Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Private ML as a Service for Natural Language Processing

Bachelor Thesis

Fabio Bertschi

July 2, 2021

Advisors: Prof. Dr. Kenny Paterson, Prof. Dr. Kaveh Razavi, Alexander Viand

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

In recent years, a lot of services have been outsourced to the cloud. Sensitive information stored online and lots of data breaches have lead to an increase in demand for secure cloud-computing solutions. One cloud-application where security is of high importance is natural language processing, which frequently deals with inherently privacy-sensitive information. For example, using machine translation can reveal the contents of otherwise private conversations. Outsourcing natural language processing tasks to the cloud while preserving the privacy of the input is possible thanks to Fully Homomorphic Encryption (FHE) schemes. FHE schemes allow a server to compute calculations on encrypted data without gaining any information about it. Natural language processing frequently relies on Recurrent Neural Network (RNN), which are challenging to implement in FHE because of their deep, recursive, nature and hard to approximate activation functions. However, recent developments in FHE have introduced a novel scheme that promises to be able to evaluate arbitrarily deep functions and non-polynomial activation functions efficiently. The goal of this thesis is to investigate the state-of-the-art in privacy-preserving natural language processing in FHE. We design, implement and evaluate a network for privacy-preserving machine translation, using this to highlight both the recent advances and remaining limitations in FHE-based machine learning.

Contents

Chapter 1

Introduction

Modern machine learning has enabled a wide variety of Natural Language Processing (NLP) applications including speech recognition and automated translation. This has enabled automation in a variety of fields that would have traditionally required prohibitively expensive human operators, translators, etc. With products like Alexa, Siri, and Google Translate, these technologies have become ubiquitous in everyday life. Speech recognition makes using technology not only more convenient but represents a significant advance in accessibility. Meanwhile, online translation services remove communication barriers and allow for an international spread of information and discourse.

Due to the size and complexity of the machine learning models, most NLP applications rely on cloud backends. In order to process the input, the cloud providers require access to the input (users' speech or text). While data is sent to the provider over an encrypted channel, it must be decrypted to compute on and return the result, e.g. the recognized sentence or translated text. In addition to this automated process, user data is frequently exposed to human operators, e.g. for quality control. Furthermore, user data might be used to get information on user interests for further advertisement strategies or just to get a general fingerprint to simplify other activities.

Leakage of information can pose significant risks to the user, especially in countries that actively monitor internet usage and heavily censor speech in general. For example, voice-based assistants are frequently activated unintentionally, leading to private conversations being leaked to the cloud provider [20]. Meanwhile, where translators are used to enable conversations, they also reveal at least half the conversation to the cloud provider. In many cases, abstaining from these services is not possible (e.g. accessibility) or feasible (e.g. too expensive to use human translators). Instead, it should be possible to benefit from such services without having to expose one's data to a potentially untrusted third party. This can be achieved by using secure computation techniques to make machine translation privacy-preserving.

FHE allows one to compute on encrypted data, enabling privacy-preserving outsourcing of, e.g., machine learning applications [15]. There has been a significant amount of work in the area of FHE-based computer vision for tasks such as classification and object recognition, showing how to evaluate deep Convolutional Neural Networks (CNNs) relatively efficiently [6, 16, 14].

Despite language models being even more privacy sensitive (e.g. Alexa), we have not seen similar progress. This is because NLP applications generally rely on Recurrent Neural Networks (RNNs) which are difficult to realize in common FHE schemes because of their deep, recursive, nature [22].

In FHE, each computation executed on the ciphertext adds some *noise*, to the ciphertext. Eventually, this noise grows to a point where decryption fails, with especially chained multiplications quickly leading to an explosion of noise. While even the very first FHE schemes introduced *bootstrapping* techniques that reset the noise, these were generally too inefficient to be of much practical use. As a consequence, these FHE schemes remained limited in the depth of neural networks they could evaluate.

Natural Language Processing (NLP) applications use especially large and complex neural networks. The vast majority use RNNs to be able to deal with the context dependent nature of natural language, both in spoken and in written form. RNNs include a feedback mechanism and can thus make future predictions based on past inputs. These models therefore work by calculating very long chains of multiplications, making it challenging for FHE to work correctly.

However, a recent development introduced a novel scheme, which significantly improves the performance of bootstrapping in this setting [11, 10]. This allows us to calculate arbitrarily long multiplicative chains in a feasible time. It is an important improvement since it may allow us to build privacy preserving deep neural network applications, in particular NLP applications.

Our goal is to implement a Recurrent Neural Network (RNN) for NLP using this new FHE scheme to assess the new state of the art limits for deep neural networks. Towards this, we want to develop a toolbox containing the most important components used for building such machine learning models. From this, we want to create an end-to-end NLP application, e.g. a privacy-preserving online translator. We want to closely document the toolbox we develop, such that they can also be used by other developers, either to learn from or to build other projects upon ours. Finally, we will benchmark our solution thoroughly, comparing it against the previous state of the art.

Chapter 2

Background

In this section, we briefly introduce core notions of Fully Homomorphic Encryption (FHE) and Machine Learning (ML).

2.1 Fully Homomorphic Encryption

In this section, we introduce the TFHE encryption scheme [9]. TFHE internally uses three different types of encryption which interact to form the overall TFHE scheme.

2.1.1 LWE-based Encrypton

The security of this scheme is based on the Learning with Errors (LWE) assumption. The underlying LWE problem is to deduce a linear function f with n arguments over a finite ring from samples $y_i = f(x_i)$ with the points being offset by small errors. The LWE assumption is that this is computationally hard to solve and therefore of good use for encryption. The important factors in setting the parameters for the TFHE scheme are security, precision of the ciphertexts and computational efficiency [17].

An LWE ciphertext c is created from the following equation:

$$c = (a, b) = (a, a * s + \mu + e)$$

where a is the mask which is sampled from a random distribution whenever a ciphertext is created, s is the secret key which is a n-dimensional vector with entries 0 or 1, μ is the (encoded) message, and e is the noise. The mask is stored with the ciphertext such that the secret key holder is able to decrypt again.

The noise e is sampled from a Gaussian distribution whose standard deviation is chosen by the user. The larger the standard deviation the higher the security, because we have a larger random number which blurs our ciphertext. If we choose a large standard deviation there has to be a greater distance between the possible encoded messages, because we need the space between the possible messages to be larger than the noise such that we can round it away again.

Symmetric Encryption

For $\mu = m/q$, m ϵ Z_q, $a \overset{\$}{\leftarrow} T_q$, $s \overset{\$}{\leftarrow} \{0,1\}^n$ and $\epsilon \overset{\chi}{\leftarrow} T_q$, symmetric encryption is defined as follows.

$$Enc_s(\mu) = \left(a_1, ..., a_n, \left[\sum a_i s_i + \mu + \epsilon\right]\right) (a_1, ..., a_n, b) \tag{2.1}$$

$$\begin{aligned} ct_0 &= \left[\sum a_i s_i + \mu + \epsilon\right] \\ ct_1 &= a_1, ..., a_n = a \\ ct &= (ct_1, ct_0) \end{aligned} \tag{2.2}$$

Decryption is correct as long as the rounding to the next valid point in the torus (i/q for $1 \leq i \leq n$), denoted by $[\dots]_1$, is correct, which is always the case for a freshly encrypted ciphertext.

$$Dec_s(ct) = ct_0 - ct_1 s = \left[b - \sum a_i s_i\right]_1 = [\mu + \epsilon]_1 = m \tag{2.3}$$

Asymmetric Encryption

From the symmetric scheme, a public-key scheme follows naturally by using an encryption of zero as the public key.

$$Enc_s(0) = \left(a_1, ..., a_n, \left[\sum a_i s_i + \epsilon\right]_1\right) = pk \tag{2.4}$$

$$\begin{aligned} pk_0 &= \left[\sum a_i s_i + \epsilon\right]_1 \\ pk_1 &= a_1, \dots, a_n = a \\ pk &= (pk_1, pk_0) \end{aligned} \tag{2.5}$$

This encryption of zero is then re-randomized during public key encryption. Note that for simplicity, we omit $[\dots]_1$ here:

$$Enc_a(\mu) = \left(a_i u + e_1, \left(\sum a_i s_i + \epsilon\right) u + \mu + e_0\right) = ct \tag{2.6}$$

$$\begin{aligned} \widetilde{a_i} &= a_i u + e_1 \\ ct_0 &= \left[\sum a_i s_i + \mu + \epsilon\right] \\ ct_1 &= (\widetilde{a_1}, \dots, \widetilde{a_n}) = \widetilde{a} \\ ct &= (ct_1, ct_0) \end{aligned} \tag{2.7}$$

Again, we assume that the parameters are selected appropriately as to guarantee correct rounding for fresh (public key encrypted) ciphertexts.

$$\begin{aligned} Dec_a(ct) = ct_0 - ct_1 s &= \sum a_i s_i u + \epsilon u + \mu + e_0 - \sum (a_i u + e_1) s_i \\ &= \mu + e_0 + \epsilon u - \sum e_1 s_i + e_1 s \\ &= \left[\mu + e_0 + \epsilon u - \sum e_1 s_i + e_1 s\right]_1 \\ &= \left[\frac{m}{q} + + e_0 + \epsilon u - \sum e_1 s_i + e_1 s\right]_1 = m \end{aligned} \tag{2.8}$$

Homomorphic Operations

One of the key aspects of FHE schemes is the fact that they support homomorphic operations over the ciphertxts. For example, it is easy to see how component-wise addition of ciphertexts leads to a message that encrypts the sum of the original messages:

$$\begin{aligned} [Enc(m_1) + Enc(m_2)] &= \left(a_1, ..., a_n, \left[\sum a_i s_i + m_1 + \epsilon\right]\right) + \left(a_1, ..., a_n, \left[\sum a_i s_i + m_2 + \epsilon\right]\right) \\ &= \left(a_1, ..., a_n, \left[\sum a_i s_i + m_1 + m_2 + \epsilon\right]\right) \\ &= Enc(m_1 + m_2) \end{aligned} \tag{2.9}$$

We will discuss how TFHE realizes multiplications between LWE ciphertexts later, when discussing bootstrapping.

