The AlexNet Moment for Homomorphic Encryption: HCNN, the First Homomorphic CNN on Encrypted Data with GPUs

Ahmad Al Badawi, Jin Chao, Jie Lin, Chan Fook Mun, Sim Jun Jie,
Benjamin Hong Meng Tan, Xiao Nan, Khin Mi Mi Aung,
Vijay Ramaseshan Chandrasekhar

{Ahmad_Al_Badawi, Jin_Chao, lin-j, Chan_Fook_Mun, Sim_Jun_Jie, Benjamin_Tan,
xiao_nan, Mi_Mi_Aung, vijay}@i2r.a-star.edu.sg

Institute for Infocomm Research, A*STAR, Singapore

November 1, 2018

Abstract

Fully homomorphic encryption, with its widely-known feature of computing on encrypted data, empowers a wide range of privacy-concerned cloud applications including deep learning as a service. This comes at a high cost since FHE includes highly-intensive computation that requires enormous computing power. Although the literature includes a number of proposals to run CNNs on encrypted data, the performance is still far from satisfactory. In this paper, we push the level up and show how to accelerate the performance of running CNNs on encrypted data using GPUs. We evaluated a CNN to classify homomorphically the MNIST dataset into 10 classes. We used a number of techniques such as low-precision training, unified training and testing network, optimized FHE parameters and a very efficient GPU implementation to achieve high performance. Our solution achieved high security level (> 128 bit) and high accuracy (99%). In terms of performance, our best results show that we could classify the entire testing dataset in 14.105 seconds, with per-image amortized time (1.411 milliseconds) $40.41\times$ faster than prior art.

Keywords— Fully Homomorphic Encryption, Deep Learning, Encrypted CNN, Privacy-preserving Computing, GPU Acceleration

1 Introduction

The next step in the machine learning revolution would be Deep Learning as a Service (DLaaS) which seeks to take advantage of the benefits that cloud computing brings. Cloud servers are excellent machine learning platforms, offering cheap data storage, near-zero deployment cost and high computational services. However, it is not all-powerful and there are important questions that need to be resolved before DLaaS can become widespread. One of the main questions is that cloud platforms do not guarantee data

privacy. In the DLaaS setting, one uploads their data into the cloud, runs the model on it and gets the results back from the cloud. At every step along the way, there are numerous opportunities for hackers and other malicious actors to compromise the data.

Privacy-preserving machine learning was considered previously by Graepel *et al.* [18] and Aslett *et al.* [5]. Following them, Dowlin *et al.* [13, 14] proposed CryptoNets, the first neural network over encrypted data, providing a method to do the inference phase of privacy-preserving deep learning. Since then, others [29, 9, 24, 21, 22] have applied a variety of cryptographic techniques, such as secure multi-party computation and oblivious transfers, to achieve similar goals. Just as AlexNet by Krizhevsky *et al.* [26] showed how image classification is viable by running convolutional neural networks (CNN) on GPUs, we show that privacy-preserving deep learning is dramatically accelerated with GPUs and offers a way towards efficient DLaaS. We follow the framework put forward in CryptoNets [13, 14] and apply our GPU-accelerated fully homomorphic encryption (FHE) techniques to realize efficient homomorphic convolutional neural networks (HCNNs).

Although the framework is available, there are still challenges to realizing performant HCNNs. FHE, first realized by Gentry [17] almost 10 years ago, allows arbitrary computation on encrypted data. Informally, it works as follows. Encryption masks the input data, called a plaintext, by a random error sampled from some distribution, resulting in a ciphertext that reveals nothing about what it encrypts. Decryption uses the secret key to filter out the noise and retrieve the plaintext as long as the noise is within some threshold. Note that during computation, the noise in ciphertexts grows, but in a controlled manner. At some point, it grows to a point where no further computation can be done without resulting in decryption failure. Bootstrapping can be used to refresh a ciphertext with large noise into one with less noise that can be used for computation. By doing this indefinitely, theoretically, any function can be computed.

However, this approach is still impractical and bootstrapping is not used in most cases. Instead, the class of functions that can be evaluated is restricted to depth L arithmetic circuits, yielding a levelled FHE scheme to avoid bootstrapping. For performance, L should be minimized which means that we have to carefully design HCNNs with this in mind. Furthermore, the model of computation in FHE, arithmetic circuits with addition (HAdd) and multiplication (HMult) gates, is not compatible with non-polynomial functions such as sigmoid, ReLU and max. This means that we should use polynomial approximations to the activation functions where possible and consider if pooling layers are useful in practice.

