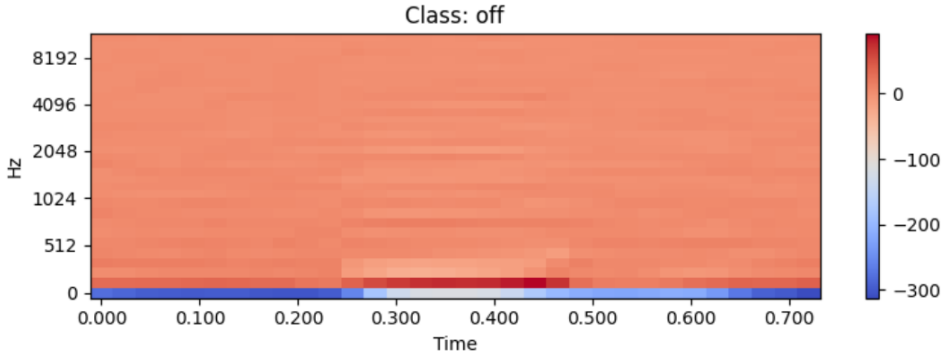


Figure 4.4: MFCC, 32x32, ”Off”

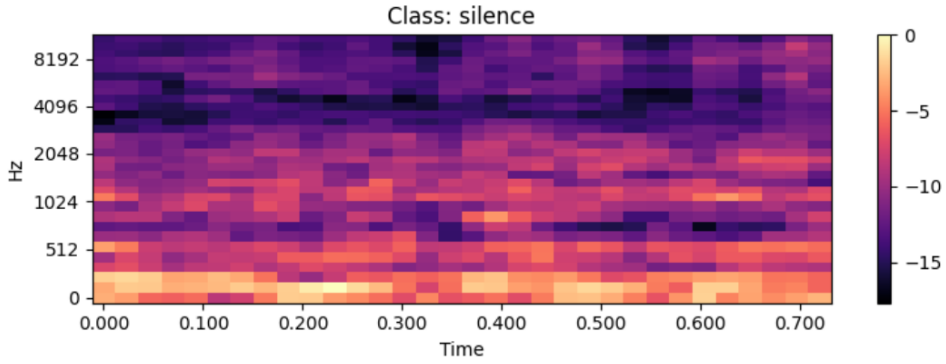


Figure 4.5: Mel spectrogram, 32x32, ”Silence”

### Data Augmentation for Training

The data is augmented during training to provide more variation in the training data and mitigate overfitting of the models. The augmentation

techniques used are applied to the spectrograms directly, as opposed to augmentation applied to the audio signals before they are transformed into images.

Blocks of time steps or frequency channels are arbitrarily masked, with the placement and width of the masking band chosen uniformly at random within specified bounds. These techniques were proposed specifically as both an effective and easy augmentation of mel-spectrograms by Park et al. in 2019 [66], together with a time warping technique. Time warping has been suggested to be the least influential of the techniques [54], and has been omitted here for simplicity.

A frequency mask and time mask is shown in Fig. 4.6 and Fig. 4.7 respectively. The augmentation is applied with a 0.5 percent probability for masking of both frequency and time. Some samples will have no augmentation and some will have both, as depicted in Fig. 4.8: This was the augmentation strategy that rendered the best results.

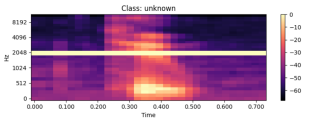


Figure 4.6: Masked frequency bandy

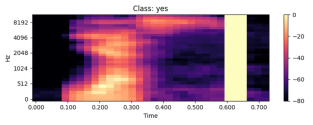


Figure 4.7: Masked time band

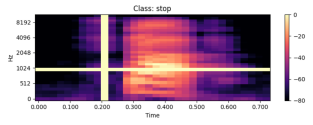


Figure 4.8: Masked frequency and time band

### 4.2.3 Model Training, Tuning and Selection

The models in the experiment were all trained using the PyTorch framework, with a NVIDIA DALI data loader for preprocessing data. The models were trained for 150 epochs with an Adam [45] or SGD optimizer [78]. Weight decay, learning rate, batch size and input format were set according to best found results from hyperparameter tuning using the validation dataset. The best model for each scheme was selected based on the results from this tuning process. Final performance was evaluated on the selected models using the test dataset with accuracy as the metric. A subset of the test dataset was used solely to evaluate the inference time and memory usage for encrypted inference. Discrepancies between plaintext and encrypted performance on the test subset was also measured.

### Hyperparameters

The models had the following adjustable hyperparameters, with the given values used as default:

- **Weight decay:** $10^{-3}$ - a regularization parameter that penalizes larger weights. Smaller weights help the model to avoid overfitting to the training data, and thus better capture the underlying patterns and generalize to unseen data.
- **Learning rate:** $10^{-3}$ - the rate by which to change the model weights based on new information during the training phase.
- **Learning scheduler:** ”plateau” - reduces learning rate when a model has stopped improving.
- **Learning scheduler patience:** 5 - number of epochs without improvement after which learning rate will be reduced.
- **Learning scheduler factor:** 0.1 - factor by which learning rate is multiplied when it is decreased (from $1 \times 10^{-3}$ to $1 \times 10^{-4}$ to $1 \times 10^{-5}$ and so forth).
- **Batch size:** 256 - number of samples that are processed before the model is updated.
- **Max epochs:** 150 - number of epochs that each model is trained for. One epoch is a complete pass through the whole training dataset.
- **Input format:** ”mel28”, ”mel32”, ”mfcc28” and ”mfcc32” - preprocessing of training data.

Hyperparameter tuning was done through manual search using the validation dataset. Batch size, input format, learning rate and weight decay were tuned.

Some hyperparameters were early set to default values after initial experimentation. These parameters were learning scheduler, learning scheduler patience and learning scheduler factor (gamma). Optimizer Adam over SGD was also chosen as default after initial experiments. Max epochs were set to 150 to give equal training resources for every model, and ensure the more

complex models would have time to converge. Mel-spectrograms or MFCCs was used based on what gave the best results during tuning, with MFCC being the default value as this is also the most commonly used format.

### Performance Evaluation

Accuracy was used as the performance metric for the models. The dataset consisted of balanced classes, and accuracy is a metric that is easily interpreted.

After hyperparameter tuning, the model with the highest accuracy was selected. The selected model was then evaluated on the test dataset to estimate its performance on unseen data. A confusion matrix was also used as a sanity check and to gain greater insight into the models' strengths and weaknesses.

Due to the increased inference time on encrypted data, a smaller subset of the test set was used to evaluate performance on encrypted data. This subset is of too small a size to accurately simulate the performance of the model on unseen data. The model performance on unseen data was conjectured to be what the full test performance indicated, while the subset was used to compare discrepancies between unencrypted and encrypted inference, as well as measure the inference time when operations were run homomorphically.

## 4.3 Models using CKKS

The models presented in this section used the CKKS scheme for inference on encrypted data. The models were trained in plaintext using PyTorch, and then the model weights and biases were loaded into models that used CKKS for encrypted inference. The models were implemented within the restrictions of the CKKS scheme and the project limitations. The final model in the section attempted to attain best possible performance given these conditions.

### 4.3.1 Restrictions

All CKKS models were run using the homomorphic functionality of Microsoft's SEAL library [82]. SEAL uses a leveled approach, such that parameters must be set beforehand in accordance with the expected multiplicative depth of the homomorphic circuit. This means that parameters must be set specifically for each model, as the number of multiplications during inference will vary according to the model's architecture.

To ensure that the encrypted models run with a satisfactory bit-level security, the number of levels must be balanced with the size of the ciphertexts. Recall that a CKKS ciphertext size and its number of slots is influenced by parameters that is set beforehand. When the size of the ciphertexts grows, so too grows the memory consumption and computation time. In this way, the complexity of the machine learning model is restricted by the number of levels that can realistically be achieved by the used library, the bit security achieved by the parameters, the ciphertext size and available slots, and the resulting memory usage and time consumption.

### 4.3.2 Parameters

Parameter choices are important for the security and efficiency of a model using CKKS. The following parameters must be set:

- **Polynomial modulus degree, $N$:** The size of $N$ directly affects the available slots of the ciphertext, as each ciphertext will have $N/2$ number of slots. It also influences the size (in bytes) of the ciphertexts, with a higher $N$ resulting in bigger ciphertexts that require more

memory. A large $N$ will infer higher computational costs, but also increase the security of the scheme.

- **Bit scale:** This is the size of the factor that real numbers are multiplied with during encoding, and divided by during decoding. The size of the bit scale influences the floating point precision of the scheme, with a higher scale resulting in higher precision.

- **Coefficient modulus sizes, $q$:** CKKS uses the Chinese Remainder Theorem to divide large numbers for homomorphic computations, using a series of prime numbers. These prime numbers are each of size $\log_2 q$ bits, and are automatically set in SEAL. The scaling is done automatically in TenSEAL: With the exception of the first and last values, each coefficient modulus is set to the same $q$, preferably the same as the bit scale used during encoding and encryption. The discrepancy between the last value and $q$ denotes the number of bits reserved for the integer part of the output value. In SEAL, and therefore also in TenSEAL, the size of $q$ cannot exceed 60 bits and must be congruent to $1 \mod 2N$. The choice of $q$ affects the size of the ciphertext elements. A larger $q$ results in weaker security.

- **Levels:** The number of primes of size $q$ decides the number of levels available for the scheme.

Microsoft SEAL enforces a minimum bit level security of 128 bits for its parameters, in accordance with the Homomorphic Encryption Standardization [6]. Parameters that violate this restriction or are not specified in the standard are deemed invalid.

### 4.3.3 Initial Benchmarking

TenSEAL provides a demo of a simple model that can run encrypted inference on the MNIST dataset [31], which consists of 28x28 pixel grayscale images. This model architecture is used as a baseline for further experimentation.

The TenSEAL demo is run on the project VM, and its performance on MNIST is recorded. A model of identical architecture is then trained and tuned on the SC dataset, according to the methodology described in

4.2.3. This model is then evaluated on test data and its performance on encrypted data is measured. The MNIST and SC models are referred to here as CNN_MNIST_5 and CNN_SC_5.

### Architecture and Parameters

The CNN_MNIST_5 model consists of the layers as described in Table 4.1. The CNN_SC_5 model is, as mentioned, identical, with the only difference being the output size of the final linear layer: MNIST has 10 classes, while the SC dataset has 12.

| **Layer** | **Details** |
|---|---|
| Convolutional layer | 4 filters (kernels). Kernel shape: $7 \times 7$. Stride: 3. |
| Activation | Square activation function: $x^2$ |
| Linear layer 1 | Input size: 256. Output size: 64 |
| Activation | Square activation function: $x^2$ |
| Final linear layer | Input size: 64. Output size: 10 |

Table 4.1: Model architecture of CNN_MNIST_5

The CKKS parameters are described in Table 4.2; both models have the same parameters, resulting in 6 levels and a precision of 5 and 19 bits for the integer and decimal parts respectively for the output values.

| **Property** | **Value** |
|---|---|
| Models | CNN_MNIST_5 , CNN_SC_5 |
| $N$ | 8192 |
| Bit Scale | 26 |
| Outer Primes of Coefficient Modulus Size | 31 |
| Inner Primes of Coefficient Modulus Size | 26 |
| Levels | 6 |
| Bits Reserved for Integer Part | 5 bits $(31-26)$ |
| Bits Reserved for Decimal Part | 19 bits $(26-5)$ |

Table 4.2: Configuration details for CNN_MNIST_5 and CNN_SC_5.