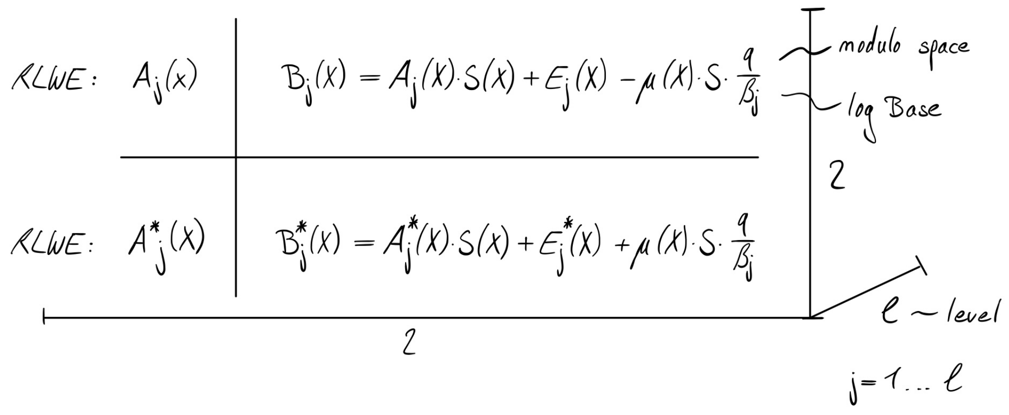
2.1.2 RLWE-based Encryption

Ring Learning with Errors (RLWE) is the Learning with Errors (LWE) problem specialized to polynomial rings over finite fields. RLWE encryption works similar to LWE encryption, but now each component is a polynomial, i.e., a ciphertext C is a tuple of polynomials

$$C = (A(X), B(X))$$

where $B(X) = A(X) * S(X) + \delta M(X) + E(X)$ and every component is of the form $F(X) = F_0 + F_1 X + ... + F_{N-1} X^{N-1}$. The coefficients, though, are of different type for every component:
$A_i \overset{\$}{\leftarrow} T_q$ is the mask element drawn from the uniform distribution.
$S_i \leftarrow \{0,1\}$ is the secret key element known only by the owner.
$E_i \overset{\chi}{\leftarrow} T_q$ is the noise element drawn from the Gaussian distribution.
$\Delta M(X) = m/q$, m ϵ Z_q is an encoded message.

The parameters are the polynomial size, the dimension of the mask and the standard deviation of the noise. The dimension of the mask refers to the



RLWE: $A_j(x)$
$B_j(X) = A_j(X)\cdot S(X) + E_j(X) - \mu(X)\cdot S\cdot \frac{q}{\beta_j}$
RLWE: $A^*_j(X)$
$B^*_j(X) = A^*_j(X)\cdot S(X) + E^*_j(X) + \mu(X)\cdot S\cdot \frac{q}{\beta_j}$
modulo space
log Base
2
2
ℓ ~ level
$j = 1 \dots \ell$

Figure 2.1: Illustration of an RGSW ciphertext

mask coefficients size. The polynomial size (usually N) refers to the degree of the polynomial of each component. Every coefficient of the polynomial can be used to store an encoded message. This means a whole vector of encoded messages can be encrypted or decrypted at once.

2.1.3 RGSW-based Encryption

The Ring-Gentry-Sahai-Waters scheme is used by TFHE in its bootstrapping algorithm. An RGSW ciphertext consists of many RLWE ciphertexts, as seen in Figure 2.1. While the encoded message is the same for each RLWE polynomial, other aspects such as the mask and the noise are dependent on their position. The advantage of RGSW ciphertexts is that they can be multiplied. There exist two different products, the internal and the external.

Formally, an RGSW ciphertext is defined as

$$C = Z + m * G_2$$

with $C \in T_N[X]^{2lx2}$ and Z a vector of $2l$ RLWE encryptions of 0 and G_2 being the gadget matrix

$$G_2 = \left[\begin{array}{c|c} g & 0 \\ \hline 0 & g \end{array}\right]$$

with $g^T = (2^{-1}, ..., 2^{-l})$.

G_2^{-1} can be used to decompose $T_N[X]$ elements (i.e., the components of an RLWE ciphertext) with respect to G_2.

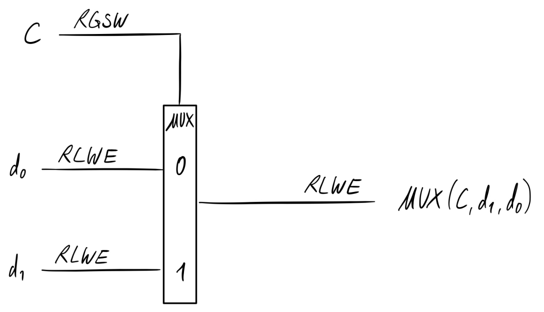


Figure 2.2: Illustration of a CMUX

Internal Product

The internal product between two RGSW ciphertexts is defined as

$$C \boxtimes D = G_2^{-1}(D) * C = \begin{pmatrix} G_2^{-1}(d_1) * C \\ . \\ . \\ . \\ G_2^{-1}(d_2l) * C \end{pmatrix} = \begin{pmatrix} C \boxdot d_1 \\ . \\ . \\ . \\ C \boxdot d_2l \end{pmatrix} \tag{2.10}$$

with $G_2^{-1}(d_i)$ having size $2l \times 2l$, C having size $2l \times 2$ and the result entry $C \boxdot d_i$ having size $2l \times 2$.

External Product

The external product between a RGSW ciphertext and an RLWE ciphertext d is defined as

$$C \boxdot d = G_2^{-1}(d) * C \tag{2.11}$$

with $G_2^{-1}(d)$ having size $1 \times 2l$, C having size $2l \times 2$ and the result having size 1×2.

Ciphertext Multiplexing

The ciphertext multiplexer is a homomorphic version of a multiplexer, i.e. a logical gate which selects an input based on the signal of a controller. A homomorphic multiplexer (CMUX) has the same logical function, but takes as inputs two RLWE vectors and as controller an RGSW encryption, as seen in Figure 2.2.

CMUX is implmented using the RGSW external product:

$$MUX(C, d_1, d_0) = C \boxdot (d_1 - d_0) + d_0$$

This is correct as seen from

$$\begin{aligned}&(C \wedge d_1) \vee (\bar{C} \wedge d_0)\\ &= C * d_1 + (1 - C) * d_0\\ &= C * d_1 + d_0 - C * d_0\\ &= C \boxdot (d_1 - d_0) + d_0\end{aligned} \tag{2.12}$$

where the first equation is the definition of a multiplexer gate.

2.1.4 Bootstrapping

During FHE operations the amount of noise, which is added to the ciphertext during encryption, might be modified as well. For example, during addition the noise of the two ciphertexts is summed up (Equation (2.9)). This is undesired, since it is only possible to round to the correct encoding point if the noise is in a certain rounding interval. If the noise has grown larger than half the interval between the encoding points, decryption will be rounded to the wrong value. The goal of bootstrapping is to reduce the noise of a ciphertext. In the following, the bootstapping procedure and its components will be explained in more depth.

Rounding Noise

The main aim of the bootstrapping is to map the noisy message $\mu^\star$ (which is $\mod q$) to a message μ (in $\mod p$) that corresponds to one of the valid encoding points. This rounding can be expressed by the following formula

$$Upper_{q,p}(\mu^\star) = \frac{q}{p}\left\lceil \frac{p}{q} * \text{lift}(\mu^\star) \right\rceil \tag{2.13}$$

where the lift operations denotes a lift of the modulo under which the operand is given. Essentially, we left shift the message by $\frac{q}{p}$ leaving only the $\log q - \log p$ bits behind the decimal point. Then we round these digits, where the noise is located, off and reverse the shift. An illustration of the process is given in Figure 2.4.

Test Vector We want to evaluate the rounding function (Equation (2.13)) on an encrypted message in order to reduce the noise. We achieve this by first building a test polynomial, which has each of it's coefficients correspond to one of the possible encoded message (after rounding), i.e. we define

$$v = v_0 + v_1 X + ... + v_{q-1} X^{q-1}$$

with the i-th coefficient set to $v_i = Upper_{q,p}(imodq)$ as seen in Figure 2.4.

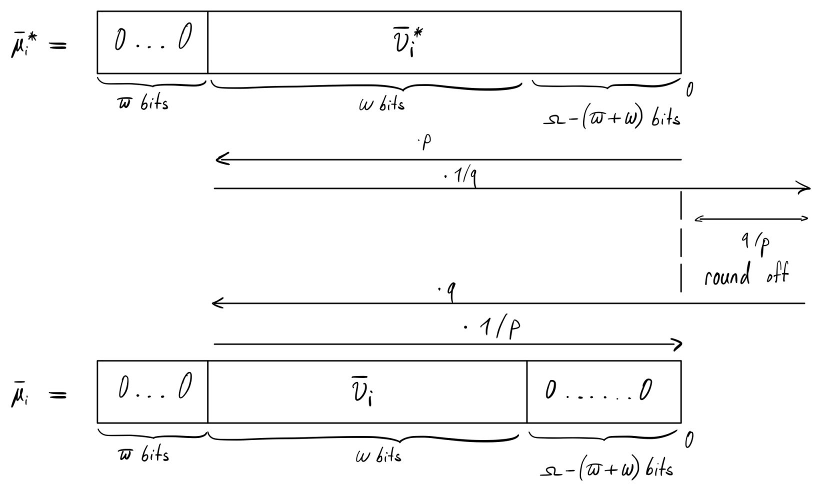


Figure 2.3: Illustration of the rounding performed by Equation (2.13) where $q = 2^{\Omega}$, $p = 2^{\tilde{\omega}+\omega}$, $\tilde{\omega}$ denotes 'padding' bits, and v is the message without those padding bits.

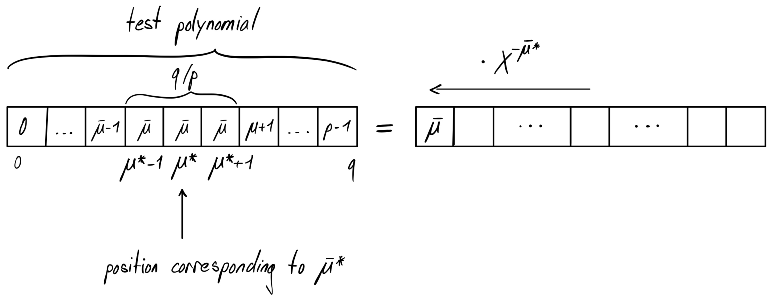


Figure 2.4: Illustration of the test vector, showing multiple copies of each possible encoded message.

