Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria
Corso di Laurea Magistrale in Computer Science and Engineering

POLITECNICO
MILANO 1863

HErBERT: a privacy-preserving natural language processing solution for text classification

Advisor: Prof. Manuel Roveri
Co-Advisor: Prof. Mark James Carman
Eng. Alessandro Falcetta

Thesis by:
Daniele Comi Matr. 944534

Academic Year 2020–2021

I want to thank the professors Manuel Roveri, Mark James Carman and the Engineer Alessandro Falcetta who helped me throughout the conception of the Thesis and the ideas behind it, and for allowing me to make the first submission of a paper for a possible publication.

I want to thank my dear Valeria for her continuous long time support making me do my best, and through the various years for always having my back, always believing in me, always having been here with me and for me. I thank her for the help in reviewing the images and making them better good-looking, improving the quality of the thesis and making it as good as it is.

I want to thank my dear Martina for her continuous support on the work I have been doing, for believing in me achieving my goals, for cheering me in doing my best with her kindness and for the help in improving the quality of the thesis by reviewing the English writing.

I want to thank my parents for their support, making all this possible.

I want to thank all my friends supporting me through the years.

I want to thank anyone else involved in what I was able to achieve.

Acknowledgments

The authors would like to thank Subcom, the company which funded and provided access to AWS in order to test the results of the privacy-preserving Deep Learning model HErBERT.

Abstract

Privacy-preserving machine and deep learning solutions will enable new and exciting breakthroughs in many different application fields in the next few years. In fact, their ability to process encrypted input data through machine and deep learning models will allow to guarantee the privacy of users during the processing, hence allowing to match the stricter and stricter legislation and recommendations in terms of data protection and user privacy. For this reason, the research interest in this field is steadily growing, with relevant results only in specific application fields. In this work, for the first time in the literature, we introduce a privacy-preserving natural language processing solution, called HErBERT, able to perform text classification on encrypted data. The proposed solution, which relies on Homomorphic Encryption to process encrypted data, is inspired by the well-known BERT architecture and introduces privacy-preserving Transformers. A computationally-efficient inference of HErBERT has been designed and developed and made available to the scientific community. Experimental results on two real-world benchmarks for text classification show the effectiveness of the proposed solution.

Sommario

I modelli di Machine Learning e Deep Learning che preservano la privacy consentiranno nuove ed entusiasmanti scoperte in svariati campi di applicazione nei prossimi anni. Infatti, la loro capacità di elaborare i dati di input crittografati attraverso modelli di machine e deep learning consentirà di garantire la privacy degli utenti durante il trattamento, consentendo così di adeguarsi alla legislazione e alle raccomandazioni più severe in termini di protezione dei dati e privacy degli utenti. Per questo motivo l'interesse della ricerca in questo campo è in costante crescita, con risultati rilevanti solo in specifici campi di applicazione. In questo lavoro, per la prima volta in letteratura, introduciamo una soluzione di elaborazione del linguaggio naturale che preserva la privacy, chiamata HErBERT, in grado di eseguire la classificazione del testo su dati crittografati. La soluzione proposta, che si basa sulla crittografia omomorfica per elaborare i dati crittografati, si ispira alla nota architettura BERT e introduce i Trasformers che preservano la privacy. Un'inferenza computazionalmente efficiente di HErBERT è stata progettata e sviluppata e resa disponibile alla comunità scientifica. I risultati sperimentali, su due dataset di riferimento su problemi reali per la classificazione del testo, mostrano l'efficacia della soluzione proposta.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Motivation

Privacy-preserving machine and deep learning is a new and promising research area aiming at designing machine and deep learning models able to operate on encrypted data, hence guaranteeing the privacy of user data. This is a crucial ability in a technological and regulation scenario aiming at enforcing the privacy of users when sensitive data (e.g., health data, genetic and biometric data, or data revealing political opinions or other personal information) are processed through Cloud-based on-line services or mobile applications [2]. The prerequisite for designing privacy-preserving machine and deep learning solutions is to integrate machine and deep learning models with privacy-preserving computation. Among the mechanisms supporting computation in a privacy-preserving manner present in the literature, we focused on Homomorphic Encryption (HE) that is a group of encryption schemes able to support the execution of a (given) set of operations directly on encrypted data. This allows third-party software or systems (e.g., Cloud-service providers or mobile app developers) to offer, in a Software-as-a-Service (SaaS) or Platform-as-a-service (PaaS) manner, advanced machine and deep learning services operating on encrypted data guaranteeing that the outcome of the computation is still encrypted and that only the user encrypting the data will be able to decrypt it. Cloud Providers provide ready-to-use services to execute Machine Learning and Deep Learning models on very powerful hard-

ware thanks to advanced Virtualization techniques which enables it to share powerful hardware resources among multiple users on-demand with very little loss, if none, in performances [3]. Paravirtualization [4] [3] techniques are commonly used so that the Virtual Machine Manager can present to the running Virtual Machines a similar, but not identical, interface to that of the underlying hardware. There are a lot of examples of the provided services for ML and DL, some are the followings [5]: classification of images and videos, object detection, instance segmentation, object-tracking, text classification, sentiment analysis, language models, sequence-to-sequence model for machine translation, text-to-speech, speech recognition, summarization, text generation and many other models solving some very hard non-linear and also dynamic problems. Cloud Computing also provides some important properties such as on-demand resources, broad network access, elasticity and scalability, availability, maintainability and pay-per-use billing mechanism [6].

Looking at all of this from a privacy perspective, it is not possible to say it is completely good. In fact, the user necessary has to upload data to the model on the Cloud Computing service in clear, which means that, even though the transmission on the network the data is encrypted, on the other end it will be decrypted in order to make any kind of inference on the model being used. These data can be highly sensitive and worth of being Privacy-Preserved, such as government data, classified data, health data, genetic and biometric data or other kind of data which might reveal political opinions or other personal information not to be disclosed without the owner's will [2]. The aim of this work is to introduce a novel architecture, HErBERT, indeed meant to preserve the privacy of the user data in the deep-learning-as-a-service computing scenario. It is inspired by the BERT [7] architecture and is able to operate on encrypted data thanks to the use Homomorphic Encryption. Homomorphic Encryption (HE) [8] is a family of encryption schemes able to support the execution of a (given) set of operations directly on encrypted data. To achieve this goal HErBERT introduces privacy-preserving Transformers, being the privacy-preserving version of the well-known and widely-used Transformers architectures.

Privacy-preserving Transformers have been carefully designed and retrained to take into account the severe constraints on the type and amount of operations that can be executed within a HE-based processing pipeline. In addition, we introduced a computationally-efficient inference of HErBERT by exploiting block-matrix multiplications to reduce the computational demand.

The ability of performing computations directly on encrypted data comes at the expense of three main drawbacks. On the first hand, the set of operations that can be executed within the privacy-preserving machine and deep learning pipeline is very limited. Indeed, in HE schemes are often allowed just the two basic operations of addition and multiplication operations. On the second hand, in order to guarantee the correctness of the decryption operation on the results, the total amount of operations that can be executed within the machine and deep learning pipeline is limited (most of the times very limited). On the third hand, the computational complexity of the HE operations is much larger than the corresponding complexity of operations on plaintexts, hence introducing a significant overhead in computational time (at least 40-50x) and memory demand (at least 10-20x) [9] when HE-based machine and deep learning models are considered. The higher computational load of scheme is also worsened by the fact that current available HE scheme are only supporting CPU instructions. No CUDA [10] implementation or implementation on GP-GPU has been ever provided. Moreover, HE schemes must be configured through some parameters that are able to trade off the accuracy in the computation with the computational loads, memory occupation and how large the deep-learning model is. These three drawbacks severely impact the design of privacy-preserving machine and deep learning solutions, since machine and deep learning models and algorithms are typically characterized by a long pipeline of non-linear operations. This is the reason why the literature about privacy-preserving machine and deep learning solutions based on HE is limited, but the research interest in this field is steadily growing. Interestingly, most of the available solutions focus on image classification/recognition, hence introducing Convolutional Neural Networks (CNNs) integrating HE mechanisms.

The field of privacy-preserving Natural Language Processing (NLP) has been rarely explored due to the very deep and nonlinear architectures of NLP solutions and models. To partially mitigate these issues, the NLP privacy-preserving solutions present in the literature do not support a fully privacy-preserving NLP processing solution, since the user is required to take part in the processing. Further details about the privacy-preserving machine and deep learning solutions based on HE can be found in Chapter 2. The proposed HErBERT architecture will show the first HE-friendly Transformer in order to preserve the privacy of textual data sent for performing Sentiment Analysis and Text Classification. Such NLP tasks are also a very good example of the reason why and when privacy preserving models are and will be a must. The proposed model will be run on AWS Cloud Computing services. we will show how it is possible with this architecture to let the user encrypt the data locally through a public key generated by the HE scheme and then send this encrypted data to an encoded Cloud-based deep-learning model as-a-service, and receive back the encrypted results of the model inference which will be locally, from the sending user/data owner, decrypted. In this way, the encryption and decryption phase is decoupled between the end-user device and the Cloud-based computing infrastructure in order to be able to fully preserve the privacy of data while still guaranteeing the previous explained properties of Cloud Computing.

1.2 Applications of Homomorphic Encryption

There is an ongoing, increasing need to create models and make predictions from confidential distributed datasets in so many different industries. [11] shows various potential real-world applications of Homomorphic Encryption. Some of the most ongoing applications are:

1.2.1 Classified information and National security

Various governments currently have resources dedicated to the only purpose of keeping relegated really important and sensible information. Sometimes this information may leak and release sensible need-to-know information, creating dangerous consequences [12]. This can happen due to the fact that while this information are sent through encrypted packets in the network, they will have to be decrypted in order to be elaborated by another computer. For example, to check the presence of a particular object in a classified image. Here, Homomorphic Encryption can quickly solve the issue of having to decrypt very sensible information, avoiding any risks of leaking too much relevant information.

1.2.2 Healthcare

Maintaining the privacy of the patients is crucial [13], it is a right guaranteed by the law; but there still is the need to share patients data and to make computations distributed across systems. We can think for example about the most common use-case of sharing patient data in order to compute the cost of a patient's treatment for the government or for the insurance companies. Making such computations may reveal the patient's treatment, diseases and medical history without the patient say-so. This is not acceptable and can be solved by using a system backed by Homomorphic Encryption.

1.2.3 Financial Services

In financial services there a lot of potential applications regarding Homomorphic Encryption such as the computation of the taxes or the evaluations of the owned shares in certain companies at the stock market exchange. All the data involved is potentially in danger of multiple privacy violations [14] if not well treated when they are manipulated and used to perform the needed computations. Using Homomorphic Encryption to perform the operations involved and automatically performed in finance can solve this problem, and it can also help the industry in making the people more involved in this area where they

may have distrust for privacy related reasons indeed.

1.2.4 Genomics

Private health data obtained from sequencing human genome can be a powerful tool to develop a cure, a therapy or to improve the researches. DNA and RNA sequences can now be obtained very easily, and often people share their own DNA data in order to better understand their health. Certainly, this has a lot of implications regarding one's privacy [15] because a DNA sequence is linked to a person, and it is unique. But other than that, it can also reveal possible diseases of the person who does not want to disclose. Current laws regarding genomics data privacy has created a lot of limitations for the researchers. Homomorphic Encryption can enable researchers to speed up sharing information while safeguarding privacy of the individuals, and thus significantly speed up research in this field.

1.3 Goal and Results

This work focuses on approaching the problem of privacy-preserving computation regarding Natural Language Processing. While the architecture being used, the Transformer, has been tailored on Sentiment Analysis and Sequence Classification, it can be easily used for other various and different NLP tasks at hand. In the proposed Transformer architecture, through the properties of HE schemes, the input data composed of the corresponding Embeddings is encrypted on the user machine (e.g., a personal computer or a mobile device) through a public key. The encrypted data is then sent to a Cloud-based special MLaaS where the prediction will be computed. After the computation is completed, the still encrypted result is sent back to the user machine, where it will be decrypted, and the result will be obtained.

This work includes an implementation of the proposed Transformer architecture. It includes the open source Python implementation of the model which also contains the necessary implementation for bridging Pytorch models to this custom approximated implementation, which will be better explained in the followings. Using this implementation,

we offer a working solution for solving privacy problems regarding the classification of raw textual data.

A detailed experimental campaign showed the effectiveness of the proposed solution. In particular, two real-world datasets for text classification have been considered: the first one is the Yelp Polarity Review [16] which contains reviews classified as positive and negative, and the second one is Yahoo! Answers [16] dataset have been considered. The Python implementation of HErBERT is released to the scientific community[1], while the efficiency of HErBERT has been tested on AWS Cloud Computing services.

1.4 Thesis Structure

Chapter 3 introduces the fundamental notions about HE, its limitations and possibilities, the advantages of its adoption and the necessary steps to use it. In particular, the peculiarities of the chosen HE scheme are presented, with guidelines to select the optimal parameters. Moreover, a basic background on ML and Transformer is exposed, along with a background on Natural Language Processing. These are concepts needed to fully understand the characteristics of the research. Chapter 2 describes the current state-of-the-art of privacy-preserving solutions for MLaaS. Chapter 4 is the core of the work, where the proposed architecture is detailed. Chapter 5 explores HErBERT, the architecture implemented in Python. Chapter 6 will show what solutions we implemented in order to improve the performances while using a Homomorphic Encryption scheme. Chapter 7 includes the experiments made on some typical use-cases in the NLP field. Conclusions are finally drawn in Chapter 8.

[1] `https://github.com/comidan/HErBERT`

Chapter 2

Related Literature

This Chapter collects studies and works with the goal of offering privacy-preserving solutions for Machine Learning and other examples of architectures similar to the one presented in this thesis. Trying to use Homomorphic Encryption scheme in order to maintain the privacy of the data involved during a certain computation has been first introduced years ago in this work [17]. This work was the first in proposing the use of HE in order to maintain the privacy of data during the computation. The idea behind their work was generated by the need of a small company using storage resources of a time-sharing service. Their problem was that if the company decides to encrypt their data before submitting writing it to storage, to maintain private the contained information, it becomes impossible to use other computational resources offered by the time-sharing service to answer query about the stored information. It can be considered as example the case in which the company needs to know the amount of income from loan payments is expected in the following months, while the data for answering this query is completely encrypted. In this first HE work, the authors indeed proposed a privacy homomorphism to encrypt the data so that computations could still occur on the stored and encrypted data. As it is clear, solving the problem of maintaining the privacy of data is an old problem still waiting for a final solution.

2.1 First HE schemes and their evolution

The first HE schemes put in practice were only Partially Homomorphic Encryption schemes given that they allow only or additions or multiplications like for example RSA [18]. Partially Homomorphic Encryption schemes are just one of three existing types available: the other two types are Somewhat Homomorphic Encryption Schemes and Fully Homomorphic Encryption schemes which they both allow to perform more than one operation but while the former a limited number of times, the latter it can perform them an unlimited number of times. The first Homomorphic Encryption scheme allowing both multiplication and additions has been proposed in this work [19]. There, the idea was to rely on ideal lattice-based cryptography to provide a scheme supporting additions and multiplications with theoretically-grounded security guarantees. But that is not the best result from that paper. Gentry has been able for the first time ever to make a Somewhat Homomorphic Encryption scheme into a Fully Homomorphic Encryption scheme. In order to be able to make this transformation Gentry introduced the concept of Recryption [19] which is the ability to change the encryption key, so to re-encrypt it, by using the first homomorphic private key just homomorphically encrypted under a new generated public key from the client itself. Then the decryption circuit of the scheme will be Homomorphically-Evaluated using this encrypted data in order to decrypt the original data and then re-encrypt it with the new public key. In this way, the noise will go back to be manageable again for another run of Homomorphic Encrypted operations: so, the Somewhat Homomorphic scheme can be virtually transoformed to a Fully Homomorphic scheme where there is no bound on homomorphic operations that can be made. But all this requires a scheme to have the property of Bootstrappability, which means being able to homomorphically evaluate its own decryption circuit schemes. Being able to do that, in practice, it is very expensive because actually slow HE operations are being performed in order to decrypt a HE encrypted data. It is so expensive that not all the Homomoprhic Encryption schemes implements it nor all the related libraries [11]. After what Gentry has done with [19], exploiting what he achieved, there has been a work on a Fully Homomorphic

Encryption scheme using bootstrapping which was relaxing the ideal lattice assumption (and its security), but allowing the usage of integer polynomial rings to define the ciphertexts [20]. The next work introduces the Brakerski-Gentry-Vaikuntanathan (BGV) scheme [21] that relies on polynomial rings to define the ciphertexts and on the learning with error (LWE) and ring learning with errors (RLWE) problems to provide theoretically-grounded security guarantees. The RLWE problem is also the basis of the Brakerski/FanVercauteren (BFV) scheme [22], detailed in Chapter 3, and the Cheon-Kim-Kim-Song (CKKS) scheme [23], that extends the polynomial rings to the complex numbers and isometric rings and leverages no more on the noise budget itself but on the error introduced by performing the various operations given the set of parameters different from BFV. The HE schemes mentioned above are theoretical and, to be applied, have then been implemented to specific processing chains.

2.2 HE schemes and libraries implementations

Various published scheme available in the literature are also available for researchers to use.[24]. One of the most commonly used in this research area is HElib [25] [26] [27]. HElib implements the BGV scheme [21] with Smart-Vercauteren ciphertext packing techniques and various other kinds of optimizations. HElib has been implemented using low-level programming, so in direct contact with the hardware constraints and the various components of the machine being used without using any kind of primitive defined in a particular programming language. It is indeed defined as the *assembly language for HE*. It was implemented using C++ library. Since 2014, it supports the introduced bootstrapping technique and since 2015, it also supports multi-threading which has been directly implemented in the library managing HElib. In a particularly important update, HElib added the homomorphic evaluation of AES [28] [29] [30], but unfortunately using HElib is quite difficult due to the sophistication needed for its low-level implementation and HE parameter selection, affecting the performances. An

interesting implementation is the "Fastest Homomorphic Encryption in the West" (FHEW) [31]. It indeed improves the computational time required to perform the bootstrapping technique to the ciphertext. It supports the evaluation of binary gates, and given that it supports NAND gates (and given that NAND gates are functionally complete) any possible function expressed with boolean gates can be computed. In this work, the usage of ciphertext packing and SIMD techniques provides an amortized cost. In this work there has been also a further improvement regarding the noise propagation by using some approximations. Finally, they also reduced the size of bootstrapping key from 1 GB to 24 MB by achieving the same security level as the one used with the 1 GB key. Another HE library called Simple Encrypted Arithmetic Library (SEAL) [32] has been released by Microsoft Research. The main reason behind the release of this library is offering a well-documented HE library which can be easily used by both crypto experts researchers and non-experts with no crypto background like Machine Learning researchers and engineers. The library does not have external dependencies like others, and it includes an automatic parameter selection for some of its parameters and more importantly noise estimator tools in order to manage the noise of HE schemes. Here there are also listed further HE implementation and relative libraries:

- HEAAN [23]: Scheme with native support for fixed point approximate arithmetic implemented in C++.
- TFHE [33]: Scheme implementing the evaluation of binary gates and homomorphic bit operations evaluation. It is implemented in C++.
- PALISADE [34]: Lattice encryption library implemented in C++ and includeds a Python wrapper.
- Pyfhel [35]: Python For HElib, which also supports BFV scheme through the SEAL API.
- cuHE and cuFHE [36]: GPU-accelerated HE library for NVIDIA CUDA-Enabled GPUs using the TFHE scheme and so evaluating binary gates, implemented in C++ and supporting Python through a wrapper.

- TenSEAL [37]: Library for HE operations on tensors, built on Microsoft SEAL, with a Python API. It supports both CKKS and BFV scheme, with regard to BFV we have also contributed in this project by adding support for multithreaded tensors operations directly in the C++ implementation and by also providing the required Python wrapper for API call [38].

2.3 Use of HE in Machine Learning

Regarding the application of HE schemes to Machine Learning and Deep-Learning there has been a lot of different works regarding its application to CNNs like [39] where a distributed deep-learning-as-a-service approach has been used but also these other two works [40] [41]. Some other works related to NLP via privacy-preserving Machine Learning have been done with [42] [43] [44] using RNNs, GRUs and LSTMs [45] where the various activation functions have been also here approximated with a working alternative which has been used during the training process on the approximated model version. In [42] they have analysed the classification of encrypted word embeddings using RNNs where they have analysed very well the noise growth in a RNN and they handled the approximation of activation functions with Chebyshev polynomials and the method described in [46]. Instead, in [43] they have used a different approach to handle the noise growth. They did not handle it reducing eventually the size of the network or by using the theoretically constructs of Bootstrapping, but they have built a communication system in order to send back the tensor data to the client in order to be decrypted and to perform on them in clear the non-polynomial operations required in order to not use any approximations reducing the final results and also to completely refresh the noise budget for further HE operations. Regarding this approach, there is also the very recent nGraph-HE2 framework [47] which is derived from the Intel nGraph deep learning (DL) compiler [48]. It is able to give the possibility to train in particular CNNs in plaintext on a particular given hardware and then deploy these trained models to HE cryptosystems. Their objective with this work was to hide most of the

complexities behind the usage of HE. While in their work the CKKS scheme has been used, their true goal is to let researchers deploy pre-trained models using their native activation functions without, or with a minimal, code refactoring. To address non-polynomials activation functions like ReLU, GeLU, etc., the framework uses indeed a similar client-server continuous communication computation approach of [43]: during inference in the model the intermediate results are sent to the client which decrypts the running tensors in the model, computes the corresponding needed activation function in clear, then re-encrypts the resulting tensor, and it sends the new encrypted tensor back again to the server where it will be further computed in the remaining step of the model inference. While those systems are able to provide encrypted results on encrypted data, they require an active participation of the users introducing various difficulties regarding data transmission, network issues, etc. For this reason, they are not considered in the comparison with HErBERT, which instead provides a fully end-to-end inference, but they are cited to show the growing interest of achieving privacy-preserving predictions also in the NLP field using various types of architectures, and now also the Transformer. Also, the same for this work [44] which is interesting because they have used an approach which involved evaluating homomorphically binary gates in order to perform some operations like activation functions, this has been achieved by them using the Gazelle library [41] to adopt a *simpler* HE scheme, namely packed additive homomorphic encryption (PAHE) scheme and garbled circuits (GC). Instead, as of our current knowledge no work has been done on applying Homomorphic Encryption to the novel state-of-the-art architecture of Transformer [49], or better no work related trying to adapt, approximate and scale down Transformer making it practical with Homomorphic Encryption. Interesting work [50] using instead Federated Learning has been used for large Transformers models like BERT has been done, where homomorphic encryption is cited as a possibility, but the paper is focusing on Federated Learning differently from us.