### Results

The results of the models are summarized in Table 4.3. CNN_MNIST_5 achieved a 98% test accuracy on the MNIST dataset for both plaintext and encrypted inference, both measured over a test set of 10000 samples. The average inference time for encrypted inference of one sample was 0.76 seconds.

CNN_SC_5 achieved a 80.56% test accuracy on the SC dataset. The plaintext result on the test subset was 81%, while the encrypted result was 73%. The average inference time for encrypted inference of one sample was 0.78 seconds. The model used 4.1 GB RAM during inference. The confusion matrix, shown in Fig. 4.9, demonstrated that its weakest class was the unknown class.

| **Model** | **CNN_MNIST_5** | **CNN_SC_5** |
|---|---|---|
| Plaintext accuracy | 98% | 80.56% |
| Plaintext subset acc | N/A | 81% |
| FHE (subset) accuracy | 98% | 73% |
| Average inference time | 0.76 s | 0.78 s |

Table 4.3: 5-layer CNN on MNIST and SC

The performance drop from CNN_MNIST_5 to CNN_SC_5 was roughly 17%, which suggests that the SC dataset is more complex than MNIST and thus also likely requires a more complex model architecture for better performance. The significant discrepancy of 8% between the plaintext and the encrypted result on the SC dataset indicates that the CKKS parameters were not sufficient for the complexity of the new dataset. Specifically, the precision of 5 bits for the integer part of the output value might have been too low for the output values.

CNN_SC_5 was run again with increases in parameters, summarized in Table 4.4. CNN_SC_5 version 2 was run with $N = 16384$, bit scale of 26, 6 levels, outer coefficient modulo sizes of 60 and inner sizes of 26. It achieved an encrypted accuracy of 79%, with an average inference time of 1.73 seconds per sample and use of 4.9 GB RAM.

CNN_SC_5 version 3 was run with a significant increase in parameters, with the bit scale set to 30 and the number of levels increased from 6 to 10. This resulted in an accuracy of 80%, which is very close to the plaintext

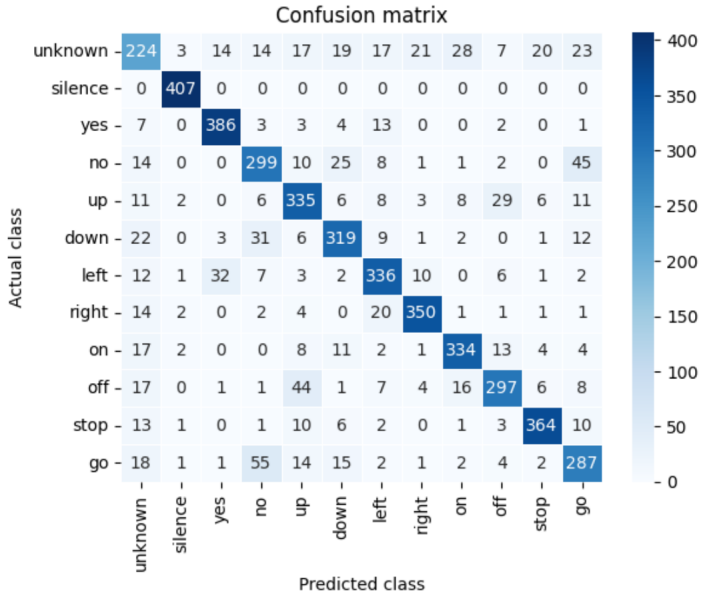


Figure 4.9: Confusion matrix of CNN_SC_5

result of 81%. The average inference time was 4.46 seconds per sample and 6.6 GB RAM was used. This goes to show how important parameter settings are for the performance of the models, and how the parameters also impact memory usage and inference time.

| **Model** | **CNN_SC_5** | **CNN_SC_5_v2** | **CNN_SC_5_v3** |
|---|---|---|---|
| N | 8192 | 16384 | 16384 |
| Bit scale | 26 | 26 | 30 |
| Levels | 6 | 6 | 10 |
| Plaintext accuracy | 80.56% | 80.56% | 80.56% |
| Plaintext subset acc | 81% | 81% | 81% |
| FHE subset accuracy | 73% | 79% | 80% |
| Average inference time | 0.78 s | 1.73 s | 4.46 s |
| Memory requirements | 4.1 GB | 4.9 GB | 6.6 GB |

Table 4.4: Comparison of different parameters for CNN_SC_5

### 4.3.4 Improved Model

CNN_6 is a more complex model than CNN_SC_5. It uses the activation function $0.5x^2 + 0.5x + 0.05$, which is an approximation of the ReLU function for values between -1 and 1, and which has given slightly better results than square activation during hyperparameter tuning. Since TenSEAL does not provide any average pooling functions, this had to be implemented locally. Average pooling and a function for packing previously modified ciphertexts were implemented locally in the TenSEAL C++ back end.

The average pooling operation uses rotations to pool one kernel at a time, following the procedure as described earlier in Section 3.3.1. It is not optimized with regards to reducing the number of keys used in CKKS rotations, and does not allow pooling several filters at the same time in the case that they can be packed into the same ciphertext. Since average pooling is a computationally intensive operation, evaluating more filters in parallel can possibly reduce the processing time for each layer and thus the final inference time by tens of seconds.

#### Architecture and Parameters

The architecture of CNN_6 is described in Table 4.5. This model uses a dropout layer [39, 71], which is a regularization technique to reduce overfitting to the training data. A dropout layer is configured using a probability ratio, which here is set to 0.5. With the given probability, an input neuron's value is set to 0 instead of retaining its original value. This essentially prevents the model from learning the training data too well, which would lead to overfitting and a poorer ability to generalize. A dropout layer is only used during training, and thus does not affect the layer structure of the model when doing encrypted inference.

The CKKS parameters of CNN_6 is described in Table 4.6. Note that the decimal parts does not have any bits reserved for its precision; this was not changed, because there was no discrepancy between the plaintext accuracy and encrypted inference accuracy, and thus the impact was assessed to be minimal.

| Layer | Details |
|---|---|
| Convolution | 4 filters (kernels). Kernel shape: 3 × 3, Stride: 1. |
| Activation | Polynomial activation function: $0.5x^2 + 0.5x + 0.05$ |
| Average pooling | Kernel size: 2. Stride: 2 |
| Linear layer | Input size: 16 × 15 × 15. Output size: 120 |
| Activation | Polynomial activation function: $0.5x^2 + 0.5x + 0.05$ |
| Dropout | Probability: 0.5 |
| Final output layer | Input size: 120. Output size: 12 |

Table 4.5: Model architecture of CNN_6

| Property | Value |
|---|---|
| Models | CNN_MNIST_5 , CNN_SC_5 |
| $N$ | 32768 |
| Bit Scale | 30 |
| Outer Primes of Coefficient Modulus Size | 60 |
| Inner Primes of Coefficient Modulus Size | 30 |
| Levels | 18 |
| Bits Reserved for Integer Part | 30 bits $(60 - 30)$ |
| Bits Reserved for Decimal Part | 0 bits $(30 - 30)$ |

Table 4.6: Configuration details for CNN_6 .

### Results

As seen in Table 4.7, CNN_6 achieved an 87.58% test accuracy on the SC dataset. The plaintext result on the test subset was 87%, and the encrypted result was 87%. The average inference time for encrypted inference of one sample was 154.54 seconds. The model used 22.7 GB RAM during inference, indicating that the ciphertext sizes are big, which is a challenge for keyword spotting models that preferably should have low memory footprint.

The confusion matrix, shown in Fig. 4.10, demonstrates that the accuracy is lowest for the unknown class, and that the model also has trouble differentiating between ”No” and ”Go”. The unknown class contains greater variation

than the other classes, which might explain why the model performance is weakest for this class.

| **Model** | **CNN_6** |
|---|---|
| Plaintext accuracy | 87.58% |
| Plaintext subset acc | 87% |
| FHE (subset) accuracy | 87% |
| Average inference time | 154.54 s |
| Memory requirements | 22.7 GB |

Table 4.7: Results of CNN_6 on the SC dataset

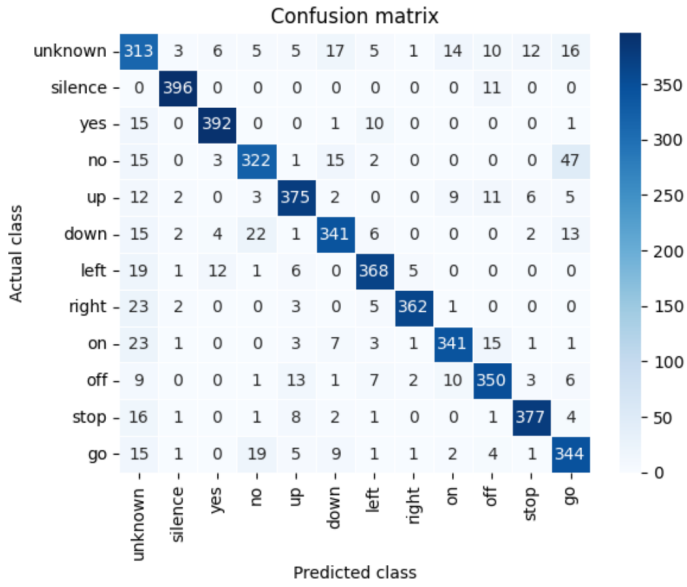


Figure 4.10: Confusion matrix for CNN_6

### 4.3.5 CKKS Models Summary

CNN_6 has a significant improvement of 7% compared to CNN_SC_5. While CNN_6 is still a simple CNN, it is notably more complex than CNN_SC_5. This complexity increases performance, but also requires a significant increase in CKKS parameters, resulting in higher ciphertext sizes and a vastly increased inference time and memory usage.

One of the challenges is related to the available slots in each ciphertext and the levels available: Linear layers are calculated homomorphically using a matrix multiplication. A higher slot count of the ciphertext means that each linear layer can be carried out using just one such multiplication.

With a more complex model with, for instance, a higher amount of convolution kernels, this necessitates a higher $N$-value, which, while increasing security, reduces computational efficiency and inflates the size of the ciphertexts. Conversely, dividing up the computations limits the operations that can be done in parallel, and each matrix multiplication accounts for a significant portion of the total inference time.

Changing the architecture to a deeper model instead requires more levels, potentially resulting in too low bit-level security. If the bit scale is set too low to alleviate the reduction in security, this in turn affects performance.

