Encrypted Image Classification with Low Memory Footprint using Fully Homomorphic Encryption*

Lorenzo Rovida[1] Alberto Leporati[1]

[1] *Department of Informatics, Systems and Communication, University of Milan-Bicocca*
Viale Sarca 336, Edificio U14, 20126 Milano, Italy
{lorenzo.rovida, alberto.leporati}@unimib.it

Abstract

Classifying images has become a straightforward and accessible task, thanks to the advent of Deep Neural Networks. Nevertheless, not much attention is given to the privacy concerns associated with sensitive data contained in images. In this study, we propose a solution to this issue by exploring an intersection between Machine Learning and cryptography. In particular, Fully Homomorphic Encryption (FHE) emerges as a promising solution, as it enables computations to be performed on encrypted data. We, therefore, propose a Residual Network implementation based on FHE which allows the classification of encrypted images, ensuring that only the user can see the result. We suggest a circuit which reduces the memory requirements by more than 85% compared to the most recent works, while maintaining a high level of accuracy and a short computational time. We implement the circuit using the well-known CKKS scheme, which enables approximate encrypted computations. We evaluate the results from three perspectives: memory requirements, computational time and calculations precision. We demonstrate that it is possible to evaluate an encrypted ResNet20 in less than five minutes on a laptop using approximately 15GB of memory, achieving an accuracy of 91.67% on the CIFAR-10 dataset, which is almost equivalent to the accuracy of the plain model (92.60%).

1 Introduction

In recent years, Neural Networks have demonstrated impressive capabilities across various tasks, and they are progressively becoming a fundamental block in delivering many online services.

Nevertheless, not enough attention is given to user privacy when these models are deployed as *Machine Learning as a Service* (MLaaS). In order to run inferences, service providers need to access plain user data, and this creates issues that are not easy to deal with. Today, data is protected by laws, like the General Data Protection Regulation (GDPR) [32] in Europe, but currently there is no *practical* protection of data, apart from laws. We propose a solution by exploring an intersection between Machine Learning and cryptography. In particular, we implement a well-known Convolutional Neural Network (CNN) called Residual Neural Network (ResNet), using a particular cryptographic

*Preprint of an article published in International Journal of Neural Systems, DOI: 10.1142/S0129065724500254

scheme that allows for operations to be performed on encrypted data. The scheme used is an example of so-called *Homomorphic Encryption* (HE) schemes.

1.1 Convolutional Neural Networks

CNNs have emerged as a powerful class of deep learning models designed to handle data with a grid-like structure, especially images, videos, time-series and so on. They were first introduced in 1989 by LeCun et al. [25] as a method to recognize handwritten digits.

The main idea behind these networks is to capture spatial dependencies, local patterns and features from data, making them highly effective for the image classification task. In particular, in this paper, we propose a HE-based version of a Residual Neural Network called ResNet20 [18]. This family of CNNs is based on Residual Connections (also known as Skip Connections), a shortcut in a Neural Network that allows the gradient to bypass one or more layers during training. It adds the original input of a layer to its output, creating a Residual Block and allowing for the network to learn and optimize the parameters effectively. Residual Blocks indeed prevent the issue of the vanishing gradient problem.

1.2 Homomorphic Encryption

Rivest et al. [34] introduced in 1978 the concept of *Privacy Homomorphism*: they noticed that the public-key scheme RSA [33] was *homomorphic* with respect to the product operation (i.e. in a basic RSA implementation, the product of two ciphertexts is equal to the encryption of the product of the corresponding two plaintexts). They theorized the idea of a scheme that could perform the operations of addition and multiplication between ciphertexts, so that a server could perform computations without having access to plain user data. This notion was put into practice in 2009, when Gentry [14] exploited the fact that ideal lattices provide both additive and multiplicative homomorphisms, and built a cryptosystem over these structures, creating the first HE scheme.

However, there is an issue about managing the growing noise: each multiplication increases the level of noise in the resulting ciphertext; when the noise grows too much, it makes the ciphertext undecryptable. The noise could be removed by decrypting the ciphertext, but this, in general, is not always feasible (the server does not have the secret key needed to do that). To address this issue, Gentry introduced the concept of *bootstrapping*, which is an operation that refreshes the level of noise inside a ciphertext by embedding the ciphertext into a fresh ciphertext, and by homomorphically evaluating the decryption circuit using an encrypted version of the secret key (sometimes called the *bootstrapping key*). In literature, it is possible to find two approaches to this matter:

- **Fast bootstrapping**: after each multiplication, the ciphertext is bootstrapped. The reference schemes are the *Fast Fully Homomorphic Encryption over the Torus* (TFHE) [11] scheme and the Dusan-Micciancio (FHEW) [12] scheme.

- **Leveled**: each ciphertext has enough "space" to handle a fixed number of multiplications; then, bootstrapping must be performed. These schemes require more memory since ciphertexts and keys are larger. The reference schemes for integer arithmetic are the Brakerski-Fan-Vercauteren (BFV) [13] scheme and the Brakerski-Gentry-Vaikuntanathan (BGV) [7] scheme. For what concerns approximate real numbers arithmetic, the reference scheme is the Cheon-Kim-Kim-Song (CKKS) [10] scheme.

When a scheme is able to evaluate any circuit of any depth, we use the term *Fully Homomorphic Encryption* (FHE).

1.3 Related works

Inference from encrypted images based on CNNs has been object of study since few years. The first generation of CNNs based on HE, called *HE-friendly*, is characterized by networks based on polynomial – in most cases linear or square – activation functions, as a consequence of the limited set of arithmetic operations in HE (i.e., only additions and multiplications).

One of the first successful attempts was CryptoNets, by Gilad-Bachrach et al. [15] in 2016, which was able to get 99% of accuracy on the MNIST dataset using a simple CNN. In 2020, Al Badawi et al. [2] proposed a CNN based on RNS-CKKS accelerated by GPU, achieving 77.55% accuracy on CIFAR-10 dataset [22].

The main issue about these HE-friendly networks is that they do not implement non-linear activation functions, and this implies lower performance on more difficult tasks. Recent developments on FHE schemes, especially on bootstrapping techniques [6], are enabling practical evaluations of deeper and more complex circuits, meaning that is it possible to use existent and pre-trained networks, instead of building and training new networks specifically for HE computations. The problem of non-linear functions is tackled using polynomial approximations.

For this second generation of networks, we find two approaches: High-throughput Networks and Low-latency Networks. The former work on many images simultaneously, minimizing the amortized inference runtime. These approaches, though, work well when dealing with big chunks of images, and not with single images.

Low-latency networks are tackled in two different ways. The first one is to use the TFHE scheme. A recent work by Benamira et al. [5], in 2023, presents an architecture that evaluates an encrypted image in approximately nine minutes, achieving a performance of 74.1% on the CIFAR-10 dataset, requiring less than 1GB of memory.

The second approach is to use the CKKS scheme. Lee et al. [26] proposed in 2022 a framework able to obtain a high level of accuracy (91.31% on CIFAR-10) in half an hour of computations. Their convolution approach follows the work of Juvekar et al. [19], and we will refer to it as *Vector Encoding*. Kim et al. [21] introduced, one year later, an open-source framework that is able to achieve a classification with a level of accuracy of 92.04% on the CIFAR-10 dataset in approximately six minutes. Their key point is in the encoding process, which enables fast convolutions based on on polynomial multiplications (which can, indeed, be interpreted as convolutions) although this approach is characterized by a high memory usage (100GB).

Vector Encoding [19, 26]: this approach requires k^2 (where k is the size of the kernel) ciphertexts representing different rotations of the input image. Each ciphertext is multiplied with the corresponding kernel elements for each channel, leading to $k^2 \cdot c$ multiplications, where c is the number of channels. This approach requires $k^2 + c$ rotations and rotation keys.

Coefficient Encoding [21]: this approach encodes an input vector directly as polynomial coefficients (i.e., *Coeff* encoding, see Section 1.4.2), and convolutions are performed in a single step by multiplying two polynomials. Nevertheless, many rotations keys are required to extract the resulting coefficients, and this implies a high memory usage.