Chapter 3

Background

This Chapter collects the main notions which are required to completely understand the various aspects of this research work. Section 3.1 presents the concepts of Homomorphic Encryption schemes in general, then the one used in this work along with considerations on the security assumptions, the permitted operations, the computational overhead, the advantages, and the drawbacks. In Section 3.2 a brief summary on Machine Learning will be presented, following a summary on Deep Learning 3.2.1 and on Natural Language Processing 3.2.2, along with more specific notions regarding Transformer in Subsection 3.2.3.

3.1 Homomorphic Encryption

Homomorphic encryption (HE) is a type of encryption scheme which enables some simple operations to be performed on encrypted data, directly without knowing any key to decrypt these data, with a very good approximation or exact computation. A general encryption scheme can be described by means of an encryption function E and a decryption function D. Where given a certain input x, $x = D(E(x))$ given a certain set of keys. Common encryption schemes, both symmetrical and asymmetrical, are not homomorphic though, they don't generally allow operations to be performed on encrypted data because the underlying algebraic structure of the plaintext is not maintained after encryption. Using instead an Homomorphic Encryption scheme operations on en-

crypted are allowed to take place, as shown in the example in Figure 3.1

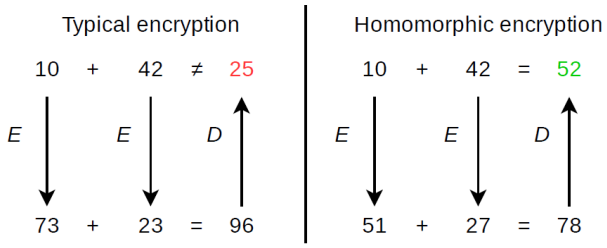


Figure 3.1: Example comparison between common Encryption Schemes and Homomorphic Encryption schemes

3.1.1 Classification of HE schemes

There exists various types of available HE schemes, the ones available are three different main categories.

Somewhat homomorphic schemes

Somewhat homomorphic schemes are the ones which allow multiplications and additions to be performed, but in a limited number of times due to the presence of a noise. This noise is always present (in order to have a certain amount of confusion and diffusion of the encrypted data) and it is added during the encryption phase. For this fact, at each operation performed, some quantity of noise is added to the new resulting ciphertext. This is an issue which needs to be addressed because the presence of the noise is tolerated until a certain threshold by the encryption scheme, meaning that having an amount of noise greater than this threshold will result in not being able to decrypt the data anymore. This threshold is called noise budget.

Partially homomorphic schemes

Partially homomorphic schemes have the characteristic of not having any bounds on the number of operations being able to be performed, but they are allowed to perform only one type of operation. Taking as an example, RSA [18] which it has the multiplicative homomorphism

allowing so to perform an unbounded number of multiplications-only on ciphertext data.

Fully homomorphic schemes

Fully homomorphic schemes have the capability of performing an unbounded number of times both addition and multiplication without any limitations from the noise or anything else.

3.1.2 The Noise

Homomorphic Encryption schemes classified as somewhat homomorphic have the dark side of always adding some noise during the encryption phase: this enables to guarantee the fundamental properties of confusion and diffusion on encrypted data. Upon the commit of an operation, its result will have more noise proportionally on the operation performed, while multiplication being the most expensive one.

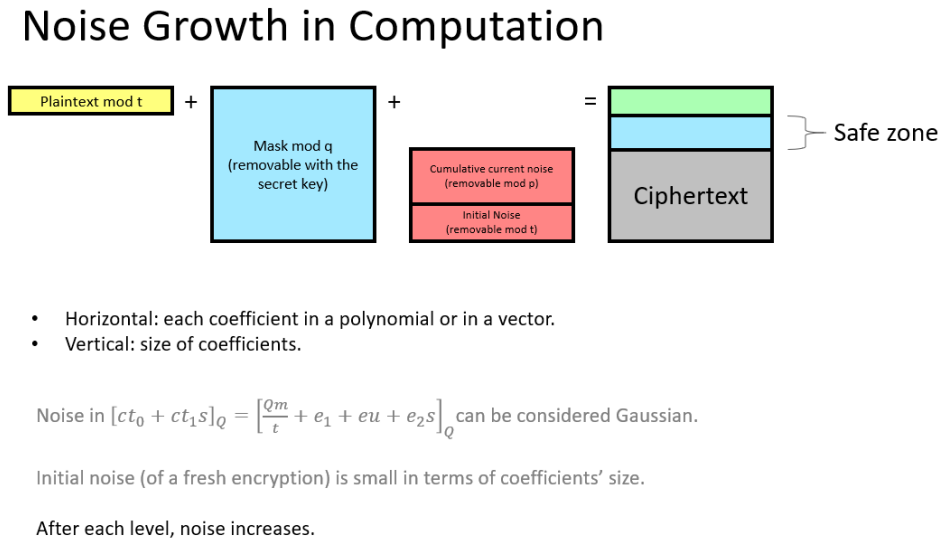


Figure 3.2: Noise relation with ciphertext during encrypted computation [1]

We can then call Noise Budget (NB) the amount of noise we are allowed to add without reaching the point in which we are no more

able to decrypt correctly as in Figure 3.2 from [1], so we will have, depending on the scheme, a limited number of consecutive operations to be performed. While this noise is strictly dependent on the encryption scheme chosen, we are allowed to assign a set of parameters Θ, which we will see later on, in order to optimize both the noise consumption and noise budget.

3.1.3 Brakersi/Fan-Vercauteren scheme (BFV)

The HE scheme considered in this work is the Brakerski/Fan-Vercauteren (BFV) [22] [51] scheme, which is based on the Ring-Learning With Errors (RLWE) problem. The BFV encryption scheme has been introduced by Junfeng Fan and Frederik Vercauteren in 2012. Their idea was to modify the scheme proposed by Brakerski, porting it from a setting in which the Learning With Error (LWE) problem was used to RLWE [52]. The main object used in the BFV scheme is the polynomial ring. The ring used in the BFV scheme is $R = \mathbb{Z}[x]/(f(x))$ where $f(x) \in \mathbb{Z}[x]$ is a monic irreducible polynomial of degree m. In particular, a cyclotomic polynomial m(x) of degree m is used; m is a positive power of 2.

Polynomial rings

The main object used in the BFV scheme is the polynomial ring. A ring is a set R with the operations of addition as $+ : R \times R \to R, (a, b) \longmapsto a+b$ and multiplication as $* : R \times R \to R, (a, b) \longmapsto a*b$ [53] satisfying the following conditions:

1. $(R, +)$ is a commutative group so where $a+b = b+a$, there exist the element $0 \in R$ which is the neutral element in the group and there exist the inverse element of $a \in R$ as $-a$.

2. The multiplication operation has associative property so, $(a*b)*c = a*(b*c)$ for all $a, b, c \in R$.

3. There is the distributive property: for all $a, b, c \in R$ $a*(b+c) = a*b+a*c$ and $(a+b)*c = a*c+b*c$.

The unit element $1 \in R$ is called unit and $1 * a = a * 1 = a$ for all $a \in R$. Given now the understanding of the algebraic structure called Ring we can understand and define the polynomial ring $R[x]$, which is the ring of all polynomials with coefficients in R:

$$R[x] = \{a_0 + a_1 x + \cdots + a_n x^n : a_i \in R \; \forall i\}$$

The addition and multiplications operations are defined as:

$$\sum_{i=0}^{n} a_i x^i + \sum_{i=0}^{n} b_i x^i = \sum_{i=0}^{n} (a_i + b_i) x^i$$

$$\sum_{i=0}^{n} a_i x^i * \sum_{j=0}^{m} b_j x^j = \sum_{i=0}^{n} \sum_{j=0}^{m} a_i b_j x^{i+j}$$

The ring used in the BFV scheme is:

$$R = \mathbb{Z}[x]/(f(x))$$

$f(x) \in \mathbb{Z}[x]$ is a monic irreducible polynomial of degree m. In particular, a cyclotomic polynomial $\Phi_m(x)$ of degree m is used; m is a positive power of 2.

Ring Learning With Errors problem

Ring Learning With Errors (RLWE) is a computational problem used in many fields of cryptography, including the development of encryption schemes resistant to attacks conducted with quantum computers. RLWE it is used in order to give assurances on the security and the strength of the scheme.

Two different versions exists of this problem: "search" and "decision". BFV uses the "decision" version.

Definition 3.1.1. RLWE-Decision Given:

- $\mathbf{a_i}$ a set of random but known polynomials from R_q, which is the ring R with coefficients in $\mathbb{Z}_q$;
- $\mathbf{e_i}$ is a set of small random and unknown polynomials relative to a bound b in the ring $\mathbb{Z}_q$;

- $\mathbf{s}$ be a small unknown polynomial relative to a bound b in the ring $\mathbb{Z}_q$.
- $\mathbf{b}_i = (\mathbf{a}_i \cdot \mathbf{s}) + \mathbf{e}_i$

And a list of polynomial pairs $(\mathbf{a}_i, \mathbf{b}_i)$, the RLWE-Decision problem consists in determining if the $\mathbf{b}_i$ polynomials were constructed as $\mathbf{b}_i = (\mathbf{a}_i \cdot \mathbf{s}) + \mathbf{e}_i$ or were generated randomly from R_q.

The security of a scheme built on the RLWE problem is explored in the work by Lyubashevsky et al. [52]. The main idea is that solving the RLWE problem is equivalent to solve the Approximate Shortest Vector Problem (α-SVP). This problem is known to be NP-hard [54].

Encryption Scheme

These are the basic operations employed in the BFV scheme:

- SecretKeyGen (1^λ): sample $\mathbf{s} \leftarrow \chi$ and output sk= $\mathbf{s}$.
- PublicKeyGen (sk): set $\mathbf{s}$=sk, sample $\mathbf{a} \leftarrow R_q$, $\mathbf{e} \leftarrow \chi$ and output

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

- EvaluateKeyGen (sk, p): sample $\mathbf{a} \leftarrow R_{p \cdot q}, \mathbf{e} \leftarrow \chi^{'}$ and return

$$\text{rlk} = ([-(\mathbf{a} \cdot \mathbf{s} + \mathbf{e}) + p \cdot \mathbf{s}^2]_{p \cdot q}, \mathbf{a})$$

  Evaluation keys are necessary to perform an operation called *relinearization*, described more in detail in Sub-Section 3.1.3.

- Encrypt (pk, $\mathbf{m}$): to encrypt a message $\mathbf{m} \in R_t$, let:

$$\mathbf{p}_0 = \text{pk}[0]$$

$$\mathbf{p}_1 = \text{pk}[1]$$

  Then, sample $\mathbf{u}$, $\mathbf{e}_1$, $\mathbf{e_2} \leftarrow \chi$ and return:

$$\text{ct} = ([\mathbf{p}_0 \cdot \mathbf{u} + \mathbf{e}_1 + \Delta \cdot \mathbf{m}]_q, [\mathbf{p}_1 \cdot \mathbf{u} + \mathbf{e}_2]_q)$$

- Decrypt (sk, ct): set $\mathbf{s} = \text{sk}, \mathbf{c}_0 = \text{ct}[0], \mathbf{c}_1 = \text{ct}[1]$. The decrypted value is:
$$\left[\left\lfloor \frac{t \cdot [\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s}]_q}{q} \right\rceil\right]_t$$
- Add $(\text{ct}_0, \text{ct}_1)$: Output $(\text{ct}_0[0] + \text{ct}_1[0], \text{ct}_0[1] + \text{ct}_1[1])$
- Multiply $(\text{ct}_0, \text{ct}_1, \text{rlk})$: compute
$$\mathbf{c}_0 = \left[\left\lfloor \frac{t \cdot (\text{ct}_1[0] \cdot \text{ct}_2[0])}{q} \right\rceil\right]_q$$
$$\mathbf{c}_1 = \left[\left\lfloor \frac{t \cdot (\text{ct}_1[0] \cdot \text{ct}_2[1] + \text{ct}_1[1] \cdot \text{ct}_2[0])}{q} \right\rceil\right]_q$$
$$\mathbf{c}_2 = \left[\left\lfloor \frac{t \cdot (\text{ct}_1[1] \cdot \text{ct}_2[1])}{q} \right\rceil\right]_q$$

If the hardness of the RLWE problem is assumed, this scheme can be shown to be semantically secure [52]. The relinearization operation described above is related to noise and noise growth in the ciphertexts. These concepts are fundamental from a practical point of view for the use of the BFV scheme as explained before.

Noise growth

Definition 3.1.2. Invariant noise Let $\text{ct} = (c_0, c_1, \ldots, c_k)$ be a ciphertext encrypting the message $m \in R_t$. Its invariant noise v is the polynomial with the smallest infinity norm such that:
$$\frac{t}{q}\text{ct}(s) = \frac{t}{q}\left(c_0 + c_1 s + \cdots + c_k s^k\right) = m + v + at$$
for some polynomial a with integer coefficients [32].

During the encryption phase, noise is added to the ciphertexts to guarantee that, being $p_1 = p_2$ two plain values to be encrypted with the same public key, the corresponding ciphertexts c_1 and c_2 are different (i.e., $c_1 \neq c_2$). However, performing operations on ciphertexts increases their noise. When the noise reaches a certain threshold, it becomes impossible to decrypt the data without corrupting it.

The Noise budget is a practical way to measure the noise of a ciphertext.

Definition 3.1.3. Noise budget Let v be the invariant noise of a ciphertext ct encrypting the message $m \in R_t$. Then the noise budget (NB) of ct is $-\log_2(2||v||)$.

The NB is an integer positive number. If it is greater than 0, it is still possible to decrypt a ciphertext and obtain the correct plaintext value. Hence, for NB, greater is better. As stated in [32]:

Lemma 3.1.1. The function Decrypt correctly decrypts a ciphertext ct encrypting a message m, as long as the NB of ct is positive.

It is useful to understand how noise grows when an operation is performed on a ciphertext. In general, the important aspect is that additions and subtractions have a small impact on the noise growth, while multiplications are the most NB-consuming operations. The precise growth bounds of the noise, for each operation, is here reported for completeness in Table 3.1 taken from [32].

What we can infer from this table is that:

- The NB is an integer positive number. If it is greater than 0, it is still possible to decrypt a ciphertext and obtain the correct plaintext value. Hence, for NB, greater is better.

- Additions between two ciphertexts cause the noise to be the sum of the noise value of the two ciphertexts.

- Multiplications not only make the noise grow with a greater factor, but also depends on the size of the two ciphertexts.

- Operations between one ciphertext and one plaintext consume less noise budget

- It is fundamental to reduce the size of the ciphertexts in order to limit the noise growth during operations, especially multiplications. The relinearization operation, which consumes noise, will perform such size reduction. Referencing the Multiply operation 3.1.3, relinearization [22] consists in writing $\mathbf{c}_2$ in base T as

Table 3.1 SEAL Noise bound of output

Operation	Input description	Noise bound of output
Encrypt	Plaintext m	$\frac{r_t(q)}{q}\|m\|N_m + \frac{7nt}{q}min\{B, 6\sigma\}$
Negate	Ciphertext ct	υ
Add/Sub	Ciphertext ct_1 and ct_2	$\upsilon_1 + \upsilon_2$
AddPlain/SubPlain	Ciphertext ct and plaintext m	$\upsilon + \frac{r_t(q)}{q}N_m\|m\|$
MultiplyPlain	Ciphertext ct and plaintext m	$N_m\|m\|\upsilon$
Multiply	Ciphertexts ct_1 and ct_2 of sizes j_i + 1 and j_2 + 1	$t\sqrt{3n}[(12n)^{j_1/2}\upsilon_2 + (12n)^{j_2/2}\upsilon_1 + (12n)^{(j_1+j_2)/2}]$
Square	Ciphertext ct of size j	Same as Multiply(ct,ct)
Relinearize	Ciphertext ct of size K and target size L, such that $2 \leq L < K$	$\upsilon + \frac{2t}{q}min\{B, 6\sigma\}(K-L)n(l+1)\omega$
AddMany	Ciphertexts ct_1,..., ct_k	$\sum_i \upsilon_i$
MultiplyMany	Ciphertexts ct_1,..., ct_k	Apply Multiply in a tree-like manner, and Relinearize down to size 2 after every multiplication
Exponentiate	Ciphertext ct and exponent k	Apply MultiplyMany to k copies of ct

$\mathbf{c}_2 = \sum_{i=0}^{l} \mathbf{c}_2^{(i)} T^i$ with $\mathbf{c}_2^{(i)} \in R_T$ and set:

$$\mathbf{c}_0^{'} = \left[\mathbf{c}_0 + \sum_{i=0}^{l} \text{rlk}[i][0] \cdot \mathbf{c}_2^{(i)} \right]_q$$

$$\mathbf{c}_1^{'} = \left[\mathbf{c}_1 + \sum_{i=0}^{l} \text{rlk}[i][1] \cdot \mathbf{c}_2^{(i)} \right]_q$$

Then, return $(\mathbf{c}_0^{'}, \mathbf{c}_1^{'})$.

Classification of HE operations

As we saw, Homomorphic Encryption schemes allow to perform a set of operations on encrypted data. There exists though two types of operations involving encrypted data, and they have different characteristics as seen in Figure 3.3.

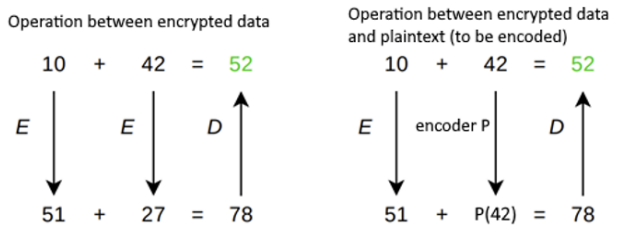


Figure 3.3: Types of Homomorphic Encryption operations

- **Encrypted-Encrypted operations**: In Encrypted-Encrypted operations both of the operands are encrypted data. This characteristic is important because it will determine how fast will the operation be performed and how much noise will be used.
- **Encrypted-Plain operations** In Encrypted-Plain operations just one of the operands in encrypted and the other one is encoded, so it is just represented in terms of a polynomial ring so to being able to be used for performing Homomorphic Encryption operations. Plain operations are less computational consuming with respect to operations between ciphertexts; moreover, they consume much less noise as shown in Table 3.1.

Encoding

As previously said, the Homomorphic Encryption schemes allow performing operation even between plaintext and ciphertext data, by encoding (changing the representation of) the plaintext. Encoding is the process always used, even when encrypting, but just to encode something is different from also encrypting it. Encoding in HE means transforming data (without encrypting it) in the same polynomial form of encrypted values, to be able to perform the various operations among other encrypted data in that form, without encrypting it with any encryption key. Obviously, an operation of decoding is needed in order to obtain the number corresponding to a (possibly previously decrypted) plain polynomial. The problem is not trivial. It is clear that the rings $\mathbb{Z}$ and R_t are very different: for example, the set of integers is infinite, whereas R_t is not. This is the main reason for which HE applications must be carefully designed: in particular, the evaluating party (the party which will have to perform the computation on encrypted data) should find appropriate parameters to guarantee a correct encoding of the involved values. While we will describe the various three encoders available in SEAL for BFV (Integer, Fractional and Batch encoders) we will use in our setting the Fractional Encoder.

Integer encoder

While we will consider the case with $B = 2$, there exist different integer encoders, one for each *base*. The base is denoted by $B \geq 2$. A possible way [32] to encode an integer in the range $-(2^n - 1) \leq a \leq 2^n - 1$ is to form the n-bit binary expansion of $|a|$, say $a_{n_1}, \cdot, a_1 a_0$. The binary encoding of a, then, is:

$$\mathsf{IntegerEncode}(a, B = 2) = \text{sign}(a) \cdot (a_{n-1}x^{n-1} + \cdot + a_1 x + a_0)$$

If $B > 2$, instead of a binary expansion, a base-B expansion must be used. The coefficients are the ones chosen from the symmetric set $[-\frac{(B-1)}{2}, \cdot, \frac{(B-1)}{2}]$, given that there is a unique representation with at most n coefficients for each integer in $[-\frac{(B^n-1)}{2}, \frac{(B^n-1)}{2}]$. $B = 2$ it is commonly used, but also $B = 3$ is usually used. The decoding of an Integer Encoded value is the evaluation of the plaintext polynomial at

$x = B$. It is important to denote that performing modulo operations during the computations may result in making impossible to decode the polynomials. There are two cases, the one is with modulo $x^n + 1$. When at least one degree in the polynomial is greater than n, the decoding operation will fail without even being noticed. The second one is due to the coefficients of the polynomials, where the modulo t operation is performed. It is sufficient that just one coefficient is greater than t and the decoding operation will not be possible, and the operation will fail silently. It is so very important to avoid such bad situations during the running of a service.