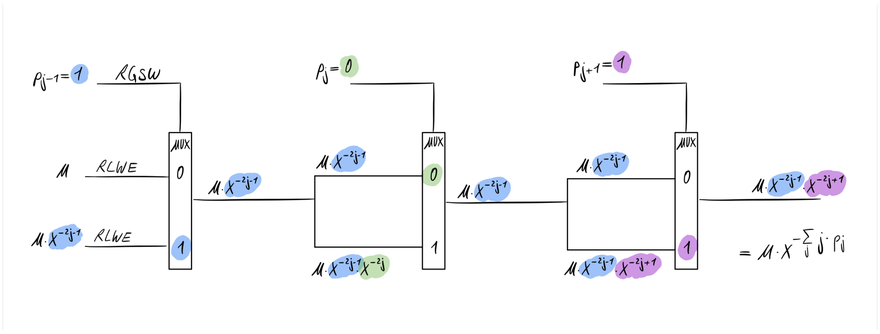


Figure 2.5: Illustration of the succession of CMUX gates used in blind rotation.

Blind Rotation In order to evaluate the Upper Function on a ciphertext, we want to treat the test vector as a look up table. We achieve this by multiplying the test vector with X^{μ^*} and extracting the constant coefficient. However, since μ^* is not available publicly, we must use a succesion of CMUX gates to perform this rotation blindly.

We pass the test vector through a series of multiplexers. Remember, a multiplexer is a logical gate which passes an input based on the signal of a controller, see Section 2.1.3 for the explanation of a homomorphic one. The *bootstrapping key*, a sequence of RGSW encryptions of the individual elements of secret key $s_i \in 0, 1$, acts as the controller. In the i-th CMUX gate, we select between the current vector or a copy of the current vector rotated by a_i. This succesion of CMUX gates is the same as choosing which a_i we rotate by, therefore the output of this sequence is equivalent to a rotation by $\sum a_i * s_i$, as seen in Figure 2.5

Drift

Polynomials used in RLWE are from $Z_N[X]$ modulo $X^N + 1$, here, the group element X has order $2N$. As a consequence, rotations by $2N$ are equivalent to not rotating at all, and rotations by $k > 2N$ are equivalent to rotations k mod $2N$. The test vector can therefore only represent 2N values. However, the noisy message $\mu^\star$ is originally modulo q. Therefore, moving to mod $2N$ leads to a rounding error, which is called drift.

$$\begin{aligned}
\mu^\star &= b + \sum_{j=1}^{n} s_j * a_j \pmod q \\
\tilde{\mu^\star} &= \tilde{b} + \sum_{j=1}^{n} s_j * \tilde{a}_j \pmod{2N} \\
\tilde{b} &= \left\lceil \frac{2N(b \bmod q)}{q} \right\rceil \\
\tilde{a}_j &= \left\lceil \frac{2N(a_j \bmod q)}{q} \right\rceil
\end{aligned} \tag{2.14}$$

Furthermore, the RLWE vector has N entries and we want to display every possible entry in the space with each coefficient, but the mod 2N space has two times the entries. We therefore cut off a padding bit, the most significant bit from $\tilde{\mu^\star}$, to reduce the space to N values:

$$v_i = Upper_{q,p}\left(\frac{q}{2N} * i \bmod q\right) \tag{2.15}$$

2.1.5 Programmable Bootstraping

One of the major features of TFHE is that we can apply a function during bootstrapping with no additional cost by simply modifying the test vector. We apply the desired function f to the coefficients v_i before using the $Upper$ function. The test vector essentially acts like a look up table, i.e. it is a list of function evaluations at different points. The step size between the evaluated values is given by the number of different possible messages. Now when executing the bootstrapping, the result has reduced noise and returns the function of the argument.

This is also used to realize ciphertext-ciphertext multiplication, by computing $a * b = \frac{1}{4}(a+b)^2 - \frac{1}{4}(a-b)^2$, where the squaring and subsequent division by four are realized using programmable bootstrapping. Since the divison by four can be folded into the same Look-Up-Table, the entire computation requires only two bootstrap operations. Correctness is easy to see:

$$\begin{aligned}
\frac{1}{4}(a+b)^2 - \frac{1}{4}(a-b)^2 &= \frac{1}{4}(a^2 + 2ab + b^2) - \frac{1}{4}(a^2 - 2ab + b^2) \\
&= \frac{1}{4}(a^2 + 2ab + b^2 - a^2 + 2ab - b^2) \\
&= \frac{1}{4}(4ab) \\
&= ab
\end{aligned} \tag{2.16}$$

Chapter 3

Introduction to Concrete

In order to implement the homomorphic operations in our program, we use the Concrete library [10]. This is a state-of-the-art library published recently (fall 2020), implementing an improved version of the Torus Fully Homomorphic Encryption (TFHE) scheme [10]. The library is written in Rust, which is designed to be a memory-safe but high-performance alternative to C/C++. While initial documentation is available [5], it is limited to a few basic operations and does not sufficiently explain how to develop more complex programs. Therefore, this chapter provides an introduction to Concrete based on the insights we gained while working and exploring this novel library and the associated literature.

3.1 Parameter Selection

In Concrete one has to choose from a predefined set of parameters, offering either 128 bit or 256 bit security. There are two settings that determine the parameters for an LWE-based encryption: the dimension of the mask and (the log of) the standard deviation of the noise (`log_std_dev`). Selecting the first requires a trade off between security and computational efficiency while the second imposes a trade off between security and the precision of the encryption. The combinations can be accessed by providing the name of a predefined parameter set to `encode_encrypt` function. The possible combinations are listed in the Concrete source code in `concrete/src/lwe_params.rs`.

3.1.1 Encoder

The encryption algorithm needs to map potentially continuous inputs to a finite number of discrete points. We need space between the encoded values, since we need to be able to round away noise between the points. This

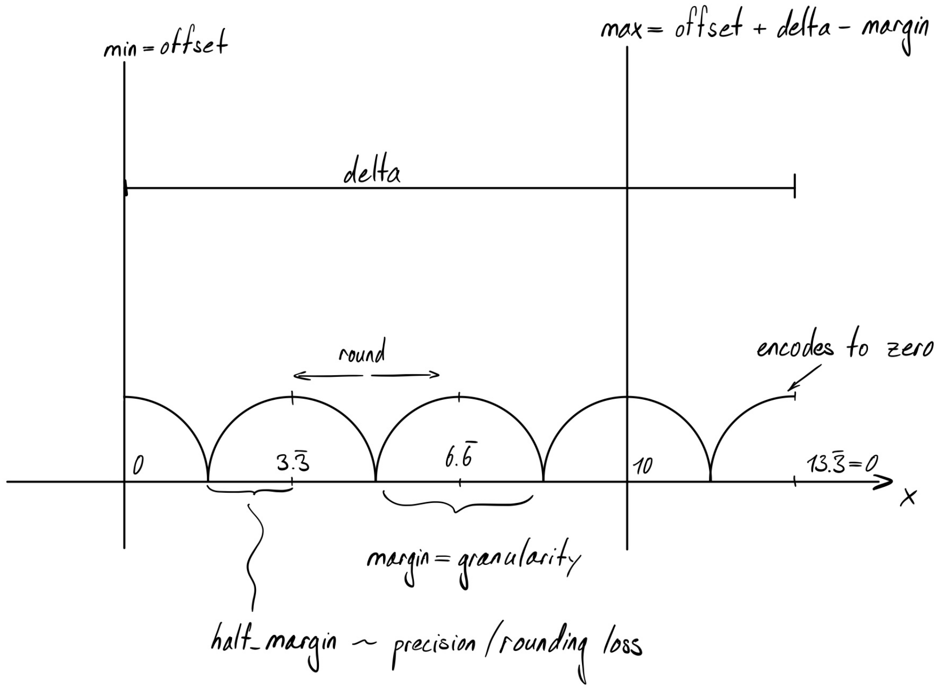


Figure 3.1: Example encoding with $min = 0$, $max = 10$ and $b_{prec} = 2$

noise is an important part of the security of the encryption, but must be removable as part of the decryption. Discrete points, with a rounding interval around them, will naturally lead to a loss of precision. This precision can be specified by the user and is a trade-off with the capacity to handle noise growth.

The encoding is defined by four encoding parameters, which the user has to specify. The Encoder takes an interval specified by max and min, and it also takes a number of precision bits b_{prec}, which lets the encoder compute the sub-intervals, whose edges it will round to. Therefore the precision defines the number of possible values in the interval. The last parameter needed is the number of padding bits b_{pad}. Padding bits are additional storage allocated to the ciphertext. They will later be used when operations require further information storage, for example when an operation maps outside of the encoding interval to keep the same precision with a larger interval a padding bit is used. They do not play any role in the initial encoding of a message. A possible encoding of an interval can be seen in Figure 3.1.

The mapping of the message to the plain text encoding is given by the following equation:

$$\mu = \left\lfloor \frac{m}{margin} \right\rceil * margin$$

where the margin is the interval between two points. It will be further explained in the next paragraph. Notice, that the encoding is actually a bijective function and therefore does not do any rounding yet. Rounding to the next margin will only happen if we use the encoder in a rounding context, which explicitly instructs Concrete to perform rounding. Otherwise, the encoding function will not round to the next margin, but just display the result on the torus with the precision of the computer.

In Concrete the encoding is computed by methods implemented in the encoder class. The encoder class is instantiated with the four parameters given above. Inside the class there exists four attributes, the offset o, the size of the interval Δ, the number of precision bits b_{prec} and the number of padding bits b_{pad}. The delta is $max - min + margin$ and represents the range of values one can encode. The $margin = \left\lceil \frac{max-min}{2^{b_{prec}}-1} \right\rceil$ is the interval between two points. Notice that the last half margin in the Δ will round to a value outside of the interval, which will then wrap around to zero. Encoding values outside of the interval will lead to undefined behavior.

Sometimes, *granularity* is used instead of *margin*. The granularity is similar to the margin but can be computed directly from the attributes of the encoder, e.g. Δ instead of *max* and *min*:

$$\begin{aligned} granularity &= \left\lceil \frac{delta}{2^{b_{prec}}} \right\rceil \\ &= \left\lceil \frac{max - min + margin}{2^{b_{prec}}} \right\rceil \end{aligned}$$

3.2 Homomorphic Operations

The main components and features implementing the homomorphic operations will be discussed in the following subsections.