## 4.4 Models using TFHE

The models presented in this section used the TFHE scheme for inference on encrypted data. The models were trained using the same plaintext PyTorch framework as the CKKS models from Section 4.3.5, and then compiled into FHE_equivalent models using Concrete ML's support for model compilation.

### 4.4.1 Restrictions

All TFHE models were run using Zama's Concrete ML library for privacy-preserving machine learning, and is restricted to the layer types and functionality provided in that library. Concrete ML uses TFHE exclusively, and only supports models that has a maximum bit width in the circuit of 16 bits. This means that computed values during the model's inference cannot exceed 16 bits, and the model inputs must therefore often be quantized to ensure that this condition holds. Quantization is done using Brevitas [65], which is a framework that supports *quantization aware training* (QAT).

#### Quantization

Brevitas provides quantization layers that can replace and be combined with non-quantized PyTorch layers. A quantized layer will quantize its input to the specified bit width. Both weights and activation values can be quantized, and the bit width for each can be set separately.

In Concrete ML, there are custom weight and activation tensor types that combine functionality from Brevitas to force quantization scales to be a power of two. This is done to speed up homomorphic inference times.

### 4.4.2 Parameters

TFHE parameters are set automatically in Concrete ML, and ensures a security of 128 bits. During compilation of a model into an FHE equivalent, a variety of parameters can be set to influence performance of the model. The parameters tweaked in this experiment is primarily the *global_p_error* , the *p_error* and the *rounding_bit_threshold*.

The global_p_error is the accepted error margin of the entire circuit, which allows the final output result to have some inaccuracies. This can speed up computations at the cost of creating a discrepancy between plaintext

and homomorphic computations. An alternative to the global_p_error is the *p_error* parameter: This denotes the accepted inaccuracy for each individual table-lookup operation. Note that only global_p_error or p_error can be set at the same time, as they are incompatible with each other.

The rounding_bit_threshold is a parameter given to an optional rounding operator, which reduces the bit width of intermediate tensors to $n$ bits. The value of $n$ cannot exceed 8. Using the rounding operator and setting a lower bit threshold can increase computation speed, and must be balanced with an increasing loss of accuracy. A more recent addition also allows a flag to be set to either ”APPROXIMATE” or ”EXACT” for this rounding operator, where ”APPROXIMATE” will potentially speed up computations at the expense of accuracy.

Finally, the quantization parameters that specifies the bit width of the weight and activation tensors affect the maximum bit width in the circuit, and thus whether the plaintext model can successfully be compiled into an FHE equivalent model. Additionally, this also affects the speed and accuracy of the resulting models: A model with a higher maximum bit width will generally be more accurate but compute slower than a model of lower maximum bit width.

### 4.4.3 Initial Benchmarking

Zama has a demonstration in Concrete ML of how to do encrypted inference on the MNIST dataset using Brevitas for QAT. This model is tested locally on MNIST, and the same model is adapted to 12 classes, retrained and run on the SC dataset. The results provide a starting point for further experimentation using Concrete ML and the TFHE scheme.

The demonstration is two years old, and the code was therefore updated to be compatible with the current version of Concrete ML. At the time the demonstration was written, Concrete ML limited the maximum bit width in the circuit to 8 bits.

#### Architecture and Parameters

The QNN_MNIST_4 layers are described in Table 4.8. Batch normalization layers are used between the linear layers to keep a current estimate of mean and variance for normalization during evaluation.

The same model is used for the SC dataset, with the exception of the final output size of the final layer being changed from 10 to 12. This version of the model is named QNN_SC_4 .

The model also uses *pruning* to reduce the size of the weights of the linear layers. The purpose of pruning is to remove unimportant neurons, so that the model retains predictive performance while being more computationally cost effective.

| **Layer** | **Details** |
|---|---|
| Quantized Linear layer | Input size: 784. Output size: 192. |
| Quantized Linear layer | Input size: 192. Output size: 192. |
| Quantized Linear layer | Input size: 192. Output size: 192. |
| Final Quantized Linear layer | Input size: 192. Output size: 10. |

Table 4.8: Model architecture of QNN_MNIST_4

The parameters used for the models are described in Table 4.9. They are compiled without the rounding_threshold_bit operator. QNN_MNIST_4 has a maximum bit width of 6 bits, whereas QNN_SC_4 has a maximum bit width of 5. The discrepancy of 1 bit is likely due to differences in the two datasets, as both the models and the parameters are otherwise identical.

| **Property** | **Value** |
|---|---|
| Models | QNN_MNIST_4 and QNN_SC_4 |
| Quantization Bits | 2 bits |
| Rounding Threshold Bit Operator | N/A |
| Global P Error | None |

Table 4.9: Parameters for QNN_MNIST_4 and QNN_SC_4

### Results

The results of the models are summarized in Table 4.10. QNN_MNIST_4 achieved a 93.53% plaintext test accuracy on the MNIST dataset, measured over a test set of 10000 samples. A subset of 100 samples was used to measure encrypted inference. QNN_MNIST_4 achieved a 92% plaintext accuracy

on the test subset, and the encrypted result was also 92%. The average inference time for encrypted inference of one sample was 9.5 seconds.

| **Model** | **QNN_MNIST_4** | **QNN_SC_4** |
|---|---|---|
| Dataset | MNIST | SC |
| Plaintext accuracy | 93.53% | 31.94% |
| Plaintext subset acc | 92% | 27% |
| FHE subset accuracy | 92% | 27% |
| Average inference time | 9.5 s | 2.67 s |

Table 4.10: Quantized model on MNIST and SC

QNN_SC_4 achieved a test accuracy of 31.94% on the SC dataset. The plaintext result on the test subset was 27%, and the encrypted result was also 27%. The average inference time for encrypted inference of one sample was 2.67 seconds. The model used 2 GB RAM during inference.

QNN_SC_4 appears unable to sufficiently capture the complexities in the SC dataset, which is also apparent in the confusion matrix in Fig. 4.11. The model struggles especially with the unknown class and the words ”No”, ”Go”, and ”Up”.

A discrepancy between the model’s accuracy on the MNIST and SC dataset of over 60% indicates that a linear model of this type and configuration is badly suited for the SC dataset, and underlines the SC dataset’s higher complexity as compared with MNIST.

### 4.4.4 Benchmarking: A more Recent MNIST model

In July 2024, Zama posted an updated overview of neural network models using TFHE and their performance on MNIST. Together with this update, they also provided a notebook that ran inference on MNIST using neural networks and the more recently added rounding_bit_threshold operator. The 20-layer model PTQNN_MNIST_20 was used to give a more updated picture of Concrete ML models on MNIST. An adapted version, PTQNN_SC_20 , was used to compare the model’s performance on the SC dataset.

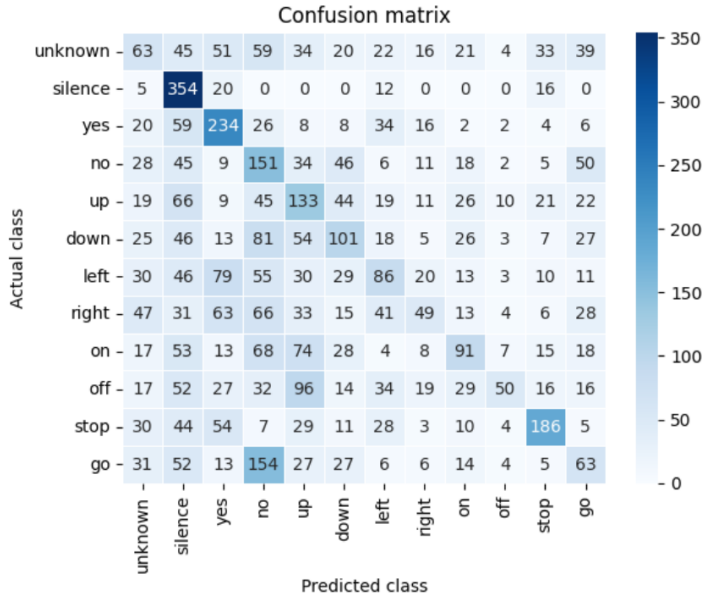


Figure 4.11: Confusion matrix of QNN_SC_4

### Architecture and Parameters

PTQNN_MNIST_20 is a neural network consisting of 20 linear layers, with ReLU activation and batch normalization between each linear layer. The first layer in the model is a single convolutional layer with ReLU activation and batch normalization. The layers are depicted in Table 4.11.

| **Layer** | **Details** |
|---|---|
| Convolutional layer | 1 kernel. Kernel shape: 3x3. Stride: 1. Padding: 1. |
| First Linear layer | Input size: 676. Output size: 92. |
| ReLU activation | |
| Batch Normalization | |
| 18 Linear Layers | Input size: 92. Output size: 92.<br>ReLU Activation<br>Batch Normalization. |
| Final Linear layer | Input size: 92. Output size: 10. |

Table 4.11: Model architecture of PTQNN_MNIST_20

PTQNN_SC_20 is identical in implementation and architecture, with the exception of 12 output classes in the final linear layer. PTQNN_MNIST_20 is compiled with a maximum bit width in circuit of 17 bits. Note that this maximum bit width is an estimate calculated on the basis of input data used for model compilation. Since TFHE is currently limited to 16 bits maximum bit width, the actual bit width is likely no more than 16 bits. PTQNN_SC_20 is compiled with a maximum bit width of 15 bits.

PTQNN_MNIST_20 and PTQNN_SC_20 are trained in plaintext, and quantization is applied after training. This approach is called Post-Training Quantization (PTQ). The models are quantized during compilation to an FHE-compatible model.

The parameters of the models are shown in Table 4.12.

| **Property** | **Value** |
|---|---|
| Models | PTQNN_MNIST_20 and PTQNN_SC_20 |
| Post Quantization Bits | 6 bits |
| Rounding Threshold Bit Operator | 6 |
| Method | ”APPROXIMATE” |
| P Error | 0.1 |

Table 4.12: Parameters for QNN_MNIST_4 and QNN_SC_4

### Results

The results of the models are summarized in Table 4.13. PTQNN_MNIST_20 achieved a 98.1% plaintext test accuracy on the MNIST dataset, measured over a test set of 10000 samples. A subset of 100 samples was used to measure encrypted inference. PTQNN_MNIST_20 achieved a 97% plaintext accuracy on the test subset, and an encrypted result of 94%. The average inference time for encrypted inference of one sample was 14.25 seconds.