Fractional encoder

Fractional encoder will enable the encoding of real numbers. Given that this encoding is done directly on an integer type, the procedure to carry out the encoding will be more complex and different. Fractional encoders, as for the previous encoders, are parameterized by an integer base $B \geq 2$ [55], on the same principle of the Integer encoder. Fractional encoding consists in encoding the integer part of the interested value, using the integer encoder. n is added to each exponent of the fractional part of the binary expansion of the value. Then, the base B is changed into the variable x; lastly, the signs of each term are flipped. [32] presents a practical example. From [32] let's consider $B = 2$ and the rational number 5.8125. It has a finite binary expansion:

$$5.8125 = 2^2 + 2^0 + 2^{-1} + 2^{-2} + 2^{-4}$$

The integer part is encoded as usual, obtaining the polynomial $\mathsf{IntegerEncode}(5, B = 2) = x^2 + 1$. Then, n is added to each exponent of the fractional part $(2^{-1} + 2^{-2} + 2^{-4})$ and the base 2 is changed into x: the result is $x^{n-1} + x^{n-2} + x^{n-4}$. Lastly, each sign of the terms is switched:

$$-x^{n-1} - x^{n-2} - x^{n-4}$$

More formally, for any rational number r with finite binary expansion it holds:

$$\begin{aligned}\mathsf{FracEncode}(r, B = 2) = \text{sign}(r) \cdot [&\mathsf{IntegerEncode}(\lfloor |r| \rfloor, B = 2) \\ &+ \mathsf{FracEncode}(\{|r|\}, B = 2)]\end{aligned}$$

The decoding is performing by applying the presented procedure in the opposite order. There are of course interesting cases, as for real numbers, which do not have a finite binary expansion. For this particular problem the generated expansion of the fractional part has to be necessarily truncated to some precision (which can be expressed as n_f bits). The simplest proposed solution [32] is to fix a number n_i to denote the number of coefficients reserved for the integer part, while the remaining $n - n_i$ coefficients will be used for the fractional part of the encoded number. The condition $n_f + n_i \leq n$ has to be respected.
We saw that the Fractional Encoder of SEAL uses $n_f + n_i <= n$ bits in order to represent the fractional and integer part of an encoded/encrypted number. The higher the n_f the better the representation of the value will be in terms of its fractional part given that this bits value will be used to truncate the expansion of the fractional part to some precision, n_f bits (equivalently, high-degree coefficients of the plaintext polynomial).
Similarly to the Integer encoder, also the decoding with the Fractional encoder can fail. The first reason is the same of the previous case: if any of the coefficients of the plaintext polynomials is greater than the plaintext modulus t, the decoding will probably fail. The second reason is that an homomorphic multiplication may cause the fractional parts of the plaintext polynomials to expand down towards the integer part, with the result that the two parts get mixed up.
it is very important to denote that BFV was born as a scheme for exact computation, more in detail it was born to handle integers. The Fractional Encoder instead allows to have a scaled representation of fractional values, through these assigned bits values, but at the cost of having at the end of the computation some error due to the truncation mentioned above. While using an Integer Encoder won't introduce any kind of error while handling integer values only.

Batch encoder

The Batch encoder will be able to encode in a ring polynomial form a vector of integer values *only*, to which the operation of rotation will be available to be used on. The Batch encoder has the very interesting

characteristic of making the allowed operation perform better than the respective encoding on single numerical values. The downside of this encoder is the support for integer only numbers, which of course can limit its application to a number of applications.

3.1.4 Tuning HE parameters

The BFV scheme is based on the following set of encryption parameters Θ:

- m: *Polynomial modulus degree*,
- p: *Plaintext modulus*, and
- q: *Ciphertext coefficient modulus*.

The parameter m must be a positive power of 2 and represents the degree of the cyclotomic polynomial $\Phi_m(x)$. The plaintext modulus p is a positive integer that represents the module of the coefficients of the polynomial ring $R_p = \mathbb{Z}_p[x]/\Phi_m(x)$, the RLWE problem is based on it. The last parameter q is a large positive integer resulting from the product of distinct prime numbers and represents the modulo of the coefficients of the polynomial ring in the ciphertext space. Choosing different values of these parameters directly affects the performances obtained in the HE scheme. It will directly affect the number of operations which can be done on a ciphertext without making it impossible to be decrypted after the computations, the computational overhead in time and memory requested to perform each operation and the security against attacks. Moreover, parameters affects also the precision of the operations' result, especially operations involving Fractional encoded values.

The Noise Budget (NB), introduced in Sub-Section 3.1.3, is a direct and numerical measure of the number of operations which can still be done on the selected ciphertext while ensuring the correctness of the final decryption operation. Every operation consumes NB like additions, but in particular multiplications between ciphertexts are the most expensive operations in terms of NB consumption. Especially the

multiplications involving two encrypted values, differently from operations involving a ciphertext and a plaintext, given that a plaintext does not have a NB, which consumes a certain amount of NB (consider that the result of such operation will be an encrypted value) strictly less than the NB consumed in the operation between two ciphertexts as stated in Sub-Section 3.1.3. When the NB of a ciphertext reaches 0, it will then become completely impossible to correctly decrypt it, corrupting definitely the resulting value. Given this fact, it is fundamental to careful tune the encryption parameters for ensuring a sufficient initial NB able to carry out all the next planned operations.

Choice of the parameters

For further details on those parameters the reader can look at SEAL [32] which is the library being used where the BFV HE scheme is implemented, in the followings we will briefly explain some of those parameters' characteristics.

In order to minimize the computational load and the noise consumption while maximizing the noise budget, we can search in the space of these three parameters:

- p : plaintext modulus, all operations will be made in modulo p. It increments the precision of the operation on encrypted data, but also increase the NB consumption per operation.

- m : polynomial ring coefficient modulus, it must be a power of 2. Higher is the modulus higher the number of operations (higher initial noise budget), unfortunately it proportionally increases computational load in terms of execution time and memory occupation.

- q : ciphertext modulus, it affects the security level of the encryption scheme, left to pre-computed values of SEAL, due to its complexity, which provides a way to automatically set q given m and the desired AES-equivalent security level *sec*, as shown in Figure 3.4 from [32].

n	Bit-length of default q		
	128-bit security	192-bit security	256-bit security
1024	27	19	14
2048	54	37	29
4096	109	75	58
8192	218	152	118
16384	438	300	237
32768	881	600	476

Figure 3.4: SEAL pre-computed q values

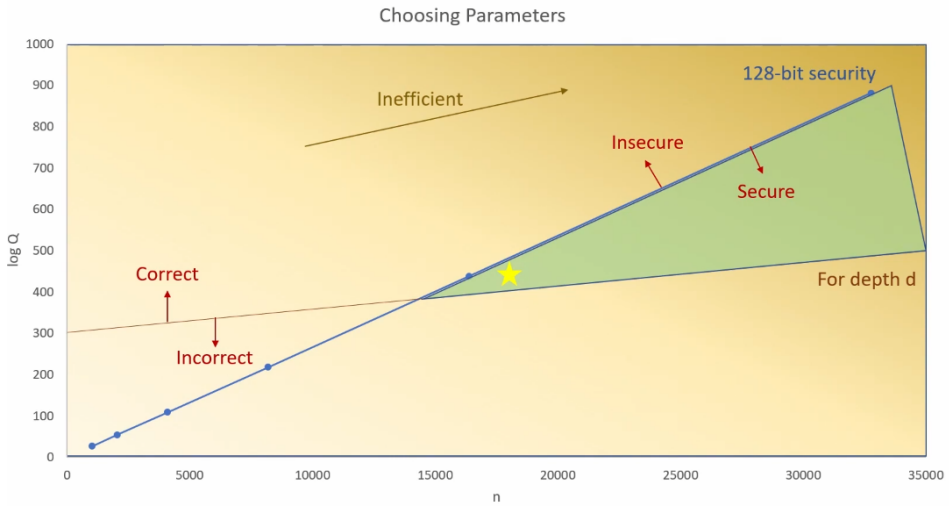


Figure 3.5: (m,q) parameters space showing the available and secure areas

it is also very important to remember that these parameters are not just here to fine-tune the HE scheme but they are also fundamental for it to work correctly and being secure as desired. This means that there exists a set of Θ parameters which make a scheme not secure or incorrect in terms of performing the computation of the operations. This effect can be clearly seen in Figure 3.5 taken from the presentation [1] of [32].

3.1.5 Homomorphic Encryption challenges

Homomorphic Encryption schemes, as it has been discussed in the years, bring quite some challenges and difficulties. The main difficulties are the current limitations involved around the computational time required to handle encrypted computation, especially using large data and/or large HE parameters Θ. It will affect the occupied memory of the homomorphically encrypted data, its encryption and decryption time but indeed more importantly the time required to process operations on the data [56]. These difficulties are the reason behind why currently to make Homomorphic Encryption practical every work is exploiting computationally powerful servers on the Cloud through Cloud Computing, as it was also stated in this work [57]. There are also difficulties related to the limitation on the possible operations available. What's available for any Somewhat Homomorphic Encryption scheme is the addition/subtraction and the multiplication. The very issue is about the missing compatibility with the division operation which is not supported in the polynomial ring algebraic structure. However, existing schemes and libraries added support for other useful operations as we can summarize in the below Table 3.2. As we can see, both operations, while fundamental in many cases, of division and comparison are not present in any of the available HE schemes up to now. This is an important drawback that can be circumnavigated with some effort. In most of the literature the division operation is just seen as a multiplication inverse of an Encrypted-Encoded operation where we can actually multiply the inverse just by encoding the value as the inverse of the division operand. The comparison operation instead is quite difficult to achieve homomorphically and indeed there has been done some work [59] about it and some methods available for certain HE schemes achieve a decent computational complexity in performing the function approximating the behavior of the comparison operation. Difficulties around Homomorphic Encryption schemes are also related to their usability, without proper libraries or frameworks they are quite challenging to be used. This is why quite a few libraries supporting one or more Homomorphic Encryption schemes have been developed. While this work will rely on SEAL [32], also other libraries currently

Table 3.2 HE schemes and available operations

Operations	SEAL(BFV, CKKS)	HElib	TFHE	Paillier	ELGamal [58]	RSA
Addition, Subtraction	Yes	Yes	Yes	Yes	No	No
Multiplication	Yes	Yes	Yes	No	Yes	Yes
Comparison	No	No	No	No	No	No
Division	No	No	No	No	No	No
Boolean operations	No	No	Yes	No	No	No
Bitwise operations	Yes	Yes	Yes	No	No	No
Matrix operations	Yes	Yes	No	No	No	No
Exponentiation	Yes	Yes	No	No	No	No
Square	Yes	Yes	Yes	No	Yes	Yes
Negation	Yes	Yes	No	No	No	No
Add Plain, Subtract Plain, Multiply Plain	Yes	No	No	No	No	No

exists such as Palisade [60], HEANN [61] and HElib [62]. Recently there has also been some new developments about HE frameworks [63] [64], so that who will have to perform a Homomorphic Encryption execution flow won't have to be so responsible for all the management and initialization process as with the provided libraries, but these frameworks are the starting point for a bright future where Homomorphic Encryption will be more and more mainstream. Indeed, more and more excellent works are being done regarding the application of Homomorphic Encryption to secure algorithms of any kinds, as these works [65] [66] [67] regarding the application of Homomorphic Encryption to substring search and pattern matching where one of their main application is Bioinformatics in DNA sequencing: in these applications Homomorphic Encryption should be a must-have! Regarding the difficulties in achieving comparable performances to in-clear computations, there has been some development with respect to GPU computations. In this ongoing work [68] it has been started to implement a working version of the CKKS scheme running on PyTorch and compatible on GPUs. This can actually solve the challenge of Homomorphic Encryption regarding the quite higher time required for its execution compared to plaintext computations.

3.2 Machine Learning

Machine Learning (ML) is a subfield of the Artificial Intelligence field. It is the study of computer algorithms able to learn how to better solve a specific task at hand. This reflection of human behavior is possible through the Statistics and Mathematics behind ML. A more precise definition for a learning algorithm is:

Definition 3.2.1. A program learns from a certain experience E, with respect to a class of tasks T, obtaining a performance P, if its performance in solving tasks of type T, measured by performance P, improves with the experience E.

ML algorithms are often used, and better shine, when there are difficult problems to solve if they were solved through a hard-coded algorithm. ML algorithms are applied to a vast number of fields and tasks such as: image classification, instance segmentation, sequence classification, sentiment analysis, text-to-speech, speech-to-text, visual-question-answering, clustering, behavior learning, etc... The various tasks listed above belong to three different paradigms used to build ML algorithms: Supervised Learning, Unsupervised Learning and Reinforcement Learning. While they are all important given the nature of this work, we will further describe the Supervised Learning process.

Supervised learning

In this case, the Machine Learning algorithm is fed with a series of inputs (X) and the corresponding outputs (Y) to learn from. During a phase called training consisting in a series of epochs call, where an epoch consists in evaluating the entire series of the data set. The algorithm will then use a loss function in order to understand how much distance there is between the real and ideal mathematical model mapping $Y = f(x)$ and the current one being trained, this will be used to accordingly update the corresponding weights of the Machine Learning model. The update of the weights happen through an optimization algorithm which can be of various types from the basic one of gradient descent, to SGD, to Adam, to AdamW and so on. Each of

these algorithms will provide a different convergence rate to the optimal model obtainable given the model architecture and provided data set. The program is given some inputs (X) and the desired outputs (Y). Through a series of attempts, the program constantly improves with the goal of finding the best possible mapping between the inputs and the outputs ($Y = f(x)$). If the examples are enough representative of the problem's domain and the model is enough flexible to correctly learn complex relationships, the final mapping will have a good performance even on new, unseen data. Supervised learning is generally used in regression and classification task. Common types of models used in Machine Learning are Linear Regression, Naive Bayes, Logistic Regression, Support Vector Machines (with or without Kernels), Neural Networks, etc... Each one of these models are good in their own way and there is no particular always-best model than others: it all depends on the task, this is essentially the No Free Lunch theorem where it is stated that we shouldn't have a preferred algorithm or model. We have to consider what's best on a particular task.

In general, ML algorithms outperform human ones in tasks which encompass a high amount of possible input data with a corresponding amount of hidden patterns in the mapping between inputs and outputs. Building ad-hoc algorithms in such domains can be time-consuming or not possible and have bad performances, while the capacity of ML algorithms of improving themselves simplifies the work of programmers. The work will shift to the identification of representative couples of input and output data and to the manual labeling of such training set.

3.2.1 Deep Learning

Deep Learning (DL) is a subset of Machine Learning. Differently from ML, Deep Learning encompass the set of algorithms and methods to create better representation of data, extract and select features automatically in order to ease the work on a classifier or a regression model: it learns both features and model end to end. So, now that the features are no more described or found by the engineers but by the computer itself, we are able to extract better and better characteris-

tics from data. It all depends on the amount of data available and the flexibility (the complexity) of the model, proportionally on the task and the data. Again, differently from ML when we are talking about DL we are talking about Neural Networks only: or better, architectures comprehending various layers of Neural Networks in order to go "deeper" in finding better and better features.

While there exists a number of DL models, this work will focus in particular on Transformers and their application on the general classification task through BERT.

3.2.2 Natural Language Processing

Natural Language Processing (NLP) is a field already present before the dawn of Machine Learning, but that it empowered itself even more thanks to the novel Machine Learning methods. Giving computers the ability to handle and to process the natural human language is a very important and old idea. There has been a lot of efforts through the years in building algorithm based (and not data based) models trying to capture human language and to process it for a variety of reasons: understanding its meaning, answering to questions, machine translation, speech recognition, speech synthesis, creating summaries etc... The real main problem for NLP has always been the ambiguity which is carried by any natural language, and that is what's most difficult to make computers handle. Since the beginning of the climbing of Machine Learning in modern times, NLP took advantage of it building better language models, better syntax parsers and a variety of its applications previously listed. NLP models are getting better and better nowadays thanks to also the arrival of Deep Learning. This was a game changer because it made it possible to find directly and automatically from data the best features for a certain NLP task. There are a lot of various applications of DL in NLP, and this has given NLP much more space for growing. There are a lot of different areas of working in NLP:

- **Part-Of-Speech (POS) tagging and Named Entity Recognition (NER)**: both of the two are sequence labeling problems over text data. POS tagging is the NLP task of assigning syn-

tactic tags to a certain text in order to better handle further parsing and controls. NER instead is the process of finding the corresponding pronoun for a certain entity, there exists algorithm for finding it but of course through Deep Learning models using Attention layers, RNN, CNN etc... you get better results. In these cases, the input is a word embedding vector and the output class computed over Softmax is a set of POS tags or NER classes.

- **Machine Translation**: this task has the responsibility to automatically translate a given utterance in one language to another language. In this task, the given input is again a set of word embeddings vectors representing each words of an utterance and the output is the probability distribution of the word sequence composing the translation in the target language. Various solutions exist like using Transformer Encoder-Decoder and Recurrent encoder-decoder Seq2Seq with LSTMs. The current state-of-the-art architecture for this task is BERT [7] as for most of the NLP tasks nowadays.

- **Question Answering**: for this task given a sequence of sentences and a final question we want the answer to that question. Various possible solutions exists like Recurrent Seq2Seq models, Memory networks with Attention mechanism, Transformer architecture like in GPT-2 [69] and GPT-3 [70]. The Question Answering task is part also of the general Chatbot application, where its implementation depends on the type of the interaction (single-turn or multi-turn) and how the core algorithm is implemented (whether generative or retrieval).

- **Speech and Sound Recognition**: before DL, speech recognition was using a predefined set of 39 feature vectors encapsulating the relevant parts of the Fourier Transform of the audio spectrum. It is a fine solution, but not if compared to the wonders offered by Deep Learning. In DL models for Speech Recognition, the input can be given as a raw input in the time domain or with more elaborated input in the frequency domain with the spectrum. In both cases those inputs are used in order to extract the acous-

tic features, and then they are passed to further layers in order to perform the recognition and classify the corresponding input, whether as a classification or as a continuous labelling. For this aspect any kind of layer can potentially be used, with different interesting advantages, like Convolutional layers, Recurrent layers or Attention layers. Many interesting architectures exist like SoundNet [71] which is CNN based with raw inputs, VGGish [72] that is again CNN based but with spectral inputs and Wav2Vec [73] in order to find a compressed representation through an auto-encoder.

- **Text Classification**: text classification includes the two similar tasks of Sequence Classification and Sentiment Analysis which are the tasks related to predicting a certain class of a sample text, whether it belongs to a subset of sentences, questions, etc... or whether it holds a certain sentiment, generally positive or negative. The general input for this kind of task is a certain utterance, represented differently than just raw words. In input we are going to give the model a certain numerical representation. There has been various different kinds of representations starting from the one-hot encoding from a dictionary, to the TF-IDF (Term Frequency — Inverse Document Frequency) to the most used one nowadays which are the Embeddings. Each embedding is representing one word numerically through a vector. Given this input, any kind of layer can be used: from 1D convolutions and poling to Recurrent layers to Attention layers. The final classification is dealt with a flattening layer like GAP, and then the Softmax activation function is used in order to get the probability distribution assigned to the set of the pre-defined classes. The presented work will focus on this specific NLP task. There exists various different architectures for this task, from the linear one represented by Logistic Regression to the most complex ones like Bi-RNN, LSTMs [45] and the Transformers [49] holding the state-of-the-art with the BERT architecture achieving really exciting results.

- and many others...

Embeddings

In the previous tasks we talked about the presence of feature vectors representing words numerically. Embeddings serve the purpose of representing the input sequence in a different dimension. Embeddings represent and map textual representation, words, into vectors $v \in \mathbb{R}^n$ of a certain dimension n. Since we move from a raw textual representation to a mathematical representation in form of a vector, embeddings are transforming the input from a high dimensional input space to a lower dimensional numerical vector space representation which has the property of having its vectors comparable and linked together dependently on the corresponding meaning of the words they are representing. This means that two similar words will be numerical near in the vector space of the embeddings: closeness in this vector space means semantic similarity and correlation as shown in Figure 3.6

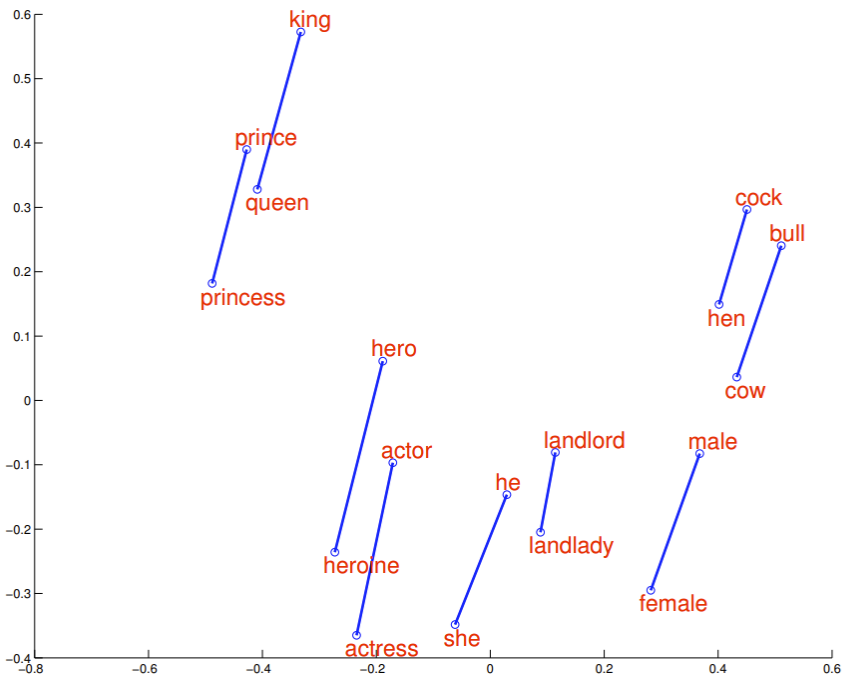


Figure 3.6: Visualization of Regularities in Embeddings Word Vector Space

We can the resume the characteristics of word embeddings as:

- **Compression (dimensionality reduction)**: the vector size d is way smaller than the original dictionary dimension, which makes embeddings easier to use in Machine Learning applications as features.

- **Smoothing**: from discrete orthogonal representations, we move to a continuous representation, making possible computation regarding the similarity between words based on their original encapsulated meaning.

- **Densification**: we move from a very sparse representation of a discrete set of words to a more dense representation where the word embedding vectors are not just now comparable, but they are quite close to each other in their sub-dimensional vector space.