3.2.1 Adding a Constant to a Ciphertext

Adding a constant m homomorphically can be done either by shifting the encoding or adding to the ciphertext. The function `add_constant_dynamic_encoder` can be used to shift the encoding by m, while the function `add_constant_static_encoder` adds a constant m to the ciphertext itself and leaves the encoding unchanged. Since the encoding stays the same, one should use the latter function if the following operation requires the same input encoding interval as the one defined in the beginning. It is advantageous if one can calculate an encoding in the beginning which matches all operations and is then able to execute the computations without the need to change the encoding again.

Shifting the encoding allows us to change the encoded message for "free" without modifying the underlying ciphertext. This means, that we use a different interval for decoding than the one initially specified. This leads to a different mapping between message and plaintext. Using an interval shifted by m for the decoding leads to a message which is exactly m larger, as we can see from Section 3.1.1. It is therefore equivalent to adding a constant. Shifting does not require any specific encoding interval and can be done with any constant. The result of the addition also does not need to be inside the initial encoding interval. The precision stays the same, since it depends only on the number of bits available and the interval size, which do not change. Since the operation is does not affect the ciphertext, it does not add noise to the ciphertext.

Alternatively, we can modify the underlying ciphertext by performing a homomorphic plaintext addition. This adds the constant m to the ciphertext's body, which leads to an encryption corresponding to the original message plus the constant. LWE encryption is linear, i.e., the ciphertext c encrypting m_0 is defined as $c = (c_0, c_1) = (a * s + m_0 + e, a)$. Therefore, we can directly add a constant m to c_0 and obtain a valid encryption of $m_0 + m$: $c + m = (c_0 + m, c_1)$. While any encoding can be used, since the output encoding is exactly the same as the one initially given, only additions in the interval are possible. Since the operation remains inside the interval, no padding bits are consumed. This operation does, on the other hand, add some small amount of noise, since it is calculated on the ciphertext itself.

3.2.2 Adding two Ciphertexts

There are two ways to add ciphertexts, which both modify the underlying ciphertext, but differ in whether or not they consume padding. Adding two ciphertexts by consuming padding is implemented in the function `add_with_padding` and there exists an equivalent operation for subtraction. Adding two ciphertexts without consuming any padding can be done with the function `add_with_new_min`, however this is only possible if we have some additional knowledge on the range of values encrypted by the ciphertext. Unfortunately, there does not exist an equivalent subtraction without padding yet.

In general a homomorphic addition works as follows. As we have previously already seen in the FHE theory section, the LWE ciphertext c encrypting m_0 is defined as $c = (c_a, c_b) = (a * s + m_0 + e, a)$. Adding two ciphertexts c and c' is done component-wise, i.e. $c + c' = (c_0 + c'_0, c_1 + c'_1)$. This operation is relatively efficient and fast. Note, that the noise of the two ciphertexts has been summed up to build the noise of the result.

Additions of large numbers can have a carry out and therefore the resulting ciphertext might need to consume a padding bit. Even though, this will not *always* be the case, because of the nature of FHE, which needs all computations and allocations to be independent of the input, the homomorphic operation always consumes one bit of padding. Decrypting with the initial interval would not give the correct result, `add_with_padding` therefore uses the interval $[min_1 + min_2, max_1 + max_2]$ to achieve correct additions. This interval covers the whole range of results possible with the initial encodings of the operands and has the same length as the size of the two previous intervals combined. A larger interval with still the intial b_{prec} leads to much less precision. We therefore limit the intervals to be the same length, such that the resulting interval is exactly double and we can always keep the original precision by consuming one padding bit. One bit more gives double the encoding points and can therefore encode a twice as large interval with the same precision. This output encoding is done by Concrete internally and does not need to be specified by the user.

We want to be assured in advance to keep the initial precision by consuming one padding bit. The function, therefore, only accepts two ciphertexts that are encoded with the same delta ($delta = max - min + margin$ see encoding section). Thus, the two need to have the same $max - min$ and number of precision bits because the margin is directly dependent on them. Furthermore, they must have the same amount of padding bits. You will produce a compile error if you input other configurations.

Adding without padding is suitable if we know that the operands are from a sub-interval of their encoding. We can then predict that the result can also just be in a sub-interval of the added input intervals. For example, if we are working in $[-10, 10]$, the naive output interval would be $[-20, 20]$. If we know that the inputs are positive, e.g., because they are the results of a squaring operation, the output interval can be restricted to $[0, 20]$. Concrete can then encode the result of the addition with the same precision without using padding bits, since the interval size remains the same.

In order to add two ciphertext without consuming padding, the user has to give a parameter min' to the function which should satisfy that the result of the addition lies inside of the interval $[min', min' + (max_1 - min_1)]$. The interval has the same size as the one from the two operands, but is translated to a position given by min'. We therefore have to know beforehand the range in which the result will lie and then deduce a min'. We can avoid using padding with this method, because we narrow down the interval the result can be in. Concrete then does not have to use the whole possible addition range for the encoding of the result and can thus circumvent using extra precision bits from the padding.

The disadvantage of this method, is that we have to know an interval of the same range as our originals, where the result is going to lie in. If we have a number of operations, with similar outputs this might make sense, but when calculating arbitrary computations the interval will not be known. The user will not be warned when the result is outside of the specified interval and undefined results will be given as output.

The preconditions for `add_with_new_min` are that the two ciphertexts have to be encoded with the same delta (same max-min and number of precision bits). Because we do not use padding bits for additional precision, both intervals need to have the same delta to keep the original precision. If different operands with different deltas are given an error message will be returned. In the end, the delta of the encoding is preserved and the result has the same interval size, precision and padding bits.

3.2.3 Multiplying a Ciphertext by a Constant

Multiplying with a constant in Concrete can be done by doing a "real" homomorphic multiplication or by only modifying the encoding, which has the same effect, but is not actually a homomorphic operation on the ciphertext. The first can only multiply with small integer constants, but preserves the precision and does not use padding. On the other hand, the second can multiply with real constants from any size, but one has to take into account that the encoding interval is enlarged, the rounding intervals therefore get larger, i.e. precision gets worse by $\frac{1}{constant}$. However, padding can be consumed to improve this precision.

Multiplying a constant without using padding can be done by using the function `mul_constant_static_encoder`. Here, the encoding is static, thus the result of the multiplication has to be inside the interval and the resulting encoding is the same as the initial one. Remember, that the LWE ciphertext c encrypting m_0 is defined as $c = (c_0, c_1) = (a * s + m_0 + e, a)$. One multiplies a constant m with both components and obtains the encryption of $c * m = (c_0 * m, c_1 * m)$. If the noise has not grown too much during computations, the last equation can be rounded to the nearest rounding interval edge, which corresponds to the result with a precision of half the margin (see Section 3.1.1). Note, that the noise was also multiplied by the constant and therefore has grown. Nevertheless, the order of magnitude of the noise is still way smaller than the rounding intervals and therefore not yet posing a problem for the correctness, only multiple multiplications will eventually lead to a false rounding.

Multiplication with constants, which are real or lead to a result outside of the interval, can be calculated with the function `mul_constant_with_padding`.

This function does not actually homomorphically compute on the ciphertext, it merely changes the encoding. The basic idea of the procedure is, that the encoding interval is multiplied by the constant m, i.e. $m * min$ and $m * max$. The interval length ($max - min$) therefore also gets m times bigger, but we still have the same number of precision bits, thus the precision decreases by a factor $\frac{1}{m}$. The value of the encoded and encrypted message we want to multiply still keeps its relative position in the encoding interval and therefore after decoding again, the procedure is equivalent to a multiplication with the constant m. However, one can double the precision by consuming a padding bit. The number of possibilities for a binary number is doubling with every bit, equivalently each extra padding bit gives us double the amount of points we can encode and therefore doubles the precision. The function takes the number of padding bits to be consumed as an argument.

3.2.4 Programmable Bootstrapping

With Bootstrapping, we want to reduce the noise of an LWE ciphertext. We refer to Section 2.1.4 for an explanation of the underlying procedure, and simply recall that bootstrapping forces a rounding context, i.e. the plaintext space is collapsed into a discrete set of points. In addition, bootstrapping introduces further precision loss (drift) and needs one padding bit, which is not consumed.

Using the the bootstrapping mechanism, we can not only reduce the encryption noise but also apply a function to the message encrypted in a ciphertext. It allows us to compute any univariate function, implemented as a Look Up Table (LUT). The LUT is automatically generated from a lambda function, making the process transparent to the developer.

In Concrete, the precision of the applied function depends, by default, only on the precision of the input and output encoder. Achieving the target precision requires sufficient capacity in the encryption key setup (bootstrapping key and RLWE secret key). For the bootstrappping key the parameters are the level and logbase. The RLWE secret key is dependent on the number of bits for security and the dimension, which need to be chosen from a predefined set of configurations (`lwe_params.rs`). The user will receive a warning if the target precision is not supported by the underlying keys.

While bootstrapping generally introduces some random *drift* in the results, we can use rounding to ensure determinism. This means the result will first be rounded to a set of $2^{b_{prec}}$ discrete points, then the function is applied, and finally the result is rounded again. In the following, we are going to show a small example, which shows how to calculate the precision of the output, when a function with bootstrapping is applied using an encoding under the rounding context.

In our example, the encoding is defined by $min = 0$, $max = 10$, $b_{prec} = 2$ and $b_{pad} = 1$, which is visualized in Figure 3.1. We apply the function $f(x) = x * x = x^2$ to a ciphertext encrypting $x = 3$. The encoded points, which the program rounds to, are therefore multiples of $3.\overline{3}$, up to $13.\overline{3}$. First, x is rounded to the next interval edge $x_{round} = 3.\overline{3}$. Then, we apply the function via the look up table $f(x_{round})$= $x_{round}^2 = 3.\overline{3}^2 = 11.\overline{1}$ Note that, in this example, the resolution of the LUT is much higher than our encoding (Resolution is of the magnitude of the RLWE polynom) and is therefore not considered. We then again round to the nearest interval edge, which leads to $f(x_{round})_{round}$ = 10, which is the result of the operation.

3.2.5 Multiplying two ciphertexts

Multiplication of two ciphertexts is not a native operation in TFHE, as it is in other schemes. Instead, it is realized by combining other homomorphic operations and programmable bootstrapping. Specifically, multiplication is realized by using $a * b = \frac{1}{4}(a+b)^2 - \frac{1}{4}(a-b)^2$, where the squaring is done via programmable bootstrapping. In the Concrete library, multiplication is implemented by `mul_from_bootstrap` which internally uses `add_with_padding`, `sub_with_padding`, `mul_constant_with_padding` and `bootstrapping_with_function`.