PTQNN_SC_20 achieved an 80.18% plaintext test accuracy on the SC dataset. The plaintext result on the test subset was 80%. After compilation and PTQ, the model achieved an accuracy of 65% for encrypted inference on the test subset, with an average inference time per sample of 12.95 seconds. The model used 2.68 GB during inference. The confusion matrix in Fig. 4.12 shows PTQNN_SC_20 ’s plaintext performance, depicting a model with a

strong performance, but struggling with the unknown class, and the words ”No”, ”Go” and ”Up”.

| **Model** | **PTQNN_MNIST_20** | **PTQNN_SC_20** |
|---|---|---|
| Dataset | MNIST | SC |
| Plaintext accuracy | 98.1% | 80.18% |
| Plaintext subset acc | 97% | 80% |
| FHE subset accuracy | 94% | 65% |
| Average inference time | 14.25 s | 12.95 s |

Table 4.13: Post quantized model on MNIST and SC

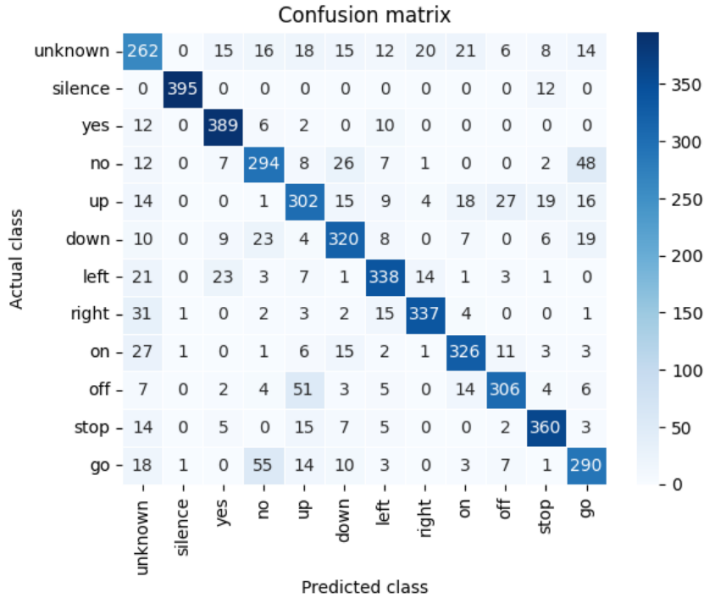


Figure 4.12: Confusion matrix of TFHE_NN_SC_1

The results show a discrepancy in plaintext performance of 18% between MNIST and SC, using the same model implementation and compilation parameters. This is reminiscent of the discrepancy of 17% observed between MNIST and SC performance for CKKS models, and is an indicator of the difference in complexity between the two classification problems.

The additional drop of 15% accuracy between PTQNN_SC_20 ’s plaintext performance and its post-quantization performance shows the impact

both quantization and the TFHE compilation parameters can have on accuracy.

To see how accuracy is impacted by the TFHE compilation parameters, PTQNN_SC_20 is compiled again with different compilation parameters. While the PTQ remains the same, the p_error is set to None and the rounding_bit_threshold operator method is set to "EXACT". The results are shown in Table 4.14. With these changes, PTQNN_SC_20 achieves an accuracy of 72% for encrypted inference, with an average inference time of 146.7 seconds and a memory use of 7.7 GB. The accuracy drop before and after PTQ is reduced from 15% to 8%. However, omitting the TFHE parameters designed to speed up inference results in a vast increase of inference time.

| **Model** | **PTQNN_SC_20** | **PTQNN_SC_20 exact** |
|---|---|---|
| Plaintext accuracy | 80.18% | 80.18% |
| Plaintext subset acc | 80% | 80% |
| FHE subset accuracy | 65% | 72% |
| Average inference time | 12.95 s | 146.7 s |
| Memory requirements | 2.68 GB | 7.7 GB |

Table 4.14: Comparison of PTQNN_SC_20 and PTQNN_SC_20 with exact parameters

### 4.4.5 Improved Model

The improved TFHE model QuantCNN_9 uses Brevitas layers for QAT in order to maintain as much accuracy as possible despite quantization. It has a deeper, more traditional CNN architecture, inspired by typical CNN classifiers such as the VGG models [83]. The model is relatively simple compared to these, in an attempt to strike a balance between performance and inference time. The model uses Average Pooling instead of Max Pooling layers, as this is recommended to speed up computations in Concrete ML documentation regarding CNNs on the CIFAR-10 dataset. The model is compiled using a recent version of Concrete ML, with a bit width limitation of 16 bits.

### Architecture and Parameters

QuantCNN_9 consists of the layers described in Table 4.15, and has a maximum bit width of 14 bits.

| **Layer** | **Details** |
|---|---|
| Quantized Conv Layer | 32 kernels. Kernel shape: 3x3. Stride: 1. Padding: 1. |
| Quantized ReLU Activation | |
| Average Pooling | Kernel shape: 3x3. Stride: 3. Padding: 0. |
| Quantized Conv Layer | 32 kernels. Kernel shape: 3x3. Stride: 1. Padding: 1. |
| Quantized ReLU Activation | |
| Quantized Conv Layer | 64 kernels. Kernel shape: 3x3. Stride: 1. Padding: 0. |
| Quantized ReLU Activation | |
| Average Pooling | Kernel shape: 3x3. Stride: 3. Padding: 0. |
| Final Quantized Linear layer | Input size: 9216. Output size: 12 |

Table 4.15: Model architecture of QuantCNN_9

The parameters of QuantCNN_9 is shown in Table 4.16. The quantization is done using Brevitas functionality that has been combined into a ”power of two”-scaling data type for quantization in Concrete ML, which aims to reduce inference time using TFHE.

| **Property** | **Value** |
|---|---|
| Models | QuantCNN_9 |
| Quantization Bits | 5 bits |
| Rounding Threshold Bit Operator | 6 |
| Method | ”APPROXIMATE” |
| Global P Error | 0.1 |

Table 4.16: Parameters for QNN_MNIST_4 and QNN_SC_4

### Results

Table 4.17 summarizes the results for QuantCNN_9 . It achieved a plaintext test accuracy of 89.55% on the SC dataset, and a plaintext accuracy of 91% on the test subset. The compiled model achieved an accuracy of 89% on the test subset with encrypted inference. The average encrypted inference time per sample was 469.8 seconds and the model used 5.91 GB during inference.

The confusion matrix of QuantCNN_9 is shown in Fig. 4.13, where it is apparent that the model follows the same pattern as previous models; it struggles most with the unknown class and the similar words ”No” and ”Go”.

| **Model** | **QuantCNN_9** |
|---|---|
| Plaintext accuracy | 89.55% |
| Plaintext subset acc | 91% |
| FHE subset accuracy | 89% |
| Average inference time | 469.8 s |
| Memory requirements | 5.91 GB |

Table 4.17: Results of QuantCNN_9 on the SC dataset

## 4.4.6 TFHE Models Summary

The improved TFHE model achieves a significantly better accuracy than the other TFHE models, but also has a much greater inference time. It is compiled using parameters that allow for inaccuracies, which results in a discrepancy of 2% between plaintext and encrypted performance on the test subset. As the experimentation with different compilation parameters for PTQNN_SC_20 demonstrated, these inaccuracies do significantly affect inference time and memory use. Notably, even allowing inaccuracies to speed up computations, QuantCNN_9 still uses minutes on encrypted inference per sample.

The framework provided by Concrete ML is flexible and supports models of various configurations. The very fast models had varying success in capturing the complexities of the SC dataset, with the most heavily quantized model QNN_SC_4 providing the poorest results. Conversely, inference time

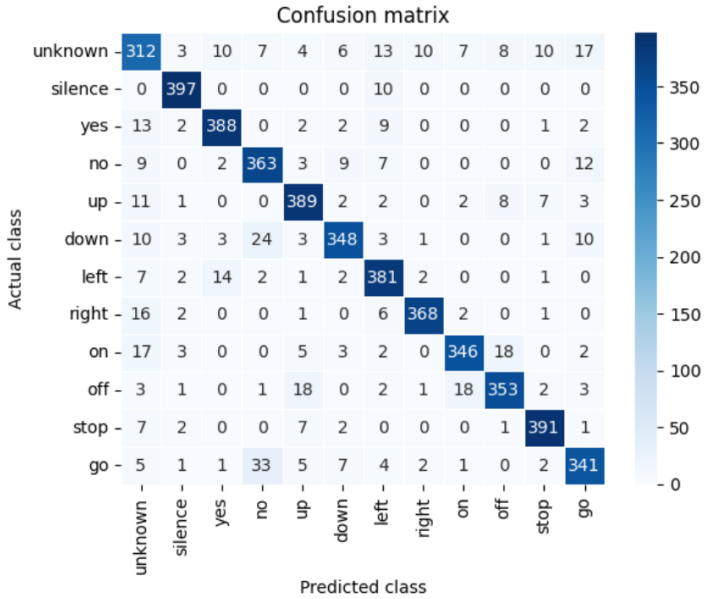


Figure 4.13: Confusion matrix of QuantCNN_9

appears to be a limitation for TFHE models when using deeper architectures with convolutional and pooling layers.

## 4.5 Model Comparisons

The CKKS and TFHE encryption schemes have different limitations, restrictions and strengths in implementing CNNs. In this section, an attempt is made at making more direct comparisons between models using CKKS and TFHE, with regards to accuracy, inference time and memory consumption. Since both the CKKS and TFHE models are implemented using libraries that ensure 128-bit security, the security level is not taken into account.

### 4.5.1 Challenges in Comparison

Specific restrictions based on both the inherent qualities in each scheme, and on the libraries in implementing them, makes it difficult to make a fair comparison of the two encryption schemes using the same machine learning model.

A model using CKKS must, for instance, either use a simpler non-linear activation function such as square activation or approximate ReLU with a higher-degree polynomial. The accuracy of such an approximation is restricted by the levels in a leveled scheme, and a larger polynomial modulus degree to support more levels will in turn affect the inference time.

A TFHE model can, however, evaluate the ReLU function while bootstrapping. Limiting such a model to a square activation function would be counter-intuitive and not indicative of the performance of a TFHE model. Thus, the choice of activation function would be ”unfair” for either the CKKS or TFHE model, giving a wrongful impression of how the scheme can be used in a CNN.

Another difference is the need for quantization. TFHE models need to quantize values to control the maximum bit width in the circuit, while CKKS models do not. Quantization, that might be detrimental to model performance, is thus a necessity only for TFHE models.

The following comparisons are thus meant to highlight the impact of the scheme-specific restrictions, and to give an intuition of the difference in inference time and memory use when the accuracy is similar. This will be done firstly by using simple linear regression models as a baseline, and investigating the effect of quantization on a plaintext model that has also been used with CKKS. Then, by looking at the results of the two improved models of Section 4.3.5 and Section 4.4.6, which has a similar level of

accuracy.

### 4.5.2 Linear Regression Models

As a baseline comparison, a single layer model was implemented and tested for both CKKS and TFHE on the SC dataset. A single linear layer in a model is equivalent to a linear regression, and no activation function is used. Three different models were tested: Linear_1, a linear regression model without quantization used for CKKS, PTQLinear_1, the same model but compiled for TFHE using PTQ, and QuantLinear_1, a linear regression model trained with QAT and compiled for TFHE.

#### Linear_1

Linear_1 achieved a plaintext accuracy of 51.45% and 51% on test set and test subset respectively. The CKKS scheme was used with $N = 8192$, bits scale of 30 and coefficient modulus bit sizes of $[60, 30, 60]$. Encrypted inference using CKKS gave an accuracy of 51% on the test subset, with an average inference time of 0.71 seconds per sample and a memory usage of 1.3 GB.