1.3.1 Our contribution

In our proposal, we utilize the CKKS scheme to implement a trained ResNet20. Differently from the most recent works [26, 21], we minimize the memory requirements. De Castro et

Table 1: *Comparison of the most recent solutions for Low-Latency CIFAR-10 HE-based CNN inference. * denotes that the authors did not provide information about the memory usage, we thus assume the RAM capacity of their machine*

Proposal	Scheme	Accuracy	Runtime	Processor	Threads	Memory
Lee et al. [28]	CKKS	90.67%	10602s	Xeon Platinum 8280 CPU	64	512 GB*
Lee et al. [26]	CKKS	91.31%	2271s	Threadripper PRO 3995WX	1	512 GB*
Benamira et al. [5]	TFHE	74.10%	570s	i7-8650U	4	0.8 GB
Kim et al. [21]	CKKS	92.04%	255s	EPYC 7402P	1	100 GB
(proposed)	CKKS	91.53%	260s	M1 Pro	1	**15.1 GB**

al. [8] shown that memory is currently the main bottleneck to be addressed in FHE circuits, although most of the works do not consider its impact when building FHE solutions. We thus introduce an efficient way to perform convolutions, namely *Optimized Vector Encoding*, requiring only five rotation keys. Refer to Figure 1 and Table 1 for a summarization of the most recent related works, compared to our proposal.

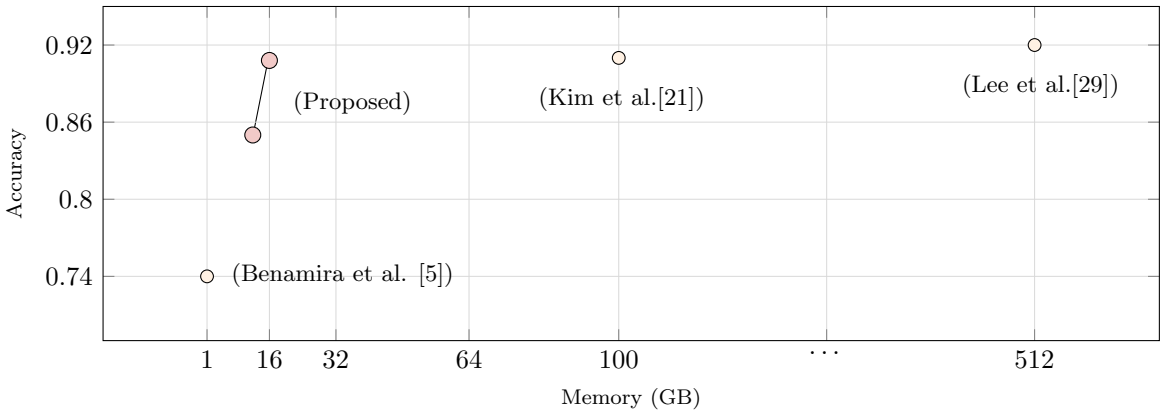

Figure 1: *Accuracy and memory usage of existent literature work and our proposal*

We also propose to use Chebyshev Polynomials in order to approximate the ReLU activation function that, differently from the composable Minimax approximation [27], gives more flexibility in terms of depth and precision.

1.4 Preliminaries

We first introduce and review some essential preliminaries concerning Fully Homomorphic Encryption and, in particular, CKKS.

1.4.1 Mathematical fundamentals

$\mathbb{Z}$, $\mathbb{R}$, and $\mathbb{C}$ represent the sets of integers, real numbers, and complex numbers, respectively. Ciphertexts are denoted using bold letters. The ring of polynomials with integer coefficients is represented as $\mathbb{Z}[X]$, and we define $\mathcal{R} := \mathbb{Z}[X]/(X^N+1)$ as the ring of polynomials modulo $X^N + 1$, where N is a power of two.

1.4.2 The CKKS scheme

The original CKKS [10] scheme takes as input a so-called *cleartext* $v \in \mathbb{C}^{\frac{N}{2}}$, and it first encodes it into a polynomial in $\mathcal{R}$. It is possible to encode it in two different ways:

- *Coeff:* the vector v is encoded directly as polynomial coefficients. It allows to pack more values ($2 \cdot \frac{N}{2}$) in a single ciphertext, but negacyclic convolutions (i.e. the product of two polynomials in $\mathcal{R}$) does not result in point-wise multiplication between coefficients.

- *Slots:* the vector v is subject to a canonical embedding $\sigma^{-1} : \mathbb{C}^{\frac{N}{2}} \to \mathbb{Z}[X]/(X^N + 1)$ which halves the number of available slots in a ciphertext[1], but negacyclic convolutions in $\mathcal{R}$ will result in a Hadamard product in the original input space $\mathbb{C}^{\frac{N}{2}}$. In other words, this encoding process enables point-wise (or slot-wise) multiplications between ciphertexts.

After the encoding, coefficients are scaled by a factor Δ that controls the precision of computations.

Remark. The scheme works with polynomials in $\mathbb{Z}[X]/X^N+1$, with N being a power of two, because they allow for fast computations using the Number Theoretical Transform (NTT), which allows to perform multiplications with complexity $\mathcal{O}(n \log(n))$ instead of the basic modular polynomial multiplication, which has a complexity $\mathcal{O}(n^2)$. NTT is a variant of the Fast Fourier Transform (FFT) that operates in a Galois Field $\mathbb{F}_{p^n}$ (sometimes referred to as $\mathrm{GF}(p^n)$) with p being a prime number, and $n \in \mathbb{Z}^+$.

In particular, we use the Residual Number System (RNS) variant of the CKKS scheme. It works on Double Chinese-Remainder-Transform (DCRT) polynomials. A DCRT polynomial is a large polynomial factorized in smaller ones using CRT and transformed in the NTT space, for performance reasons.

After the encoding, the plaintext polynomial is encrypted using a variant of the Learning with Errors (LWE) [31] hard problem. In particular, the ring-variant (RLWE), which has been shown [30] to be as hard as the worst-case lattice problems, and therefore resistant to quantum attacks.

Informally, we take a RLWE sample $a, b \in (\mathbb{Z}_q[X]/(X^N + 1))^2$ – notice that each of a, b is a *pair* of polynomials – such that a is uniformly sampled and $b = a \cdot s + e$ for a small e. The ciphertext $\mathbf{c}$ encrypting the input polynomial $m(X)$ is thus obtained as follows:

$$\begin{aligned} \mathbf{c} = (b', a') &= v \cdot (b, a) + (m + e_0, e_1) \\ &\in (\mathbb{Z}_q[X]/(X^N + 1))^2 \end{aligned}$$

An important feature of RLWE-based cryptosystems is that it is possible to pack multiple values in a single polynomial, leading to a Single Instruction Multiple Data (SIMD) computational paradigm, which unlocks parallel computations.

Our proposal is based on the OpenFHE implementation [1] of the RNS-CKKS [9] scheme, which allows to perform operations on encrypted $\mathbb{R}^{\frac{N}{2}}$ vectors. Computations are approximate, meaning that:

$$a = Decrypt(Encrypt(a)) + \varepsilon$$

where ε is a small error introduced by different factors (by the rounding in encoding process, by the scaling factor and because of RLWE encryption) which grows when performing

[1] Refer to [10] for a more detailed explanation of this encoding process

operations. The available operations are addition ($c_1 \oplus c_2$), multiplication ($c_1 \otimes c_2$) and rotation $\text{ROT}_i(c)$ (when $i > 0$, coefficients are rotated to the left, to the right otherwise). In case of multiple rotations on the same ciphertext, the Hoisted Automorphisms technique [16] $\text{FASTROT}_i(c)$ is used.

Notice that additions and multiplications can also be evaluated between ciphertexts and plaintexts.

1.4.3 Moduli chain and Bootstrapping

Ciphertexts are encrypted using a large modulus Q, which is constructed as a moduli chain $Q = q_0 \cdot q_1 \cdot \cdots \cdot q_\ell$. In our case, ℓ represents the depth of the circuit, i.e. the number of multiplications that can be performed on a ciphertext. When performing a multiplication, the scale of the resulting ciphertext is squared to Δ^2, meaning that a rescaling procedure is required. This operation brings back the scale to Δ, but it *consumes* one level, meaning that the modulus of the ciphertext is reduced from Q to Q/Δ. Since the q_i are close to Δ, rescaling can be seen as removing an element of the moduli chain Q. Intuitively, when a ciphertext reaches its smallest possible modulus $Q = q_0$ (therefore, no further rescaling is applicable), it needs to be bootstrapped. We use the bootstrapping [4] procedure introduced by Boussuat et al. [6]. This operation is, in general, the most complex operation in FHE circuits.

Rotations need a Key Switching procedure, which requires a key for each rotation index. We use the Hybrid Key Switching (HKS) [17, 20] procedure, in which a rotation key (sometimes referred to as Automorphism Key or Galois Key) requires approximately

$$2 \cdot \ell \cdot d_{num} \cdot N \cdot \left(1 + \frac{1}{d_{num}}\right) \tag{1}$$

bits, where ℓ is (roughly) the depth of the circuit, d_{num} is the number of digits in HKS, and N is the ring dimension. The number of rotation keys is the main factor to take into account when analyzing the memory requirements, since they are the largest objects in a RNS-CKKS based circuit.