Besides that, we have to encode decimals in a form that is compatible with FHE plaintext data, which are usually integers. These can have high precision which mean that they will require integers of large bit-size to represent them in the commonly used *scalar encoding*. In this encoding, decimals are transformed into integers by multiplying them with some scaling factor Δ and then operated on with HAdd and HMult normally. The main drawback of this encoding is that we cannot re-scale encoded data mid-computation; therefore, successive homomorphic operations will cause data size to increase rapidly. Managing this scaling expansion is a necessary step towards scaling HCNNs to larger datasets and deeper neural networks.

Our Contributions.

1. We present the first GPU-accelerated Homomorphic Convolutional Neural Networks (HCNN) that runs a pre-learned model on encrypted data from the MNIST dataset.
2. We provide a rich set of optimization techniques to enable easy designs of HCNN and reduce the overall computational overhead. These include low-precision training, optimized choice of HE scheme and parameters, and a GPU-accelerated implementation.
3. We reduced the HCNN for the MNIST dataset to only 5 layers deep for both training and inference, smaller than CryptoNets [13] which used 9 layers during training.
4. We compute predictions for 10,000 28×28-pixel images in 14.105 seconds, more than $40.41\times$ improvement over the current record (by CryptoNets) and with higher (128-bit) security.

Related Work. The research in the area of privacy-preserving deep learning can be roughly divided into two camps: those using homomorphic encryption or combining it with secure multi-party computation (MPC) techniques. Most closely related to our work are CryptoNets by Dowlin *et al.* [13, 14], FHE-DiNN by Bourse *et al.* [9] and E2DM by Jiang *et al.* [22], who focus on using only fully homomorphic encryption to address this problem. Dowlin *et al.* [13, 14] were the first to propose using FHE to achieve privacy-preserving deep learning, offering a framework to design neural networks that can be run on encrypted data. They proposed using polynomial approximations of the most widespread ReLU activation function and using pooling layers only during the training phase to reduce the circuit depth of their neural network. However, they used the YASHE′ scheme by Bos *et al.* [8] which is no longer secure due to attacks proposed by Albrecht *et al.* [3]. Also, they require a large plaintext space of over 80 bits to contain their neural network's output. This makes it very difficult to scale to deeper networks since intermediate layers in those networks will quickly reach several hundred bits with their settings.

Following them, Bourse *et al.* [9] proposed a new type of neural network called discretized neural networks (DiNN) for inference over encrypted data. Weights and inputs of traditional CNNs are discretized into elements in $\{-1, 1\}$ and the fast bootstrapping of the TFHE scheme proposed by Chilotti *et al.* [12] was exploited to double as an activation function for neurons. Each neuron computes a weighted sum of its inputs and the activation function is the sign function, $\mathsf{sign}(z)$ which outputs the sign of the input z, i.e. $\mathsf{sign}(z) = -1$ if $z < 0$ and 1 otherwise. Although this method can be applied to arbitrarily deep networks, it suffers from lower accuracy, achieving only 96.35% accuracy on the MNIST dataset with lower amortized performance. Very recently, Jiang *et al.* [22] proposed a new method for matrix multiplication with HE and evaluated a neural network on the MNIST data set using this technique. They also considered packing an entire image into a single ciphertext compared to the approach of Dowlin *et al.* [13, 14] who put only one pixel per ciphertext but evaluated large batches of images at a time. They achieved good performance, evaluating 64 images in slightly under 29 seconds but with worse amortized performance.

Some of the main limitations of pure FHE-based is the need to approximate non-polynomial activation functions and high computation time. Addressing these problems, Liu *et al.* [29] proposed MiniONN, a paradigm shift in securely evaluating neural networks. They take commonly used protocols in deep learning and transform them into oblivious protocols. With MPC, they could evaluate neural networks without changing the training phase, preserving accuracy since there is no approximation needed for activation functions. However, MPC comes with its own set of drawbacks. In this setting, each computation requires communication between the data owner and model owner, thus resulting in high bandwidth usage. In a similar vein, Juvekar *et al.* [24] designed GAZELLE. Instead of applying levelled FHE, they alternate between an additive homomorphic encryption scheme for convolution-type layers and garbled circuits for activation and pooling layers. This way, communication complexity is reduced compared to MiniONN but unfortunately is still significant.