#### PTQLinear_1

PTQLinear_1 is a post-training quantized version of Linear_1. It was quantized to 6 bits during compilation to an TFHE-compatible model, resulting in a maximum bit width of 14 bits. Both global_p_error, p_error and rounding_threshold_bits were set to None. Encrypted inference using TFHE gave an accuracy of 30% on the test subset, with an average inference time of 0.11 seconds per sample and a memory usage of 1.39 GB.

#### QuantLinear_1

QuantLinear_1 achieved a plaintext accuracy of 49.8% and 50% on the test set and the test subset respectively. It was trained using a quantization of 6 bits. It was compiled into a TFHE-compatible model of maximum bit width of 15 bits, with the rounding_bit_threshold operator set to 6 bits with the ”APPROXIMATE” flag, and a global_p_error of 0.1. The accuracy for

encrypted inference on the test subset was 36%, with an average inference time of 16.4 seconds and a memory usage of 3.16 GB.

Removing the allowed error margins for the global_p_error, and setting the rounding_bit_threshold flag to ”EXACT”, the same model achieved an accuracy for encrypted inference of 48%, with an average inference time of 55.7 seconds and a memory usage of 6.1 GB.

#### Summary of Regression Models

The results from the linear regression models are summarized in Table 4.18. The models show which impact the compilation parameters has for the models using TFHE. Where the model using CKKS can aim to achieve the same accuracy as the plaintext model, the TFHE models must balance quantization, bit width constraints and inference time.

PTQLinear_1, which was quantized after training, had both a low accuracy and a low inference time. QuantLinear_1, which was trained using QAT, had almost the same plaintext accuracy as the unquantized model. It also had a higher potential to retain accuracy after compilation, but - depending on the parameters - at a cost of increase in inference time.

| **Model** | **Linear_1** | **PTQLinear_1** | **QuantLinear_1** | **QuantLinear_1 exact** |
|---|---|---|---|---|
| Plaintext accuracy | 51.45% | 51.45% | 49.8% | 49.8% |
| Plaintext subset acc | 51% | 51% | 50% | 50% |
| FHE subset accuracy | 51% | 30% | 36% | 48% |
| Average inference time | 0.71 s | 0.11 s | 16.4 s | 55.7 s |
| Memory requirements | 1.3 GB | 1.39 GB | 3.16 GB | 6.1 GB |

Table 4.18: Comparison of linear regression models

### 4.5.3 Quantization of Improved CKKS Model

The CNN_6 model from Section 4.3.5 had a plaintext accuracy of 87.58% on the SC dataset, and 87% on the test subset for both plaintext and encrypted inference. To get an indication of the differences between this CKKS model and a similar TFHE model, a corresponding quantized model was trained, compiled and used for TFHE inference. This model, QuantCNN_6 , has the same model architecture as CNN_6 , but is trained using QAT and uses the ReLU activation function.

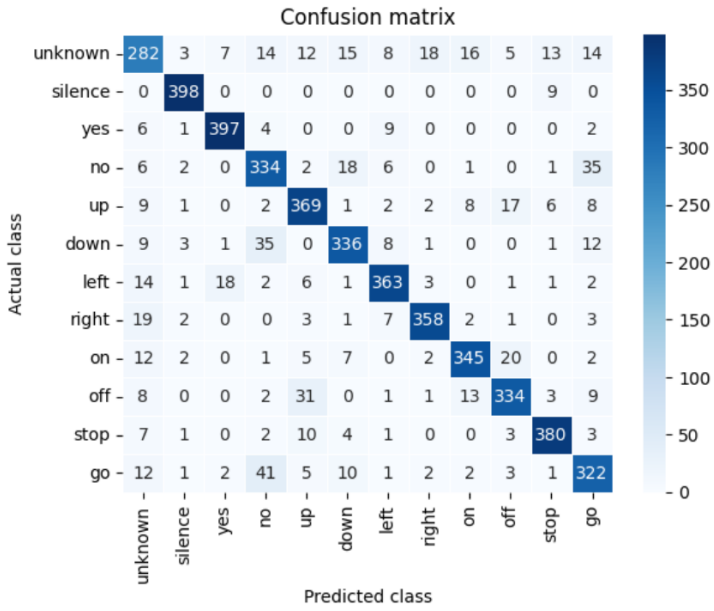


Figure 4.14: Confusion matrix for QuantCNN_6

QuantCNN_6 was trained with a bit quantization of 5 bits using Brevitas layers. It was compiled with a rounding_bit_threshold set to 6 bits, using the ”APPROXIMATE” flag. The global_p_error was set to 0.1. The maximum bit width of the model was 14 bits.

### Results

QuantCNN_6 achieved a plaintext test accuracy of 86.29% and 86% on the test set and test subset respectively. Encrypted inference using TFHE gave an accuracy of 84%, with an average inference time of 288.64 seconds and a memory use of 4.91 GB. The confusion matrix for the model is shown in Fig. 4.14.

Table 4.19 gives an overview of how QuantCNN_6 with TFHE compares to CNN_6 with CKKS. Quantization seem to have only minimally reduced the plaintext accuracy of QuantCNN_6 , but the compilation into a TFHE model cost an additional 2% in accuracy. Even compiling with an allowance for inaccuracies, the inference time was much higher than that of CNN_6 . Thus, QuantCNN_6 is both less accurate and slower than CNN_6 with the same model architecture. However, with regards to memory usage,

QuantCNN_6 required only a fraction of the memory of CNN_6 .

| **Model** | **CNN_6** | **QuantCNN_6** |
|---|---|---|
| Plaintext accuracy | 87.58% | 86.29% |
| Plaintext subset acc | 87% | 86% |
| FHE subset accuracy | 87% | 84% |
| Average inference time | 154.5 s | 288.64 s |
| Memory requirements | 22.7 GB | 4.9 GB |

Table 4.19: Comparison of CNN_6 and QuantCNN_9

### 4.5.4 Comparison of Improved Models

CNN_6 has a plaintext test accuracy of 87.58%, while QuantCNN_9 had an accuracy of 89.55%. The plaintext performances of the models are close, with the TFHE model outperforming CNN_6 slightly by 2%. The results from the two models are summarized in Table 4.20. Notably, the TFHE model is approximately 5 minutes slower per inference, but needs only a fourth of the memory.

The CKKS model retains its accuracy when doing encrypted inference, while the TFHE model loses 2%. This loss of accuracy is due to the parameters set for the TFHE model when compiled into an FHE-compatible model. As shown with QuantLinear_1 and PTQNN_SC_20 from section 4.4.6, allowing some inaccuracies greatly speeds up inference time for TFHE models. Given QuantCNN_9 inference time of 470 seconds, switching to parameters enforcing a more exact approach would likely result in a great increase in inference time.

CNN_6 has a simpler architecture than QuantCNN_9 which likely contributes to the large difference in inference time. Still, as seen with QuantCNN_6 , the difference in model complexity does not in itself account for the full discrepancy in inference time.

Furthermore, even though CKKS is an approximate scheme, CNN_6 has no discrepancy between plaintext and encrypted inference. TFHE has the potential to be exact, but the high inference time necessitates allowing some inaccuracies to speed up computations, resulting in discrepancies for QuantCNN_9 higher than its CKKS counterpart.

The memory consumption of CNN_6 already exceeds what is typically found on a personal laptop, while QuantCNN_9 could likely be run on most computers. This makes QuantCNN_9 a more available model for private use and experimentation, as there are no demanding hardware requirements to run it.

| **Model** | **CNN_6** | **QuantCNN_9** |
|---|---|---|
| Plaintext accuracy | 87.58% | 89.55% |
| Plaintext subset acc | 87% | 91% |
| FHE subset accuracy | 87% | 89% |
| Average inference time | 154.5 s | 469.8 s |
| Memory requirements | 22.7 GB | 5.9 GB |

Table 4.20: Comparison of CNN_6 and QuantCNN_9

### 4.5.5 Comparison Summary

#### Accuracy, Speed and Memory

Several CKKS and TFHE models have been compared with regards to accuracy, inference time and memory consumption. The picture that emerges is that CKKS is generally faster than TFHE when applied in this machine learning setting. For the simplest models and those of equal complexity, CKKS also had higher accuracy. However, the difference between a non-quantized and quantized plaintext model was generally small when the quantized models were trained with quantization aware training. The improved TFHE model achieved the highest accuracy, and was also the most complex.

While CKKS models had none or negligible discrepancies between plaintext and encrypted inference accuracy, the TFHE models lost additional accuracy after compilation. This was true for both post-quantized and QAT models. The accuracy loss was intentional; both to ensure that the TFHE-compiled model met its bit width requirement through quantization, and to achieve more practically feasible inference times through the allowance of error margins. Pushing a TFHE model towards an exact result vastly increased inference time.

The CKKS models had a much higher memory requirement, and the memory requirement increased with model complexity. Even the improved

TFHE model remained at a level where it could be run on an older personal laptop.

Improving the accuracy came at a great cost to inference time for both the TFHE and CKKS scheme, and for the latter also gave a significant rise in required memory.

### Other Considerations

The TFHE models were implemented using Concrete ML, which is to a large degree compatible with the machine learning library PyTorch. TFHE models could thus be easily designed, configured and trained in a plaintext version using PyTorch and/or Brevitas. FHE-specific parameters were set automatically. It was easy to implement models with an arbitrary number of layers, since Concrete ML provides programmable bootstrapping.

The CKKS models were also initially PyTorch models, but exporting learned parameters to the encrypted versions were done manually. FHE parameters for CKKS had to be set specifically for each model architecture to get the required levels, ciphertext sizes, and precision.

This difference is due to the libraries used. Zama provides state-of-the-art TFHE implementations for machine learning in Concrete ML. They have a high focus on compatibility with existing machine learning frameworks and user accessibility. There is currently no CKKS equivalent to such a library. This means that TFHE is, per now, easier to use in machine learning applications, especially for those without cryptographic expertise and those more familiar with Python as the ”traditional” machine-learning and data-science programming language. It also means, however, that some of the CKKS models in this project might have a greater potential for optimizations in performance, as more of the code is self-written with a focus on functionality, and parameter choices are set based on experimentation.

# Chapter 5

# Results and Discussions

## 5.1 Overview

This thesis has sought to investigate whether FHE systems can be used on a speech command recognition problem. As part of this investigation, the practical applications of different FHE systems through representative cryptographic libraries have been explored and their suitability for the speech command recognition problem considered.

### 5.1.1 Two Main FHE Schemes

CKKS and TFHE was introduced as the two main FHE schemes in use today, and consequently also the FHE systems used in the thesis experiment. As with all practical use of current FHE systems, they have challenges with high runtime and high memory usage. Both schemes have previously been used with machine learning models, including deep learning models such as convolutional neural networks. The inherent qualities of each scheme gives them unique strengths and challenges in these applications.