1.4.4 Chebyshev Polynomials

Chebyshev polynomials are a set of orthogonal polynomials that are widely used for approximating non-linear functions over $[-1, 1]$. The Chebyshev Polynomials of the first kind are defined as follows:

$$T_0(x) = 1$$
$$T_1(x) = x$$

Then, recursively:

$$T_n(x) = 2x\,T_{n-1}(x) - T_{n-2}(x)$$

These polynomials, of degree n, have n different roots in $[-1, 1]$, called the Chebyshev Roots. It can be shown (refer to Trefthen [36] for a detailed introduction) that they are distributed as:

$$x_k = \cos\left(\frac{k}{n}\pi\right) \quad \text{with } 0 \leq k < n$$

By using these roots, instead of a set of uniformly distributed values, the oscillation and divergence of high-degree polynomial interpolants near the endpoints of the interval (sometimes known as the Runge Phenomenon), is minimized. In our proposal, these polynomials are used in approximating the ReLU activation function and the modular reduction performed in the boostrapping procedure.

2 Circuit overview

We now present the FHE-based circuit that implements ResNet20.

2.1 Ciphertexts Depth

The multiplicative depth of a ciphertext is distributed as follows:

- 1 level consumed for the Convolutional Layer and the Batch Normalization.

- 5, 6 or 7 levels, depending on the desired precision, consumed for the evaluation of the Chebyshev Polynomial approximating ReLU.

After that, we refresh the level of the ciphertext via the bootstrapping procedure (refer to Section 5 of Boussat et al.[6] for a detailed exploration):

- 1 level consumed after the `ModRaise` procedure.

- 4 levels consumed by the `CoeffToSlot` procedure, i.e. the homomorphic version of the encoding.

- 5 levels consumed by the Chebyshev approximation of $g_0(x)$, that is the base case of the modular reduction iterative procedure (with r equal to the number of iterations):

$$g_0(x) = \frac{1}{\sqrt[2^r]{2\pi}} \cos\left(\frac{2\pi}{2^r}\left(x - \frac{1}{4}\right)\right)$$

- 3 levels consumed by the double-angle iterations used to conclude the evaluation of the homomorphic modular reduction:

$$g_{i+1}(x) = 2\, g_i^2 - \left(\frac{1}{\sqrt[2^r]{2\pi}}\right) \text{ with } 0 \leq i < r$$

- 4 levels consumed by the `SlotToCoeff` procedure, i.e. the homomorphic version of the decoding.

2.2 Residual Networks Structure

ResNet20 is a particular type of Residual Network, constructed as shown in Figure 2.

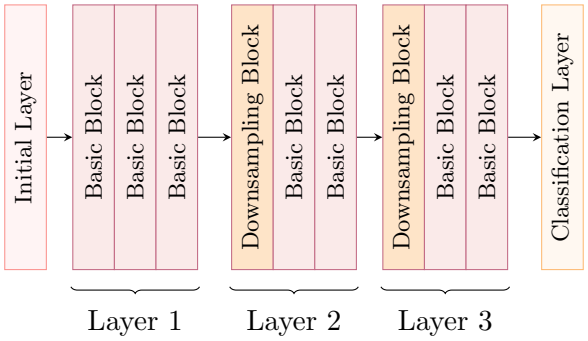

Figure 2: *ResNet20 high level structure*

The network is composed of five layers:

• Initial Layer: takes as input a RGB image I represented as a tensor $\mathbb{R}^{3\times32\times32}$ and evaluates a Convolutional Layer, a Batch Normalization and the ReLU function; it returning a $\mathbb{R}^{16\times32\times32}$ tensor.

• Layer 1: this layer is composed of three Basic Block, each of them built as shown in Figure 3. Each block works with $16 \times 32 \times 32$ input tensors, Convolution Layers have a padding of $\{1, 1\}$, a kernel width $k = 3$, and a stride of $\{1, 1\}$ The output is a tensor $\in \mathbb{R}^{16\times32\times32}$.

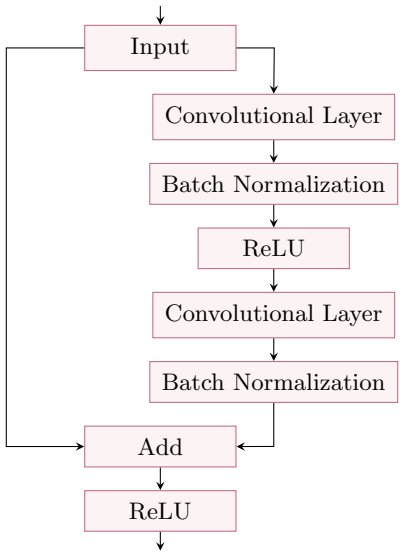

Figure 3: *Structure of the Basic Block in ResNet20*

• Layer 2: this layer performs a downsampling on the input image by performing the first convolution with a stride of $\{2, 2\}$. This halves the width of the $n_{out} = 32$ channels. As a result, the output of the downsampling block is a tensor $\in \mathbb{R}^{32\times16\times16}$. Then, two Basic Blocks follow.

• Layer 3: as before, there is another downsampling at the beginning of this layer. In this case $n_{out} = 64$, so the output tensor $\in \mathbb{R}^{64\times8\times8}$. After that, two Basic Blocks complete the layer.

• Classification Layer: an Average Pooling Layer is used to extract features from each channel. This is followed by a Fully Connected Layer with 10 output neurons, each corresponding to a specific class. The classification decision is made by selecting the neuron with the highest output score.

The size of the ciphertexts will change through the evaluation of the circuit: at Layer 1, ciphertexts have $s = 16384$ slots, then s is reduced to $s = 8192$, and so on (see Table 2). This allows to perform computations on smaller ciphertexts, which are faster.

Table 2: *Image shapes in different ResNet20 layers*

Layer	Channels (c)	Size (w)	Total values ($c \cdot w^2$)
1	16	32	$16384 = 2^{14}$
2	32	16	$8192 = 2^{13}$
3	64	8	$4096 = 2^{12}$

2.3 Optimized Vector Encoding

We propose an optimized version of the Vector Encoding procedure that, in case of odd-sized kernels with $\{1, 1\}$ padding, uses a constant number of five rotation keys. Notice that the number of rotation keys does not depend on the size of the kernel k^2 or on the number of channels c, while in previous implementations [19] the number of rotation keys was $(k^2 - 1) + c$. This allows to drastically reduce the memory footprint of the rotation keys.

2.3.1 Incorporating the Batch Normalization

The Optimized Vector Encoding procedure allows to evaluate both the Convolutional Layer and the following Batch Normalization layer at the same time. When a trained ResNet is evaluated, all the Batch Normalization layers use precomputed values of mean and variance, making it, de facto, an affine transformation. The original formula of the Batch Normalization is the following:

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}(x) + \varepsilon}} \cdot \gamma + \beta$$

Since we consider the inference process, the value of the mean $\mathrm{E}[x]$ and the variance $\mathrm{Var}(x)$ are not computed (instead, constant values are used), and we can thus re-write the equation as follows:

$$\begin{aligned} y &= \frac{x - E}{V} \cdot \gamma + \beta \\ y &= \frac{\gamma}{V}\, x - \frac{E\gamma}{V} + \beta \\ y &= Ax + b \quad \text{with } A = \frac{\gamma}{V},\ b = \frac{E\gamma}{V} + \beta \end{aligned}$$

Now, since Convolutional Layers can be seen as affine transformations, they can be combined with Batch Normalization, consuming only one multiplicative depth.

2.3.2 Applying the kernel

We introduce the kernel application along with an example, in particular we want to replicate the convolution shown in Fig. 4. Notice that all the convolutions (excluding the ones performed in downsampling) are performed with $\{1, 1\}$ padding.

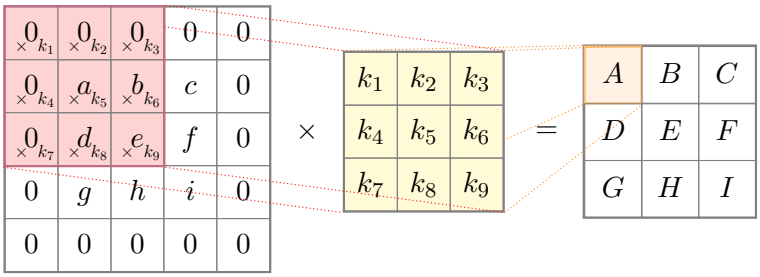

Figure 4: *Convolution on a width $w = 3$ channel with padding $= \{1, 1\}$ and a kernel of size $k = 3$*

To begin with, the application of the kernel to the image requires a ciphertext containing all the input channels next to each other, from the first to the last one (Figure 5).