Organization of the Paper. Section 2 introduces fully homomorphic encryption and neural networks, the main components of HCNNs. Following that, Section 3 discusses the challenges of adapting convolutional neural networks to the homomorphic domain. Next, we describe the components that were used in implementing HCNNs in Section 4. In Section 5, we report the results of experiments done using our implementation of HCNNs on the MNIST dataset. Lastly, we conclude with Section 6 and discuss some of the obstacles that will be faced when extending HCNNs can be scaled to larger datasets.

2 Preliminaries

In this section, we review a set of notions that are required to understand the paper. We start by introducing FHE, thereby describing the BFV scheme, an instance of levelled FHE schemes. Next, we introduce neural networks and how to tweak them to become compatible with FHE computation model.

2.1 Fully Homomorphic Encryption

First proposed by Rivest *et al.* [31], fully homomorphic encryption (FHE) was envisioned to enable arbitrary computation on encrypted data. FHE would support operations on ciphertexts that translate to functions on the encrypted messages within. It remained unrealized for more than 30 years, until Gentry [17] proposed the first construction. The blueprint of this construction remains the only method to design FHE schemes. The (modernized) blueprint is a simple two-step process. First, a somewhat homomorphic encryption scheme that can evaluate its decryption function is designed. Then, we perform bootstrapping, which decrypts a ciphertext using an encrypted copy of the secret key. Note that the decryption function here is evaluated homomorphically, i.e., on encrypted data and the result of decryption is also encrypted.

As bootstrapping imposes high computation costs, we adopt a levelled FHE scheme instead, which can evaluate functions up to a pre-determined multiplicative depth without bootstrapping. We chose the Brakerski-Fan-Vercauteren (BFV) scheme [10, 16], whose security is based on the Ring Learning With Errors (RLWE) problem proposed by Lyubashevsky *et al.* [30]. This problem is conjectured to be hard even with quantum computers, backed by reductions (in [30] among others) to worst-case problems in ideal lattices.

The BFV scheme has five algorithms (KeyGen, $\mathsf{Encrypt}$, $\mathsf{Decrypt}$, HAdd, HMult). KeyGen is the algorithm that generates the keys used in an FHE scheme given the parameters chosen. $\mathsf{Encrypt}$ and $\mathsf{Decrypt}$ are the encyption and decryption algorithms respectively. The differentiation between FHE and standard public-key encryption schemes is the operations on ciphertexts; which we call HAdd and HMult. HAdd outputs a ciphertext that decrypts to the sum of the two input encrypted messages while HMult outputs one that decrypts to the product of the two encrypted inputs.

We informally describe the basic scheme below and refer to [16] for the complete details. Let $k, q, t > 1$ with $N = 2^k$, t prime and $\mathcal{R} = \mathbb{Z}[X]/\langle X^N + 1\rangle$, we denote the ciphertext space as $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$ and message space as $\mathcal{R}_t = \mathcal{R}/t\mathcal{R}$. We call ring elements "small" when their coefficients have small absolute value.

- $\mathsf{KeyGen}(\lambda, L)$: Given security parameter λ and level L as inputs, choose k, q so that security level λ is achieved. Choose a random element $a \in \mathcal{R}_q$, "small" noise $e \in \mathcal{R}_q$ and secret key $s \in \mathcal{R}_2$, the public key is defined to be $pk = (b = e - as, a)$.
- $\mathsf{Encrypt}(pk, m)$: Given public key pk and message $m \in \mathcal{R}_t$ as input, the encryption of m is defined as $\mathbf{c} = (br' + e' + \lfloor q/t \rfloor m, ar')$, for some random noise $e', r' \in \mathcal{R}_q$.
- $\mathsf{Decrypt}(sk, \mathbf{c})$: Given secret key sk and ciphertext $\mathbf{c} = (c_0, c_1) \in \mathcal{R}_q^2$ as inputs, the decryption of $\mathbf{c}$ is
$$m = \lceil (t/q)(c_0 + c_1 s \bmod q) \rfloor \bmod t.$$
- $\mathsf{HAdd}(\mathbf{c}_1, \mathbf{c}_2)$: Given two ciphertexts $\mathbf{c}_1 = (c_{0,1}, c_{1,1}), \mathbf{c}_2 = (c_{0,2}, c_{1,2})$ as inputs, the operation is simply component-wise addition, i.e. the output ciphertext is $\mathbf{c}' = (c_{0,1} + c_{0,2}, c_{1,1} + c_{1,2})$.
- $\mathsf{HMult}(\mathbf{c}_1, \mathbf{c}_2)$: Given two ciphertexts $\mathbf{c}_1 = (c_{0,1}, c_{1,1}), \mathbf{c}_2 = (c_{0,2}, c_{1,2})$ as inputs, proceed as follows:
  1. (Tensor) compute
  $$\mathbf{c}^* = (c_{0,1}c_{0,2}, c_{0,1}c_{1,2} + c_{1,1}c_{0,2}, c_{1,1}c_{1,2}); \tag{1}$$
  2. (Scale and Relinearize) output
  $$\mathbf{c}' = \lceil (t/q)\mathsf{Relinearize}(\mathbf{c}^*) \rfloor \bmod q. \tag{2}$$