There are different architectures for word embeddings one of the most known are Word2Vec [74] (with Skip-Gram architecture where the surrounding context is predicted or with the continuous bag-of-words architecture where the missing word is predicted in order to train the network for finding the correct embeddings), Glove [75] and FastText [76]. Other than word embeddings there exists also sentence embeddings, which are compressing the meaning of a sentence in common vector of continuous values.

One of the advantages of NLP, or better of the text-related tasks of NLP, is the less difficulty in finding data in order to train models. Through the Internet, an enormous amount of natural language data in textual form can be accessed and prepared for being used in training a model for a certain task.

NLP combined with DL is changing and reshaping the world with the better and better applications they find themselves fitting it or that they create. One thing for sure is that NLP will contribute more and more in the future, it will be fundamental in building the Computer-Human user interface of the future.

3.2.3 Transformers

Transformers [49] are a recently proposed architecture for Natural Language Processing (NLP) in a variety of tasks: Sequence Classification, Sentiment Analysis, Machine Translation, speech-to-text, Question Answering, etc... but it also found applicability in various other different or correlated fields like image recognition or Visual Question Answering (VQA). The architecture outperformed the previous state-of-the-art defined by Recurrent Neural Networks (RNNs). The Transformer is a complex yet simple architecture composed of a variety of different modules. For its simplicity it is meant that it is way more efficient than previous RNNs architectures where the parallelization among a sequence was not possible given those architectures nature. In the Transformer architecture instead we are not bound to compute in sequence each data in input, but we can parallelize it, and we can so remove the dependence on the sequence length. Indeed, the Transformer big-O complexities are the followings:

- complexity per layer: $\mathcal{O}(n^2 \cdot d)$ where n is the sequence length and d the embedding dimension.
- sequential operations: $\mathcal{O}(1)$
- maximum path length: $\mathcal{O}(1)$, no dependence on sequence length, allowing to capture very long range dependencies easily.

The Transformer uses, as well as any model in Deep Learning, Artificial Neural Networks (ANNs). It isn't composed by ANNs only, but it contains a series of other different modules: Self-Attention, Layer Normalization with residual connections and the initial Embedding layer.

The architecture of the Transformer allows it to elaborate the sequences in input in order to gather all the obtainable relevant features required to perform the said NLP task.

What's different from previous existing architecture in NLP is the presence of the Self-Attention module, which will be better explained.

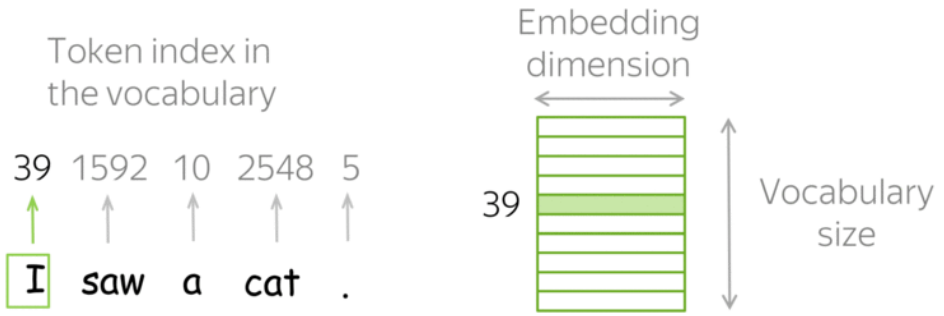


Figure 3.7: Extracting the corresponding word embeddings

Embeddings

The Embedding module serves the purpose of representing the input sequence in a different dimension. The embedding module is always placed at the beginning because it is the essential part of the model, making the input ready to be fed to the model no more as a sparse discrete representation but as compressed dense representation through continuous vectors. In this module, the embeddings are generated from words just by making an indexing operation. The embedding module contains a dictionary where for each word present inside the dictionary there is the corresponding word embedding vector that is then extracted as shown in Figure 3.7 from [77]. The corresponding word embedding vector is essentially generated through auto-encoder self-trained networks like Word2Vec.

Self-Attention

Self-Attention is the module which has the responsibility to find on which data of a sequence better focus on, better pay attention to. In Self-Attention module there are various operations, as it is not just one single operation, but it encompasses a series of different operations inside it. Inside it every word in a sentence is weighted through different operations which involve three principal vectors: Query, Key and Value. (They're abstractions that are useful for calculating and thinking about attention.) Essentially, Q helps keeping focus on a word on a sentence being analysed, whereas Key it is used to help computing the score among all the words on a sentence. Value will be used to

know which words are worth being focused on.
The first step in calculating self-attention is to generate Q, K and V vectors from each of the encoder's input vectors (the word embeddings). So for each word in input, a Query vector, a Key vector, and a Value vector are created. The three vectors are generated by multiplying the word embedding by the three matrices that have been trained during the training process of the model.
The second step of self-attention is to compute a score. Starting from the self-attention of the first word of input. The score of each word of the input sentence against this first word is needed. The score is indeed what determines how much "attention" and focus to place on other parts of the input sentence while encoding a word at a certain position. The score is calculated through the dot product of the query vector with the key vector of the respective word its score is being computed. So if the model is processing the self-attention for the word in first position, the first score would be the dot product of q_1 and k_1. The second score would be the dot product of q_1 and k_2.

$$Q \cdot K^T$$

The third and fourth steps are to divide the scores by the square root of the dimension of the key vectors. This operation has the functionality of being a scaling factor, it is used to having more stable gradients during the training phase.

$$\frac{Q \cdot K^T}{\sqrt{d\_model}}$$

After having performed this operation, the result is then passed through a softmax operation: this will normalize the scores making them positive and add up to 1, given that they will be mapped to a probability distribution.

$$softmax(\frac{Q \cdot K^T}{\sqrt{d\_model}})$$

The fifth step involves the multiplication of each value vector V by the result of the previous softmax operation. The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for

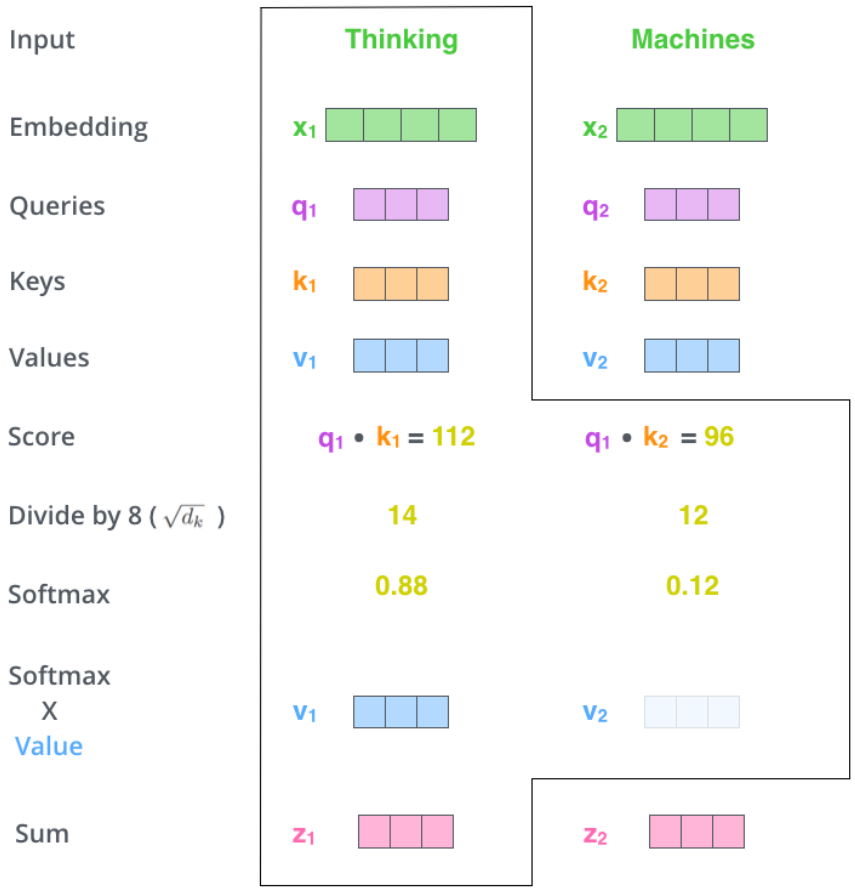


Figure 3.8: Self-Attention

example). [78]
The sixth and final step is the summation the weighted value vectors. This produces the output of the self-attention layer at the first position, so for the first word. In the following Figure 3.8 from [78] we can better see how all the procedure works.

$$Z = softmax(\frac{Q \cdot K^T}{\sqrt{d\_model}}) \cdot V$$

After this important computation to better generate the context of a sentence, the output is a set of vectors which can be sent along to the feed-forward neural network, where each vector can be forwarded independently. This concept is really important because even the self attention can be computed independently, and this means that here differently from RNNs the parallelism can be exploited way better.

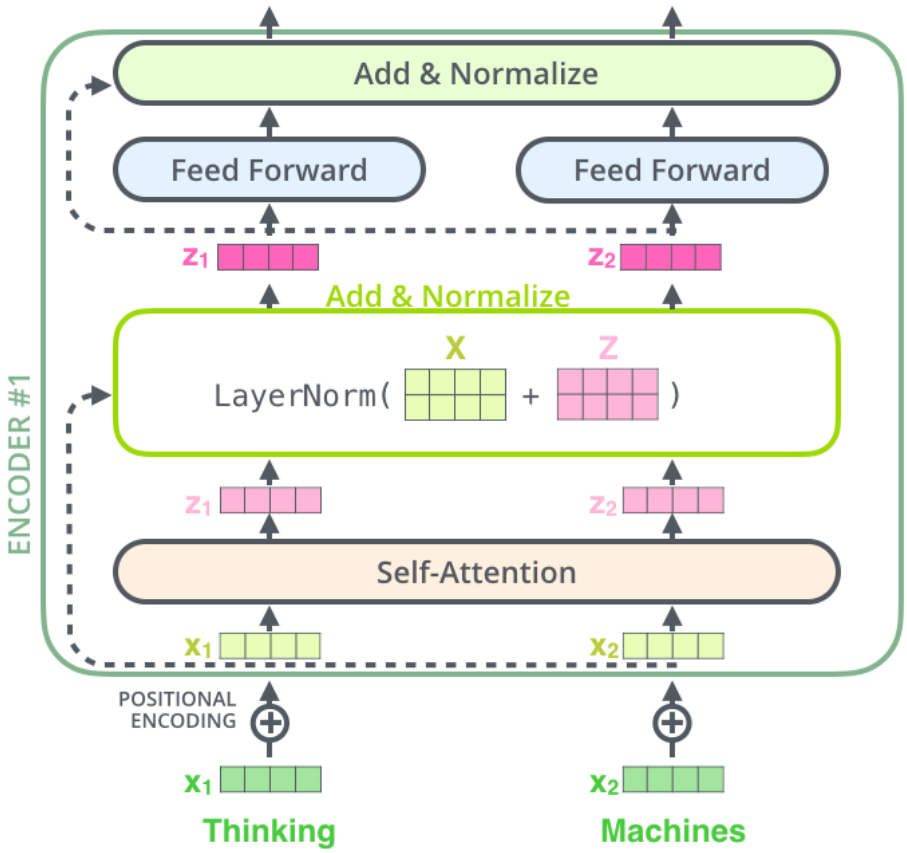


Figure 3.9: Transformer Encoder

Also, here the order is not important, or better: by using a vector indexing the order of the words there are no issues like in RNN to give in the network inputs in a specific correct order (could be important for some applications). In the end the architecture of the Encoder (the Deep-Learning feature extractor) looks like this in Figure 3.9 from [78]:

Activation functions

Between the two Feed Forward layers shown in Figure 3.9 there is an activation function. The activation is used to modify the output of the neurons before passing it to the next ones, this approach enables the network to better generalize over more complex data, to be more flexible. In order to do that only non-linear functions will be useful to

represent non-linearities of complex data better described, usually, by Neural Networks models. There a various number of existing activation functions but in the Transformer architecture two common types are usually used which are ReLU and GeLU. They are defined as the followings, ReLU $= max(0, x)$ and GeLU $= x \cdot \frac{1}{2}[1 + erf(x/\sqrt{(2)})]$ but it can be also approximated for faster computation as $0.5x(1 + tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$. ReLU (Rectified Linear Unit) was first introduced for Neural Networks in order to solve a very important problem, which is the vanishing gradient. Using activation functions previously common in ML such as tanh or sigmoid, there was the problem of not being able to make Deep Learning models learn due to the various layers present to be trained. This problem, it is particularly important for Feed Forward Neural Networks (FFNNs) and Recurrent Neural Networks (RNNs). For RNNs it is due to the fact that while doing back-propagation through time unrolling the network for U steps back in time we are computing the part of the gradient as product of the various other weights of the "static" part of the network but also of the recurrent part of the network. In the recurrent part of the network and its following output we will have as derivative the weight_of_recurrence $\cdot \frac{\partial f}{\partial x}$ output_function: if we compute the norm of this we see that the norm is less than or equal than the product of the norm of these two terms, multiplied by themselves U times: if the norm of one of the two is very low or just < 1 it will cause a vanishing gradient after a certain time of k<U steps because it will make the product converge to 0 and so all the back-propagation converge to 0. The problem with ReLU it is not solved in RNN yet, but it will need to use the Circular Error Carousel introduced in the Long-Short Term Memory (LSTM) model in order to avoid the opposite problem, the exploding gradient. In the FFNNs ReLU and Leaky-ReLU solves the vanishing gradient problem by substituting tanh and sigmoid functions which are the main cause of vanishing gradient due to their saturation. The ReLU activation solves it by setting its gradient either to 1 if input is greater than 0 or to 0 otherwise. Leaky-ReLU solves the problem of not being to represent negative values by setting its derivative to 1 if input > 0 otherwise to $0.01 \cdot x$. This actually means that in the Transformer as long as in the FFNN module which is applied on each part

of the input sequence in the various modules is being used as activation function ReLu or Leaky-ReLU, then it won't suffer from vanishing gradient because the ReLu activation function solves it by setting its gradient either to 1 if input is greater than 0 or to 0 otherwise. The self-attention part instead is not a cause of vanishing gradient because it is essentially computing self-attention by performing as seen before the softmax of the various operations between the Q, K and V tensors which are just the values obtained to compute self-attention on the input sequence in order to decide on which part the model should better focus on.

Layer Normalization

Layer Normalization is a layer used to re-scale data in a better representation for further computations, reducing it to values between 0 and 1. It consists in computing the standard deviation and mean of the current tensor in input to the layer and the output will consist of $y = \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}}$. The main difference between this normalization and another common one such as Batch Normalization is how the mean μ and the variance σ^2 are computed: in this case they are computed during each inference (even not in the training phase) while in Batch Normalization μ and σ^2 are learnt, and so they won't be computed no more. As we will see in the next chapters, Batch Normalization will play an important part in our work.

Feed Forward Neural Networks layer

In an encoder of the Transformer architecture there are two Feed Forward Neural Network layers being used in order to better extract features and compute the prediction after having computed the Self-Attention. It consists of a weight tensor learnt during training together with a bias vector. They are then used to multiply and sum the input value to the module in order to transform it accordingly to the weight previously learnt. These layers will also be used again for the final classification being performed after completing the computation in the Transformer encoder.

Pooling layer

Used for end-classification purposes, pooling layers can use different functions to implement such simplification: the most common ones are average poolings and max poolings. In each case, the pooling layer will extract the more relevant data depending on the function, by averaging a tensor into a matrix or by extracting the maximum matrix from the tensor.

Softmax layer

While it is usually used at the end of the classification, in this case Softmax has also the particular function of being inside the Self-Attention module for the Transformer architecture. Its function has the property of mapping the tensor values along a certain dimensions to a probability distribution.

Chapter 4

Architecture of the proposed solution

The proposed architecture is shown in Figure 4.1.
Privacy-preserving computation comprises two different actors: the user U and the service provider S. The user U aspires to execute a service, e.g., a Cloud-based service or a mobile application, provided by the service provider S for his/her purposes but he/she requires the service provider not to access his/her data. To achieve this goal, the service provider S make available the requested service in a privacy-preserving "as-a-service" where the designed and developed service is able to process data that have previously encrypted by U. The result of this processing is still encrypted and only the user U can decrypt it.

In our specific case, HErBERT is a privacy-preserving text classifi-

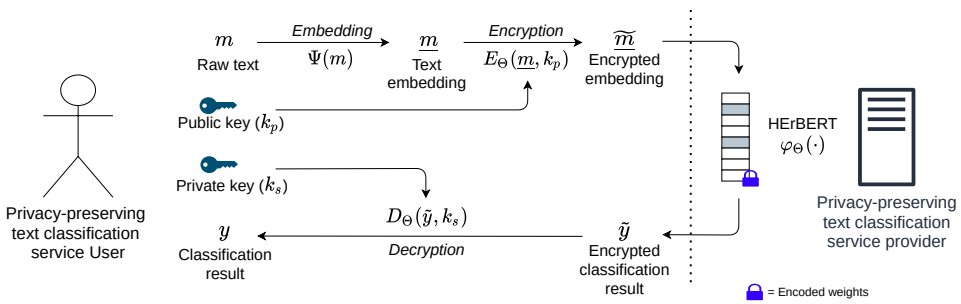


Figure 4.1: Computing chain of a text classification service, working on encrypted inputs.

Table 4.1 HErBERT symbols

Symbol	Meaning
U	user
S	service provider
m	raw text
k_p	public (shared) key
k_s	private (secret) key
$\Psi(m)$	embedding function of raw text m
$\underline{m}$	text embedding of raw text m
$E_\Theta(\underline{m}, k_p)$	encryption function of text emedding $\underline{m}$ using the public key k_p
$\widetilde{\underline{m}}$	encrypted text embedding $\underline{m}$
$\varphi_\Theta(\widetilde{\underline{m}})$	HErBERT making inference on $\widetilde{\underline{m}}$
$\hat{y}$	encrypted prediction result of HErBERT
$D_\Theta(\hat{y}, k_s)$	decryption function of encyrpted result $\hat{y}$ using the private secret key k_s
y	decrypted prediction result of HErBERT

cation service (e.g., to distinguish between spam or not) provided by a service provider S and the user U is interested in using HErBERT on his/her text data m without sending it in plaintext. An overview of the proposed HErBERT solution is shown in Figure 4.1 encompassing user U, service provider S, the encryption/decryption phases, the embedding phase and the inference of the HErBERT NLP deep neural network for the text classification.

In more detail, given m the raw text to be classified, the user U generates the secret and public keys, i.e., k_s and k_p, and performs the text embedding on his/her own device (e.g., the personal computer or the mobile device). In this last step, i.e., the text embedding, the words composing the raw text m are embedded by means of an embedding algorithm $\Psi(\cdot)$, obtaining the text embedding $\underline{m}$. The embedding algorithm together with the encryption/decryption phases and the HErBERT architecture, training and inference will be detailed in Chapter 5. We emphasize that the text embedding represents a form of extraction of numerical features from text m. This embedding phase, requiring operations (e.g., the indexing operation) that are not allowed in the HE scheme, is executed on the user device, but this does not limit HErBERT since $\underline{m}$ it is encrypted before being sent to the service provider S. Without any loss of generality $\Psi(\cdot)$ can be provided by the service provider S (being trained from existing provided models such as HErBERT) or downloaded from public-services by choosing a pre-trained word-embedding with a compatible dimensionality to HErBERT.

Once $\underline{m}$ is computed, it is encrypted by the user U through the encryption step

$$\widetilde{\underline{m}} = E_{\Theta}(\underline{m}, k_p)$$

by means of the public key k_p obtaining the encrypted version $\widetilde{\underline{m}}$ of the text embedding $\underline{m}$. The encrypted embedding $\widetilde{\underline{m}}$ is then sent to the service provider S for the privacy-preserving text classification $\varphi_{\Theta}(\cdot)$ through HErBERT:

$$\hat{y} = \varphi_{\Theta}(\widetilde{\underline{m}}).$$

This is the *core* of this work: designing an effective and efficient $\varphi_\Theta(\cdot)$ requires both to completely re-design the NLP text-classification solution taking into account the constraints on the type and number of operations characterising the BFV scheme and to efficiently implement it by means of novel algorithmic solutions to reduce the inference time (since traditional GPU-based solutions cannot be here considered). These aspects will be better explained in Chapter 6.

We emphasize that, to support the processing of encrypted data, $\varphi_\Theta(\cdot)$ is encoded by means of the encryption parameters Θ. The output $\hat{y}$ of $\varphi_\Theta(\widetilde{\underline{m}})$ is the encrypted version of the text classification provided by HErBERT. Only the user U, by relying on the private key k_s, can then decrypt it and get the final result

$$y = D_\Theta(\hat{y}, k_s).$$

The Deep-Learning models considered in this work on which HErBERT is based are Transformers (BERT) models for Sequence Classification, but as it will be clear the HErBERT architecture can be easily adapted to any other NLP task. Approximating the original model, so actually changing its composing modules, where it is needed, is a must in order to make it possible to run the model using Homomoprhic Encryption operations, so with just additions and multiplications. This is unfortunately something usually needed when Homomoprhic Encryption and Neural Networks come together because of the presence of non-linear functions present in these type of models.
The general model architecture is completely based on the Transformer's BERT general architecture, but approximated and reduced. So it will have the central part of the Transformer which is Self-Attention, but it has been approximated in order to be used with Homomoprhic Encryption. The same goes for the further parts of the Transformer encoder, which are the normalizations and the activation function of the two Feed Forward Neural Network layers. The approximated architecture, with the various details and reasons of the approximations, is shown in Figure 4.2.

it is important to denote the various steps in order to make the HEr-BERT possible.