In order to make all the underlying homomorphic operations work, we need to have compatible input encodings. Both encodings need to have the same interval size. The multiplication also needs two ciphertexts encoded with the same number of padding bits. The operation consumes two padding bits, because it uses the addition and subtraction which consume a padding bit. The output encoding gets calculated internally by a function called `_square_divided_by_four`. This function adjusts the encoding to the addition, squaring and division, such that the result will be inside the output encoding.

Chapter 4

Design

Our goal is to implement a neural network which is able to translate German sentences to English while preserving the privacy of the text being translated. In order to approach this task we have to answer multiple design questions. The neural network architecture and internal structure should be suitable to learn long input sequences. The input should be transformed to an embedding which can be efficiently processed by the network. In the next step, we have to implement the neural network under the limitations given by the Concrete FHE library. Specifically, we have to choose suitable encoding parameters, like the precision and range. We have to select from the Concrete library the appropriate functions which let us efficiently compute operations such as matrix multiplication, vector addition and execution of functions on a vector.

4.1 Neural Network Architecture

In order to process sentences which are composed of many letters and spaces, we need to learn on long sequences. Natural Language Processing therefore mostly relies on Recurrent Neural Networks (RNNs), which are a type of neural networks that are able to learn sequences. This is because an RNN has so called recurrent layers. These recurrent layers have a feedback mechanism, which is a loop from the output back to the input of the recurrent layers. This lets the system learn the relationship between sequential data points. For an RNN, the next output is dependent on not just the current data point but also previous ones. Effectively, the RNNs inner state is able to remember the sequence.

4.1.1 LSTM Cell

Today, the most common form of RNN is the Long-Short-Term-Memory (LSTM) cell, which is an improvement to regular recurrent cells. Regular recurrent units only have a hidden state h, which is bounded in the range $[-1, 1]$ by the hyperbolic tangent. For longer sequences the bounded hyperbolic tangent is prone to vanishing gradients. Because of its boundedness the derivative of the hyperbolic tangent starts to be negligible for large values. A negligible gradient leads to the chain rule used in the back-propagation multiplying very small values and therefore quickly being below the computer precision and not able to go back further. The vanishing gradient problem is a common problem in deep networks, because the multiplication chains in the back propagation are so long. In RNN, the loops lead to a similar length of multiplicative chains.

A LSTM cell has, in addition to h, also a cell state c which is unbounded, since it includes (a product with) the old cell state (c_{t-1}), as seen in Equation (4.1). This is helpful for creating a "long-term" memory. If the activation function of a node is unbounded, also for large inputs the derivative is not getting negligible. The cell state is unbounded and has its input passed by a previous cell state which is also unbounded, this leads to the back-propagation being able to pass through these nodes with the gradient staying non-negligible for much longer. This is equivalent with being able to learn also from layers which are very far back. In our case, layers which are many loops back.

$$\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{ih}h_{t-1} + b_{ih}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{ig} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \circ c_{t-1} + i_t \circ g_t \\
h_t &= o_t \circ \tanh c_t
\end{aligned} \tag{4.1}$$

The $\circ$ denotes the Hadamard product, which is the entry-wise multiplication of two matrices. The other operations consist of vector-matrix multiplications, additions of vectors and a function application to a vector.

4.1.2 Translation Task

We use the Multi30k dataset [3], which includes a set of translation mappings from German to English sentences. A neural network internally only calculates with tensors, which are higher-dimensional matrices. We, therefore, have to transform our sentences, which are strings to another representation. We process our sentences word for word and thus we only have to give a tensor representation of each word. This is done using standard dictionaries, for example, the language processing library Spacy can convert the German sentence strings to a sequence of one-hot vectors referencing a dictionary that has 7854 words in the German part and 5893 words in the English. For any word in the dictionary, Spacy returns a one-hot vector corresponding to the index of the word.

These one-hot vector representations have size equal to the amount of words in the dictionary, but vectors of size 5000-8000 are too large to efficiently work with. Hence, a further step in processing a word is creating an embedding. The one-hot index is passed through an embedding layer [1]. The embedding layer is basically a pre-trained Look-Up-Table, which maps every of the 7854 one-hot vectors to a smaller (real-valued) vector of a chosen size, in our case we select a dimension of 400 as a trade-off between compactness and accuracy.

4.1.3 Encoder-Decoder Architecture

For our implementation of the private machine translator, we used an encoder-decoder architecture [4]. In the last years this type of architecture has proven to be the best for sequence-to-sequence predictions [2]. The encoder-decoder architecture is able to directly train on source and target sentences and can handle input and outputs with variable size.

The embedding vector is the input to our recurrent encoder unit. The recurrent unit is composed of two stacked LSTM units. The output of the LSTM units is not used until the input sequence has reached the end-of-sentence token. If reached, the inner state represents the seen sequence and is passed to the output.

This output is then passed through a fully connected layer, which returns logits representing the probability for each entry to be one in the one-hot encoding. If the model trains well this should then be approximately one at one entry and approximately zero at all others. We then apply argmax to get the index of the one-hot entry. The index of the encoded German sequence part is then passed to the decoder. The decoder takes the index and creates a new embedding. We pass the embedded vector through a recurrent unit, existing again of two stacked LSTM units. This recurrent unit learns how to map the embedding to a sequence of English language. At every step of processing done by the recurrent unit an output one-hot vector corresponding to an English word is returned, which ultimately is our translated sentence.

4.2 Privacy-Preserving Translation

We want the machine learning model to be able to translate encrypted vectors. Therefore it should be able to execute a forward pass on encrypted sentences. The model is trained in Pytorch on unencrypted data (Multi30k dataset). After training our model on the data for 30 epochs, we export the weights. We use the plaintext weights and homomorphically operate with them on the encrypted input to achieve a forward pass and therefore a translation of a sentence.

Our goal is to be able to homomorphically evaluate a forward pass on a translation model stored on a server. The encoder embedding is evaluated client-side as it is outside the core network and does not have to be trained over sensitive data. However, the decoder embedding has to be evaluated under FHE as it is nested in the middle of the model. Converting the one-hot indexed output of each step to an English word stored in the dictionary could also be done outside of the model and therefore is feasible to be done by the client. However, in order to conserve bandwidth, instead of sending the one-hot vector, we send a single ciphertext containing the index. This index needs to be computed for the next stage anyway, making this bandwidth optimization effectively free.

In Concrete, one has to consider multiple parameters when computing multiple operations after each other. One has to decide in advance for all the operations which encoding and Concrete operation suits best. The user has to make sure all the operations result inside the output encoding-interval of the specific operation. Also one has to ensure that all the results have the correct encoding for the next operation, which most of the time has to be the same as the second operand. If the computations are longer and operands are used multiple times, it gets very complicated if the encoding intervals change after each operations. Therefore we recommend using Concrete functions which do not change the encoding interval, such that one can specify the same encoding interval for all operands in the beginning and execute all operations with no further adjustments required. With this strategy one has to calculate in advance a range which holds all results.

4.2.1 Matrix Multiplication

One of the most important steps in implementing the LSTM is the matrix-vector multiplication. The vector is an encrypted tensor which represents a word of the sentence that a client wants to have translated privately. We are going to look a the more general form of the matrix-vector multiplication the matrix-matrix multiplication. As we can see from Equation (4.2), the matrix multiplication basically consist of multiple additions of multiplications.

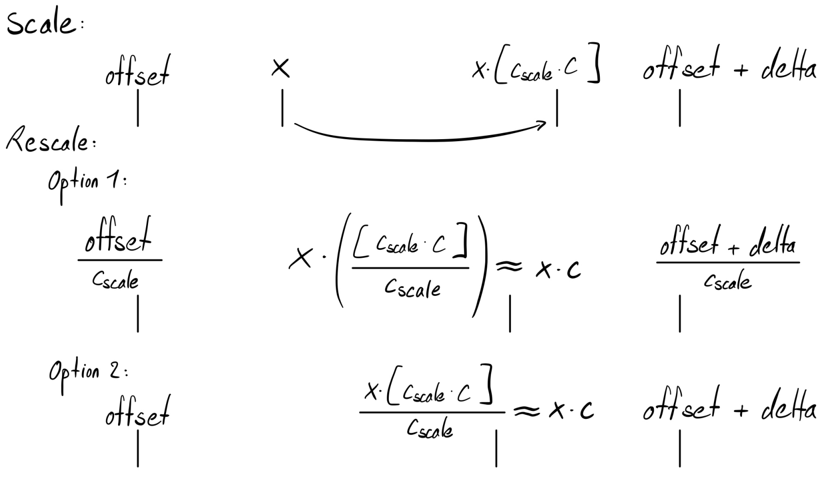
$$c_{ij} = \sum_{j=1}^{k} a_{ik} b_{kj} \tag{4.2}$$

For the multiplications in Equation (4.2) we choose to use the `mul_const_static_encoder`. It requires both operands to have the same encoding interval, but does also return the same encoding. This operation has a restriction though. It can only multiply with integer numbers, which is not the case for the weights in our use case. We decided to still use it but work around it. We use a very large encoding interval for our ciphertext, multiply the constant with a large number and round off the part after the decimal point, which is in our case after the 6th digit. This maps all constants to integers. We can then use `mul_const_static_encoder` to multiply this integer number with our ciphertext and get the result scaled by a known magnitude. We do this with every product and therefore in the end after summation have all our ciphertexts scaled up.

We choose the constant for scaling looking at the trade off between the precision bits needed for the larger interval and the precision loss of the small constant when not scaled enough before rounding. From Table 5.1 we see that we can use at most 50 precision bits for the encoding points of our interval. If we want to have precision up until the 6th digit we should use a constant that is larger than 10^6 this is given with 2^{22}. The interval size is then given by the available precision bits divided by the constant, which is $2^{50}/2^{22} = 2^{28}$. We should then use an interval size which is of size 2^{27} on the positive and negative axis. When unscaling, the range in which the number can lie is $2 * 2^{27}/2^{22} = 2 * 2^5 = 2 * 32$ (on the negative and positive axis). Scaling is a good idea because the constants which in the model really are weights are mostly smaller than 1 and therefore all will be in the interval when multiplied with a large constant. We can also make sure that it is not possible to scale a value outside the encoding interval by setting a minimum and maximum on the weight range by looking in which range 90% of the weights are. Since all multiplications result in the same large interval we can easily continue the matrix multiplication by adding multiplications with the encoding interval which is the same for all. Besides using the function `mul_const_static_encoder`, which returns the same encoding interval, for multiplication, we use `add_with_new_min` for addition and set the new minimum to the old one, which essentially leads to addition with a static encoder as well.