Correctness of the Scheme. For the scheme to be correct, we require that $\mathsf{Decrypt}(sk, \mathbf{c})$ for $\mathbf{c}$ output from $\mathsf{Encrypt}(pk, m)$, where $(pk, sk = s)$ is a correctly generated key-pair from KeyGen. We characterize when decryption will succeed in the following theorem.

Theorem 1. *Let* $\mathbf{c} = (c_0 = \lfloor q/t \rfloor m' + e - c_1 s, c_1)$ *be a ciphertext. Then,* Decrypt *outputs the correct message* $m = m'$ *if* $\|e\|_\infty < q/2t$*, where* $\|e\|_\infty$ *is the largest coefficient of the polynomial* $e \in \mathcal{R}_q$*.*

Proof. Recall that the decryption procedure computes $m = \lceil (t/q)(c_0 + c_1 s) \bmod q \rfloor \bmod t$. Therefore, to have $m = m'$, we first require $c_0 + c_1 s < q$ which means that $\lfloor q/t \rfloor m' + e < q$. Finally, we need the rounding operation to output m' after scaling by t/q which requires that $\|e\|_\infty < q/2t$ since $(t/q) \cdot e$ must be less than $1/2$. □

To see why HAdd works, part of the decryption requires computing

$$\begin{aligned} c_{0,1} + c_{0,2} + (c_{1,1} + c_{1,2})s &= (c_{0,1} + c_{1,1}s) + (c_{0,2} + c_{1,2}s) \\ &= \lceil q/t \rfloor \, m_1 + e_1 + \lceil q/t \rfloor \, m_2 + e_2 \\ &= \lceil q/t \rfloor \, (m_1 + m_2) + e_1 + e_2. \end{aligned}$$

This equation remains correct modulo q as long as the errors are small, i.e. $\|e_1 + e_2\|_\infty < q/2t$. Therefore, scaling by (t/q) and rounding will be correct which means that we obtain the desired message.

For HMult, the procedure is more complicated but observe that

$$(c_{0,1} + c_{1,1}s)(c_{0,2} + c_{1,2}s) = c_{0,1}c_{0,2} + (c_{0,1}c_{1,2} + c_{1,1}c_{0,2})s + c_{1,1}c_{1,2}s^2. \tag{3}$$

This means that we need s as well as s^2 to recover the desired message from $\mathbf{c}^*$. However, with a process called relinearization (Relinearize), proposed by Brakerski and Vaikuntanathan [11] and applicable to the BFV scheme, $\mathbf{c}^*$ can be transformed to be decryptable under the original secret key s.

Computation Model with Fully Homomorphic Encryption. The set of functions that can be evaluated with FHE are arithmetic circuits over the plaintext ring $\mathcal{R}_t$. However, this is not an easy plaintext space to work with; elements in $\mathcal{R}_t$ are polynomials of degree up to several thousand. Addressing this issue, Smart and Vercauteren [34] proposed a technique to support single instruction multiple data (SIMD) by decomposing $\mathcal{R}_t$ into a product of smaller spaces with the Chinese Remainder Theorem over polynomial rings. For prime $t \equiv 1 \bmod 2N$, $X^N + 1 \equiv \prod_{i=1}^{N} (X - \alpha_i) \bmod t$ for some $\alpha_i \in \{1, 2, \ldots, t-1\}$. This means that $\mathcal{R}_t = \prod_{i=1}^{N} \mathbb{Z}_t[X]/\langle X - \alpha_i \rangle \cong \prod_{i=1}^{N} \mathbb{Z}_t$. Therefore, the computation model generally used with homomorphic encryption is arithmetic circuits with modulo t gates.