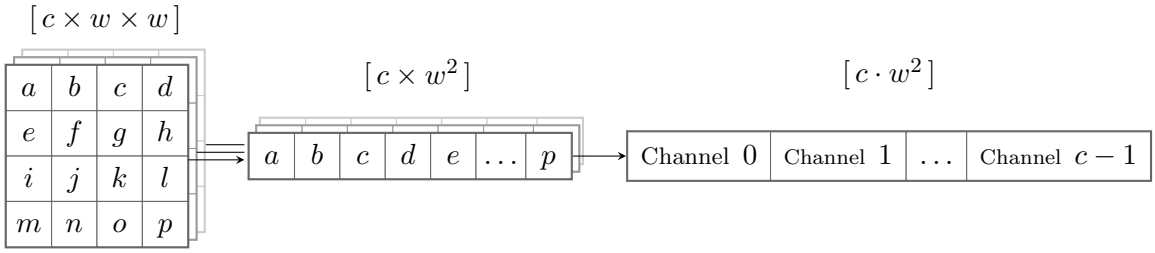

Figure 5: *The reshaping process applied before the encryption*

More formally, let $I \in \mathbb{R}^{c \times w \times w}$ be the input feature map with c channels of dimension $w \times w$. The slots of the ciphertext $\mathbf{c}$ encrypting I are

$$\mathbf{c}_{(w^2 \cdot i)+(w \cdot j)+z} \leftarrow I_{i,j,z}$$

where $0 \leq i < c$ indexes the channel and $0 \leq j, z < w$ the spatial position. Notice that the values of c and w change during the evaluation of the circuit depending on the considered ResNet20 layer, as shown in Table 2. The evaluation of the convolution consists of three steps.

(1) **Preparing the rotations**: We start by considering the following proposition:

Proposition. *All possible rotations of a ciphertext can be obtained with a single rotation key, given an index that is a generator for the ring $\mathbb{Z}_{\frac{N}{2}}$.*

For instance, we can obtain any rotation r of a ciphertext $\mathbf{c}$ by applying r times $\text{ROT}_1(\mathbf{c})$, and this only requires one rotation key. Of course this approach is not suggested since it is extremely time-consuming. Instead, we propose a hybrid approach by finding *possible combinations* of rotation indexes.

Recall our objective, that is to create nine (k^2) ciphertexts which align the nine required values for each filter application. Considering the example in Figure 4, we want to obtain

all the required rotations like is shown in Figure 6, where the first column contains the elements needed for the first convolution, the second column contains the elements needed for the second convolution, and so on.

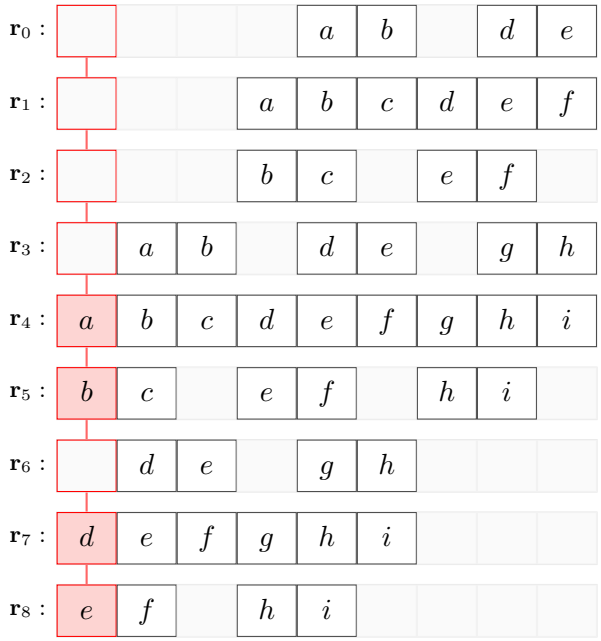

Figure 6: *Rotations required for the application of a filter on the input feature map presented in Figure 4. The red column represents the elements used in the first filter application*

In this example, eight keys would be required to store the required rotation indexes (i.e., $\{-6, -5, -4, -1, +1, +4, +5, +6\}$), but this set can be reduced to $\{1, -5, +5, +1\}$ by noticing that some of the elements can be written in terms of these four (e.g., $6 = 5 + 1$). Moreover, this method does not require more rotations, since we simply compose existing rotations.

In general, the kernel operates on slices of the feature map that are stored in different rows. Considering that the ciphertext encodes a bi-dimensional feature map in row-major order, we need keys to rotate "horizontally" inside the slice and "vertically" to rotate into another slice.

Given the feature map size w, and the padded size $w' = w + 2$, we create keys to rotate horizontally for indexes 1 and -1 (right and left), and for indexes w' and $-w'$ (below and above) in order to rotate vertically. We thus align the nine elements required for the filter application as follows:

- $\mathbf{r}_0 \leftarrow \textsc{Rot}_{-1}(\mathbf{r}_1)$
- $\mathbf{r}_1 \leftarrow \textsc{FastRot}_{-w}(\mathbf{r}_4)$
- $\mathbf{r}_2 \leftarrow \textsc{Rot}_{+1}(\mathbf{r}_1)$
- $\mathbf{r}_3 \leftarrow \textsc{FastRot}_{-1}(\mathbf{r}_4)$
- $\mathbf{r}_4 \leftarrow \mathbf{c}$
- $\mathbf{r}_5 \leftarrow \textsc{FastRot}_{+1}(\mathbf{r}_4)$
- $\mathbf{r}_6 \leftarrow \textsc{Rot}_{-1}(\mathbf{r}_7)$

- $\mathbf{r}_7 \leftarrow \text{FastRot}_w(\mathbf{r}_4)$
- $\mathbf{r}_8 \leftarrow \text{Rot}_{+1}(\mathbf{r}_7)$

The general procedure is presented in Algorithm 1. In particular, the first for-loop creates the vertical rotations, rotating by w and $-w$. On the other hand, the second loop creates the other rotations (rotating by -1 and 1), using the previously generated ciphertexts.

The number of required rotations for this phase remains $k^2 - 1$, but the number of Automorphism Keys is constant and equal to four, whereas in previous works[26] based on Vector Encoding convolutions, the number of rotation keys is equal to $k^2 - 1$.

(2) **Applying a set of filters**: given a set of k^2 rotations, each rotation must be multiplied by the corresponding weight. Intuitively, the resulting k^2 ciphertexts are summed, and the result will contain the output of the filter application.

A kernel K is defined as a four-dimensional tensor $K \in \mathbb{R}^{n_{out} \times n_{in} \times k \times k}$ containing n_{out} filters, one for each output channel. Each filter is a three-dimensional tensor $\mathbb{R}^{n_{in} \times k \times k}$ in which n_{in} is the number of input channels.

The intuition is that each rotation $\mathbf{c}_i$ represents the i-th element on which the filter is applied. For instance, $\mathbf{c}_0$ represents the values that will be multiplied with the first element of the filter window (upper-left), $\mathbf{c}_1$ with the second element of the filter window (upper), and so on.

Algorithm 1 Procedure to generate the rotations

```
1:  procedure GenerateRotations(c, k)
2:      r ← {}
3:      mid ← (k^2 − 1)/2
4:      half ← ⌊k/2⌋
5:      r_mid ← c
6:      for i ← 0 to half do
7:          r_{mid+i·k} ← Rot_w(r_{mid+(i−1)·k})
8:          r_{mid−i·k} ← Rot_{−w}(r_{mid−(i−1)·k})
9:      end for
10:     for i ← 0 to k do
11:         row ← k · i
12:         for j ← 0 to half do
13:             idx ← row + half + (j + 1)
14:             r_idx = Rot_1(r_{idx−1})
15:             idx ← row + half − (j + 1)
16:             r_idx = Rot_{−1}(r_{idx−1})
17:         end for
18:     end for
19: end procedure
```

We therefore create k^2 plaintexts containing w^2 repetitions of the corresponding kernel value. Considering the example filter presented in Figure 4, we encode k^2 plaintexts $\mathbf{k}_i$ as shown in Figure 7.

This method encodes a $k \times k$ filter by using w^2 slots in k^2 plaintexts. The repetitions are necessary as they allow the parallel evaluation of the application of a filter to an entire channel.

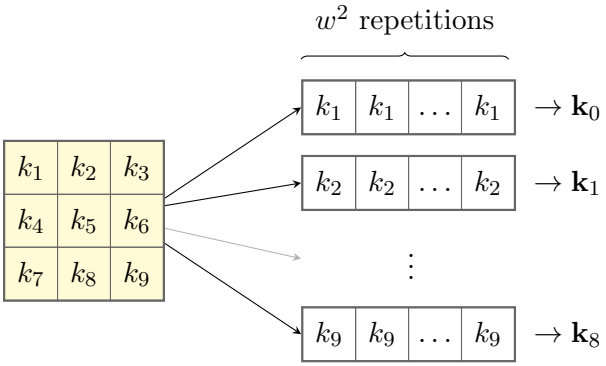

Figure 7: *Encoding a filter in nine plaintexts* $\mathbf{k}_i$

Some values will be masked, in order to avoid to align values coming from other channels (notice gray values in Figure 6), we therefore put zeros in those slots. We can thus compute a temporary ciphertext $\mathbf{t}$ as shown in Figure 8.