We had two ideas on how to re-scale to the original magnitude. The first one is that we divide the encoding interval after the matrix multiplication, but this leads to different encoding intervals when adding multiple matrix-vector product results, which has to be dealt with again (Figure 4.1).

Secondly, we decided to implement it differently and to undo this scaling at the last step of our LSTM evalution, at the activation function eq. (4.1). By reversing the scaling with a division of the ciphertext(Figure 4.1). When applying the activation function via bootstrapping, we can simply divide the argument by 10. This returns us a look-up-table which is scaled, but gives us the actual result of our computations. The idea behind this method



Scale:
offset
x
$x \cdot \lceil c_{scale} \cdot c \rceil$
offset + delta
Rescale:
Option 1:
$\frac{offset}{c_{scale}}$
$x \cdot \left(\frac{\lceil c_{scale} \cdot c \rceil}{c_{scale}} \right) \approx x \cdot c$
$\frac{offset + delta}{c_{scale}}$
Option 2:
offset
$\frac{x \cdot \lceil c_{scale} \cdot c \rceil}{c_{scale}} \approx x \cdot c$
offset + delta

Figure 4.1: Illustration of the scaling

is to calculate all the matrix-vector multiplications and vector additions with scaled weights and only in the end at the activation function, we rescale to the real value. This is helpful, since we will have a static encoding interval for all operations. The idea applied to the first LSTM equation can be seen in Equation (4.3).

$$\sigma(\frac{1}{\Delta}(\Delta * W_1 * x + \Delta * b_1 + \Delta * W_2 * X + \Delta * b_2) \tag{4.3}$$

4.2.2 Activation Functions

One of the main challenges in previous FHE schemes was the implementation of activation functions. Previous FHE schemes were not able to efficiently implement arbitrary functions, instead being limited to calculating polynomials. However, it has been shown [19] that networks with activation functions which are not polynomial are much better for approximating arbitrary functions, i.e. machine learning.

The TFHE scheme based library Concrete is able to compute any function, which is a leap forward in this problem. The problem although is only partially solved, since the function evaluation algorithm which is based on bootstrapping can only compute with ciphertexts at low precision. In order for the bootstrapping to support a maximum of precision, there are many different variables one can tweak like the LWE key with its gaussian noise standard deviation and mask dimension, the RLWE key with the mask dimension, the noise standard deviation and polynomial size and for the bootstrapping key the level and logbase, as well as the encoding interval size. The precision supported by the bootstrapping is also the precision we can evaluate our function at. Often larger parameter lead to higher precision, but it does not seem to be true in general. The paper [18] shows that the highest precision they could find with mask dimension not higher than 4096 was 7 bits of precision, which is not that much considering that at this size evaluation is already prohibitively expensive in terms of runtime. In our experiments with the unmodified Concrete library, we were able to bootstrap with at most 5 bits of precision using the provided settings. At this precision, bootstrapping remains very efficient. This is in line with recent follow-up work [12] to Concrete's programmable bootstrapping, that identifies about 6 bits of bootstrapping precision as the limit of efficiency. While this follow-up work proposes improvements that increase precision, an implementation of these techniques is not yet available.

One way we can counter the problems arising from limited precision during bootstrapping is using quantization. Quantization is a technique which lets one compute with neural networks at lower bit sizes. Therefore all the tensors are represented and trained not with standard 64 bits, but at lower bit numbers. In Pytorch, quantization to 8-bit integers is natively supported. Therefore, 8 bits would be desirable to have as a minimal precision. However, we have to work with even lower precision because of the bootstrapping restrictions we encountered.

4.2.3 Embedding One-Hot Vectors

In the encoder-decoder architecture, the outputs of the decoder cells are converted into logit vectors by a fully connected layer. Each vector element represents the estimated probability for this index to be the correct index. This logit vector is then converted into a true one-hot vector by applying argmax to identify the index of the largest element. This now represents a single word in the target language and is output as part of the translated sequence. Due to the recurrent nature of the decoder, this output is also passed into the next stage of the network. However, first an embedding is applied, similar to the embedding used to prepare the input sequence for the decoder. These processes have a lot of similar elements which ought to be implemented wisely.

Argmax The fully connected layer takes the output of the recurrent cell and returns a logit vector over which we need to apply argmax to identify the output word. Evaluating argmax over large vectors is computationally expensive in FHE, especially in Concrete, which does not easily allow us to use batching to pack many elements into a single ciphertext. In Concrete, a simple linear scan updating the maximum value and associated index is therefore the most suitable algorithm to find the index of the largest value in the vector.

In order to calculate the maximum value and index, we need to implement both comparisons and conditionally updates. A comparison between two ciphertexts a and b can easily be realized by bootstrapping $a - b$ with $f(x) = 0$ for $x < 0$ and $f(x) = 1$ else. This works reliably even at low precision. Conditional updates, i.e. setting a variable to a or b depending on a binary condition c can be evaluated using multiplexers, i.e. by computing $c * (a - b) + b$. This requires computing a ciphertext-ciphertext product, which requires two bootstrapping operations as described in Section 2.1.5. However, these bootstrapping operations need to maintain much more precision, which makes this approach difficult with the limitations present in current versions of Concrete.

Instead of using multiplications (realized via bootsrapping), one could alternatively use the native CMUX operation in concrete. However, this requires the condition to be an RGSW ciphertext. While it is theoretically possible to convert an LWE ciphertext into an RGSW ciphertext homomorphically, this is expensive and not possible using Concrete's relatively high-level API. As a result, the limits of Concrete currently make realizing the argmax step difficult. While recent follow-up work [12] proposes methods to increase bootstrapping precision, these techniques are not yet currently implemented.

Embedding Embedding one-hot vectors on the decoder side happens in the middle of the model and therefore has to be done under encryption. While evaluating the embedding look-up-table using bootstrapping sounds like a straight-foward approach, the large sizes of the data makes it considerably less straight forward. For example, the embedding has 400 dimensions, so one has to bootstrap 400 copies of the index with 400 different look-up-tables, corresponding to the different dimensions of the embedding. In addition, bootstrapping is also (currently) limited in precision to around 5 bits, where as this would require $\lceil \log(400) \rceil = 9$ bits. While higher-precision versions of bootstrapping are not yet available, one could replace the look-up-table based embedding by a fully connected layer that is trained during model training. A fully connected layer requires only matrix-vector multiplications, additions and activation functions, and is more robust in the context of lower precision bootstrapping.

Chapter 5

Implementation & Evaluation

In this chapter, we briefly describe our implementation before presenting the results of our evaluation.

5.1 Implementation

We re-used a model architecture from the 2019 Microsoft Research Private AI bootcamp [22]. We trained the model for 30 epochs, until we observed a convergence in validation loss (see Figure 5.1). We export the weights and biases to `csv` files for later re-import into the FHE computation.

In Concrete, we reconstruct the weights matrices and bias vectors from the `csv` files generated by the Python-based training. We implemented all the matrix and vector operations needed to rebuild a forward pass on the model. These are also of independent interest, as they are key components of more general machine learning and other applications.

In the Design chapter, we already mentioned the issues arising from Concrete's limited precision. Here, we investigate these limits experimentally. We evaluated a wide selection of parameters and an overview over the available parameters and the supported precision can be found in Table 5.1. To-

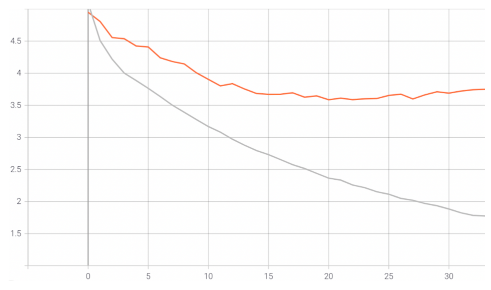


Figure 5.1: Training loss in grey and validation loss in orange.

n	$\log(\sigma)$	p	p_b	Bootstrap (s)	KeyGen (s)	Size (MB)
256	-5	2	0	2.669	759	49
512	-11	8	5	1.78	20161	97
630	-14	11	4	2.05	932	119
650	-15	12	4	5.531	2695	122
688	-16	13	5	5.056	2929	130
710	-17	14	5	3.917	2992	134
750	-18	15	5	3.741	3064	141
800	-19	16	5	2.549	3121	151
830	-20	17	5	2.643	3181	156
1024	-25	22	4	2.639	3116	193
2048	-52	49	3	6.761	45313	385
4096	-105	-	-	-	-	-

Table 5.1: Overview over the precision and performance of bootstrapping in Concrete for different predefined parameter sets using different LWE key sizes n. $\log(\sigma)$ is the log of the standard deviation of the Gaussian error distribution, p is the precision of linear operations in bits, and p_b is the precision of the bootstrapping operation, also in bits. Bootstrap refers to the runtime of the bootstrapping operation, while KeyGen measures the time required to generate the bootstrapping keys, both in seconds. Finally, Size indicates the size of the bootstrapping keys in megabytes. All settings use 128 bit security, base 5 and level 3 (these settings determine how operations are broken down internally). There is a known bug that prevents execution using parameters with $n = 4096$.

wards the end of this thesis, two papers were published which confirmed our observation that only about 6 bits of bootstrapping precision can be achieved in practice [18, 12].

A further problem concerning the bootstrapping was the time it takes to generate the bootstrapping key and their size, as seen in the KeyGen and Size columns of Table 5.1. Generating a bootstrapping key can take several hours for higher precision settings. While Concrete provides functions to save the bootstrapping keys to disk and load it again, we discovered bugs in this functionality that led to computations with de-serialized keys failing. Therefore, we had to refrain from using this functionality and re-generate keys on each run, which made development significantly more tedious.

5.2 Evaluation

We begin our evaluation with micro benchmarks of Concrete's homomorphic operations, to provide a general intuition of the cost of FHE operations in this setting. Then, we evaluate the runtime of our design and compare it briefly to other approaches to deep neural networks in FHE. We conducted our evaluation of Concrete and our design using an AWS `m5.xlarge` instance with 4 cores and 16 GB of RAM.