For efficiency, the circuits evaluated using the HAdd and HMult algorithms should be levelled. This means that the gates of the circuits can be organized into layers, with inputs in the first layer and output at the last, and the outputs of one layer are inputs to gates in the next layer. In particular, the most important property of arithmetic circuits for HE is its depth. The depth of a circuit is the maximum number of multiplication gates along any path of the circuit from the input to output layers.

A levelled FHE scheme with input level L can evaluate circuits of at most depth L which affects the choice of parameter q due to noise in ciphertexts. In particular, the HMult operation on ciphertext is the main limiting factor to homomorphic evaluations. From Equation (3), we have

$$\begin{aligned} c_{0,1}c_{0,2} &= (\lceil q/t \rfloor m_1 + e_1)(\lceil q/t \rfloor m_2 + e_2) \\ &= (\lceil q/t \rfloor m_1)(\lceil q/t \rfloor m_2) + \underbrace{\lceil q/t \rfloor m_1 e_2 + \lceil q/t \rfloor m_2 e_1 + e_1 e_2}_{e'}. \end{aligned}$$

Even after scaling by t/q, the overall noise ($\approx t/q \cdot e'$) in the output $\mathbf{c}'$ is larger than that of the inputs, $\mathbf{c}_1$ and $\mathbf{c}_2$. Successive calls to HMult have outputs that steadily grow. Since decryption only succeeds if the error in the ciphertext is less than $q/2t$, the maximum depth of a circuit supported is determined by the ciphertext modulus q. To date, the only known method to sidestep this is with the bootstrapping technique proposed by Gentry [17].

2.2 Neural Networks

A neural network, by which we mean artificial feed-forward neural networks, can be seen as a circuit made up of levels called layers. Each layer is made up of a set of nodes, with the first being the inputs to the network. Nodes in the layers beyond the first take the outputs from a subset of nodes in the previous layer and output the evaluation of some function over them. The values of the nodes in the last layer are the outputs of the neural network.

In the literature, many different layers are used but these can generally be grouped into three categories.

1. Activation Layers: Each node in this layer takes the output, z, of a single node of the previous layer and outputs $f(z)$ for some function z.
2. Convolution-Type Layers: Each node in this layer takes the outputs, $\mathbf{z}$, of some subset of nodes from the previous layer and outputs a weighted-sum $\langle \mathbf{w}, \mathbf{z} \rangle + b$ for some weight vector $\mathbf{w}$ and bias b.
3. Pooling Layers: Each node in this layer takes the outputs, $\mathbf{z}$, of some subset of nodes from the previous layer and outputs $f(\mathbf{z})$ for some function f.

The functions used in the activation layers are quite varied, including sigmoid ($f(z) = \frac{1}{1+e^{-z}}$), softplus ($f(z) = \log(1 + e^z)$) and ReLU, where

$$\mathsf{ReLU}(z) = \begin{cases} z, & \text{if } z \geq 0; \\ 0, & \text{otherwise.} \end{cases}$$

Although commonly used in practice, some have questioned the utility of pooling layers. Springenberg *et al.* [36] proposed to remove pooling layers completely from convolutional neural networks and Kamnitsas *et al.* [25] showed that pooling was unnecessary for some cases of image analysis. To adapt neural networks operations over encrypted data, we do not use pooling and focus on the following layers:

- *Convolution (weighted-sum) Layer*: At each node, we take a subset of the outputs of the previous layer, also called a filter, and perform a weighted-sum on them to get its output.
- *Square* Layer: Each node linked to a single node z of the previous layer; its output is the square of z's output.
- *Fully Connected* Layer: Similar to the convolution layer, each node outputs a weighted-sum, but over the entire previous layer rather than a subset of it.

3 Homomorphic Convolutional Neural Networks

Homomorphic encryption (HE) enables computation directly on encrypted data. This is ideal to handle the challenges that machine learning face when it comes to questions of data privacy. We call convolutional neural networks (CNN) that operate over encrypted data as homomorphic convolutional neural networks (HCNN). Although HE promises a lot, there are several obstacles, ranging from the choice of plaintext space to translating neural network operations, that prevent straightforward translation of standard techniques for traditional CNNs to HCNNs.