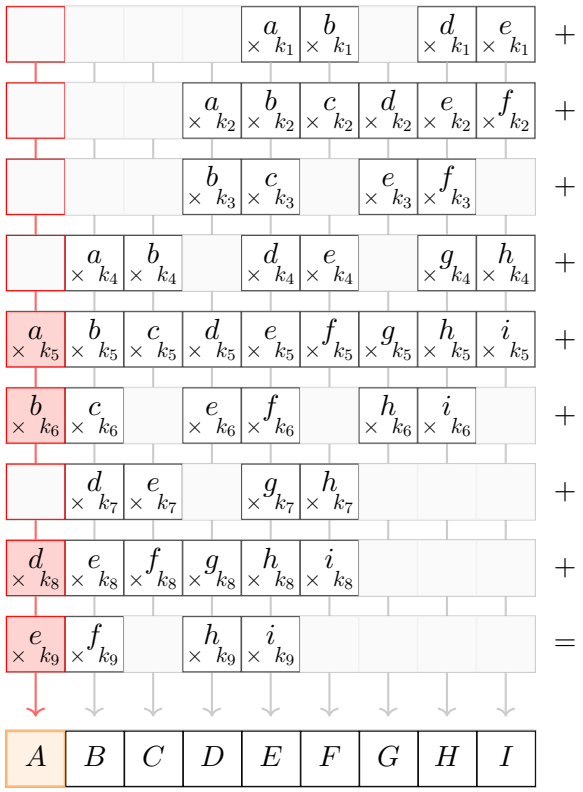

Figure 8: *Evaluating a convolution by adding* k^2 *ciphertexts up, following the example illustrated in Figure 4*

More formally, each line computes $\mathbf{r}_i \otimes \mathbf{k}_i$ and $\mathbf{t} = \sum_{i=0}^{8} \mathbf{r}_i \otimes \mathbf{k}_i$ is the sum of the nine lines. Nevertheless, there are a lot of empty slots in plaintexts, since we are aligning a kernel in the first w^2 slots only. We extend the procedure to apply c filters over c channels by arranging the values as shown in Figure 9.

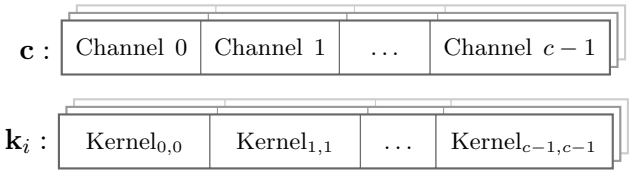

Figure 9: *Elements alignment before the first iteration*

We thus obtain a single ciphertext $\mathbf{t}$ in which each i-th *block* (consisting of w^2 elements) contains the application of the i-th filter on the i-th channel, arranged as follows:

$$\mathbf{t}: \quad \boxed{\mathbf{c}_0 \cdot \mathbf{K}_{0,0} \mid \mathbf{c}_1 \cdot \mathbf{K}_{1,1} \mid \ldots \mid \mathbf{c}_{c-1} \cdot \mathbf{K}_{c-1,c-1}}$$

(3) **Applying the whole kernel**: previously, we applied a single filter to each channel. To complete the convolution process, all the n_{out} filters of the kernel need to be applied to each of the n_{in} input channels. As a first step, we rotate by w^2 the ciphertext $\mathbf{t}$, obtaining $\mathbf{r} = \textsc{Rot}_{w^2}(\mathbf{t})$:

$$\mathbf{r}: \quad \boxed{\mathbf{c}_1 \cdot \mathbf{K}_{1,1} \mid \ldots \mid \mathbf{c}_{c-1} \cdot \mathbf{K}_{c-1,c-1} \mid \mathbf{c}_0 \cdot \mathbf{K}_{0,0}}$$

In the second iteration, we apply the second two-dimensional convolutional window, obtaining a new $\mathbf{t}$, and add it to $\mathbf{r}$.

$$\begin{aligned} \mathbf{r}: &\quad \boxed{\mathbf{c}_1 \cdot \mathbf{K}_{1,1} \mid \ldots \mid \mathbf{c}_{c-1} \cdot \mathbf{K}_{c-1,c-1} \mid \mathbf{c}_0 \cdot \mathbf{K}_{0,0}} \\ \mathbf{t}: &\quad \boxed{\mathbf{c}_0 \cdot \mathbf{K}_{1,0} \mid \ldots \mid \mathbf{c}_{c-2} \cdot \mathbf{K}_{c-1,c-2} \mid \mathbf{c}_{c-1} \cdot \mathbf{K}_{0,c-1}} \end{aligned}$$

At the last iteration, the first channel will be back to the first block, since we rotated by w^2 for c times, which is the length of the ciphertext. Assuming $c = 16$, our procedure sums the following ciphertexts up:

$$\begin{array}{|c|c|c|c|} \hline \mathbf{c}_0 \cdot \mathbf{K}_{0,0} & \mathbf{c}_1 \cdot \mathbf{K}_{1,1} & \ldots & \mathbf{c}_{15} \cdot \mathbf{K}_{0,15} \\ \hline \mathbf{c}_1 \cdot \mathbf{K}_{0,1} & \mathbf{c}_2 \cdot \mathbf{K}_{1,2} & \ldots & \mathbf{c}_0 \cdot \mathbf{K}_{15,0} \\ \hline \mathbf{c}_2 \cdot \mathbf{K}_{0,2} & \mathbf{c}_3 \cdot \mathbf{K}_{1,3} & \ldots & \mathbf{c}_1 \cdot \mathbf{K}_{15,1} \\ \hline \end{array}$$

$$\vdots$$

$$\begin{array}{|c|c|c|c|} \hline \mathbf{c}_{15} \cdot \mathbf{K}_{0,15} & \mathbf{c}_0 \cdot \mathbf{K}_{1,0} & \ldots & \mathbf{c}_{14} \cdot \mathbf{K}_{15,14} \\ \hline \end{array}$$

We define the algorithm used for the generation of the encoded kernel plaintext in Algorithm 2, and the complete procedure for the Optimized Vector Encoding in Algorithm 3.

Algorithm 2 Procedure that encodes the kernel

procedure EncodeKernel($K, iter$)
  $kernel \leftarrow \{\}$
  for $i \leftarrow 0$ to n_{out} **do**
    $filters \leftarrow \{\}$
    for $j \leftarrow 0$ to k^2 **do**
      $idx \leftarrow (i + iter) \bmod n_{in}$
      $rep \leftarrow \text{Repeat}(K_{i,idx,j}, w^2)$
      $filters \leftarrow \text{Append}(filters, rep)$
    end for
    $kernel \leftarrow \text{Append}(kernel, filters)$
  end for
  return $kernel$
end procedure

In usual Vector Encoding, each iteration requires a different key, for a total of c keys. We thus improved the keys requirements for Vector Encoding from (k^2-1+c) to a constant value 5 ($\approx 94\%$ of memory usage reduction in Layer 3, when $c = 64, k = 3$).

Algorithm 3 Optimized Vector Encoding

procedure OptimizedVectorEncoding(c)
  $\mathbf{r} \leftarrow 0$
  $\mathbf{c} \leftarrow \text{GenerateRotations}(c)$
  for $i \leftarrow 0$ to n_{out} **do**
    $\mathbf{k} \leftarrow \text{EncodeKernel}(K, i)$
    $\mathbf{t} \leftarrow 0$
    for $j \leftarrow 0$ to k^2 **do**
      $\mathbf{t} \leftarrow \mathbf{t} \oplus (\mathbf{c}_j \otimes \mathbf{k}_j)$
    end for
    $\mathbf{r} \leftarrow \mathbf{r} \oplus \text{Rot}_{w^2}(\mathbf{t})$
  end for
end procedure

2.4 ResNet20 Circuit

Recalling the structure of a ResNet20 (Section 2.2), we want to build an equivalent sequence of blocks based on FHE. The first issue is about the position of bootstrapping operations. We consider a Basic Block (Fig. 3), and we build the FHE equivalent as shown in Fig. 10.

Notice that we merged Convolutional Layer and Batch Normalization in a single ConvBN layer (refer to Section 2.3.1) that consumes one level.