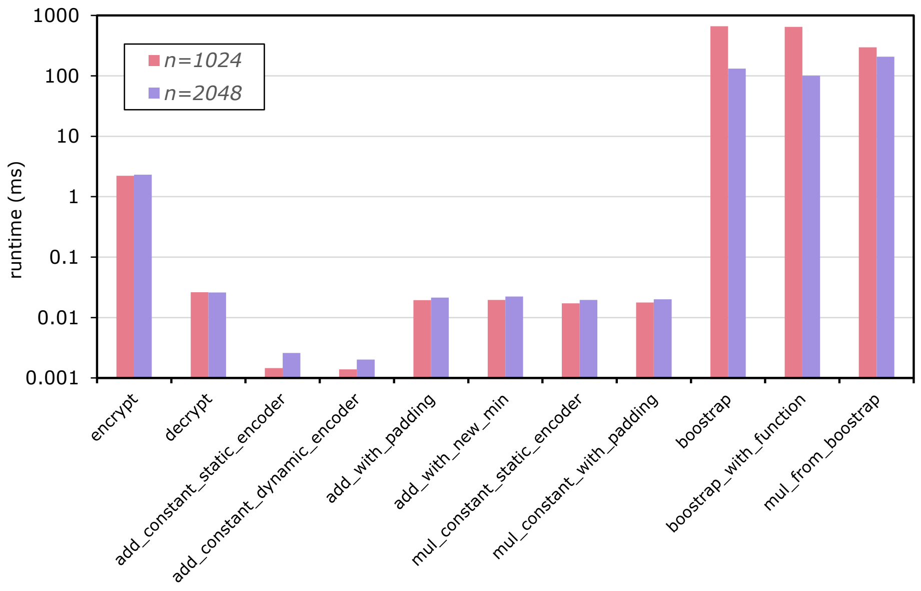


Figure 5.2: Runtime of different operations in Concrete, in milliseconds on a logarithmic scale. $n = 1024$ refers to the setting `RLWE_1024_LWE_512` while $n = 2048$ refers to `RLWE_2048_LWE_512`.

Microbenchmarks We evaluated the operations in the Concrete library by running a series of microbenchmarks. We provide benchmarks for each of the linear operations and bootstrapping based operations we covered in Section 3.2. For the measurements, we used parameters with 128 bits of security, 512-dimensional LWE keys and either $n = 1024$ or $n = 2048$ dimensional RLWE keys. The measurements presented in Figure 5.2 are the average over 100 (linear operations) or 10 (bootstrapping operations) executions of each operation.

We note that encryption is two orders of magnitude slower than decryption, which might be due to the need to generate a large number of random values for the mask and the Gaussian noise. Interestingly, `add_constant_dynamic_encoder` is only slightly faster than `add_constant_static_encoder` even though adding a constant with a dynamic encoder only modifies the encoding without actually performing homomorphic operation as the static version does. The addition of two ciphertexts takes roughly an order of magnitude longer than addition with a constant, which is coincidentally about the same time as that of ciphertext-plaintext (i.e.,ciphertext-constant) multiplications. For all these operations, the size of the RLWE key does not lead to significant differences.

However, for bootstrapping-based operation, we can see roughly an order of magnitude difference between the two parameter sets. As expected, bootstrapping in general takes much longer than the other operations. Bootstrapping without a function or with a function performs the same, proving

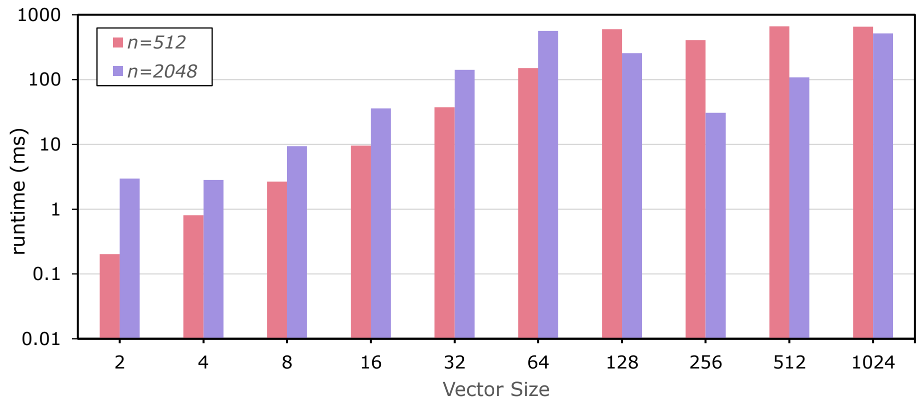


Figure 5.3: Duration of matrix multiplication in milliseconds for different vector/(square) matrix dimensions. n refers to the size of the LWE keys.

Concrete's claim of "free" function evaluation. Oddly, `mul_from_bootstrap` performs similar to standard bootstrapping even though the underlying algorithm internally uses two bootstrapping procedures. It is not clear which, if any, internal optimizations in Concrete, are causing this behaviour.

Matrix-Vector Multiplication The most important operation for implementing a neural network is the matrix-vector multiplication that appears in both fully connected layers and RNN cell functions. It is the most computational intensive besides the activation function and therefore it should be as efficient as possible. For our measurements we used settings with 128 bit security and either a 512 dimensional or 2048 dimensional LWE key. The first setting provides 8 bits of guaranteed precision, which corresponds to PyTorch's natively supported quantization size. Since matrix-vector multiplications between an encrypted vector and a plaintext matrix do not require bootstrapping, the RLWE key size does not matter here. In Figure 5.3, we report the runtimes for a variety of vector, and therefore (square) matrix, sizes. All measurements are the average over ten executions.

We observe some unexpected behavior when we increase the dimension of the matrix, especially for $n = 2048$. For vector sizes above 256, we see unexpected drops in the runtime compared to smaller instances. Since the number of homomorphic operations increases with increasing size and all operations are independently computed, the most plausible reason might be OS or compiler optimizations kicking in, for example AVX512 vectorization or advanced pipelining optimizations.

Bias Addition An other important operation in the evaluation of the neural network is the bias addition. The bias addition is the addition of a network intrinsic bias-weight vector which is unencrypted to an encrypted vector

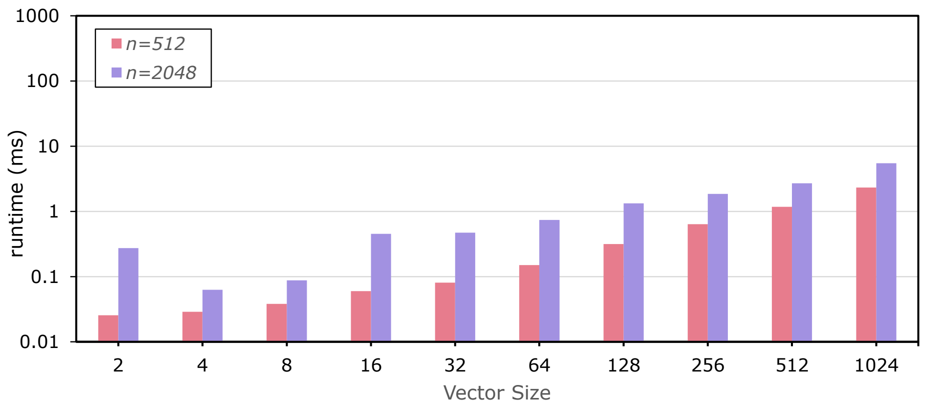


Figure 5.4: Runtime of adding a ciphertext and plaintext vector, in milliseconds, for different vector dimensions. n is the size of the LWE keys.

which results from the weight-matrix multiplication with the input-vector. Illustrated in Figure 5.5, we show the duration of this operation in milliseconds over different vector dimensions. We use the same parameters (128-bit security, LWE key size 512 or 2048) as before and also report the average over 10 executions.

The bias addition is an entry-wise addition and therefore as expected the duration increases according to the dimension.

Encrypted Vector Addition Furthermore, we measure the addition of two vectors consisting of LWE ciphertexts, an operation which appears inside the LSTM cell equations when the result of two matrix-vector products followed by bias additions are added together. The addition was implemented by entry-wise applying `add_with_new_min`. As usual, Figure 5.5 reports the average runtime of ten executions, for 128-bit security and either 512 or 2048-dimensional LWE keys.

Activation Function Finally, we evaluate the performance of applying programmable bootstrapping to a vector to compute a non-polynomial activation function (e.g., tanh as used in LSTM cells). In fig. 5.6, we report the average runtime of ten executions, for 128-bit security. Since bootstrapping uses the RLWE keys, we report results with either 1024 or 2048-dimensional RLWE keys.

Since the bootstrapping is applied entry-wise the time to execute the bootstrapping should increase with the vector size, but oddly, the measurements do not bear this out. Even after verifying our benchmarking setup multiple times, we cannot account for this oddity. However, given that we already saw unusual runtimes with multiple bootstrapping operations in the context

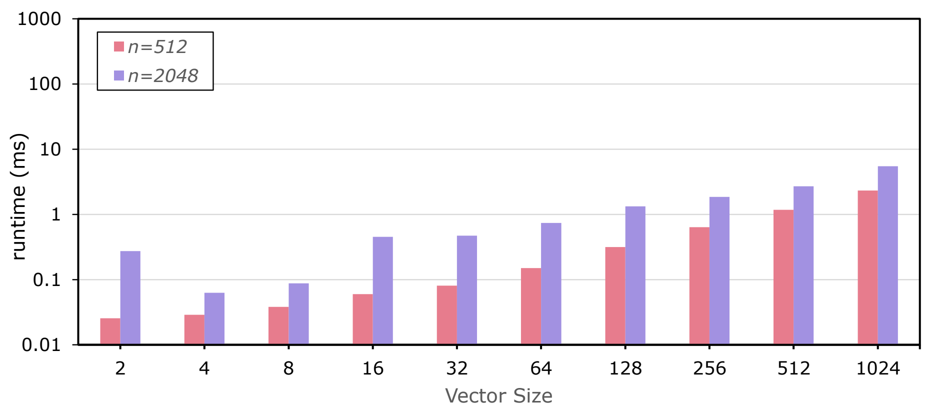


Figure 5.5: Duration of adding two ciphertext vectors in milliseconds for different vector dimensions. n is the size of the LWE keys.