CKKS is an approximate scheme using real numbers, where separate values can be packed into the same ciphertexts and evaluated in parallel for efficiency. Very large ciphertexts can, however, lead to a high memory usage. It can evaluate polynomial expressions; any functions that cannot be expressed as a polynomial, such as typical activation functions, must be approximated. CKKS is often, also here, used with a leveled approach. While this is faster, it also requires careful consideration of cryptographic parameters to ensure that the scheme can handle the needed number of multiplications.

TFHE uses its special programmable bootstrapping to both evaluate non-polynomial univariate functions and reduce noise at the same time. While bootstrapping is a costly procedure, it allows for an indefinite number of multiplications. TFHE is also an exact scheme over the integers; this gives the potential to exactly match a plaintext result, but also means floating point values generally used in machine learning must be quantized.

### 5.1.2 Encrypted Inference

FHE systems can be used to achieve privacy-preserving machine learning, either separately or in combination with other approaches. Often, the FHE

systems are used to ensure a machine learning model does not read sensitive data, but instead only processes data in its encrypted form. In this way, the owner of the data retains full control, even when data is sent for processing by a model. This means that FHE systems can be used for both encrypted training and inference, where the latter was the focus in the thesis experiments. There is research on both the CKKS and TFHE schemes used for encrypted inference in a variety of different machine learning models, demonstrating that the schemes are applicable, but also highlighting challenges in runtime and memory usage.

The speech command recognition problem is a machine learning problem where short commands are classified in accordance with a predefined set of speech command classes. A dataset was released from Google in 2018 to facilitate development of speech command recognition models. The one second utterances in the dataset can be processed into spectrograms where the speech data is presented as an image, and thus the problem can be approached as an image recognition problem. This opened up the possibility to use convolutional neural networks, a type of model that is suitable and often preferred for image classification tasks. The use of such models together with the CKKS and TFHE schemes has been explored in previous research, providing some insight into how this can be implemented practically using existing cryptographic libraries.

### 5.1.3 Cryptographic Libraries

CKKS have several comprehensive cryptographic libraries, both closed and open-source, but none specifically designed for machine learning. A leading and widely used library from Microsoft called SEAL was chosen for CKKS implementations. SEAL is written in C++, which means it is challenging to use with other machine learning frameworks where Python is the preferred programming language. There are many Python wrappers for SEAL, where TenSEAL in particular is a Python wrapper that also provides additional functionality for machine learning, such as the tensor datatype from which it is named.

TFHE has one state-of-the-art library called Concrete ML from Zama, which is designed for machine learning and runs over the TFHE framework Concrete from the same company. Concrete ML is a Python library and is aimed at those without cryptographic expertise, and who also use other

machine learning frameworks.

SEAL with TenSEAL and Concrete ML was used to implement the models with CKKS and TFHE that was used in the experiment. They both enforce a 128-bit security level. They also have some limitations, such that the use of these libraries requires some additional considerations.

TenSEAL's machine learning functionality is limited, which means that additional functions had to be manually programmed and thus are not necessarily optimal in their implementations. Cryptographic parameters must be set specifically for each machine learning model, and there is no way to import or automatically transform a plaintext model from a common framework into a model using CKKS. As TenSEAL runs over SEAL, it adheres to SEAL's restrictions on ciphertext size and its leveled approach.

Concrete ML sets cryptographic parameters automatically, and can compile plaintext models directly into a TFHE counterpart. Most common machine learning functionality is supported, such as traditional layers used for deep learning models and different activation functions. The TFHE implementation is, however, restricted in the number of bits it can handle in its bootstrapping operation. This limitation on maximum bit width in a cryptographic circuit often requires input to be quantized to a smaller bit size, where quantization can affect the accuracy of the models. Compilation parameters must be chosen carefully to strike a balance between runtime optimizations and accuracy.

## 5.2 Findings from Experiment

A variety of different models have been tested on the speech commands (SC) dataset. Both cryptographic libraries TenSEAL and Concrete ML have presented models using the respective FHE schemes on the MNIST dataset of handwritten digits. These models were applied to the SC dataset to give an initial starting point, and to give some intuition on the difference in complexity between the SC spectrograms and the MNIST digits.

A new set of models with more complex architectures was then tested on the SC dataset. They were implemented with the goal of achieving a better performance on the SC dataset, and showcase the performance and limitations of models using CKKS and TFHE when the complexity increases.

Finally, some models were implemented and tested in an attempt to conduct a fair comparison of the two schemes. This included linear models, quantized and compiled TFHE versions of plaintext models that were also the basis for CKKS models, and models with a comparable architecture. The comparison was concluded by looking at the results of the two best performing models for CKKS and TFHE.

### 5.2.1 Using FHE for Speech Command Recognition

The experiments show that inference on the SC dataset is more difficult than that of MNIST. The accuracy of the MNIST models drop and the resource use increases when the models are retrained on the SC dataset and tested using encrypted inference. Regardless, it is apparent that even simple MNIST models can achieve a decent accuracy, such as for the CKKS model, with an accuracy of 80%. Even the linear models exceeded that of guessing, which given the class distribution and number of classes would land at a performance of roughly 8%.

Another takeaway is that increasing the complexity, and consequently the performance, of the models came at a high cost in inference time and memory usage. When the performance increased towards 90% accuracy, the resource use was too high for the model to be feasibly applied to the speech command recognition problem. This was true both for models designed for CKKS and TFHE.

For a speech command recognition problem, the user experience depends on the models' ability to both quickly and accurately recognize a keyword. Often, one would want to use such a model on an IoT device, requiring a low footprint. It is demonstrably possible for those without cryptographic expertise to do encrypted inference with CNNs on the SC dataset using existing libraries for CKKS and TFHE, and to run such a model on consumer hardware. However, accuracies below 90% with inference times of minutes and memory consumption measured in several GBs, show that there is a way to go before such an approach would be viable to use in practice.

### 5.2.2 Comparison of Models using CKKS and TFHE

The overall picture emerging from comparing the use of the CKKS and TFHE schemes is that CKKS is faster. A simple CKKS model achieved an

accuracy of 80% on the SC dataset in a few seconds. The more complex CKKS model generally achieved higher accuracy with lower inference time than equivalent TFHE counterparts. This speed advantage is likely in part due to how the packing of CKKS ciphertexts enable parallel calculations. Another contributing factor is probably that the SEAL library provides a leveled CKKS scheme; while this necessitates more careful parameter choices, it also means that time consuming bootstrapping operations are avoided. A drawback for all CKKS models was the high memory consumption, with the most complex CKKS model requiring more memory than what is commonly found on most laptops.

The models using TFHE were slower, and they also lost more accuracy in the transition from plaintext to encrypted inference. This accuracy loss was caused to some degree by the need for quantization, but also by optimizations that introduced error margins to speed up computations. The TFHE models only required a fraction of the memory needed for the CKKS models.

The model with the highest accuracy was a TFHE model with a deeper architecture than the most complex CKKS model. This model took advantage of how TFHE uses programmable bootstrapping, and therefore is not restricted by levels and corresponding parameter choices in model design. The model was, however, several minutes slower per inference than the best CKKS model.

While a fair comparison was attempted, the performance of the different schemes also depends on the state of the cryptographic libraries used. Since Concrete ML is a library specially designed for machine learning using TFHE, one must assume that its implementation is also optimized for peak machine learning performance. The lack of such a comprehensive and specialized machine learning library for CKKS possibly means that the results could be better given additional optimizations.

### 5.2.3 Availability and Compatibility

While accuracy, speed, and memory usage are the basis for the comparisons of TFHE and CKKS, there are other factors that influence the practical appliance of an FHE scheme to a machine learning problem. These factors are closely linked to the state of the current cryptographic libraries.

Currently, the CKKS scheme is a less available and user-friendly alternative for machine learning: It requires more cryptographic knowledge for

setting parameters, it is less flexible with regards to supported functionality, its leveled approach and the need to hard-code cryptographic parameters manually for each different model. It also lacks compatibility with common machine learning frameworks, making it less available for those seeking to explore its use from a machine learning point of view or experiment with it on existing projects.

For comparison, TFHE provides extensive machine learning functionality and automatic setting of cryptographic parameters in Concrete ML. Here, compatibility with both scikit-learn and PyTorch is also a focus area: A PyTorch model can for instance, with few adjustments, be directly imported and compiled into an equivalent model using TFHE. TFHE can therefore easily be tested on projects using popular machine learning frameworks, and can also be run on most consumer laptops.

The runtime of a single encrypted inference is the most obvious challenge to overcome before either CKKS or TFHE is viable for practical use on the SC dataset, and CKKS is currently faster. Yet this does not mean that the runtime differences will be the most important factor in the years to come. Technological advancements can evolve quickly, and there are research areas that could revolutionize the runtime of FHE systems, for instance through the development of specialized hardware. In such a case, could ease-of-use and compatibility with popular machine learning frameworks come to play the more important role when choosing a suitable FHE scheme?

A scheme that is easy to experiment with on consumer hardware, compatible with other frameworks, and requires little prior cryptographic knowledge could for many machine learning problems be more preferable than a scheme which is faster. While these are areas where TFHE currently outperform CKKS, it remains to be seen whether those differences would remain if more mature machine-learning specific CKKS libraries were developed.

### 5.2.4 Future Work

The experiment approach has been to use CNNs for speech command recognition. This approach was chosen due to its straightforwardness, and because there is previous research on CNNs that use either CKKS or TFHE for encrypted inference. The experiment focused on the FHE schemes and sought to take advantage of existing research, demonstrations and implemented functionality.

There are other ways to tackle this classification problem, either using different types of models or CNNs in combination with other techniques; this can result in both a lower footprint and higher accuracy. Thus, there is unexplored potential in addressing the problem primarily from a machine learning perspective. Could refining the approach with an aim to reduce footprint, increase accuracy and limit the number of large multiplications result in a model that both performs better and potentially is also better suited for use with homomorphic encryption?

An additional area worth exploring would be to combine such a refined model with an implementation of an FHE scheme specifically optimized for the model in question, and where cryptographic parameters and model architecture were meticulously tuned to each other.

### 5.2.5 Concluding Remarks

CKKS is currently the most suitable scheme for achieving a feasible runtime on the SC dataset. This does not mean that CKKS necessarily is the most promising scheme going forward, as memory use, availability and compatibility might come to play a more important role. Regardless, both models using CKKS and TFHE were too slow and required too much memory to be a practically feasible approach for speech command recognition.

Speech command recognition is a small part of achieving encrypted inference on speech data. It deals with keywords that might activate a microphone, but not the speech data that is subsequently captured. In the long term, encrypted inference would ideally be available for all kinds of speech data sent for remote processing, not only smaller keywords that also potentially could be processed on a local device. While an image classification approach on spectrograms might not be directly related to all other kinds of speech recognition, the results still implies that there is a long way to go before data such as live speech can be processed using FHE.