We fix the level of the input ciphertext at $\ell - 1$. ConvBN is therefore evaluated before bootstrapping. In our example (Fig. 10) we assume six levels before bootstrapping, but this, in general, depends on the degree d of the ReLU Chebyshev Polynomial approximation. Different experiments will have different circuit depths, but the general structure of the Basic Block is always the same.

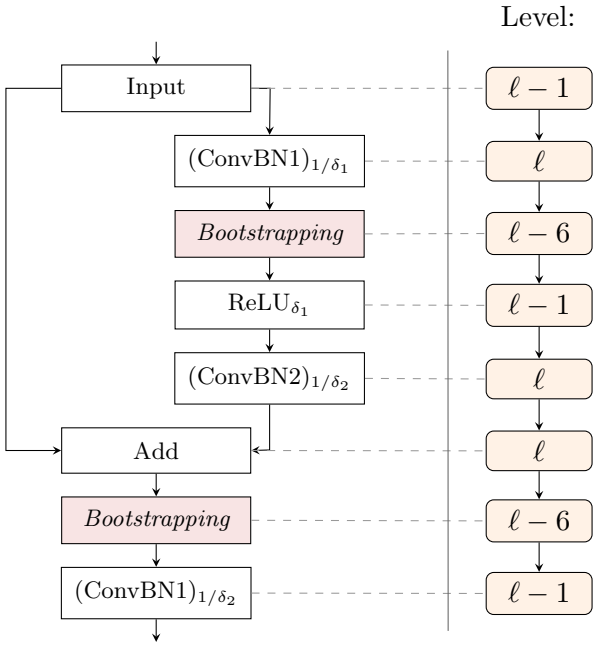

Figure 10: *FHE-based structure of a ResNet block. The boxes on the right represent the ciphertext modulus level, assuming a circuit depth ℓ. Here we assume that the evaluation of the ReLU function consumes five levels*

2.4.1 ReLU Approximation Interval

When using Chebyshev Polynomials in order to approximate a non-linear function, an approximation interval must be defined. Using an interval different from $[-1, 1]$ is not convenient as it requires an additional level consumption, since a generic $[a, b]$ must be scaled down to $[-1, 1]$.

Our approach is to study the range $[-\delta_i, \delta_i]$ of values before all the ReLU evaluations in the plain ResNet20, and to run the ConvBN with a scale $1/\delta_i$ (i.e. $\text{ConvBN}_{1/\delta_i}$) such that all the inputs given to ReLU are in $[-1, 1]$:

$$\text{ReLU}_{\delta_i}\left(\text{ConvBN}_{1/\delta_i}\right) = \begin{cases} \delta_i\, x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

We ended up with the experimental values for the approximating factors δ_i, $i \in \{1, 2\}$, in each block of each layer, reported in Table 3.

Table 3: *Experimentally obtained values for δ_i's, the approximating factors for intervals in ReLU*

Layer	Block 1		Block 2		Block 3	
	δ_1	δ_2	δ_1	δ_2	δ_1	δ_2
1	1.00	0.52	0.55	0.36	0.63	0.42
2	0.57	0.40	0.76	0.37	0.63	0.25
3	0.63	0.40	0.57	0.33	0.69	0.10

2.4.2 Downsampling Layers

Given that the total number of values decreases during the evaluation of the network (refer to Table 2), it is possible to reduce the number of slots in ciphertexts accordingly. In particular, downsamplings are performed by evaluating a Convolution Layer with stride $\{2, 2\}$ in *Downsampling Blocks* (Figure 2). Our Optimized Vector Encoding method is no longer sufficient, since it will blank slots in the output ciphertexts. A reshaping operation is therefore needed in order to fill these empty spaces. This process is performed by consuming six levels, and it intuitively works as shown in Figure 11.

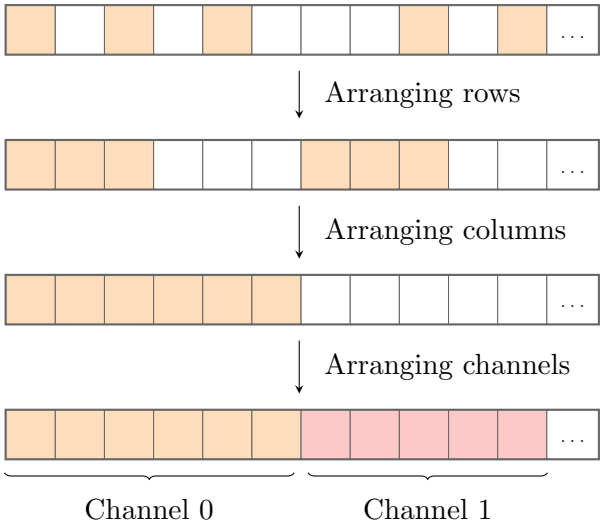

Figure 11: *Reshaping values after a Convolutional Layer with stride* $\{2, 2\}$

Re-arranging values requires new rotation keys. Assuming that the ciphertext has s slots, our strategy is to perform this procedure when the bootstrapping keys for s slots are no longer needed, so that they can be cleared from the memory. We can then load the keys needed for the downsampling, perform the procedure and set the number of slots for the ciphertexts to $s/2$. After that, we load the new bootstrapping keys for $s/2$ slots. The following layer will be evaluated with smaller ciphertexts, although with twice the number of channels.

3 Experiments and evaluations

We now present the experiments run using four sets of parameters. Each experiment will be evaluated according to three main factors: computations precision, computation time and memory requirements.

The precision of the FHE circuit is evaluated by running a parallel circuit working on plain values; we evaluate it, considering that CKKS works using fixed-point arithmetic, as:

$$p(v, v') = 1 - \left(\sum_i \left(\frac{|v_i - v'_i|}{|max(v)|} \right) \cdot ||v||^{-1} \right) \tag{3}$$

where v and v' are two vectors, computed on the plain and on the FHE circuit, respectively. The considered CIFAR-10 test set is composed of 1000 test images of size 32×32, each representing one out of 10 classes.

3.1 Circuit parameters

All the experiments have been run on 1.000 images from the CIFAR-10 test set, with the constraint of using less than 16GB of RAM and to satisfy the level of $\lambda \geq 128$ security bits according to the *Homomorphic Encryption Security Standard*[3].

Some common RLWE parameters for all the experiments are the following: Hamming weight (the number of non-zero elements in the secret key) set to $h = 64$, standard deviation of error distribution $\sigma = 3.19$. On the other hand, each experiment has different values for the following parameters:

- *Precision* (Δ, q_i, d) : this set of parameters has a big impact on the precision of the result. However, larger values mean slower and heavier computations.

- *Memory* $(N, q_i, d_{num}, \ell, \texttt{CtoS}, \texttt{StoC})$: as shown by Eq. 1, these parameters impact the memory requirements. Also, the levels dedicated to the `CoeffToSlot` (`CtoS`) and the `SlotToCoeff` (`StoC`) are important, because larger values mean less complex bootstrapping.

- *Time* $(N, q_i, d, \texttt{CtoS}, \texttt{CtoS})$: the first parameter, N, has the heaviest impact in computation time, as it defines the magnitude of all calculations. Smaller q_i and d result in smaller ciphertexts (hence, faster computations). Lastly, larger levels given to `CtoS` and `StoC` results in faster bootstrapping procedures.

The output of the circuit is a vector containing ten values representing the activation of each output neuron, each associated with an output class. A sample successful output is presented in Figure 12.

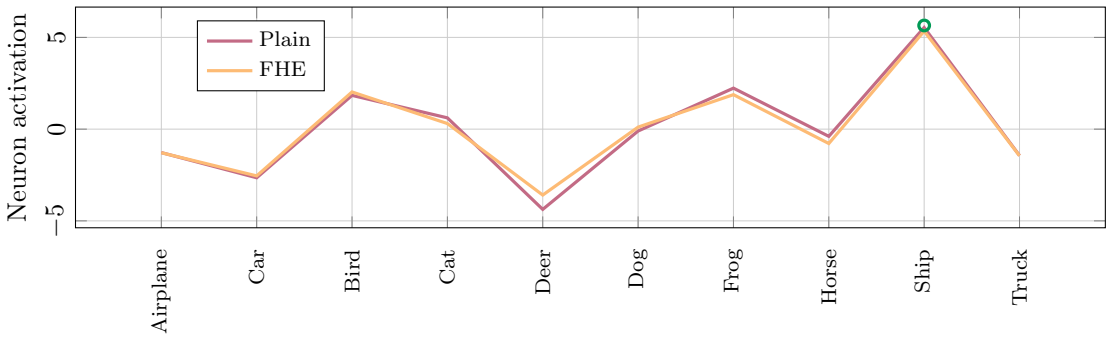

Figure 12: *An example of successful FHE output*

When the difference between the first and the second most active neurons is large, even approximate computations are successful. On the other hand, approximate classifications fail when the error changes the index of the maximum value (Figure 13).