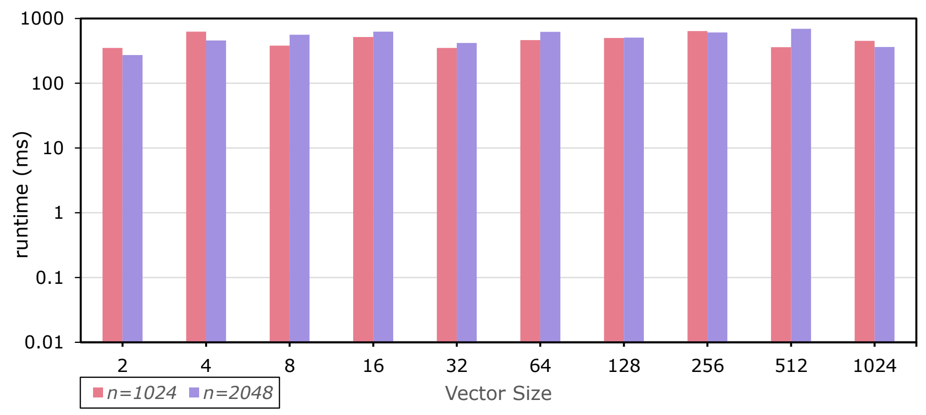


Figure 5.6: Duration of evaluating an activation function on a vector using bootstrappings, in milliseconds for different vector dimensions. Here, n refers to the RLWE key sizes.

of `mul_from_bootstrap`, there might be internal optimizations in Concrete that make subsequent bootstrapping more efficient.

LSTM cell Based on our evaluations, we can now determine the runtime of an entire LSTM cell. Recalling the equations

$$\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{ih}h_{t-1} + b_{ih}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{ig} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \circ c_{t-1} + i_t \circ g_t \\
h_t &= o_t \circ \tanh c_t
\end{aligned} \tag{5.1}$$

	runtime (s)	
Operation	5 bit	8 bit
i_t, f_t, g_t, o_t	0.521	0.688
c_t	118.477	118.477
h_t	118.803	118.969
LSTM Cell	237.801	238.134

Table 5.2: Projected runtime for (the components of) an encrypted LSTM cell (Equation (5.1)). The 5-bit setting represents the highest-precision bootstrapping that is actually possible in current versions of Concrete, while the 8-bit setting would be the target precision, if bootstrapping in this setting were equally precise.

we see that i_t, f_t, g_t and o_t all consist of two matrix-vector products and two bias additions followed by a sum between two encrypted vectors before finally an activation function is applied. c_t differs in that it instead features two component-wise multiplications between encrypted vectors before the products are added together. Finally, h_t requires an activation function followed by component-wise multiplications between encrypted vectors. In our design, all vectors are 400-dimensional and all matrices are 400x400.

In Table 5.2, we provide predicted run-times for the individual components of the LSTM and the overall cell, for parameters that achieve either 5 bits of precision (largest setting where bootstrapping still succeeds) or, as a hypothetical comparison, parameters that would provide 8 bits of precision if bootstrapping were equally precise. We see that it takes just under four minutes to evaluate one LSTM cell, which is many times slower than the plaintext version but still practical for non-interactive settings. We can also see that the vast majority of the time is spent computing c_t and h_t, which require ciphertext-ciphertext multiplications, which in turn requires expensive bootstrapping operations. Recent follow-up work [13] extends TFHE to include non-bootstrapped homomorphic ciphertext-ciphertext multiplications, as seen in other schemes, and could potentially eliminate most of the overhead of our approach.

While our evaluation does not cover an entire end-to-end RNN, it goes significantly further than prior work. For example, the BFV-based RNN cell implemented in [22] consists of only i_t and f_t, uses x^2 as an approximate activation function and takes over 90 seconds to evaluate 5 cells, which would take less than 7 seconds in our system. Since BFV is a levelled scheme, going beyond five cells would require either increasing the parameters to infeasible sizes, or performing the prohibitively expensive BFV bootstrapping procedure. In contrast, our solution is indefinitely composable since all cell outputs are already bootstrapped.

Chapter 6

Discussion

We compare our results to existing work and close with a discussion of the current state of deep machine learning in Concrete and FHE in general.

6.1 Related Work

There are two main branches in the exploration of neural network inference using FHE. The first uses polynomials to approximate ReLU or sigmoid activation functions, as in CryptoNets [23]. These approaches have been extended to support batching techniques [8]. Nevertheless, this line of work has not been able to perform well on deep networks such as ImageNet. On the other hand, there is a branch of researchers working on achieving inference of networks with arbitrary depth by using techniques such as binarized and discretized networks [7].

By using Concrete, we improve on the problem of evaluating activation functions by using the - comparatively - efficient bootstrapping-with-a-function functionality, which at the same time addresses the problem of calculating deep networks by resetting noise.

To compare the performance of our system to other works targeting neural networks, we can have a look at the very recently published DOReN system [21], where the authors used HElib, which is a library based on the BGV FHE scheme, and various optimization techniques to optimize the evaluation of a neural network. Since BGV does not support computing non-polynomial functions over integer plaintexts, DOReN uses binary emulation, i.e. each bit of a number's binary representation is encrypted into its own ciphertext. This requires evaluating complicated arithmetic circuits homomorphically to perform integer arithmetic. Our system is faster for the bootstrapping, application of weights and addition than DOReN at smaller to medium vector sizes, but gets overtaken at large dimensions by DOReN due to their ability to use SIMD batching. More importantly, their overall (non-amortized) time to compute, e.g. something similar to i_t is several orders of magnitude larger. Thus, our Concrete-based solution would be able to perform well also in comparison with the latest developments. What is leaving Concrete in a weaker position, is that it can only use low-precision (limited LUT bitwidth) evaluations of a given function.

6.2 Discussion

This work explores the state-of-the-art in Fully Homomorphic Encryption based privacy-preserving machine learning inferences. We show how the recent introduction of programmable bootstrapping has enabled progress for FHE in neural network inference. We saw that using programmable bootstrapping to evaluate activation functions is an attractive alternative to polynomial approximations, although the current precision limits of 6 bits is too restrictive in practice. During the evaluation, we have seen that Concrete at the current state allows us to evaluate LSTM cells in under four minutes.

While improvements to the limited precision of the bootstrapping have recently been proposed [12], these are yet to be made available in Concrete. Once this happens, we expect that a higher-accuracy version of our system should be a straight-forward extension. Besides improving bootstrapping precision, our project identified other aspects of Concrete, which are hindering the user and could be improved. There is a lack of in-depth documentation, especially regarding the choice of secret keys, and the differences between the various operations offered in the API. We hope to have somewhat filled this gap with our Introduction to Concrete (Chapter 3) and our overview of bootstrapping precision in Table 5.1. It would have also been beneficial to mention in the documentation that about 6 bits of precision is the limit for bootstrapping at feasible time. Further examples would also help the users to get a grip of the library and its possibilities to achieve computations by using FHE-specific approaches. Implementing standard operations like matrix multiplications in Concrete is unnecessarily complex for beginners. We have created implementations of standard operations and plan to make them available as an open-source toolbox for the community.

Besides improving the Concrete library, the developers (Zama) have also announced that they are working towards creating a machine learning compiler targeting Concrete. An FHE compiler converts code in a high-level programming language to FHE, thus the user does not have be familiar with FHE or the library. While this should significantly improve user experiences for machine learning tasks, general-purposes applications will have to continue to use Concrete directly for the foreseeable future and will therefore continue to benefit from usability improvements to the library.

Bibliography

[1] Embedding — pytorch 1.9.0 documentation. https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html. (Accessed on 06/17/2021).

[2] Encoder-decoder recurrent neural network models for neural machine translation. https://machinelearningmastery.com/encoder-decoder-recurrent-neural-network-models-neural-machine-translation/. (Accessed on 06/17/2021).

[3] multi30k/dataset: Multi30k dataset. https://github.com/multi30k/dataset. (Accessed on 06/17/2021).

[4] Seal/seq2seqmodel.py at privateai2019 · privateai-group1/seal. https://github.com/privateai-group1/SEAL/blob/PrivateAI2019/seq2seqmodel.py. (Accessed on 06/30/2021).

[5] Concrete documentation. https://concrete.zama.ai/, 2020. Accessed: 2021-6-24.

[6] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. nGraph-HE2: A High-Throughput framework for neural network inference on encrypted data. 12 August 2019.

[7] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Advances in Cryptology – CRYPTO 2018*, volume 10993 of *Lecture Notes in Computer Science*, pages 483–512. Springer, 2018.

[8] Alon Brutzkus, Oren Elisha, and Ran Gilad-Bachrach. Low latency privacy preserving inference. *CoRR*, abs/1812.10659, 2018.

[9] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. Cryptology ePrint Archive, Report 2018/421, 2018.

[10] Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. CONCRETE: Concrete operates on ciphertexts rapidly by extending TfhE. In *WAHC 2020 – 8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 15 December 2020.

[11] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. Technical report, Zama, 15 October 2020.

[12] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe. Cryptology ePrint Archive, Report 2021/729, 2021. https://eprint.iacr.org/2021/729.

[13] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe. Cryptology ePrint Archive, Report 2021/729, 2021.

[14] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–156, New York, NY, USA, 8 June 2019. ACM.

[15] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In Maria Florina Balcan and Kilian Q Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 201–210, New York, New York, USA, 2016. PMLR.

[17] Jakub Klemsa. Setting up efficient TFHE parameters for multivalue plaintexts and multiple additions. https://eprint.iacr.org/2021/634.pdf, 2021. Accessed: 2021-5-25.

[18] Jakub Klemsa. Setting up efficient tfhe parameters for multivalue plaintexts and multiple additions. Cryptology ePrint Archive, Report 2021/634, 2021. https://eprint.iacr.org/2021/634.

[19] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993.

[20] Dorian Lynskey. ’alexa, are you invading my privacy?’ – the dark side of our voice assistants. *The Guardian*, 9 October 2019.

[21] Souhail Meftah, Benjamin Tan, Chan Mun, Khin Aung, Bharadwaj Veeravalli, and Vijay Chandrasekhar. Doren: Towards efficient deep convolutional neural networks with fully homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, PP:1–1, 06 2021.

[22] Travis Morrison, Bijeeta Pal, Sarah Scheffler, and Alexander Viand. Private outsourced translation for medical data. Technical report, December 2019.

[23] Pengtao Xie, Misha Bilenko, Tom Finley, Ran Gilad-Bachrach, Kristin E. Lauter, and Michael Naehrig. Crypto-nets: Neural networks over encrypted data. *CoRR*, abs/1412.6181, 2014.