Still, the experiment demonstrates that FHE schemes and their implementations are available for experimentation those without cryptographic expertise, using consumer hardware and open source libraries. While the current state of FHE and its cryptographic libraries impose a series of limitations and might not be feasible for very complex problems or problems requiring quick inference times, FHE might still be a viable approach for other kinds of machine learning problems. It is thus an alternative for de-

velopers and data scientists interested in privacy preserving machine learning to test FHE as a way of securing user data during model processing.

# Bibliography


[1] (2022). Clara Train examples. URL: https://github.com/NVIDIA/clara-train-examples.

[2] (2022). NVIDIA Clara Imaging. URL: https://developer.nvidia.com/clara-medical-imaging.

[3] (2024). Concrete ML GitHub repository. URL: https://github.com/zama-ai/concrete-ml/issues/390.

[4] (2024). NVIDIA Data Loading Library (DALI). URL: https://docs.nvidia.com/deeplearning/dali/user-guide/docs/index.html.

[5] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., & Zheng, X. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. URL: https://www.tensorflow.org/ software available from tensorflow.org.

[6] Albrecht, M., Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Halevi, S., Hoffstein, J., Laine, K., Lauter, K., Lokam, S., Micciancio, D., Moody, D., Morrison, T., Sahai, A., & Vaikuntanathan, V. (2018). *Homomorphic Encryption Security Standard*. Technical Report HomomorphicEncryption.org Toronto, Canada. URL: https://homomorphicencryption.org/standard/.

[7] Arora, M. (2012). EE Times. URL: https://www.eetimes.com/how-secure-is-aes-against-brute-force-attacks/?_ga.

[8] Atali, G., & Kırcı, S. (2021). Real-Time Bicycle Detection with Deep Learning Network Using TensorFlow. URL: https://www.researchgate.net/publication/354691567.

[9] Badawi, A. A., Alexandru, A., Bates, J., Bergamaschi, F., Cousins, D. B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., Liu, Z., Micciancio, D., Pascoe, C., Polyakov, Y., Quah, I., R.V., S., Rohloff, K., Saylor, J., Suponitsky, D., Triplett, M., Vaikuntanathan, V., & Zucca, V. (2022). OpenFHE: Open-Source Fully Homomorphic Encryption Library. Cryptology ePrint Archive, Paper 2022/915. URL: https://eprint.iacr.org/2022/915 https://eprint.iacr.org/2022/915.

[10] Barker, E., Chen, L., Roginsky, A., Davis, R., & Simon, S. (2019). Recommendation for Pair-Wise Key Establishment Using Integer Factorization Cryptography. doi:https://doi.org/10.6028/NIST.SP.800-56Br2.

[11] Benaissa, A., Retiat, B., Cebere, B., & Belfedhal, A. E. (2021). TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption. arXiv:2104.03152.

[12] Benaissa, A., Retiat, B., Cebere, B., & Belfedhal, A. E. (2021). TenSEAL: A Library for Encrypted Tensor Operations Using Homomorphic Encryption. URL: https://arxiv.org/abs/2104.03152. arXiv:2104.03152.

[13] Bharti, R., Khamparia, A., Shabaz, D. M., Dhiman, G., Pande, S., & Singh, P. (2021). Prediction of Heart Disease Using a Combination of Machine Learning and Deep Learning. *Computational Intelligence and Neuroscience*, *2021*. doi:10.1155/2021/8387680.

[14] Bishop, C. M. (2007). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. (1st ed.). Springer. URL: https://www.amazon.com/Pattern-Recognition-Learning-Information-Statistics/dp/0387310738.

[15] Bost, R., Popa, R. A., Tu, S., & Goldwasser, S. (2015). Machine Learning Classification over Encrypted Data. In *Proceedings 2015 Network and Distributed System Security Symposium* NDSS 2015. Internet Society. URL: http://dx.doi.org/10.14722/ndss.2015.23241. doi:10.14722/ndss.2015.23241.

[16] Brakerski, Z., Gentry, C., & Vaikuntanathan, V. (2012). (Leveled) fully homomorphic encryption without bootstrapping. In S. Goldwasser (Ed.), *ITCS* (pp. 309–325). ACM. URL: http://dblp.uni-trier.de/db/conf/innovations/innovations2012.html#BrakerskiGV12.

[17] Bridle, J. (1989). Training Stochastic Model Recognition Algorithms as Networks can Lead to Maximum Mutual Information Estimation of Parameters. In D. Touretzky (Ed.), *Advances in Neural Information Processing Systems*. Morgan-Kaufmann volume 2. URL: https://proceedings.neurips.cc/paper_files/paper/1989/file/0336dcbab05b9d5ad24f4333c7658a0e-Paper.pdf.

[18] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., & Amodei, D. (2020). Language Models are Few-Shot Learners. URL: https://arxiv.org/abs/2005.14165. arXiv:2005.14165.

[19] Brutzkus, A., Gilad-Bachrach, R., & Elisha, O. (2019). Low Latency Privacy Preserving Inference. In K. Chaudhuri, & R. Salakhutdinov (Eds.), *Proceedings of the 36th International Conference on Machine Learning* (pp. 812–821). PMLR volume 97 of *Proceedings of Machine Learning Research*. URL: https://proceedings.mlr.press/v97/brutzkus19a.html.

[20] Bzdok, D., Altman, N., & Krzywinski, M. (2018). Statistics versus machine learning. *Nature Methods*, *15*, 233–234. URL: https://doi.org/10.1038/nmeth.4642. doi:10.1038/nmeth.4642.

[21] Bäckström, T., Räsänen, O., Zewoudie, A., Zarazaga, P. P., Koivusalo, L., Das, S., Mellado, E. G., Mansali, M. B., Ramos, D., Kadiri, S., Alku, P., & Vali, M. H. (2022). *Introduction to Speech Processing*. (2nd ed.). URL: https://speechprocessingbook.aalto.fi. doi:10.5281/zenodo.6821775.

[22] Chee, F. Y. (2024). Reuters. URL: https://tinyurl.com/yc6ad6mn.

[23] Chen, G., Parada, C., & Heigold, G. (2014). Small-footprint keyword spotting using deep neural networks. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 4087–4091). doi:10.1109/ICASSP.2014.6854370.

[24] Cheon, J. H., Kim, A., Kim, M., & Song, Y. (2016). Homomorphic Encryption for Arithmetic of Approximate Numbers. Cryptology ePrint Archive, Paper 2016/421. URL: https://eprint.iacr.org/2016/421 https://eprint.iacr.org/2016/421.

[25] Cheon, J. H., Kim, A., Kim, M., & Song, Y. (2017). Homomorphic Encryption for Arithmetic of Approximate Numbers. In T. Takagi, & T. Peyrin (Eds.), *Advances in Cryptology – ASIACRYPT 2017* (pp. 409–437). Cham: Springer International Publishing.

[26] Chillotti, I. (2022). Zama. URL: https://www.zama.ai/post/tfhe-deep-dive-part-1.

[27] Chillotti, I., Gama, N., Georgieva, M., & Izabachène, M. (2018). TFHE: Fast Fully Homomorphic Encryption over the Torus. Cryptology ePrint Archive, Paper 2018/421. URL: https://eprint.iacr.org/2018/421 https://eprint.iacr.org/2018/421.

[28] Cho, A. (2024). Medium. URL: https://medium.com/rlracingproject.

[29] Choi, H., Kim, J., Kim, S., Park, S., Park, J., Choi, W., & Kim, H. (2024). UniHENN: Designing Faster and More Versatile Homomorphic Encryption-Based CNNs Without im2col. *IEEE Access*, *12*, 109323–109341. URL: http://dx.doi.org/10.1109/ACCESS.2024.3438996. doi:10.1109/access.2024.3438996.

[30] Cireşan, D., Meier, U., & Schmidhuber, J. (2012). Multi-column Deep Neural Networks for Image Classification. URL: https://arxiv.org/abs/1202.2745. arXiv:1202.2745.

[31] Deng, L. (2012). The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, *29*, 141–142. URL: https://ieeexplore.ieee.org/document/6296535.

[32] Dubey, S. R., Singh, S. K., & Chaudhuri, B. B. (2022). Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark. URL: https://arxiv.org/abs/2109.14545. arXiv:2109.14545.

[33] Er, Y., & Atasoy, A. (2016). The Classification of White Wine and Red Wine According to Their Physicochemical Qualities. *International Journal of Intelligent Systems and Applications in Engineering*, *4*, 23–23. doi:10.18201/ijisae.265954.

[34] Folkerts, L., Gouert, C., & Tsoutsos, N. G. (2023). REDsec: Running Encrypted Discretized Neural Networks in Seconds. In *Network and Distributed System Security Symposium (NDSS)* (pp. 1–17). URL: https://tinyurl.com/yb6x6f99.

[35] Gentry, C. (2009). Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher (Ed.), *STOC* (pp. 169–178). ACM. URL: https://www.bibsonomy.org/bibtex/2fcbbc53f119b352e68169a37082f8714/ytyoun.

[36] Gouert, C., Joseph, V., Dalton, S., Augonnet, C., Garland, M., & Tsoutsos, N. G. (2023). Accelerated Encrypted Execution of General-Purpose Applications. Cryptology ePrint Archive, Paper 2023/641. URL: https://eprint.iacr.org/2023/641.

[37] Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: data mining, inference and prediction*. (2nd ed.). Springer. URL: http://www-stat.stanford.edu/~tibs/ElemStatLearn/.

[38] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. URL: https://arxiv.org/abs/1512.03385. arXiv:1512.03385.

[39] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. URL: https://arxiv.org/abs/1207.0580. arXiv:1207.0580.

[40] Holdsworth, J., & Scapicchio, M. (2024). Ibm. URL: https://www.ibm.com/topics/deep-learning.

[41] Huyhn, D. (2020). openmined. URL: https://blog.openmined.org/ckks-explained-part-1-simple-encoding-and-decoding/.

[42] Iniyan, S., Akhil Varma, V., & Teja Naidu, C. (2023). Crop yield prediction using machine learning techniques. *Advances in Engineering Software*, *175*, 103326. URL: https://www.sciencedirect.com/science/article/pii/S0965997822002277. doi:https://doi.org/10.1016/j.advengsoft.2022.103326.

[43] Ishiyama, T., Suzuki, T., & Yamana, H. (2020). Highly accurate cnn inference using approximate activation functions over homomorphic encryption. (pp. 3989–3995). doi:10.1109/BigData50022.2020.9378372.

[44] Jung, W., Kim, S., Ahn, J. H., Cheon, J. H., & Lee, Y. (2021). Over 100x Faster Bootstrapping in Fully Homomorphic Encryption through Memory-centric Optimization with GPUs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2021*, 114–148. URL: https://tches.iacr.org/index.php/TCHES/article/view/9062. doi:10.46586/tches.v2021.i4.114-148.

[45] Kingma, D. P., & Ba, J. (2017). Adam: A Method for Stochastic Optimization. URL: https://arxiv.org/abs/1412.6980. arXiv:1412.6980.

[46] Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. URL: https://api.semanticscholar.org/CorpusID:18268744.