- Reducing the size of the model
- Approximating the model for HE-compatibility
- Training the model from scratch
- Encoding the model for privacy-preserving inference

The architecture was in the need of reduction compared to the common size of Transformer and the Transformer base architecture, BERT, which has been used. This was needed in order to deal with the noise budget: those model dimensions were really too large to handle with Homomorphic Encryption, we can't just use tensor with dimensions of thousands in magnitude with these ciphers because even the highest parameters will fail to handle such large data, and also the required time for computation is simply unfeasible.
BERT needs to be reduced and approximated in order to be compatible with HE available operations: the various approximations are explained in high details in this Section.
Training the model from scratch is required in order to get the best weights for this approximated and reduced BERT architecture, given that the architecture is changed from the required resizing and the approximations.
The final step, after having trained the model, is to encode its weights found during training. It has to be encoded so that the Homomoprhic Encryption scheme BFV can perform privacy-preserving operations with the received encrypted inputs. Encoding the model means encoding all its plaintext parameters into an encoded polynomial ring representation through the HE parameters Θ, so that now $\varphi_\Theta(\hat{I})$ can perform operations on the encrypted inputs $\hat{I}$.

4.1 Embeddings, Encryption and Decryption

In NLP in general we make use of Embeddings, an ordered subspace representation of the unordered set of possible words present in a certain vocabulary. Given the nature of Embeddings and how they are obtained during inference from a text representation, we have seen that they have to be made part of the actual pre-processing (and it is actually a pre-processing) of text, and it needs to be performed on the client side. This is due to the fact that to obtain Embeddings from a word we need to index integer values, corresponding to a certain dictionary of words, to a tensor containing the real values associated with the subspace representation of that word that we call the embedding. This indexing operation, among other operations being performed for the process of Embeddings, is not inside the set of available operation in HE schemes. So, the client before encrypting will have to pre-process the input text through the trained Embedding layer of Transformer and then send the HE encrypted Embeddings of the sample to the Cloud Computing service. This means that the encrypted input, $E(I, \Theta, k_p)$, which it will send, already contains the indexed embeddings. The server at the end of the classification task will send back still encrypted data. When the user will decrypt them will see a set of values on which it will be able to perform the Softmax operation in order to obtain the highest predicted class probability among all the present ones.

4.2 HErBERT

HErBERT is a privacy-preserving DL model for NLP, in particular for text classification. The proposed HErBERT architecture is shown in Figure 4.2, while a thorough description is provided in the remaining of the Section. The definition of the model has taken into account the limitations imposed by HE, because HErBERT is specifically designed to be used on encrypted data.

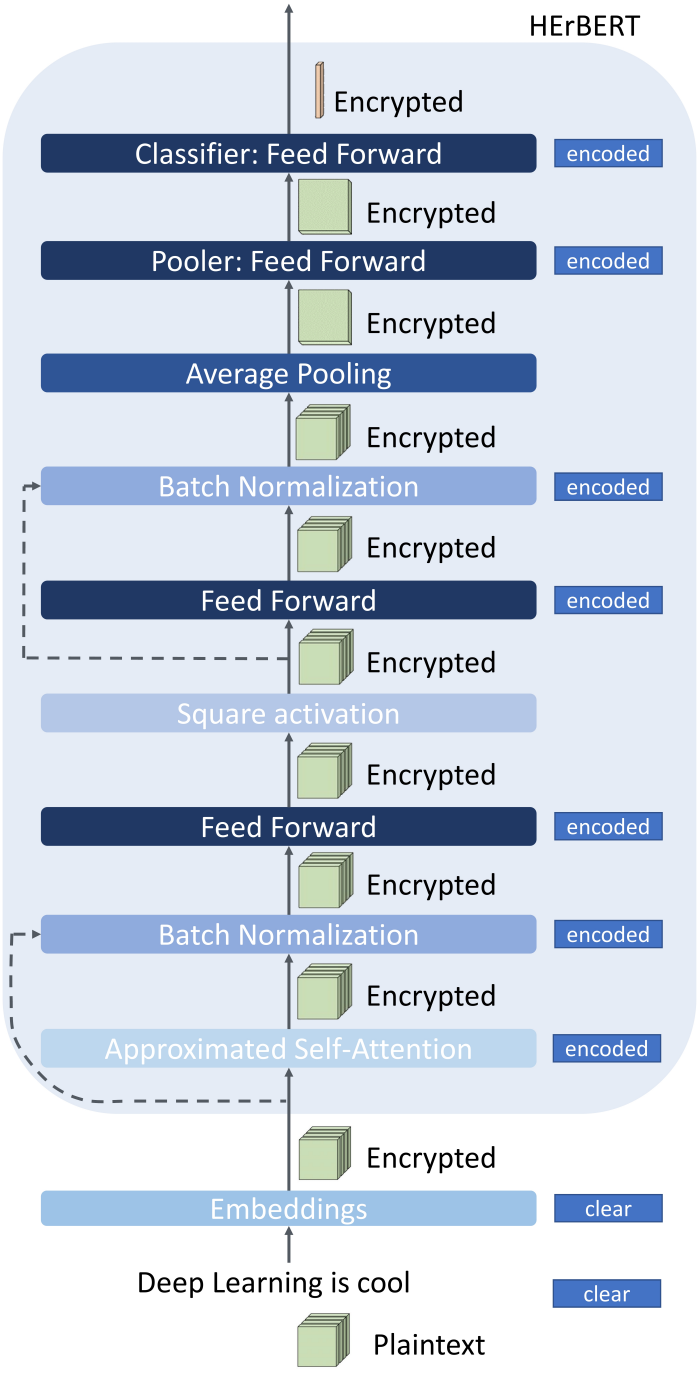


Figure 4.2: HErBERT architecture and classifier

Embedding layer

The first layer of HErBERT is the Embedding layer. As we can see from its describing figure it is receiving input in plaintext and that's because, as it will be better explained, the limitations of HE won't allow to perform the required indexing operation to extract the corresponding embeddings from the words of the input text. Anyway, the Embedding layer as in BERT is constituted in order to let the embedding vectors determine the position of each word, or the distance between different words in the sequence. Adding these positional values to the embeddings provides meaningful distances between the embedding vectors once they're projected into Q/K/V vectors and during dot-product Self-Attention. These positional encodings have the same dimension of the embeddings, given that the two are to be summed. There are many choices of positional encodings, learned and fixed [79]. In [49] it has been experimented in using both fixed and learned positional embeddings. While it has been found out that the two versions produced nearly identical results, a fixed mapping function $f : \mathbb{N} \to \mathbb{R}^k$ is better because it allows the model to extrapolate sequence lengths longer than the ones encountered during the training phase of the model. In HErBERT's embedding layer, it has been used the same mapping function constituted of sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d\_model})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d\_model})$$

where pos is the position and i is the dimension. So each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. This function is suggested [49] because it has shown the ability of allowing the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}. The final output of the Embedding layer will be then the indexed word embeddings plus the fixed positional embeddings which will model the order of words in the input text, in HErBERT the embedding dimension will be of 4 or 8.

Approximated Self-Attention

The second layer of HErBERT is constituted by the approximated Self-Attention. As can be seen from Figure 4.2 this layer has its parameters encoded so that HE operations can be successfully be performed. The first operation will be applying the optimized operation suggested in [49] and [80] which through a Linear Layer will compute the Query (Q), Key (K) and Value (V) tensors. Then the Self-Attention dot-product in HErBERT will be then computed as

$$Z = \frac{Q \cdot K^T}{\sqrt{head\_size \cdot heads}} \cdot V$$

where the scaling factor $head\_size \cdot heads$ is equal to d_model. This layer involves an high number of encrypted-encrypted multiplications, which are the most NB consuming operations, as explained in Chapter 3. The output of the Self-Attention layer will be an encrypted tensor which will be used as input of the following layer.

Normalization layer and residuals

The following layer is the normalization layer. In HErBERT Batch Normalization has been used to normalize the tensor flowing through the model during inference. Even in this case, given that Batch Normalization has its parameters, the parameters are encoded as indicated in the Figure 4.2. Although this may seem just a Batch Normalization, HErBERT also maintained the residual connections. Together with Batch Normalization (but also other Normalization techniques), residual connections are a standard procedure used to help Deep Neural Networks train faster and more accurately. So, the output of Batch Normalization won't just be the normalized Self-Attention but:

$$Y = Z + BatchNorm(Z)$$

it is important to note that HErBERT apply the Batch Normalization over the embedding dimension through a temporary reshaping of the input tensor: this is done in order to make the model faster while preserving the required Normalization.

Feed Forward layers

In BERT architectures, as well as in HErBERT, following the residual connection with the normalization layer there is the first Feed Forward Neural Network layer which is still encoded in its parameters, as any other parameterized layer of HErBERT. In these layers HErBERT, as BERT does, will make use of the computed tensors in Self-Attention to perform a parallelized computation, differently from other RNN architectures, over the various words of the input text given the absence of any recurrent connection. Following the first Feed Forward layer, HErBERT applies as activation function the non-linear Square function. This function has been chosen accordingly to be able to make HErBERT still enough flexible to learn non-linearities in the input data while keeping under control the growth of the Noise. The activation function is not a layer composed of learned parameters and indeed in this case there is nothing in need to be encoded, as the Figure 4.2 suggests. After having performed the activation function, HErBERT maintains the common repetition of the second Feed Forward Neural Network layer and of the second Normalization (using Batch Normalization) with the residual connection, keeping under control the gradient. This module of HErBERT is then consisting of two linear transformation with a Square activation in between:

$$FFN(x) = (xW_1 + b_1)^2 W_2 + b_2$$

Being two different layers, the two FFNNs use different parameters. In HErBERT the hidden dimension hyperparameter (the dimension of the FFNNs) is set to be equal to the chosen `d_model` for the embeddings.

Pooling and classification

HErBERT from this last layer begins the final phase of the classification. HErBERT is using an Average Pooling layer to extract a matrix from the computed tensor of the DNN, representing the average of the extracted features which have been found. Then a Feed Forward Neural Network layer, encoded, has been trained as further pooling function, which is directly followed by the final Feed Forward Neural Network

layer for the final classification reshaping the output as a series of values to be further transformer as probability distribution through the Softmax operation after they have been correctly decrypted. In HErBERT the pooling layer and the classification layer given that they operate on simpler operation were able to be larger in dimension, and they were assigned 20 dimension in output for the Pooling layer and 20 dimension in input respectively for the classifier:

$$pooler = LinearLayer(d\_model, 20)$$

$$classifier = LinearLayer(20, \#classes)$$

4.3 Approximated and encoded Deep Learning processing

As said in the previous section, the proposed HErBERT architecture is general enough in its nature to be used in future works for also other NLP tasks. The Transformer is in its nature an architecture which is able to be applied among different kind of tasks. While there could be the need of the Decoder part of the Transformer for sequence-to-sequence tasks, the proposed architecture will work as a reference to easily add that other approximated needed component. As mentioned in the Introduction, the Transformer in order to work with HE data needs to be approximated in order to have polynomial operations only. This means that all the non-linear functions not already expressed in a polynomial form will have to be in some way approximated in a polynomial form. For example, one of the most common non-linear and non-polynomial activation functions is ReLU. Given that ReLU is non-polynomial, it can't be performed through HE schemes (they can't perform the comparison operation). So ReLU will have to be substituted by another working non-linear and polynomial activation function. In the following Table 4.2 there are shown the various approximation that has been done on the original BERT architecture.

Table 4.2 BERT modules approximations

Original	Approximation
Softmax	NULL or Batch Normalization
Layer Normalization	Batch Normalization
ReLU or GeLU	Square
Max pooling	Average pooling

Let's now analyse in more details the chosen approximations needed to be made:

- Softmax: this is a highly non-linear and non-polynomial function involving exponential functions and divisions, $\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$. Unfortunately, these are two types of operations unable to being performed using HE schemes. Approximating it through polynomials is not very good because we would need very high degree polynomials [81] compromising all the available noise budget. So, the first approximation we thought about to try was Batch Normalization [82]. Given that Softmax scale all values among a certain axes between $[0, 1]$ and make their summation exactly equal to 1, we wanted to let the model maintain at least an approximation of one of these two properties. With Batch Normalization we were able to do that. We also saw that Softmax is not the Self-Attention's best paired function. Completely removing the Softmax operation and not adding any of its approximations the model during training phase was able to reach the same performances or better than the model using Batch Normalization to partially substitute Softmax. The fact that our model being used is smaller than common BERT architectures in clear and without having multiple stacked Transformer's encoders, it is reason why Softmax seems to have less importance in this case and even without it the model will actually perform well.

- Layer Normalization [83]: this module has one problem, which is the computation of the square root of the variance, but also has a more subtle issue. The computed-during-inference mean

and variance are encrypted data, they won't be encoded like the weights of the model. This is because in Layer Normalization, mean and variance are computed during inference for each sample. That's not completely an issue if not for the higher computational load and noise consumption. The issue is what we do with this data. Layer Normalization is computed in this way: $y = \frac{x-mean}{\sqrt{variance+\epsilon}}$. As you can see, this would mean to perform a division between encrypted data, but remember that the only two atomic type of operations available are the sum and multiplication between encrypted data, not the division. So instead of using Layer Normalization we are able to perform Batch Normalization [82] on the tensor received in the corresponding part of the Transformer and learn during training the estimation of the mean and variance of the population of the training dataset, which will be a good approximation of the man and variance of the test dataset population. Furthermore, in order to ease the noise budget degradation we decided to apply Batch Normalization not on the batch column but on the feature columns so that we would consume less noise and time during computation, the Section 7 will show the good accuracy that Batch Normalization on a temporary reshaped tensor achieves.

- ReLU [84]: ReLU is not a polynomial function: it isn't even derivable technically. We can't perform $ReLU = max(0, x)$ because we can't perform the comparison operation or at least not in a feasible time using the current scheme and implementation. So by following other works we are using the interestingly easy and working polynomial activation function $Square = x^2$ which showed in other works bringing good results [85].

- Max pooling: we are not able again to perform comparison operation so we can't compute the maximum among a certain axes of a tensor. The solution is to use the other working and common pooling method, which is to perform the mean. The mean operation is completely HE compatible given that we are performing a summation between encrypted data and dividing them to a plaintext, the number of values: $\frac{1}{n} * \sum_{i=1}^{n} x_i$. The division here is

not a problem because we can easily change it to the multiplication of the inverse of the number of values, which is possible to compute given that it is not an encrypted value.

As briefly explained above, before reaching this final result, we initially investigated other solutions which we thought to be valid. The first solution with regard to the Self-Attention was trying to partially substitute Softmax with another BatchNorm which tried to normalize the values and partially emulating Softmax behaviour of limiting the range of the values in the input tensors.
With regard to the activation function, before reaching to the Square activation, we tried to see whether was possible to run the approximation of the GeLU function homomorphically. Unfortunately, the approximation involves a power with a degree of 3 which made our available noise budget quickly go down to 0 or to near-0 values given the already present multiple encrypted-encrypted type of operations in the module of Self-Attention.
Regarding instead the final classifier, we initially thought of adding an approximation of an activation function like the hyperbolic tangent function. At the beginning we thought of using Taylor series or Chebyshev polynomials which are good for a general purpose approximation but not so good when you have a very strict amount of operations available, given that those approximations were involving polynomials with degrees of 3 or more. An interesting approximation we thought about was from an intuition thanks to a simple numerical analysis of the tensors running inside the model. We saw that the range of values was quite small, and so in this range the hyperbolic tangent can be approximated linearly. While mathematically correct, this approximation did not bring any improvement due to the lack of non-linearities which are giving the model more flexibility to learn more complex patterns.
The final proposed architecture was chosen not only thanks to analysing the feasibility of other solutions but also by finding out the performances of various different solutions and comparing them: we were able to see that the chosen solution was performing quite good or better compared to other solutions we thought about, more on that will

be shown in 7. After having chosen the solution, indicated in the Table 7.2 we investigated, as it will be explained furthermore, the feasibility of an embedding dimension of 4 which improved furthermore our architecture with respect to the usage of Homomorphic Encryption schemes.
Anyway all these approximations of the final model of course will alter the final results in accuracy which will be addressed in the following Sections. We also emphasize that we cannot use an already trained BERT and approximate it, but we had to train an approximated and reduced BERT from scratch in order to obtain the best results from it. This is due to let the model adapt its weights to the various approximations now in place, which are of course different from the original model. The general model will then have the following form:

$$y(x) = \sum_i \sum_j (w_{ij} \cdot \frac{1}{n} \cdot \sum_k ((\sum_l \sum_m w_{lm} \cdot ((\sum_n \sum_o (w_{no} \cdot ((Q \cdot x) \cdot (K \cdot x)) \cdot (V \cdot x) + b_{no}) + b\_norm_1)/\sigma_1)^2) + b\_norm_2)/\sigma_2 + b_{lm}) + b_{ij}) \quad (4.1)$$

In the above Equation 4.1 we can see how our model is now completely free of non-polynomial non-linearities, we just have a set of summations and multiplication operations among tensors and the non-linearity of x^2 is added in order to still give the model the power of being more flexible and to better represent complex data. The division operations while shown are actually just multiplications: $\frac{1}{n}$, $\frac{1}{\sigma_1}$ and $\frac{1}{\sigma_2}$ are just constant values or values obtained through the initial training of the model. The first term is for the average pooling while the other two are for the Batch Normalization. Being just known values of the model itself, which is as we said *encoded*, we are able to perform the division by just encoding not those values at the denominator but by encoding the whole fraction computed in clear just before so that then we will be completely free of division, and we will have the full HErBERT. Also note the presence of the Q (query), K (key) and V (values) tensors for the Self-Attention while the absence of the Softmax operation which was removed as explained in the current Section.
In the HErBERT architecture as previously shown, it has also been

considered the final classifier, which is not technically a part of BERT, but it is just a module used for classification tasks. It is important to denote that we also wanted to let the classification be privacy-preserving, but it is completely possible to just consider a feature-extraction privacy-preserving-deep-learning model while letting the less computational demanding classifier module run on the client in order to consume less noise and of course to make computation slightly shorter. In this way, we would have a Transfer Learning model where a classifier can be trained and used in clear from the features extracted by the deep learning model run in a privacy-preserving setting.

4.4 Dimensioning the embeddings

For achieving a working HErBERT we necessarily needed to scale down a bit the dimensions of the model. One of the most influencing dimension of the model is the size of the embeddings. Without doing this, we would not be able to make computations in a feasible time, and we would not be able to find the feasible parameters needed to handle such large tensors. We introduced in this Section that embeddings of 4, 8, 16 and 32 were initially used while then reducing them to 4 and 8 due to constraints imposed by HE. The first question which comes to mind is that they seem to be quite small embeddings compared to what the original implementation of the Transformer used.
While the analysis of the number of dimensions have not received enough attention [86] and also previous works [87] show the performance for some tasks degrading for higher embeddings dimensions. This justifies the need for study of bounds for dimensions of embeddings. And it also shows that smaller embeddings are not unusable, it is all dependent on the level of generalization and complexity a model need to achieve with respect to a certain dataset and especially how large such dataset is. Coming back to our embeddings dimensions, the choice other just made out of trial and error was also backed by this work of Google [88] where they provided a general formula for dimensioning the embeddings in function of how large is a vocabulary of a certain dataset. The general provided formula is

$embeddingDimension = \sqrt[4]{vocabularySize}$. The results in the next Sections will show that in this work, small embeddings for the HErBERT architecture performed with good results on the various dataset chosen to be analysed. Further details about the dimensioning of the embeddings will be shown in there.

4.5 Parallelizing computations

Homomorphic Encryption makes everything slower in computation. In order to make things faster we decided not to just parallelize over the various sample of a batch and over multiple batches but we decided to also parallelize the computation of a single sample. In the used implementation of SEAL, further presented later in the next Section, this parallelism was missing. This was a problem because for the most common use-case in production, a model would not receive batches of data but mostly single samples and so here it is the first reason of why we added this parallelization. The other main reason is that in this way we can further reduce the inference time of the whole test set in order to measure the performances of the model.

4.6 Encryption parameters

Choosing the parameters Θ is fundamental in order to make possible the inference through the model and to get correct results back. While the q parameter, as already mentioned, is more difficult to choose and for that we just refer to pre-computed values present in SEAL [32] for the other parameters we made a throughout search for the ones which provided best results. In the previous subsection we explored the various approximations and changes to be made in order to make HErBERT working. This also gave suggestion on which parameters put our focus better on. In HErBERT there are in total a larger encrypted-encrypted tensor multiplication for the obtaining the Query, Key and Value tensors from Self-Attention and another encrypted-encrypted tensor multiplication for the Square activation function. Then multiple others encrypted-encoded tensor multiplication for the inference

through the various layers' weights of the model. We already talked about the higher noise consumption for encrypted-encrypted tensor multiplications and this was what limited our choice of parameters. We could not choose, $m \in 1024, 2048, 4096$ but we had to choose a baseline of 8192 in order to do not finish all the noise budget available before ending the computation. This particularly influenced the inference time given that this large parameters while bringing more noise budget, it also has the counter-effect of making computations way slower. particularly difficult and influences the security of the scheme. The p parameter, now that the m parameter had been fixed, was chosen by just exploring a certain range of values in order which one was providing good results in terms of low computational error and noise budget consumption. By doing in this way we saw that the parameter p = 2100000 was a good choice in terms of those two metrics. Regarding security, in this work we considered the sec parameter equals to 128 bits, which is the default value of SEAL. There are also what we could call other hyperparameters which are the integer and fraction bits assigned for the Fractional Encoder, their influence will also be shown in the Section for Experimental Results regarding their effect of lowering the computational error.

Chapter 5

Implementation of the proposed solution

In this Chapter, the HErBERT architecture implementation will be introduced. This implementation is just one of the many possible ones. HErBERT is the Python implementation of a Crypto Transformer. What's new differently from other different Crypto NLP models is the usage for the first time of the BERT architecture, while reduced in dimension. The following implementation has the characteristics of completely working as if it was in clear, all the encrypted operations are hidden from whom will call the model for performing an inference on some data. The implementation has been used to perform predictions and tests on Amazon AWS environments, which shows the applicability in real-world scenarios.