3.1 Plaintext Space

The first problem is the choice of plaintext space for HCNN computation. Weights and inputs of a neural network are usually decimals, which are represented in floating-point. Unfortunately, these cannot be

directly encoded and processed in most HE libraries and thus require some adjustments. For simplicity and to allow inference on large datasets, we pack the same pixel of multiple images in a single ciphertext as shown in Figure 1. Note that we can classify the entire MNIST testing dataset at once as the number of slots is more than 10,000.

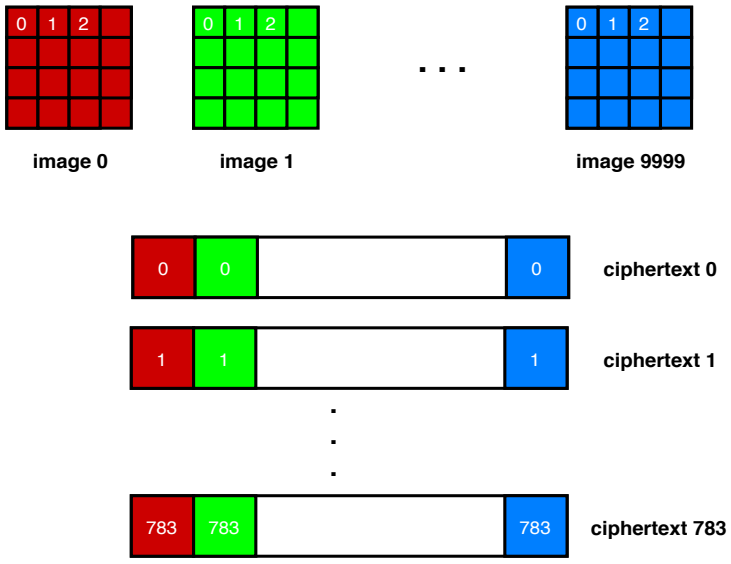

Figure 1: Packing MNIST testing dataset. Ciphertext i contains pixel i from all images.

Encoding into the Plaintext Space. We adopt the *scalar encoding*, which approximates these decimals with integers. It is done by multiplying them with some scaling factor Δ and rounding the result to the nearest integer. Then, numbers encoded with the same scaling factor can be combined with one another using integer addition or multiplication. For simplicity, we normalize the inputs and weights of HCNNs to between $[0, 1]$ and Δ (initially) corresponds to the number of bits of precision of the approximation, as well as the upper bound on the approximation.

Although straightforward to use, there are some downsides to this encoding. The scale factor cannot be adjusted mid-computation and mixing numbers with different scaling factors is not straightforward. For example, suppose we have two messages $\Delta_1 m_1, \Delta_2 m_2$ with two different scaling factors, where $\Delta_1 < \Delta_2$:

$$\Delta_1 m_1 + \Delta_2 m_2 = \Delta_2(m_2 + \Delta_2/\Delta_1 m_1); \quad \Delta_1 m_1 \times \Delta_2 m_2 = \Delta_1 \Delta_2 (m_1 m_1).$$

Multiplication will just change the scaling factor of the result to $\Delta_1 \Delta_2$ but the result of adding two encoded numbers is not their standard sum. This means that as homomorphic operations are done on encoded data, the scaling factor in the outputs increases without a means to control it. Therefore, the plaintext modulus t has to be large enough to accommodate the maximum number that is expected to result from homomorphic computations.

With the smallest scaling factor, $\Delta = 2$, 64 multiplications will suffice to cause the result to potentially overflow the space of 64-bit integers. Unfortunately, we use larger Δ in most cases which means that the expected maximum will be much larger. Thus, we require a way to handle large plaintext moduli of possibly several hundred bits.