[47] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira,

C. Burges, L. Bottou, & K. Weinberger (Eds.), *Advances in Neural Information Processing Systems*. Curran Associates, Inc. volume 25. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[48] Laur, S., Lipmaa, H., & Mielikäinen, T. (2006). Cryptographically private support vector machines. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* KDD '06 (p. 618–624). New York, NY, USA: Association for Computing Machinery. URL: https://doi.org/10.1145/1150402.1150477. doi:10.1145/1150402.1150477.

[49] LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., & Jackel, L. D. (1989). Handwritten Digit Recognition with a Back-Propagation Network. In *Neural Information Processing Systems*. URL: https://api.semanticscholar.org/CorpusID:2542741.

[50] Lee, J.-W., Kang, H., Lee, Y., Choi, W., Eom, J., Deryabin, M., Lee, E., Lee, J., Yoo, D., Kim, Y.-S., & No, J.-S. (2022). Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access*, *10*, 30039–30054. doi:10.1109/ACCESS.2022.3159694.

[51] Li, J., Cheng, J.-h., Shi, J.-y., & Huang, F. (2012). Brief Introduction of Back Propagation (BP) Neural Network Algorithm and Its Improvement. In D. Jin, & S. Lin (Eds.), *Advances in Computer Science and Information Engineering* (pp. 553–558). Berlin, Heidelberg: Springer Berlin Heidelberg. URL: https://link.springer.com/chapter/10.1007/978-3-642-30223-7_87.

[52] Li, T., Sahu, A. K., Talwalkar, A., & Smith, V. (2020). Federated Learning: Challenges, Methods, and Future Directions. *IEEE Signal Processing Magazine*, *37*, 50–60. doi:10.1109/MSP.2020.2975749.

[53] Lin, G., & Shen, W. (2018). Research on convolutional neural network based on improved Relu piecewise activation function. *Procedia Computer Science*, *131*, 977–984. URL: https://www.sciencedirect.com/science/article/pii/S1877050918306197. doi:https://

doi.org/10.1016/j.procs.2018.04.239. Recent Advancement in Information and Communication Technology:.

[54] Ma, E. (2019). Towards Data Science. URL: https://towardsdatascience.com/data-augmentation-for-speech-recognition-e7c607482e78.

[55] MacKay, D. J. (1992). A practical Bayesian framework for backpropagation networks. *Neural computation*, *4*, 448–472. URL: https://ieeexplore.ieee.org/document/6796869.

[56] Makri, E., Rotaru, D., Smart, N., & Vercauteren, F. (2019). EPIC: Efficient Private Image Classification (or: Learning from the Masters): The Cryptographers' Track at the RSA Conference 2019, San Francisco, CA, USA, March 4–8, 2019, Proceedings. (pp. 473–492). doi:10.1007/978-3-030-12612-4_24.

[57] Marcolla, C., Sucasas, V., Manzano, M., Bassoli, R., Fitzek, F. H., & Aaraj, N. (2022). Survey on Fully Homomorphic Encryption, Theory, and Applications. Cryptology ePrint Archive, Paper 2022/1602. URL: https://eprint.iacr.org/2022/1602. doi:10.1109/JPROC.2022.3205665 https://eprint.iacr.org/2022/1602.

[58] Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (2001). *Handbook of Applied Cryptography*. CRC Press. URL: http://www.cacr.math.uwaterloo.ca/hac/.

[59] Mitchell, T. M. (1997). *Machine learning* volume 1. McGraw-hill New York. URL: https://www.bibsonomy.org/bibtex/2ea9f893d9d19c182bcf2822eb590fe4f/msteininger.

[60] Mohassel, P., & Zhang, Y. (2017). SecureML: A System for Scalable Privacy-Preserving Machine Learning. (pp. 19–38). doi:10.1109/SP.2017.12.

[61] Montero, L., Frery, J., Kherfallah, C., Bredehoft, R., & Stoian, A. (2024). Neural Network Training on Encrypted Data with TFHE. URL: https://arxiv.org/abs/2401.16136. arXiv:2401.16136.

[62] Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning* ICML'10 (p. 807–814). Madison, WI, USA: Omnipress. URL: https://dl.acm.org/doi/10.5555/3104322.3104425.

[63] Nassif, A., Shahin, I., Attili, I., Azzeh, M., & Shaalan, K. (2019). Speech Recognition Using Deep Neural Networks: A Systematic Review. *IEEE Access*, *PP*, 1–1. doi:10.1109/ACCESS.2019.2896880.

[64] O'Shea, K., & Nash, R. (2015). An Introduction to Convolutional Neural Networks. URL: https://arxiv.org/abs/1511.08458. arXiv:1511.08458.

[65] Pappalardo, A. (2023). Xilinx/brevitas. URL: https://doi.org/10.5281/zenodo.3333552. doi:10.5281/zenodo.3333552.

[66] Park, D. S., Chan, W., Zhang, Y., Chiu, C.-C., Zoph, B., Cubuk, E. D., & Le, Q. V. (2019). SpecAugment: A Simple Data Augmentation Method for Automatic Speech Recognition. In *Interspeech*. URL: https://api.semanticscholar.org/CorpusID:121321299.

[67] Park, S., Byun, J., Lee, J., Cheon, J. H., & Lee, J. (2020). HE-Friendly Algorithm for Privacy-Preserving SVM Training. *IEEE Access*, *8*, 57414–57425. doi:10.1109/ACCESS.2020.2981818.

[68] Park, S. H. (2023). *heaan.sdk.R: R Interface to 'HEaaN-SDK'*. URL: https://heaan.it/docs/stat/python/sdk.html r package version 0.2.0.

[69] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., & Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32* (pp. 8024–8035). Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[70] Peikert, C. (2015). A Decade of Lattice Cryptography. Cryptology ePrint Archive, Paper 2015/939. URL: https://eprint.iacr.org/2015/939 https://eprint.iacr.org/2015/939.

[71] PyTorch (2023). PyTorch Docs - Dropout. URL: https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html.

[72] Reddy, G. T., Reddy, M. P. K., Lakshmanna, K., Kaluri, R., Rajput, D. S., Srivastava, G., & Baker, T. (2020). Analysis of Dimensionality Reduction Techniques on Big Data. *IEEE Access*, *8*, 54776–54788. doi:10.1109/ACCESS.2020.2980942.

[73] Rieke, N. (2019). nvidia. URL: https://blogs.nvidia.com/blog/what-is-federated-learning/.

[74] Rieke, N. (2024). zama. URL: https://docs.zama.ai/concrete-ml/built-in-models/training.

[75] Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, *21*, 120–126. URL: https://doi.org/10.1145/359340.359342. doi:10.1145/359340.359342.

[76] Roth, H., Zephyr, M., & Harouni, A. (2021). nvidia. URL: https://developer.nvidia.com/blog/federated-learning-with-homomorphic-encryption/.

[77] Rovida, L., & Leporati, A. (2024). Encrypted image classification with low memory footprint using fully homomorphic encryption. *International journal of neural systems*, *34*, 2450025. doi:10.1142/S0129065724500254.

[78] Ruder, S. (2017). An overview of gradient descent optimization algorithms. URL: https://arxiv.org/abs/1609.04747. arXiv:1609.04747.

[79] Ruehle, V. e. a. (2021). microsoft. URL: https://tinyurl.com/4fa4z4cv.

[80] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., &

Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, *115*, 211–252. doi:10.1007/s11263-015-0816-y.

[81] Sainath, T. N., & Parada, C. (2015). Convolutional neural networks for small-footprint keyword spotting. In *Interspeech 2015* (pp. 1478–1482). doi:10.21437/Interspeech.2015-352.

[82] SEAL (2023). Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[83] Simonyan, K., & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. URL: https://arxiv.org/abs/1409.1556. arXiv:1409.1556.

[84] Stoian, A., Frery, J., Bredehoft, R., Montero, L., Kherfallah, C., & Chevallier-Mames, B. (2023). Deep Neural Networks for Encrypted Inference with TFHE. Cryptology ePrint Archive, Paper 2023/257. URL: https://eprint.iacr.org/2023/257.

[85] strojax (). reddit. URL: https://www.reddit.com/r/MachineLearning/comments/19f5z25/p_training_ml_models_on_encrypted_data_with_fully/.

[86] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2014). Going Deeper with Convolutions. URL: https://arxiv.org/abs/1409.4842. arXiv:1409.4842.

[87] Tian, Y., Yao, H., Cai, M., Liu, Y., & Ma, Z. (2021). Improving RNN Transducer Modeling for Small-Footprint Keyword Spotting. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 5624–5628). doi:10.1109/ICASSP39728.2021.9414339.

[88] Titus, A. J., Kishore, S., Stavish, T., Rogers, S. M., & Ni, K. (2018). PySEAL: A Python wrapper implementation of the SEAL homomorphic encryption library. URL: https://arxiv.org/abs/1803.01891. arXiv:1803.01891.

[89] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., & Lample, G. (2023). LLaMA: Open and Efficient Foundation Language Models. URL: https://arxiv.org/abs/2302.13971. arXiv:2302.13971.

[90] Vasanthi, M., & Seetharaman, K. (2022). Facial image recognition for biometric authentication systems using a combination of geometrical feature points and low-level visual features. *Journal of King Saud University - Computer and Information Sciences*, *34*, 4109–4121. URL: https://www.sciencedirect.com/science/article/pii/S1319157820305577. doi:https://doi.org/10.1016/j.jksuci.2020.11.028.

[91] Venkataramanan, K., & Rajamohan, H. R. (2019). Emotion Recognition from Speech. *ArXiv*, *abs/1912.10458*. URL: https://api.semanticscholar.org/CorpusID:209444761.

[92] Vygon, R., & Mikhaylovskiy, N. (2021). Learning Efficient Representations for Keyword Spotting with Triplet Loss. In A. Karpov, & R. Potapova (Eds.), *Speech and Computer* (pp. 773–785). Cham: Springer International Publishing. URL: https://arxiv.org/abs/2101.04792.

[93] Warden, P. (2018). Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. URL: http://arxiv.org/abs/1804.03209 cite arxiv:1804.03209.

[94] zama (). zama. URL: https://docs.zama.ai/concrete-ml.

[95] Zama (2022). Concrete ML: a Privacy-Preserving Machine Learning Library using Fully Homomorphic Encryption for Data Scientists. https://github.com/zama-ai/concrete-ml.

[96] Zama (2022). Concrete: TFHE Compiler that converts python programs into FHE equivalent. https://github.com/zama-ai/concrete.

[97] Zama (2022). TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data. https://github.com/zama-ai/tfhe-rs.

[98] Zhang, J. X., Yordanov, B., Gaunt, A., Wang, M. X., Dai, P., Chen, Y.-J., Zhang, K., Fang, J. Z., Dalchau, N., Li, J., Phillips, A., & Zhang, D. Y. (2021). A deep learning model for predicting next-generation sequencing depth from DNA sequence. *Nature Communications*, *12*, 4387. URL: https://doi.org/10.1038/s41467-021-24497-8. doi:10.1038/s41467-021-24497-8.