5.1 External libraries and dependencies

For the implementation of HErBERT the following external libraries have been used:

- NumPy [89]: library used for performing advanced scientific computation on the tensors flowing throughout the model during inference.
- multiprocessing [90]: internal dependency of Python for enabling the model of having the capability for parallel computations.

- Pyfhel [35]: Homomorphic Encryption library adding support for the used BFV [21] scheme under SEAL [32].
- PyTorch [91]: Deep Learning Python library used in order to run HErBERT with the correct weights and parameters.

5.2 Structure

HErBERT is implemented with the Python 3 language. It relies on some libraries and dependencies previously listed. The Pyfhel library 2.3.1 [35], Laurent (SAP) and Onen (EUROCOM) licensed under the GNU GPL v3 license for the tasks of HE; Pyfhel is a Python wrapper of the C++ implementation of the Microsoft SEAL library [32] which includes the BFV scheme [21] which has been used in this work. The Numpy library is implementing in Python all the various matrix and tensor operations required to perform inference in Neural Networks and Deep Learning models, NumPy is essentially wrapping fast and optimized C++ code for performing computationally intensive operations such as tensor multiplication. The PyTorch library is required in order to use the trained models in PyTorch and make them run in HErBERT, this library will fundamental in order to run the saved models and extract the corresponding weights which will be then further encoded in order to enable the Homomoprhic Encryption operations to take place accordingly. HErBERT can be divided into sub-packages:

- activations: code containing required activation functions for final predictions.
- attention: code containing the Self-Attention approximated implementation of HErBERT.
- core: fundamental code for making HErBERT work as desired. It contains code for performing cryptographic operations such as encrypting/decrypting and encoding/decoding tensors, the code for the fundamental mathematical operations performed in parallel with the multiprocessing Python dependency through the HE scheme and the base module code for the various HErBERT Deep Learning layers.

- embeddings: code for performing the input transformation and compression into a lower dimensional vector space from the pre-trained weights.
- modules: fundamental code containing the various approximated Deep Learning layers needed for HErBERT.
- utils: code containing the functions for loading accordingly any required data from a certain dataset and the functions for loading the trained PyTorch model which is desired to run. It will load its weights, biases and any parameters present inside the file containing the model.
- tests: code for running tests of the overall implementations and sub-parts of it, fundamental during the implementation for making sure of a correct model implementation.

HErBERT itself is able to handle various types of required layers in the Transformer architecture:

- transformer: Transformer encoder implementation
- feed_forward: Transformer's Feed Forward Neural Network layer implementation
- linear_layer: Linear layer implementation
- pooler: Pooler linear layer for the classification
- batch_norm: Batch Normalization layer implementation for Layer Normalization approximation
- activation_function: Activation function layer implementation for performing approximated non-linear activation functions such as Square function.

Every module in HErBERT has been built following the PyTorch paradigm of having the two methods to be overridden:

- ___init___: this functions builds a Layer object which will be further initialized with correct parameters once loaded.

- ___call___: this function receives in input the input tensor on which a layer will make the corresponding computations on and then returns the correct output.

Other than these two methods, the abstract Module class inside the core package also includes the init_param method which it is used recursively to initialize the entire HErBERT model parameters from the outer container.

All this abstractions are required in order to make the code easier to read, understand and maintain for future and further developments.

5.3 Model loading

The code contained in the utils package can retrieve all the parameters of a PyTorch model regarding a Transformer, with restriction on the layers name using the ones of the official implementation. It will load all the model parameters, encode them accordingly and store them in a data structure which will be used by HErBERT to initialize the encoded Transformer model for HE inference. A Transformer model made in PyTorch to be loaded in HErBERT has to respect the following constraints:

- It must be a Transformer model;
- It must come from a compatible and already-trained approximated and reduced BERT model for HE;
- It must have been saved in PyTorch with the corresponding API using the save() function;
- It must be a model saved with the .pt or .pth extension.

The load_model Python file under the utils package is built in order to load the corresponding model parameters in a functional way, making the code very understandable and more efficient. The general method for parameters loading is the one presented in the Appendix A.

5.4 Model

Given that SEAL [32] is a C++ implementation of the Homomorphic Encryption scheme being used, we needed a Python wrapper in order to make it work with the overall project being implemented using Python. For this purpose, one of the most complete and supported wrapper libraries for SEAL, and especially BFV, is Pyfhel [35]. We chose it given also that it has also implemented in it the Fractional Encoder needed to scale float values into integer representation of BFV as described in previous sections. While Homomorphic Encryption is gaining more and more popularity, there is still a huge lack of support from major Deep Learning frameworks like Pytorch [91], Tensorflow [92] and Keras [93]. This is also due to the counterpart missing support of Homomorphic Encryption schemes implementations for GPU and CUDA [10]. So, given the total absence of support, we had to implement completely from scratch our model and each of its parts and its submodules.

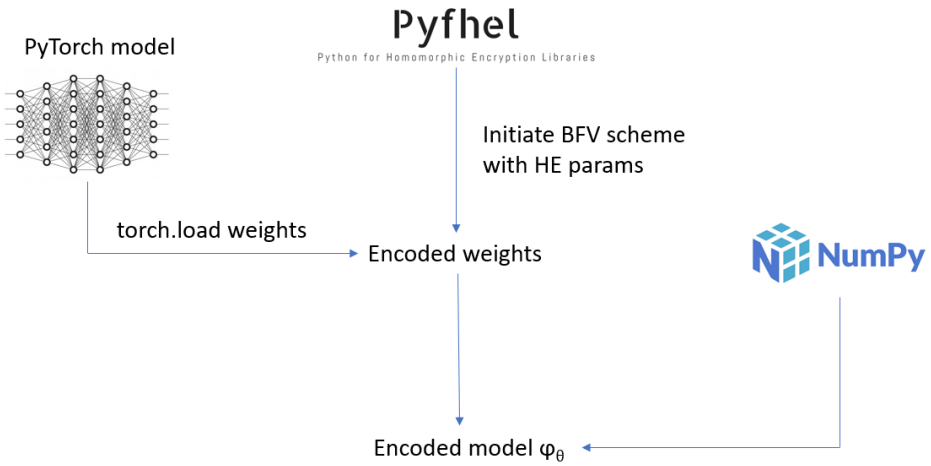


Figure 5.1: HErBERT implementation

Pyfhel was also chosen because it eased the work for us, given that it offers support for using Numpy [89] to perform various computation regarding tensors of homomorphically encrypted values. For the correct implementation[1] we followed the official implementation of the Trans-

[1]Code is available at `https://github.com/comidan/HErBERT`

former architecture provided by Pytorch [80], we completely refactored the various operations from the lowest level using NumPy. As just said we then had to re-implement everything starting from the base modules made of Linear Layer, Batch Normalization, Feed Forward and Pooler. Then of course in the module Attention we re-implemented, also here following the official implementation [80], the Multi Headed Self Attention and the actual Self Attention. These are of course the main parts of the Transformer in order to make it work. To give a glimpse of the whole implementation we show here the code of the MultiHeadedSelfAttention where it is shown how the code completely resembles the original one from PyTorch but it is refactored with NumPy methods and the usage of Homomorphic Encryption, in the Appendix A As you can see everything is completely independent of any PyTorch code or any other DL framework, everything had to be made from scratch. In the core modules instead we implemented the actual way of handling data with the Homomorphic Encryption scheme: initializing, encrypting, decrypting and all the various pre-computation needed operations. The important methods for handling encoding/decoding and encrypting/decrypting are shown in the Appendix A. There is also the operational part where the parallelization code is also implemented, and we will talk about that in the following Chapter 6, where the various processes are continuously created and run. Each process during the computation will then call the actual NumPy primitive for tensor multiplications using completely encrypted data and encoded weights of the model. The overall implementation logic is summarized in the Figure 5.1 where it can be seen that to actually make HErBERT work with Homomorphic Encryption there are a lot of considerations to be done.

5.5 Software testing

Given that the implementation had to be made fully from scratch using NumPy for tensors operations and Pyfhel for handling HE operations, it required a precise implementation to make sure everything worked as expected making various tests on the developed classes and function-

alities during their development was a must. The Transformer module and the Homomoprhic Encryption operations are not directly straightforward to use, so implementing tests in parallel it favored their complete and correct implementation, with respect to the original one in PyTorch, while also debugging them. Three main classes for the tests were made which focused on three different parts of the architecture:

- test_functions.py: testing class for the fundamental operations such as tensor multiplications, activation functions and pooling.
- test_layers.py: testing class for the various layer of the model, from the LinearLayer to MultiHeadedSelfAttention and Normalization classes.
- test_loading.py: testing the encoding of the parameters and the further loading of those encoded parameters into the HErBERT model.

All the various tests have been made using the unittest library provided under the IntelliJ IDE on which PyCharm is based on.

5.6 Model training and testing

The training of the models were performed using PyTorch and a combination of the Google Colab resources and of the local available resources, consisting in a Nvidia GTX 1060 3 GB GPU. The training process has been performed taking in account the following summarized characteristics in Table 5.1. For the weight initialization with Xavier Initialization, we used GlorotNormal to better initialize weights W and letting back propagation algorithm start in advantage position, given that final result of gradient descent is affected by weights initialization.

$$W \sim \mathcal{N}\left(\mu = 0,\ \sigma^2 = \frac{2}{N_{in} + N_{out}}\right)$$

Other than the testing of the actual software implementation, also the testing of the model was mandatory. In order to successfully test the trained model, we used an evaluation set for the various models being

Table 5.1 Training parameters

Training argument	Parameter	Details
loss	Categorical cross entropy	it is a multi-label classification.
epochs	24	Empirically shown to be enough for training these reduced and approximated models.
steps_per_epochs	Depends on the dataset	Using a batch-size of 32.
validation_steps	Depends on the datasets	Using a batch-size of 32.
Weight initialization	Xavier	To have better starting weights, which should speed up the training and let us have a better final result of gradient descent.
optimizer	AdamW	In [94] it is said that it should give better results than Adam which is the de-facto standard.
learning_rate	dynamically scheduled starting from 0.001	Value which allowed the model to learn correctly.
beta_1	0.9	Value suggested in [94].
beta_2	0.999	Value suggested in [94].
Callback	Early Stopping	To stop the model when it reaches convergence, and it acts as weight decay which helps preventing over-fit.
delta	0.001	A small enough value, it is $\approx \frac{1}{100}$ of the final loss of the models.

trained on which we monitored whether the training process was going on successfully. For visualizing the partial results and directly plotting training and validation accuracy/loss epoch after epoch, it has been used Tensorboard [95].
In order to get the best model during the training process we used the Early-Stopping technique which saved the current best model and stopped in case in the following epochs the loss was continuing to get higher and higher without further improvements with respect to the previous local minimal. The further final testing results were instead performed on the pre-built tests set of the used Datasets described in the Chapter 7. The training and testing process are particularly important in this case because we don't have an already pre-trained model, but we must train from scratch a new kind of approximated model, and so we have to monitor in a very peculiar way the progress of these whole processes.

Chapter 6

Efficiency optimizations

6.1 Limitations

Homomorphic Encryption brings with it a lot of limitations and drawbacks in terms of freedom, computational load and time. A multiplication between two tensors being performed under Homomorphic Encryption is way slower than one performed in clear for a variety of reasons:

- CPU-only support: no CUDA or GP-GPU code support is present for the implementation of the HE scheme, this will result in slower computation in general.
- HE cipher scheme: from how HE schemes work, they are in nature slower in performing the same operations that in clear would be way faster.
- HE Fractional Encoder: it enables to manage only one scalar encrypted value per operation, so we can't speed up using a vector representation because we would still have to explore the whole vector of fractional values.
- With large models come large tensors: the larger the model and the corresponding tensors, the way larger the execution time, memory consumption and noise consumption will be.
- Missing parallelism in the used HE scheme implementation: the

HE scheme being used does not offer by itself any kind of parallelism among the various cores of a multiprocessor.

6.2 Parallelizing computations

In order to optimize the overall execution and actually make HErBERT possible, we decided to focus on the two last points. Regarding the largeness of the Transformer architecture, we decided to sensibly reduce the size of its parameters. Parameters with the same or similar size to the one used by BERT [7] or by the original implementation are really too large not just to perform inference in slower time but to actually being able to decrypt the results without letting the noise budget reach the 0 value. So we investigated one depth Transformers with small embedding dimensions of 4, 8, 16 and 32 as we have shown in the previous Chapter. The last point is about the missing parallelism. In order to achieve it we exploited two properties, the first is to actually perform in parallel on multiple cores the inference of various sample in one batch. But in order to also reduce the execution of a single sample among a batch of samples, or even in the case of batch size of 1, we decided to exploit the block properties of matrix multiplications [96]. It is a technique which enables us to perform in parallel multiple sub-matrix multiplication and combine all of them together at the end. Even about that, more will be explained in the corresponding following subsection.

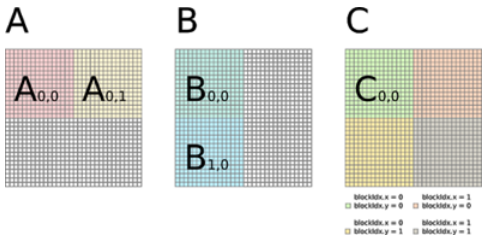


Figure 6.1: Block matrix multiplication

The parallelization technique is based on dividing up in different blocks the two matrices being multiplied, as shown in Figure 6.1 in order to perform in parallel those sub-matrices multiplications. The

level of parallelization is so dependent on the block sizes and so dependent with the size of the two matrices. The above Algorithm 1 shows

Algorithm 1 Parallel Block Matrix Multiplication algorithm

Input: A, B, C, q
Output: C

function parallelBlockMatMul(A, B, C, q)

```
1: Reshape accordingly A,B
2: for i to q − 1 do
3:    for j to q − 1 do
4:       Initialize all elements of C_{i,j} to 0;
5:    end for
6: end for
7: Reshape accordingly C
8: for k = 0 to q − 1 do
9:    p := Process(target_function=matmul, args=(A[i, 0, :, :], B[0,
      j, :, :], C[i, j, :, :]));
10:   Processes_list.add(p);
11:   p.start();
12: end for
13: for p in Processes_list do
14:   p.join();
15: end for
16: return C
```

Input: A, B, C
Output: C

function matmul(A, B, C)

```
17: C := C + A ∗ B;
18: return C
```

the pseudocode implementation of the parallelized through processes block matrix multiplication involving in this example case two matrices A and B of shape (n, n) with a block size of $(n/q, n/q)$ where q must be a divisor n, hence the dependence on the original shape. This procedure will then memorize in C, the output matrix, the multiplication result of the blocks done through different parallel instantiated

processes. The procedure ends with waiting all the processes to join the main caller in order to be sure to return the correct computation of the C output matrix. Now that theoretically we know what to do to make things a little better, we have to face the problem that neither SEAL [32] nor Pyfhel [35] are providing any kind of support for parallelized computation. So we needed to parallelize it on our own on top of Pyfhel in Python. With the library wrapping SEAL in Python there are different factors which creates the problem of not being able to speed up the computation by scaling it to multiple cores:

- Python's GIL [97] for CPython : it's a necessary mutex used mainly because CPython's memory management is not thread-safe and it will block concurrent threads accessing particular resources like CPython code, which is indeed our case. Moreover, according to python documentation and reports no one has managed to get rid of it in a new Python release, unfortunately.

- No native tensor C++ classes in SEAL nor in the Pyfhel wrapper parallelizing code a-priori before GIL.

- BFV being native for integers, while using Fractional Encoder which just keeps track of the scale of fractional values represented in int64, won't be parallelizable through Python BLAS [98] libraries: they are not supporting integer types.

The real imposed limitation between these three factors is the Python Global Interpreter Lock (GIL). The GIL is a mutex variable allowing or disallowing threads to run their code through the CPU. It is a shared mutex across multiple threads for which they are concurring to acquire in order to lock it so that they are authorized to run and to do not wait in the threads' queue. The GIL must be acquired and released accordingly as shown in the Figure 6.2 from [99] where each thread is running only when has the GIL of the process managing those threads.

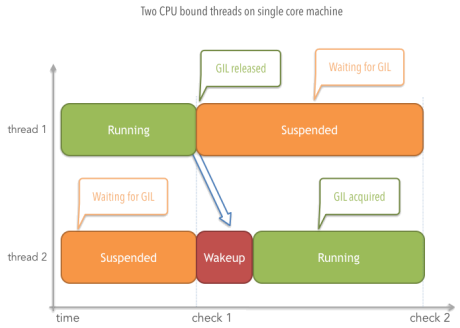


Figure 6.2: Global Interpreter Lock in Python multi-threading

In Python source code [100] the GIL is represented as this shared mutex:

```
static PyThread_type_lock
interpreter_lock = 0; /* This is the GIL */
```

Each thread in order to run will have to run the blocking call of the following primitive in order to run their code:

```
void PyEval_AcquireLock(void)
{
    PyThread_acquire_lock(interpreter_lock, 1);
}
```

And the following for releasing it:

```
void PyEval_ReleaseLock(void)
{
    PyThread_release_lock(interpreter_lock);
}
```

So as it is shown, here we are dealing with a *global* mutex among multiple thread, making it impossible to access the needed bytecode of CPython in parallel from different threads. So this is a serious limitation generated by how Python manages multi-threading. This can be easily bypassed by thinking outside the threads' realm and without limiting us to one process only. As it is usually done in Python for getting real parallelism, we move to multiprocessing where each process won't have the problem to wait for acquiring the GIL because each one will have its own GIL, so no race condition for a single GIL

in order to acquire access to CPython code. So, in order to solve these problems we just have to bypass the problem of threads for the GIL and performing dot product of tensors using the block matrix multiplication algorithm shown before. From multiple threads we have to move to multiple processes, in this way we will bypass the limitation imposed by GIL. we will be able to scale with multiple CPU cores, but the memory won't be shared anymore. There is the need of concurrent structures which are able to work across the different virtual address spaces of different stacks in memory for the processes. Python in the multiprocessing module provides such data structure like the Queues.

6.3 Implementation

All the presented reasoning can now be shown as actually implemented Python code enabling to have parallelism in Homomorphic Encryption tensors operations through Pyfhel and directly by using Python.

```
def rebuild_matrix(arr):
    rows, cols, n, m = arr.shape
    arr = arr.swapaxes(1, 2).reshape(rows*n, cols*m)
    for i in range(arr.shape[0]):
        for j in range(arr.shape[1]):
            arr[i][j]._pyfhel = cm.HE
    return arr


def block_matrix(arr, rows, cols):
    h, w = arr.shape
    n, m = h // rows, w // cols
    return arr.reshape(rows, n, cols, m).swapaxes(1, 2)

def dot_product(a, b, out, q, i, j):
    out[:] = np.matmul(a, b)
    q.put([out[:], i, j])
    q.task_done()

def __pardot(a, b, nblocks, mblocks, func=dot_product):
    n_jobs = nblocks * mblocks
```

```
out = np.empty((a.shape[0], b.shape[1]), dtype=PyCtxt)
out_block_matrix = block_matrix(out, nblocks, mblocks)
a_block_matrix = block_matrix(a, nblocks, 1)
b_block_matrix = block_matrix(b, 1, mblocks)

processes = []
q = JoinableQueue()
for i in range(nblocks):
    for j in range(mblocks):
        p = Process(target=func,
                    args=(a_block_matrix[i, 0, :, :],
                          b_block_matrix[0, j, :, :],
                          out_block_matrix[i, j, :, :], q, i, j))
        p.start()
        processes.append(p)
result = []
for _ in range(n_jobs):
    result.append(q.get(block=True))

for p in processes:
    p.join()
    p.terminate()
for data in result:
    out_block_matrix[data[1], data[2], :, :][:] = data[0]
return np.expand_dims(rebuild_matrix(out_block_matrix), axis=0)
```

From the above source code, we can see the implementation of the parallelized HE matrix multiplication operation directly in Python on the Pyfhel HE library. The `blockshaped` method is required to adjust as seen in the pseudo-code the 2-dimensional tensors being multiplied in blocks that can be multiplied in parallel through the do_dot method which is going to save partial results in the `JoinableQueue`, the shared-memory data structure. After having performed the multiplication, the `unblockshaped` is going to reshape the output matrix accordingly to a 2-dimensional tensor and it is going to re-add the reference to the HE Pyfhel object handling the HE operations on the various elements present in the Numpy tensor. This is needed because when handling the HE values in a different process than the original one, the reference

to the HE object is lost simply because in their process' assigned memory that object does not exist anymore. In order to be able to perform further operations, monitoring the noise budget values and any other operation on encrypted values, there is the need to re-associate the HE Pyfhel object reference to each one of the elements present in the Numpy tensor. While this may seem not really efficient as a solution, it is the only way given this implementation restriction of Pyfhel which was not thought to handle multiprocessing for its operations through Numpy. While it has been talked about parallelizing tensor multiplications and matrix dot-products, the parallelization technique had also been applied to other layers of the model where it has been able to reduce the computational time. Parallelization has been applied also for the Square activation function and for the Batch Normalization multiplication to the inverse of the square root of the variance σ. The particular thing about parallelizing these other parts of the model was that should have changed was just target function in the multiple processes, to be adapted depending on the operation to be performed.

Chapter 7

Experimental results

In this Section, we show the evaluation of the accuracy and the computation load of the HE Deep Learning provided in various configurations.

7.1 Description of HErBERT settings

The HErBERT model being used have embeddings dimension of 4 and 8. Furthermore, for the heavy computational constraints imposed by the use of HE, HErBERT will employ a 1-layered Transformer and one attention head. Given that SEAL [32], one of the most important HE frameworks, is a C++ implementation of the BFV scheme, a Python wrapper is needed. One of the most complete and supported Python wrappers for SEAL is Pyfhel [35]. While HE is gaining more and more popularity, for now no major DL frameworks has support for this technology. Additionally, HE libraries supporting the use of GPU are at the very early stage. For these reasons, HErBERT has been implemented from scratch using NumPy [89]. However, the reference implementation of the various layers is PyTorch [80].