This is likely to happen when the difference between the first and the second most active neurons is small. We studied how the difference between the first and the second most active neuron is distributed, considering the complete set of images of the CIFAR-10 test set, using the plain ResNet20 model[2]. Given y_i, the sorted vector containing the classification of the i-th image, we define a vector D:

$$D = \{d_i : d_i = (y_i)_0 - (y_i)_1\}$$

[2] `https://github.com/chenyaofo/pytorch-cifar-models`

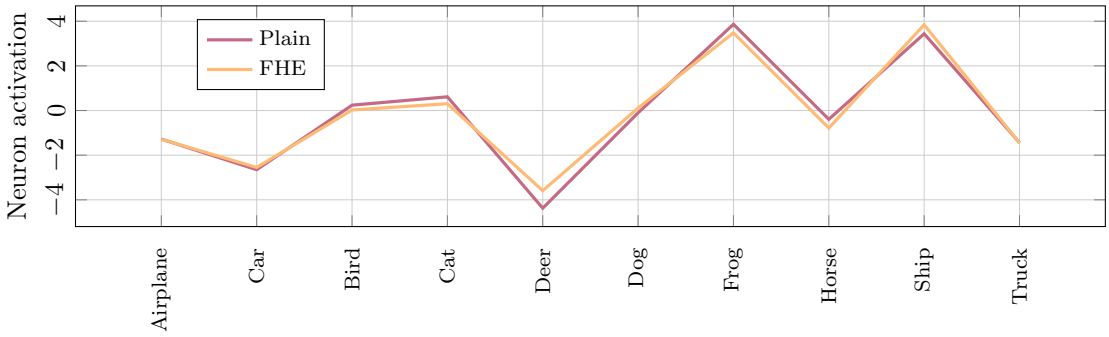

Figure 13: *Example of wrong FHE classification with respect to the plain one (Frog for plain, Ship for FHE)*

Table 4: *Parameters for the RNS-CKKS scheme, grouped by experiment.*

#Exp	Ring (N)	Scaling factor (Δ)	Bit length of q_i's	HKS digits (d_{num})	Moduli chain $\log(qp)$	Degree ReLU (d)	Circuit depth (ℓ)	`CtoS` and `StoC` levels
1	2^{16}	2^{52}	2^{48}	2	1756 bits	59	23	$\{3,3\}$
2	2^{16}	2^{50}	2^{46}	3	1772 bits	200	27	$\{4,4\}$
3	2^{16}	2^{50}	2^{46}	3	1772 bits	119	27	$\{5,4\}$
4	2^{16}	2^{48}	2^{44}	2	1748 bits	59	26	$\{4,4\}$

The average value of differences is ≈ 3.12, with a standard deviation $\sigma(D) \approx 2.64$. In general, more than 98% of the images have a difference d_i that is larger than 10^{-1}. This means that, when approximation errors are less than 10^{-1}, at least 98% of the results is equal to the plain model.

All the results presented in the experiments are obtained by evaluating the corresponding FHE circuit on a set of one thousand encrypted images from the CIFAR-10 test set, using a M1 Pro CPU and 16GB of RAM. The set of parameters for each experiment is given in Table 4.

It is possible to reproduce all the experiments using our open-source code[3] in C++. Algorithm 1 and 2 are available as Python notebooks, along with a notebook that shows how values from Table 3 have been found. As far as we are aware, ours and Kim et al. [21] (written using the Lattigo [24] library, in Go language) are the only recent and available open-source FHE-based CNN implementations.

3.2 Discussion

We present all the results obtained by the four experiments in Table 5 and in Fig. 14.

Experiments show interesting results. To begin with, all the considered sets of parameters result in at least 0.93 of relative accuracy. Since the original model has a level of accuracy of 0.92, this results in at least 0.85 of final accuracy, using any set of parameters. The set relative to Experiment 4, in particular, is the one having the lightest memory requirements (i.e. only 11.6GB) and the lowest runtime (less than 5 minutes).

By increasing the complexity of the parameters, we find the set of Experiment 1 to be slightly more complex than the previous one. In particular, we increased the values of Δ

[3] `https://github.com/narger-ef/LowMemoryFHEResNet20`

Table 5: *Comparison of results obtained in the experiments. Relative accuracy is computed with respect to the plain results on a subset of 1000 images from the test set*

#Exp	Total runtime	Bootstrapping precision	Memory requirements	Output layer precision p	Relative accuracy	Final accuracy
1	$291s \pm 4s$	≈ 9.4 bits	≈ 13.6GB	0.93 ± 0.02	945/1000	87.97%
2	$336s \pm 6s$	≈ 7.7 bits	≈ 15.2GB	0.98 ± 0.01	986/1000	**91.67**%
3	$285s \pm 6s$	≈ 7.7 bits	≈ 12.5GB	0.97 ± 0.01	978/1000	90.75%
4	$260s \pm 3s$	≈ 7.6 bits	$\approx$ **11.6GB**	0.92 ± 0.02	931/1000	86.12%

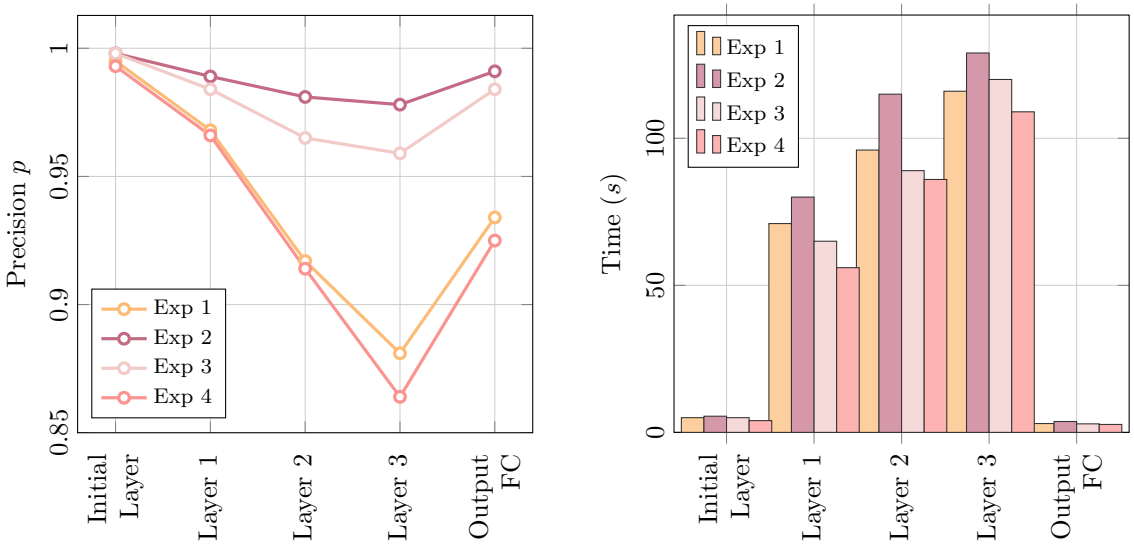

Figure 14: *Visual representation of precision and runtime of the experiments for the encrypted circuit*

and q_i for a higher accuracy in computations. In order to maintain the required level of security, we reduced by one the levels dedicated to `CtoS` and `StoC`. As a result, we obtained an increase on the relative accuracy (+0.02) at the cost of 2 more GB of memory and a longer runtime ($+7s$).

Surprisingly, with the set of parameters defined in Experiment 3, we obtained a larger relative accuracy (0.98) even though the required memory is increased by only 1GB with respect to Experiment 4. This confirms that the main bottleneck in computations precision is the degree d of the ReLU polynomial approximation. We then furthermore increased the degree to $d = 200$ in Experiment 2, obtaining the highest level of relative accuracy (0.99). This costs in terms of memory, since it requires around 15.2GB. Computational times, too, are higher.

To recap, we propose Experiment 3 as the one having the best ratio between precision of computations, memory requirements and execution time.

4 Conclusion

We presented a FHE-based circuit that classifies encrypted images with a high level of accuracy (0.9167) on the CIFAR-10 test set, evaluable in less than five minutes on a MacBook laptop equipped with M1 Pro CPU and 16GB of memory. The order of magnitude of the

achieved accuracy is similar to other recent works [21, 26], but our approach has a reduction of memory requirements by about 85% from Kim et al. [21] and by about 98% from Lee et al. [26].

Execution times are lower, although it is not easy to compare circuits executed on different hardware. Also for this reason, our work is available open-source. As far as we are aware, ours and Kim et al. [21] are the only CNNs based on FHE available open-source, as other recent works do not share any source code.