Plaintext Space CRT Decomposition. One way to achieve this is to use a composite plaintext modulus, $t = \prod_{i=0}^{r-1} t_i$ for some primes $t_0, \ldots, t_{r-1}$ such that t is large enough. Recall that the Chinese

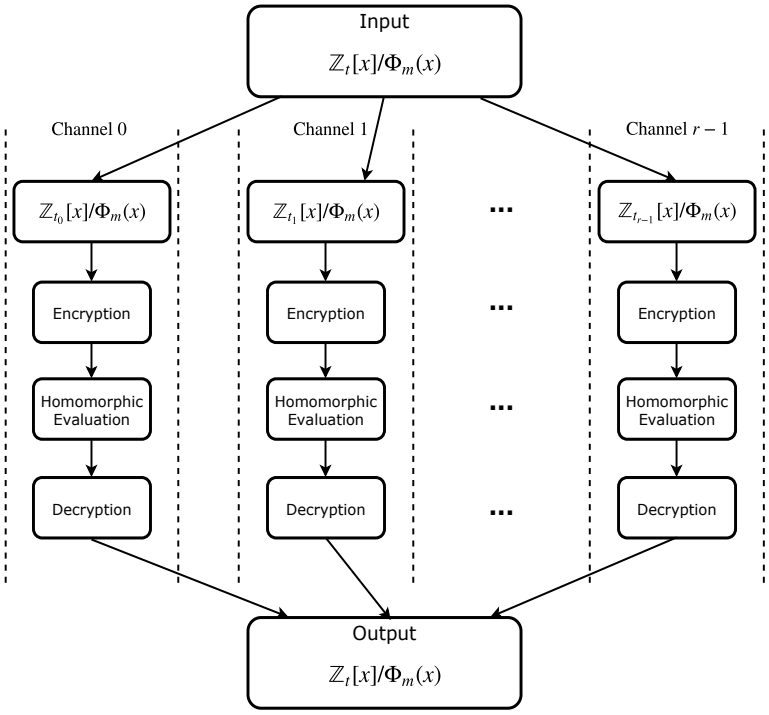

Figure 2: Plaintext CRT decomposition for a HE arithmetic circuit

Remainder Theorem (CRT) gives us an isomorphism between $\mathbb{Z}_t$ and $\prod_{i=0}^{r-1} \mathbb{Z}_{r_i}$:

$$\begin{aligned}
\mathsf{CRT}:\ & \mathbb{Z}_{t_0} \times \cdots \times \mathbb{Z}_{t_{r-1}} \longrightarrow \mathbb{Z}_t \\
& \mathbf{m} = (m_0,\ m_1,\ \ldots,\ m_{r-1}) \longmapsto \mathsf{CRT}(\mathbf{m}),\ \text{where } m_i \in \mathbb{Z}_{t_i}; \\
\mathsf{ICRT}:\ & \mathbb{Z}_t \longrightarrow \mathbb{Z}_{t_0} \times \cdots \times \mathbb{Z}_{t_{r-1}} \\
& m \longmapsto \mathsf{ICRT}(m) = (m \bmod t_0,\ m \bmod t_1,\ \ldots,\ m \bmod t_{r-1});
\end{aligned}$$

where for any $m \in \mathbb{Z}_t$, we have $\mathsf{CRT}(\mathsf{ICRT}(m)) = m$.

For such moduli, we can decompose any integer $m < t$ into a length-r vector with ICRT. Arithmetic modulo t is replaced by component-wise addition and multiplication modulo the prime t_i for the i-th entry. We can recover the output of any computation as long as it is less than t because the inverse map CRT will return a modulo t result.

As illustrated in Figure 2, for homomorphic operations modulo t, we separately encrypt each entry of $\mathsf{CRT}(m)$ in r HE instances with the appropriate t_i and perform modulo t_i operations. At the end of the homomorphic computation of function f, we decrypt the r ciphertexts, one per HE instance, to obtain the vector $\mathbf{v} = \mathsf{ICRT}(f(m))$. The actual output $f(m)$ is obtained by applying the CRT map to $\mathbf{v}$, i.e. $f(m) = \mathsf{CRT}(\mathbf{v})$.

3.2 Neural Network Layers

Computation in HE schemes are generally limited to addition and multiplication operations over ciphertexts. As a result, it is easy to compute polynomial functions with HE schemes. As with all HE schemes, encryption injects a bit of noise into the data and each operation on ciphertexts increases the noise within it. As long as the noise does not exceed some threshold, decryption is possible. Otherwise, the decrypted results are essentially meaningless.