7.2 Datasets

Two datasets have been used to test our model:

- Yelp Polarity Review [16] is a dataset provided by Yelp containing reviews. It is extracted from the Yelp Dataset Challenge 2015

data. The Yelp reviews polarity dataset is constructed by considering stars 1 and 2 negative, and 3 and 4 positive. It contains a set of 560000 samples for training, and 38000 for testing.

- Yahoo! Answers [16] is a dataset provided by the now closing Yahoo! Answer platform. The dataset is the Yahoo! Answers corpus as of 10/25/2007. It includes all the questions and their corresponding answers. The Yahoo! Answers topic classification dataset is constructed using 10 largest main categories equally distributed. Therefore, with 10 different classes the total number of training samples is 1,400,000 and testing samples 60,000 in this dataset. From all the answers and other meta-information, only the best answer content and the main category information were kept.

A further analysis of these two datasets about their properties and relevant information which were important for the choice of embedding dimensions can be found in these works [101] [102] where other than just providing the average review length in words, it was also provided the size of the opinion vocabulary of the Yelp Polarity Review dataset [102]: the size of the set of words which actually was relevant for the Text Classification task.

7.3 Parallelization

In this section we show the interesting results provided with parallelizing the dot product on a single encrypted-encoded multiplication bypassing the GIL on two matrices with different square shapes (4, 8, 16 and 32). The parameters being used are 8192 for m and 2100000 for p. As we can see from the Table 7.1, we are always getting a useful improvement from parallelizing the dot product. These results have been sampled from the execution on a c5.18xlarge AWS istance using 72 cores (vCPU) and 144 GB of RAM for parallelizing. Note that given the availability of 72 cores, the maximum q used to divide up in blocks has been 8 so that no process of the 64 created was going to be put in the waiting queue of the CPU. This means that higher number of cores can furthermore increase performances.

Table 7.1 HE multiplication timings comparison using m = 8192 and p = 2100000

Shape	Serial	Parallel
(4,4)	0.235 s	0.058 s
(8, 8)	1.868 s	0.359 s
(16, 16)	14.939 s	2.006 s
(32, 32)	119.503 s	14.652 s

7.4 Experimentation details

The following subsection for the Results will show the empirical results organized for various different metrics, which we considered important to analyse for both Homomorphic Encryption and Deep Learning with the type of task at hand. The following metrics are being considered:

- Model accuracy: we are running a model for Sentiment Analysis and Text Classification, so accuracy over the test set is the right metric for evaluating the goodness of the approximated and encoded model.

- Inference time: the running time for making both a single review inference and an overall test set inference is really important. That's because Homomorphic Encryption slows execution time of the various operations more than just a bit. We are no more talking about inference is milliseconds but in terms of seconds, both for the fact of running encrypted operations where modular arithmetic is impacting a lot and for not having available the possibility of moving to very high performing GPUs.

- Main memory usage: memory occupation here is very important. Using the large parameters we chose, the ring polynomials will be very large, and so they will take a lot of memory: we are talking of already a few GBs occupied for making one review inference in the best case.

- Noise budget and its consumption: the noise budget will be fun-

Table 7.2 Transformer approximation accuracy in clear on Yelp Polarity Review dataset, in the columns the numbers are referring to (embedding dimension, maximum utterance length)

Architecture	4, 32	4,64	8, 32	8, 64
Approximated Transformer + Self Attention with BatchNorm	0,834	0,881	0,834	0,881
Approximated Transformer (no lin tanh) + Self Attention with BatchNorm	0,836	0,882	0,839	0,884
Solution: Approximated Transformer (no lin tanh) + Self Attention without BatchNorm 1 head	0,835	0,88	0,839	0,882
Approximated Transformer + Self Attention with BatchNorm 1 head	0,834	0,879	0,838	0,884
Original reduced Transformer	0,761	0,781	0,835	0,877

damental to track, monitor and record throughout a review inference so that we can better understand how to better tune parameters but also how much more we can potentially push the model with larger parameters.

7.5 Results

On testing HErBERT we got interesting results. Before exploring the main results we just want to show the results involving also other approximations of the Transformer we took in consideration as stated in 4, so here with the Table 7.2 we show these results. From this experimental data, we inferred that:

- Max utterance length has more influence in having better results for small dataset.

- Small changes in small embeddings do not bring much change [75] for small dataset in understanding and capturing words relationships.

- The architecture without softmax or any substitutes performs very well.

- The various types of reduced and approximated Transformer models behave like the original, statistically speaking.

- Reducing to 1 attention head from 2 attention heads performed well.

After having analysed the result we chose the model indicated as the solution in the Table 7.2 and from further on we will consider only that model architecture. The reason behind this choice was not just regarding the obtained accuracy, but with a necessary eye to the constraints imposed by HE. The chosen solution is performing better or practically equal to other models while being essentially smaller in terms of number of operations to be performed (1 attention head and no Batch Normalization in the Self-Attention module) so we are able to save further noise budget.
it is also interesting to see in reference to what has been analysed in Section 4 that in the original model where there is the Softmax function inside Self-Attention we have a drop in accuracy when reducing furthermore the embeddings dimension.

On both datasets, the model performed well while dropping a lot of the available Noise Budget. The tested model were trained with the corresponding training sets using 12 epochs and AdamW as optimization algorithm with a weight decay of 0.01.
In Table 7.3 we summarized the results and compared them with BERT-base, the approximated and reduced BERT and Logistic Regression. Even with Logistic Regression, we used to perform training and testing embeddings using Word2Vec in order to make a valid comparison on similar inputs given the size magnitude of our HE model in terms of parameters and embedding sizes. In all the following results,

Table 7.3 Results accuracy on Yelp Polarity Review test set using HErBERT (4, 32)

Model	Accuracy
BERT	92%
Approximated and reduced BERT	83.4%
HErBERT	80.92%
Logistic regression	63.4%

the word embedding have been trained from scratch. While using already pre-trained word embeddings such as Glove [75], Word2Vec [74] and furthermore could have helped in achieving better results, we did not pursue this solution given that the smallest dimensional sized pre-trained word embedding is Glove-50 with 50 dimensions, and it is way larger than what we are able to compute without ending up our noise budget. There is a really interesting work [103] which is successfully reducing pre-trained embeddings dimension up to 50% their original dimension. While reducing Glove-50 to 25 dimension still wasn't a solution, it has been tried to reduce the dimensions further more to 4 using the same proposed method consisting of successive application of PCA. Unfortunately, after having reduced the embedding dimension and loading them in the model for training, the model did not perform well due to the high compression performed. So, it has been chosen to optimize the results by sticking to training from scratch even the embedding layer. In this first Table 7.3 results in terms of accuracy on the test set are shown for the Yelp Polarity Review data set using HErBERT with a word embedding dimension of 4 and the maximum review length as 32. In this other second Table 7.4 results in terms of accuracy on the same test set using instead HErBERT with a word embedding dimension of 4 and the maximum review length as 64. As we clearly see, there is an increment in accuracy for HErBERT. In this other third Table 7.5 results in terms of accuracy on the test set are shown for the Yahoo! Answers data set using HErBERT with a word embedding dimension of 8 and the maximum review length as

Table 7.4 Results accuracy on Yelp Polarity Review test set using HErBERT (4, 64)

Model	Accuracy
BERT	92%
Approximated and reduced BERT	88%
HErBERT	83.04%
Logistic regression	63.4%

Table 7.5 Results accuracy on Yahoo! Answers test set using HErBERT (8, 32)

Model	Accuracy
BERT	77.62%
FastText [76]	72.3%
Approximated and reduced BERT	60.9%
HErBERT	59.26%
Seq2CNN [104]	55.39%
Logistic regression	22.7%

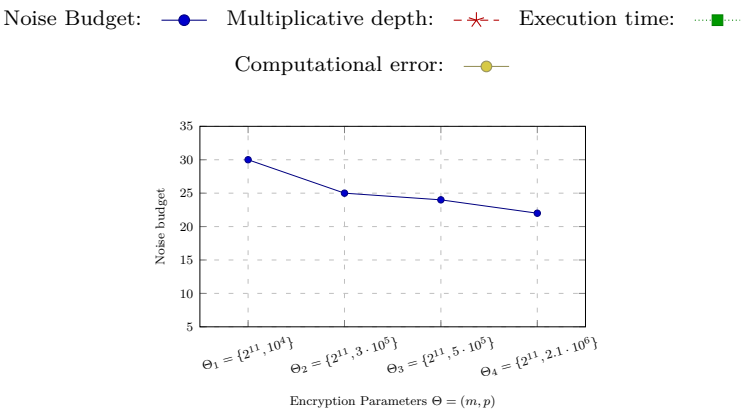
32. Given the presence of a more difficult task (multi-class instead of binary) we decided to provide better results moving to 8 dimensional word embeddings but staying unfortunately as will be shown later very close to 0 noise budget available. Lastly, in Table 7.6 results in terms of accuracy are shown for the same test set using HErBERT with a word embedding dimension of 4 and the maximum review length as 32. It was interesting to show what are the implications of reducing furthermore the size of the model with respect to the required approximations and the loss of accuracy during HE inference, while keeping the right maximum word number constant (given the data set being used). In this other Table 7.8 we summarize instead the noise budget consumption sampled during the inference of a sample. It is interesting to see how dependent is its consumption on how large the tensors

Table 7.6 Results accuracy on Yahoo! Answers test set using HErBERT (4, 32)

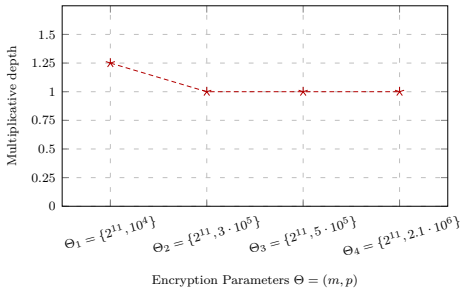
Model	Accuracy
Original BERT	77.62%
FastText [76]	72.3%
Approximated and reduced BERT	57.4%
HErBERT	55.42%
Seq2CNN [104]	55.39%
Logistic regression	22.7%

involved in the operations are, but also how there is a slightly more consumption of noise in parallel than there is in serial execution, while the output remains exactly the same.

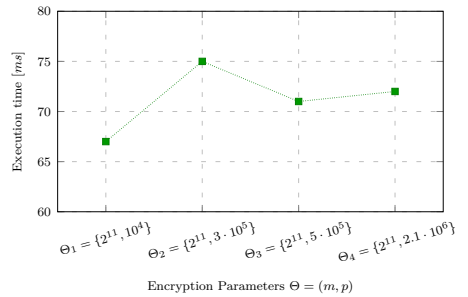
The HE parameters being used here are again as in the previous tests $\Theta = (p, m, s) = (2100000, 8192, 128)$ which were found from a parameter search made on various different criteria as it is shown in the Figures 7.1 7.2 7.3, where a search of parameters for a set of encrypted-encoded multiplications of square matrices were made ranging between a set of p parameters $(10^4, 3{\cdot}10^5, 5{\cdot}10^5, 2.1{\cdot}10^6)$ and the m parameter switching between 2048, 4096 and 8192. In these Figures representing a sample of the benchmarks made for the HE parameters search, it is possible to directly see the effect of those very parameters. While it may seem that reducing the plaintext modulus will help in achieving more NB and so a better HE computation, the other plots will instead prove that a lower plaintext modulus is as good as an higher one only when you are performing very few operations. Keeping the p parameter lower, it will be possible to achieve low errors only with one or more multiplications. The Figure 7.1d shows that for $p = 10^4$ the computational error after just five multiplication of small square matrices, it will be quite high. The successive plots are showing the same benchmarks on the other polynomial modulus degree parameters 4096 and 8192. It is very interesting to see that for performing correctly multiple multipli-



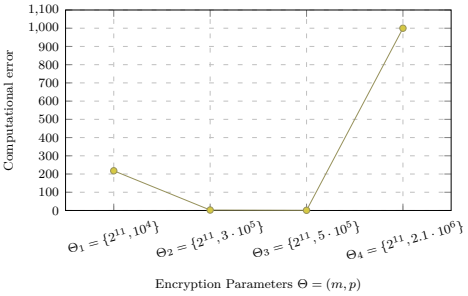
(a) Noise budget using the indicated HE parameters Θ_i



(b) Multiplicative depth using the indicated HE parameters Θ_i

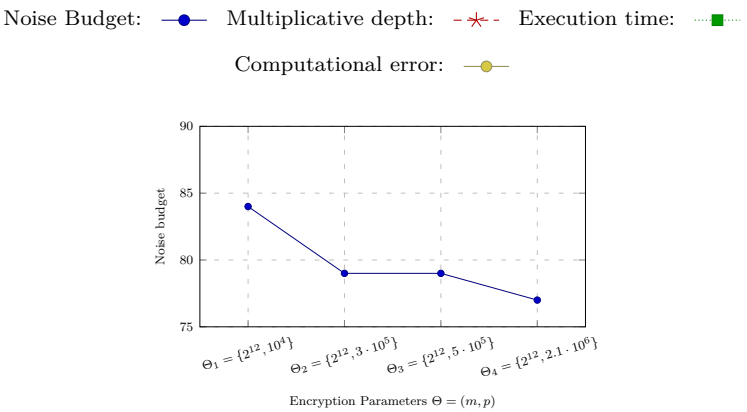


(c) Execution time using the indicated HE parameters Θ_i

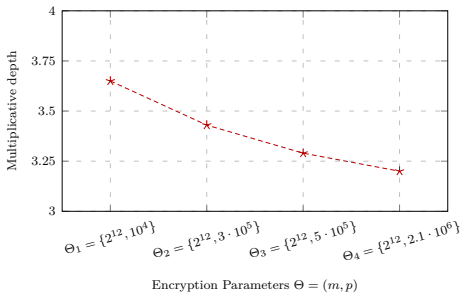


(d) Computational error using the indicated HE parameters Θ_i

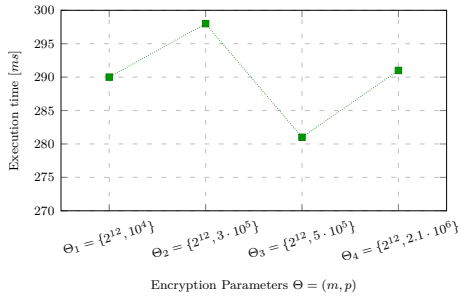
Figure 7.1: HE parameters benchmarks after five Encrypted-Encoded multiplications between 4-dimensional square matrices using the indicated HE parameters $\Theta_i = (2^{11}, p)$



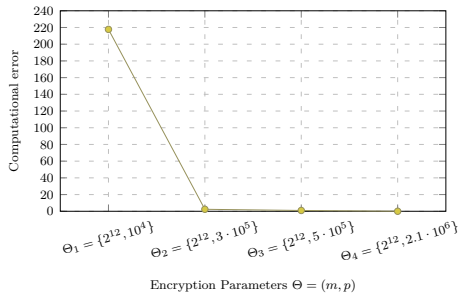
(a) Noise budget using the indicated HE parameters Θ_i



(b) Multiplicative depth using the indicated HE parameters Θ_i

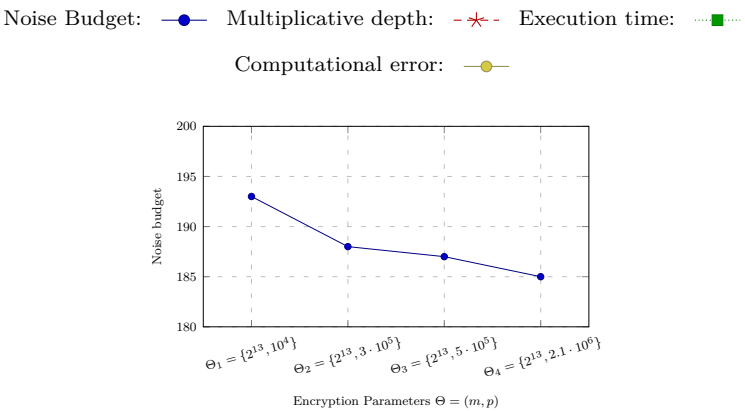


(c) Execution time using the indicated HE parameters Θ_i

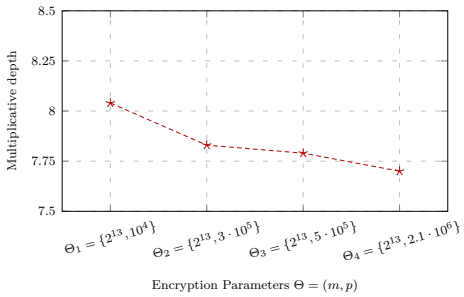


(d) Computational error using the indicated HE parameters Θ_i

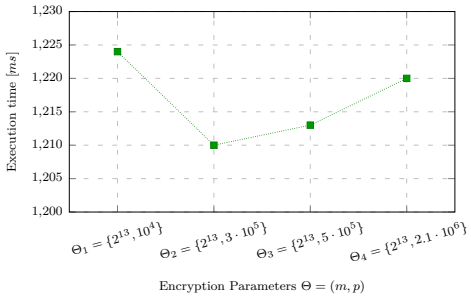
Figure 7.2: HE parameters benchmarks after five Encrypted-Encoded multiplications between 4-dimensional square matrices using the indicated HE parameters $\Theta_i = (2^{12}, p)$



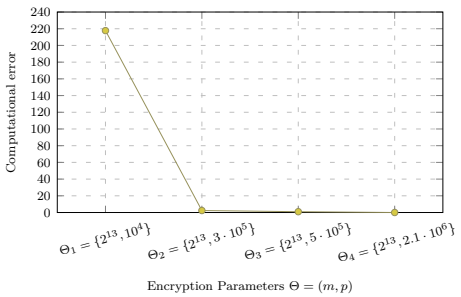
(a) Noise budget using the indicated HE parameters Θ_i



(b) Multiplicative depth using the indicated HE parameters Θ_i



(c) Execution time using the indicated HE parameters Θ_i



(d) Computational error using the indicated HE parameters Θ_i

Figure 7.3: HE parameters benchmarks after five Encrypted-Encoded multiplications between 4-dimensional square matrices using the indicated HE parameters $\Theta_i = (2^{13}, p)$

Table 7.7 Noise budget consumption over using m = 2^{14}

Model (embeddings)	Noise Budget reduction
HErBERT (4)	402 → 234

cations it is important to have an higher p parameter: while the noise budget available after performing them will be slightly lower as seen in Figures 7.1a 7.2a 7.3a, which is normal because it will consume more noise budget a higher p parameter, the computational error is almost zero and so it fits perfectly for HErBERT and other applications in DL and ML where usually lower-than 1 numbers are being treated during inference. Experimental data can be seen in Figures 7.1d 7.2d 7.3d it is clear from 7.8 that larger models are at risk of not being able to be computed given how close HErBERT is to 0 noise budget available: using $m = 2^{13}$ was a choice to balance feasibility of the HE computation in HErBERT while minimizing the execution time. Increasing m to 2^{14} will give way much more noise budget at the cost of having way more computational load, as can be seen from Table 7.7 the available noise budget after the end of the inference is very high compared to the chosen m parameter for HErBERT. Interestingly, there is a slightly more consumption of NB when the computation is executed in parallel w.r.t to the same operation executed with no parallelization, even though the output remains exactly the same. A possible explanation is the fact that the HE library currently used [35] does not support parallelism, the reason behind the efficiency optimizations, and when moving computation to multiple different processes (which created different isolated stacks in the RAM) the various encrypted elements lost the reference to the main Pyfhel object handling the operation through NumPy. This reference needed to be restored afterwards the parallel computation for continuing the inference. This process may be the reason behind a slightly more NB consumption. it is also important to denote that differently from other works, HErBERT had to use such a high parameter as polynomial modulus degree $m = 8192$, given the type of operations which are present. Self-Attention together with the Square Activation function are the most expensive operations being

Table 7.8 Noise budget consumption over a single review of 32 words

Model (embeddings)	Test set	Serial	Parallel
HErBERT (4)	Yelp Polarity Review	182 → 22	182 → 17
HErBERT (8)	Yahoo! Answers	182 → 10	182 → 7

Table 7.9 Comparison of inference output between various HErBERT executions on a 32 word sample using the same Θ parameters

Modality	Time	Classification results
HErBERT serial	40 s	[1.29405295, -1.30774493]
HErBERT parallel	15 s	[1.29405295, -1.30774493]
HErBERT serial in clear	0.01 s	[1.2215196, -1.2390065]

performed given that both operands are encrypted, and not just one with the other encoded. So much expensive that the huge drop of the noise budget is mostly related to these three operations and without a baseline of 8192 as polynomial modulus degree we would not be able to perform such computations and indeed as shown in the Tables 7.3, 7.4, 7.5 and 7.6 and in the following Table 7.9 the parameter m chosen is actually the most correct.

From the above Table 7.9 we can clearly how close the results are, we have a 0.96% difference between the output in clear and the output from encrypted data: which is clearly well reflected from the good results obtained in the overall accuracy on the datasets. Also, we can clearly notice the computational overhead present during the usage of Homomorphic Encryption scheme with respect to an in-clear execution.

The good results were also made better thanks to the usage of the fractional bits of the Fractional Encoder, explained in the previous sections. In the Table 7.10 we experimented on the influence of the fractional bits on the final approximation error of the performed op-

Table 7.10 Fractional Encoder bits and their influence using the same Θ parameters

Fractional bits	Maximum matrix multiplication error
int: 64 bits, frac: 32 bits	4.381428286137634e-05
int: 16 bits, frac: 80 bits	5.829292604175862e-11

Table 7.11 Inference time of a single review of 32 words

Model (embeddings)	Test set	Serial	Parallel
HErBERT (4)	Yelp Polarity Review	40 s	15 s
HErBERT (8)	Yahoo! Answers	110 s	34 s

eration when changing their values. In this example we change the values from the standard ones to a set where we give more importance to the fractional part of the number, given that in Machine Learning and Deep Learning we usually work with relatively small numbers. As we clearly see, there is an improvement in the generated approximated error after the computation is performed, while keeping all the other HE parameters constant.