Furthermore, the circuit can be made even more suitable for real applications by using dedicated hardware [23, 35], which allows for extreme speed-up for FHE circuits. Theoretically, a 4.600x speed-up [35] would result in approximately five encrypted classifications per second using our circuit, enabling privacy-preserving inferences to be used in real-world tasks.

Acknowledgments

We express our gratitude to the OpenFHE forum for the help and hints in navigating the complexities of FHE and lattice-based cryptography.

Preprint of an article published in International Journal of Neural Systems, DOI: 10.1142/S0129065724500254

References

[1] Ahmad Al Badawi et al. "OpenFHE: Open-Source Fully Homomorphic Encryption Library". In: *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC'22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 53–63. ISBN: 9781450398770. DOI: 10.1145/3560827.3563379. URL: https://doi.org/10.1145/3560827.3563379.

[2] Ahmad Al Badawi et al. "Towards the AlexNet Moment for Homomorphic Encryption: HCNN, the First Homomorphic CNN on Encrypted Data With GPUs". In: *IEEE Transactions on Emerging Topics in Computing* 9.3 (2021), pp. 1330–1343. DOI: 10.1109/TETC.2020.3014636.

[3] Martin Albrecht et al. *Homomorphic Encryption Security Standard*. Tech. rep. Toronto, Canada: HomomorphicEncryption.org, Nov. 2018.

[4] Ahmad Al Badawi and Yuriy Polyakov. *Demystifying Bootstrapping in Fully Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2023/149. https://eprint.iacr.org/2023/149. 2023. URL: https://eprint.iacr.org/2023/149.

[5] Adrien Benamira et al. *TT-TFHE: a Torus Fully Homomorphic Encryption-Friendly Neural Network Architecture*. https://doi.org/10.48550/arXiv.2302.01584. 2023. arXiv: 2302.01584 [cs.CR]. URL: https://arxiv.org/abs/2302.01584.

[6] Jean-Philippe Bossuat et al. "Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys". In: *Advances in Cryptology – EUROCRYPT 2021*. Ed. by Anne Canteaut and François-Xavier Standaert. Cham: Springer International Publishing, 2021, pp. 587–617. ISBN: 978-3-030-77870-5.

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) Fully Homomorphic Encryption without Bootstrapping". In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ITCS '12. Cambridge, Massachusetts: Association for Computing Machinery, 2012, pp. 309–325. ISBN: 9781450311151. DOI: 10.1145/2090236.2090262. URL: https://doi.org/10.1145/2090236.2090262.

[8] Leo de Castro et al. *Does Fully Homomorphic Encryption Need Compute Acceleration?* https://doi.org/10.48550/arXiv.2112.06396. 2021. arXiv: 2112.06396 [cs.CR].

[9] Jung Hee Cheon et al. "A Full RNS Variant of Approximate Homomorphic Encryption". en. In: *Sel Areas Cryptogr* 11349 (Jan. 2019), pp. 347–368.

[10] Jung Hee Cheon et al. "Homomorphic Encryption for Arithmetic of Approximate Numbers". In: *Advances in Cryptology – ASIACRYPT 2017*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Cham: Springer International Publishing, 2017, pp. 409–437. ISBN: 978-3-319-70694-8.

[11] Ilaria Chillotti et al. "TFHE: Fast Fully Homomorphic Encryption Over the Torus". In: *Journal of Cryptology* 33.1 (Jan. 2020), pp. 34–91. ISSN: 1432-1378. DOI: 10.1007/s00145-019-09319-x. URL: https://doi.org/10.1007/s00145-019-09319-x.

[12] Léo Ducas and Daniele Micciancio. "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second". In: *Advances in Cryptology – EUROCRYPT 2015*. Ed. by Elisabeth Oswald and Marc Fischlin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 617–640. ISBN: 978-3-662-46800-5.

[13] Junfeng Fan and Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2012/144. https://eprint.iacr.org/2012/144. 2012. URL: https://eprint.iacr.org/2012/144.

[14] Craig Gentry. "Fully Homomorphic Encryption Using Ideal Lattices". In: *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*. STOC '09. Bethesda, MD, USA: Association for Computing Machinery, 2009, pp. 169–178. ISBN: 9781605585062. DOI: 10.1145/1536414.1536440. URL: https://doi.org/10.1145/1536414.1536440.

[15] Ran Gilad-Bachrach et al. "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy". In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 201–210.

[16] Shai Halevi and Victor Shoup. "Algorithms in HElib". In: *Advances in Cryptology – CRYPTO 2014*. Ed. by Juan A. Garay and Rosario Gennaro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 554–571. ISBN: 978-3-662-44371-2.

[17] Kyoohyung Han and Dohyeong Ki. "Better Bootstrapping for Approximate Homomorphic Encryption". In: *Topics in Cryptology – CT-RSA 2020*. Ed. by Stanislaw Jarecki. Cham: Springer International Publishing, 2020, pp. 364–390. ISBN: 978-3-030-40186-3.

[18] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778.

[19] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. "GAZELLE: A Low Latency Framework for Secure Neural Network Inference". In: *27th USENIX Security Symposium*. Baltimore, MD, Aug. 2018, pp. 1651–1669. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar.

[20] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. "Revisiting Homomorphic Encryption Schemes for Finite Fields". In: *Advances in Cryptology – ASIACRYPT 2021*. Ed. by Mehdi Tibouchi and Huaxiong Wang. Cham: Springer International Publishing, 2021, pp. 608–639. ISBN: 978-3-030-92078-4.

[21] Dongwoo Kim and Cyril Guyot. "Optimized Privacy-Preserving CNN Inference With Fully Homomorphic Encryption". In: *Trans. Info. For. Sec.* 18 (Mar. 2023), pp. 2175–2187. ISSN: 1556-6013. DOI: 10.1109/TIFS.2023.3263631. URL: https://doi.org/10.1109/TIFS.2023.3263631.

[22] Alex Krizhevsky, Geoffrey Hinton, et al. *Learning multiple layers of features from tiny images*. Tech. rep. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf. 2009.

[23] Banafsheh Saber Latibari et al. "A Survey on FHE Acceleration". In: *2023 IEEE 16th Dallas Circuits and Systems Conference (DCAS)*. 2023, pp. 1–6. DOI: 10.1109/DCAS57389.2023.10130256.

[24] *Lattigo v5*. Online: https://github.com/tuneinsight/lattigo. EPFL-LDS, Tune Insight SA. Nov. 2023.

[25] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

[26] Eunsang Lee et al. "Low-Complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Parallel Convolutions". In: *Proceedings of the 39th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri et al. Vol. 162. Proceedings of Machine Learning Research. PMLR, 17–23 Jul 2022, pp. 12403–12422. URL: https://proceedings.mlr.press/v162/lee22e.html.

[27] Eunsang Lee et al. "Minimax Approximation of Sign Function by Composite Polynomial for Homomorphic Comparison". In: *IEEE Transactions on Dependable and Secure Computing* 19.6 (2022), pp. 3711–3727. DOI: 10.1109/TDSC.2021.3105111.

[28] Joon-Woo Lee et al. "Privacy-Preserving Machine Learning With Fully Homomorphic Encryption for Deep Neural Network". In: *IEEE Access* 10 (2022), pp. 30039–30054. DOI: 10.1109/ACCESS.2022.3159694.

[29] Junghyun Lee et al. *Optimizing Layerwise Polynomial Approximation for Efficient Private Inference on Fully Homomorphic Encryption: A Dynamic Programming Approach*. https://doi.org/10.48550/arXiv.2310.10349. 2023. arXiv: 2310.10349 [cs.CR].

[30] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors over Rings". In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by Henri Gilbert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–23. ISBN: 978-3-642-13190-5.

[31] Oded Regev. "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography". In: *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*. STOC '05. Baltimore, MD, USA: Association for Computing Machinery, 2005, pp. 84–93. ISBN: 1581139608. DOI: 10.1145/1060590.1060603. URL: https://doi.org/10.1145/1060590.1060603.

[32] Protection Regulation. "Regulation (EU) 2016/679 of the European Parliament and of the Council". In: *Regulation (EU)* 679 (2016), p. 2016.

[33] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: https://doi.org/10.1145/359340.359342.

[34] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. "On data banks and privacy homomorphisms". In: *Foundations of secure computation* 4.11 (1978), pp. 169–180.

[35] Nikola Samardzic et al. "CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data". In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22. New York, New York: Association for Computing Machinery, 2022, pp. 173–187. ISBN: 9781450386104. DOI: 10.1145/3470496.3527393. URL: https://doi.org/10.1145/3470496.3527393.

[36] Lloyd N. Trefethen. *Approximation Theory and Approximation Practice, Extended Edition*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2019. DOI: 10.1137/1.9781611975949. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611975949. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611975949.