We also sampled and summarized in Table 7.11 the inference time of a sample in order to give a glimpse of its usefulness during inference and for further use in productions contexts. As it is clear the introduced parallelism is further reducing the execution of a single sample, which unfortunately before was not possible to do. Now we can parallelize not just at multiple samples level but also at one sample level using Pyfhel, which is good to further push the usage of Homomorphic Encryption in Deep Learning and of privacy-preserving Machine Learning. it is reminded as we said in the previous Optimizations Section that the parallelism introduced at single sample level is strictly dependent on the shapes of the tensor and given that they are not square shapes the improvement introduced is not as high as the one showed in the

Table 7.12 Memory peak consumption of a single review of 32 words

Model (embeddings)	Test set	Serial	Parallel
HErBERT (4)	Yelp Polarity Review	3.01 GB	3.51 GB
HErBERT (8)	Yahoo! Answers	4.55 GB	6.64 GB

Table 7.1. In the last Table 7.12 there are summarized the results of the sampling of the peak RAM consumption during inference of the HE-model. It is interesting, and it is right, to see a slightly higher consumption when parallelization is being performed. It is expected given the overhead introduced in creating multiple processes and to manage their results with proper safe data structures. Of course by increasing how large the model is, the larger the memory consumption will be as we can see in the Table 7.12 when moving to 8 sized embeddings from 4 sized embeddings.

Chapter 8

Conclusions

In this Chapter conclusions regarding the proposed architecture are drawn (Section 8.1), and some ideas for future works are presented (Section 8.2).

8.1 Conclusions

The aim of this work was to introduce HErBERT, a Deep Learning model for Natural Language Processing based on BERT able to work on encrypted data, using Homomorphic Encryption. The design process of HErBERT, which took into account the hard constraints imposed by the use of HE, has been carefully detailed and a highly-optimized open-source Python implementation is released to the scientific community. A wide experimental campaign, shows its performances on two different test set among the Sentiment Analysis and Sequence Classification tasks, demonstrating the value of the proposed solution. It has been seen that to get a working BERT under an HE scheme we needed to come to various compromises, especially regarding the model dimension, which is what HErBERT differs more with respect to BERT model. This means that, currently, running a full or even slightly smaller BERT model or Transformer model is not feasible with Homomorphic Encryption for practical use, given the constraints imposed by the noise budget on the very much larger tensors dimensions which would have been used. While the HE schemes will further improve in the years, enabling new achievement and making the goal of

running larger BERT models closer and closer, the application of HE to DL tasks is gaining traction in the academic field, with industrial solutions capable of offering privacy-preserving DLaaS expected to be deployed in the next years.

8.2 Future Works

Future works will consider improving further more performances of reduced Transformers in order to get further good results when made privacy-preserving and the use of different HE schemes capable of performing also the operations of division and comparison, which could succeed in moving also the embedding parts of the architecture on the server. The very important next step is also moving not just to a privacy-preserving inference of a trained model, but making one step forward: privacy-preserving training. This is really important because it would make it possible to preserve the privacy of the data used during training while still being able to achieve a working trained model. This research will involve the refactoring of the back propagation algorithms behind Neural Networks in order to be compatible with Homomorphic Encryption computations.
Also, future works will consider the automatic configuration of Homomorphic Encryption parameters through AutoML. As seen in the previous Chapters, wrong choices of encryption parameters can lead to really poor results in terms of precision, accuracy and available Noise Budget for further computations. Computational times are also heavily affected together with memory occupation. An algorithm capable to automatically find the best possible parameters in a quite reasonable amount of time may help to mitigate this important issue, and making HE usable by more and more users, lowering the learning barrier currently required even in the research field. Another field on which it is important to focus on is the application of Homomorphic Encryption to Internet-of-Things as client end-devices, where the concerns around Privacy have to be addressed even there. These devices are characterized by having constraints on power consumption and memory, which go against the constraint imposed by Homomorphic Encryption. This

is an important research area to address, which will involve the trade-off between the required HE parameters of a certain scheme and the constraints imposed by the Internet-of-Things hardware.

Bibliography

[1] Microsoft Research. Intro to homomorphic encryption. `https://www.youtube.com/watch?v=SEBdYXxijSo&ab_channel=MicrosoftResearch`, 2019.

[2] 2018 reform of eu data protection rules.

[3] James E. Smith and Ravi Nair. *Virtual Machines Versatile Platforms for Systems and Processes*. Elsevier, 2005.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Michael L. Scott and Larry L. Peterson, editors, *SOSP*, pages 164–177. ACM, 2003.

[5] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and S. S. Iyengar. A survey on deep learning: Algorithms, techniques, and applications. *ACM Computing Surveys*, 51(5):92, 2018.

[6] Thomas Erl, Ricardo Puttini, and Zaigham Mahmood. *Cloud Computing: Concepts, Technology & Architecture*. Pearson, 2013.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina N. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2018.

[8] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys*, 51(4):79, 2018.

[9] Flavio Bergamaschi and Eli Dow. Homomorphic encryption comes to linux on ibm z. Technical report, IBM Research, July 2020.

[10] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[11] Sai Sri Sathya, Praneeth Vepakomma, Ramesh Raskar, Ranjan Ramachandra, and Santanu Bhattacharya. A review of homomorphic encryption libraries for secure computation. *arXiv preprint arXiv:1812.02428*, 2018.

[12] Mark D Young. National insecurity: The impacts of illegal disclosures of classified information. *ISJLP*, 10:367, 2014.

[13] Harsh Kupwade Patil and Ravi Seshadri. Big data security and privacy issues in healthcare. In *2014 IEEE International Congress on Big Data*, pages 762–765, 2014.

[14] Abhishek Mahalle, Jianming Yong, Xiaohui Tao, and Jun Shen. Data privacy and system security for banking and financial services industry based on cloud computing infrastructure. In *2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design ((CSCWD))*, pages 407–413, 2018.

[15] Catherine Heeney, Naomi Hawkins, Jantina de Vries, Paula Boddington, and Jane Kaye. Assessing the privacy risks of data sharing in genomics. *Public health genomics*, 14(1):17–25, 2011.

[16] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level Convolutional Networks for Text Classification . *arXiv:1509.01626 [cs]*, September 2015.

[17] R L Rivest, L Adleman, and M L Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978.

[18] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of The ACM*, 21(2):120–126, 1978.

[19] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[20] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *EUROCRYPT'10 Proceedings of the 29th Annual international conference on Theory and Applications of Cryptographic Techniques*, pages 24–43, 2010.

[21] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Leveled) fully homomorphic encryption without bootstrapping. *innovations in theoretical computer science*, 6(3):13, 2014.

[22] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[23] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437, 2017.

[24] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation, 2017.

[25] HElib. Helib. `https://github.com/homenc/HElib`, 2011.

[26] Shai Halevi and Victor Shoup. Algorithms in helib. Cryptology ePrint Archive, Report 2014/106, 2014. `https://ia.cr/2014/106`.

[27] Shai Halevi and Victor Shoup. Design and implementation of helib: a homomorphic encryption library. Cryptology ePrint Archive, Report 2020/1481, 2020. `https://ia.cr/2020/1481`.

[28] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[29] Specification for the advanced encryption standard (aes). Federal Information Processing Standards Publication 197, 2001.

[30] Craig Gentry, Shai Halevi, and Nigel Smart. Homomorphic evaluation of the aes circuit. *Cryptology EPrint Archive*, 7417:850–867, 08 2012.

[31] Léo Ducas and Daniele Micciancio. Fhew: bootstrapping homomorphic encryption in less than a second. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 617–640. Springer, 2015.

[32] Hao Chen, Kim Laine, and Rachel Player. Simple encrypted arithmetic library - seal v2.1. In *International Conference on Financial Cryptography and Data Security*, pages 3–18, 2017.

[33] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[34] New Jersey Institute of Technology, Duality Technologies, Raytheon BBN Technologies, MIT, University of California, San Diego. Palisade. `https://palisade-crypto.org/`, 2017.

[35] M. O. Alberto Ibarrondo, Laurent Gomez. Pyfhel: Python for homomorphic encryption libraries. `https://github.com/ibarrond/Pyfhel`, 2018.

[36] Wei Dai and Berk Sunar. cuhe: A homomorphic encryption accelerator library. In Enes Pasalic and Lars R. Knudsen, editors, *BalkanCryptSec*, volume 9540 of *Lecture Notes in Computer Science*, pages 169–186. Springer, 2015.

[37] Ayoub Benaissa, Bilal Retiat, Bogdan Cebere, and Alaa Eddine Belfedhal. Tenseal: A library for encrypted tensor operations using homomorphic encryption, 2021.

[38] TenSEAL. Tenseal. `https://github.com/OpenMined/TenSEAL/pull/285`, 2021.

[39] Simone Disabato, Alessandro Falcetta, Alessio Mongelluzzo, and Manuel Roveri. A privacy-preserving distributed architecture for deep-learning-as-a-service. *arXiv preprint arXiv:2003.13541*, 2020.

[40] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N. Wright. Privacy-preserving machine learning as a service. *Proc. Priv. Enhancing Technol.*, 2018(3):123–142, 2018.

[41] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, 2018.

[42] Robert Podschwadt and Daniel Takabi. Classification of encrypted word embeddings using recurrent neural networks. *PrivateNLP@WSDM*, pages 27–31, 2020.

[43] Maya Bakshi. Cryptornn - privacy-preserving recurrent neural networks using homomorphic encryption. *International Symposium on Cyber Security Cryptography and Machine Learning*, pages 245–253, 2020.

[44] Bo Feng, Qian Lou, Lei Jiang, and Geoffrey C. Fox. Cryptogru: Low latency privacy-preserving text analysis with gru. *arXiv preprint arXiv:2010.11796*, 2020.

[45] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[46] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. Cryptodl: Deep neural networks over encrypted data. *arXiv preprint arXiv:1711.05189*, 2017.

[47] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. ngraph-he2: A high-throughput framework for neural network inference on encrypted data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, volume 2019, page 947, 2019.

[48] Intel. Intel ngraph deep learning compiler. `https://www.intel.com/content/www/us/en/artificial-intelligence/ngraph.html`, 2019.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, volume 30, pages 5998–6008, 2017.

[50] Priyam Basu, Tiasa Singha Roy, Rakshit Naidu, Zumrut Muftuoglu, Sahib Singh, and Fatemehsadat Mireshghallah. Benchmarking differential privacy and federated learning for bert models, 2021.

[51] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.

[52] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM*, 60(6):43, 2013.

[53] J. Liesen and V. Mehrmann. *Linear Algebra*. Springer Undergraduate Mathematics Series. Springer International Publishing, 2015.

[54] Daniele Micciancio. The shortest vector in a lattice is hard to approximate to within some constant. *SIAM Journal on Computing*, 30(6):2008–2035, 2001.

[55] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Manual for using homomorphic encryption for bioinformatics. *Proceedings of the IEEE*, 105(3):552–567, 2017.

[56] Khalid El Makkaoui, Abdellah Ezzati, and Abderrahim Beni Hssane. Challenges of using homomorphic encryption to secure cloud computing. In *2015 International Conference on Cloud Technologies and Applications (CloudTech)*, pages 1–7, 2015.

[57] Kristin E. Lauter, Michael Naehrig, and Vinod Vaikuntanathan. Can homomorphic encryption be practical. *IACR Cryptology ePrint Archive*, 2011:405, 2011.

[58] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[59] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. Efficient homomorphic comparison methods with optimal complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 221–256, 2020.

[60] David Cousins, Lloyd Greenwald, Yuriy Polyakov, Kurt Rohloff, Steve Schwab, Shai Halevi, Andrey Kim, Daniele Micciancio, Yuriy Polyakov, Vincent Zucca. Palisade. `https://gitlab.com/palisade/palisade-release`, 2017.

[61] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Heaan. `https://github.com/snucrypto/HEAAN`, 2016.

[62] Shai Halevi and Victor Shoup. Algorithms in helib. In *34rd Annual International Cryptology Conference, CRYPTO 2014*, volume 2014, pages 554–571, 2014.

[63] Adrià Gascón and Oliver Strickson. Sheep. `https://github.com/alan-turing-institute/SHEEP`, 2017.

[64] Eduardo Chielle, Oleg Mazonka, Nektarios Georgios Tsoutsos, and Michail Maniatakos. E 3 : A framework for compiling c++ programs with encrypted operands. *IACR Cryptology ePrint Archive*, 2018:1013, 2018.

[65] Nicholas Mainardi, Alessandro Barenghi, and Gerardo Pelosi. Privacy preserving substring search protocol with polylogarithmic communication cost. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 297–312, 2019.

[66] Nicholas Mainardi, Davide Sampietro, Alessandro Barenghi, and Gerardo Pelosi. Efficient oblivious substring search via architectural support. In *Annual Computer Security Applications Conference*, pages 526–541, 2020.

[67] Nicholas Mainardi, Alessandro Barenghi, and Gerardo Pelosi. Privacy-aware character pattern matching over outsourced encrypted data. *Digital Threats: Research and Practice*, 2021.

[68] TFEncrypted. tf-seal. `https://github.com/tf-encrypted/tf-seal`, 2019.

[69] Paweł Budzianowski and Ivan Vulić. Hello, it's gpt-2 - how can i help you? towards the use of pretrained language models for task-oriented dialogue systems. In *Proceedings of the 3rd Workshop on Neural Generation and Translation*, pages 15–22, 2019.

[70] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30(4):681–694, 2020.

[71] Yusuf Aytar, Carl Vondrick, and Antonio Torralba. Soundnet: Learning sound representations from unlabeled video. In *Advances in Neural Information Processing Systems*, volume 29, pages 892–900, 2016.

[72] Qiuqiang Kong, Turab Iqbal, Yong Xu, Wenwu Wang, and Mark D. Plumbley. Dcase 2018 challenge baseline with convolutional neural networks. *arXiv: Sound*, 2018.

[73] Steffen Schneider, Alexei Baevski, Ronan Collobert, and Michael Auli. wav2vec: Unsupervised pre-training for speech recognition. In *Interspeech 2019*, pages 3465–3469, 2019.

[74] Tomas Mikolov, Kai Chen, Greg S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *ICLR (Workshop Poster)*, 2013.

[75] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[76] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5(1):135–146, 2017.

[77] Lena Voita. Word embeddings. `https://lena-voita.github.io/nlp_course/word_embeddings.html`, 2020.

[78] Jay Alammar. The illustrated transformer. `https://jalammar.github.io/illustrated-transformer/`, 2018.

[79] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning. *CoRR*, 2017. cite arxiv:1705.03122.

[80] Pytorch. Transformer, pytorch official implementation. `https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html`, 2019.

[81] A. Tuerk and S.J. Young. Polynomial softmax functions for pattern classification. Technical report, Cambridge University Engineering Department Trumpington Street Cambridge CB2 1PZ England, 2001.

[82] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of The 32nd International Conference on Machine Learning*, volume 1, pages 448–456, 2015.

[83] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *arXiv: Machine Learning*, 2016.

[84] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[85] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: applying neural networks to encrypted data with high throughput and accuracy. In *ICML'16 Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, pages 201–210, 2016.

[86] Kevin Patel and Pushpak Bhattacharyya. Towards lower bounds on number of dimensions for word embeddings. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, volume 2, pages 31–36, 2017.

[87] Oren Melamud, David McClosky, Siddharth Patwardhan, and Mohit Bansal. The role of context types and dimensionality in learning word embeddings. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1030–1040, 2016.

[88] Google, TensorFlow Team. Introducing tensorflow feature columns. `https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html`, 2017.

[89] Stéfan van der Walt, S Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science and Engineering*, 13(2):22–30, 2011.

[90] Python. multiprocessing, python. `https://docs.python.org/3/library/multiprocessing.html`, 2008.

[91] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai,

and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, pages 8026–8037, 2019.

[92] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, pages 265–283, 2016.

[93] François Chollet. Keras: The python deep learning library. *Astrophysics Source Code Library*, 2018.

[94] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

[95] Tensorflow. Tensorboard. `https://github.com/tensorflow/tensorboard`, 2017.

[96] Liu Zhong, Chen Shuming, Dou Qiang, Guo Yang, Liu Hengzhu, Tian Xi, Gong Guohui, Chen Haiyan, Peng Yuanxi, Wan Jianghua, Liu Sheng, Chen Yueyue, Hu Xiao, and Wu Jiazhu. Block matrix multiplication vectorization method supporting vector processor with multiple mac (multiply accumulate) operational units, 2013.

[97] Python. Global interpreter lock. `https://wiki.python.org/moin/GlobalInterpreterLock`, 2007.

[98] Z. Xianyi, Wang Qian, and Zhang Yunquan. openblas: a high performance blas library on loongson 3a cpu. *Journal of Software*, 22, 2011.

[99] Vishal Kanaujia and Chetan Giridhar. Python threads: Dive into gil! In *Pycon India 2011*, Sep 2011. `https://in.pycon.org/2011/talks/41-python-threads-dive-into-gil!/`.

[100] Python. Global interpreter lock implementation under ceval.c. `https://github.com/python/cpython/blob/e62a694fee53ba7fc16d6afbaa53b373c878f300/Python/ceval.c#L238`, 2016.

[101] Dusit Niyato Ngoc Thanh Nguyen, suphamit Chittayasothorn and Bodgan Trawiński (Eds.). Intelligent information and database systems. In *Intelligent Information and Database Systems: 13th Asian Conference, ACIIDS 2021 Phuket, Thailand, April 7-10, 2021 Proceedings*, pages 125–126, 2021.

[102] Ziqian Zeng and Yangqiu Song. Variational weakly supervised sentiment analysis with posterior regularization. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 3259–3268, 2021.

[103] Vikas Raunak, Vivek Gupta, and Florian Metze. Effective dimensionality reduction for word embeddings. In *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*, pages 235–243, 2019.

[104] Taehoon Kim and Jihoon Yang. Abstractive text classification using sequence-to-convolution neural networks. *arXiv preprint arXiv:1805.07745*, 2018.

Index

Appendix A

Source code

This appendix includes the referred source code section in the thesis.

A.1 Loading and encoding of model's parameters

This first method is the general one for preparing the encoded loaded parameters in order to initialize HErBERT.

```
def __load_params(self, layer, params, enc):
    """Return requested parameters"""
    out = [self.__load_tensor(module, enc) if layer is None else
           self.__load_tensor(self.layer_id + str(layer) + "."
                              + module, enc)
           for module in params]
    return out[::2], out[1::2]
```

While the other following method is in charge of returning the HE encoded version of the parameters' tensor for each layer:

```
def __load_tensor(self, module, enc=True):

    tensor = self.model.get(module)

    if tensor is not None:
        return cm.encode(tensor.cpu().detach().numpy()) if enc
        else tensor.cpu().detach().numpy()
    else:
```

```
        return None
```

A.2 HE Multi Headed Self-Attention implementation

```
import numpy as np
from attention.self_attention import SelfAttention
from modules.linear_layer import LinearLayer
from modules.dropout import Dropout
from core.module import Module
from core.crypto import CryptoManager as cm


class MultiHeadedSelfAttention(Module):
    """Multi Headed Self Attention implementation for the encoder"""

    def __init__(self, heads, model_features, dropout=0.1):

        self.d_k = model_features // heads
        self.model_features = model_features
        self.h = heads
        self.total_h_size = self.h * self.d_k
        self.pruned_heads = set()

        self.attention_layers = LinearLayer(model_features,
                                            model_features)
        self.output_linear = LinearLayer(model_features,
                                         model_features)
        self.self_attention = SelfAttention(self.d_k, self.h)
        self.dropout = Dropout(p=dropout)
        #Dropout implementation just to resemble original code
        #but it is disabled of course

    def init_param(self, weights, biases):
        self.attention_layers.init_param(weights[0], biases[0])
        self.output_linear.init_param(weights[1], biases[1])
```

```
def __transpose_for_scores(self, x):
    batch_size = x.shape[0]
    return np.reshape(np.transpose(x, (0, 2, 1, 3)),
                      (batch_size * self.h,
                      self.model_features * self.h, self.d_k))

def __call__(self, x, mask=None):
    batch_size = x.shape[0]
    t, b, e = x.shape
    s = e // self.h
    scaling = float(self.h) ** -0.5

    query, key, value = np.split(self.attention_layers(x),
                                 3, axis=-1)
    if cm.logging() and cm.is_enabled():
        print(cm.get_noise(value))

    query = np.transpose(np.reshape(query, (t, b * self.h,
                         self.d_k)), (1, 0, 2))
    key = np.transpose(np.reshape(key, (-1, b * self.h,
                       self.d_k)), (1, 0, 2))
    value = np.transpose(np.reshape(value, (-1, b * self.h,
                         self.d_k)), (1, 0, 2))

    # Apply self attention using the scaled dot product
    x = self.self_attention(query, key, value, mask=mask,
                            dropout=self.dropout)

    x = np.transpose(x, (1, 0, 2)).reshape(t, b, e)

    return self.output_linear(x)
```

A.3 Fundamental HE tensor operations implementation

```
    @staticmethod
    def encrypt(tensor: np.ndarray):
        try:
            return np.array(list(map(CryptoManager.HE.encryptFrac,
                                     tensor)))
        except TypeError:
            return np.array([CryptoManager.encrypt(t)
                             for t in tensor])

    @staticmethod
    def decrypt(tensor: np.ndarray):
        try:
            return np.array(list(map(CryptoManager.HE.decryptFrac,
                                     tensor)))
        except TypeError:
            return np.array([CryptoManager.decrypt(t)
                             for t in tensor])

    @staticmethod
    def encode(tensor: np.ndarray):
        try:
            return np.array(list(map(CryptoManager.HE.encodeFrac,
                                     tensor)))
        except TypeError:
            return np.array([CryptoManager.encode(t)
                             for t in tensor])

    @staticmethod
    def decode(tensor: np.ndarray):
        try:
            return np.array(list(map(CryptoManager.HE.decodeFrac,
                                     tensor)))
        except TypeError:
            return np.array([CryptoManager.decode(t)
                             for t in tensor])
```