Approximating Non-Polynomial Activations. For CNNs, a major stumbling block for translation to the homomorphic domain is the activation functions. These are usually not polynomials, and therefore unsuitable for evaluation with HE schemes. The effectiveness of the ReLU function in convolutional neural networks means that it is almost indispensable. Therefore, it should be approximated by some polynomial function to try to retain as much accuracy as possible. The choice of approximating polynomial depends on the desired performance of the HCNN. For example, in this work, we applied the square function, $z \mapsto z^2$, which Dowlin *et al.* [13, 14] found to be sufficient for accurate results on the MNIST dataset with a five layer network.

The choice of approximation polynomial determines the depth of the activation layers as well as its complexity (number of HMults). The depth and complexity of this layer will be $\lceil \log d \rceil$ and $d - 1$ respectively, where d is the degree of the polynomial. However, with the use of scalar encoding, there is another effect to consider. Namely, the scaling factor on the output will be dependent on the depth of the approximation, i.e. if the scaling factor of the inputs to the activation layer is Δ, then the scaling factor of the outputs will be roughly $\Delta^{1+\lceil \log d \rceil}$, assuming that the approximation is a monic polynomial.

Handling Pooling Layers. Similar to activations, the usual functions used in pooling layers, maximum ($\mathsf{max}(\mathbf{z}) = \max_{1 \le i \le n} z_i$), ℓ_2-norm and mean ($\mathsf{avg}(\mathbf{z}) = \frac{1}{n} \sum_{i=1}^{n} z_i$) for inputs $\mathbf{z} = (z_1, \ldots, z_n)$, are generally non-polynomial. Although avg is a linear function, division in HE schemes is more involved and requires different plaintext encoding methods (see Dowlin *et al.* [15]). In CryptoNets [13, 14], a variant of the mean function, called scaled-mean ($\sum_{i=1}^{n} z_i$) is which introduces an additional n factor over avg and does not impact its performance. Still, that is not the only choice that is available. Several works [36, 25] have shown that pooling is not strictly necessary and good results can be obtained without it. For a simpler CNN, we chose to remove the pooling layers used in CryptoNets during training and apply the same network for both training and inference, with the latter over encrypted data.

Convolution-Type Layers. Lastly, we have the convolutional-type layers. Since these are weighted sums, they are straightforward to compute over encrypted data; the weights can be multiplied to encrypted inputs with HMult and the results summed with HAdd. Nevertheless, we still have to take care of the scaling factor of outputs from this layer. At first thought, we may take the output scaling factor as $\Delta_w \Delta_i$, multiply the scaling factor of the weights and the inputs, denoted with Δ_w and Δ_i respectively. But, there is actually the potential for numbers to increase in bit-size from the additions done in weighted sums. Recall that when adding two Δ-bit numbers, the upper bound on the sum is $\Delta + 1$ bits long. Therefore, the maximum number that can appear in the worst-case in the convolutions is about $\Delta_w \Delta_i \times 2^{\lceil \log n \rceil}$ bits long, where n is the number of terms in the summands. In practice, this bound is usually not achieved since the summands are almost never all positive. With negative numbers in the mix, the actual contribution from the summation can be moderated by some constant $0 < c < 1$.

4 Implementation

Implementation is comprised of two parts: 1) training on unencrypted data, and 2) classifying encrypted data. Training on unencrypted data is performed using the 5-layer network whose details are shown in Table 1. We use the Tensorpack framework [37] to train the network and compute the model. This

Table 1: HCNN architecture for training and testing MNIST dataset

Layer Type	Description	Layer Size
Convolution	5 filters of size 5×5 and stride $(2, 2)$ without padding.	$12 \times 12 \times 5$
Square	Outputs of the previous layer are squared.	$12 \times 12 \times 5$
Convolution	50 filters of size 5×5 and stride $(2, 2)$ without padding.	$4 \times 4 \times 50$
Square	Outputs of the previous layer are squared.	$4 \times 4 \times 50$
Fully Connected	Weighted sum of the entire previous layer with 10 filters, each output corresponding to 1 of the possible 10 digits.	$1 \times 1 \times 10$

part is quite straightforward and can be simply verified by classifying the unencrypted test dataset. For neural networks design, one of the major constraints posed by homomorphic encryption is the limitation of numerical precision of layer-wise weight variables. Training networks with lower precision weights would significantly prevent the precision explosion in ciphertext as network depth increases, and thus speed up inference rate in encrypted domain. To this end, we propose to train low-precision networks from scratch, without incurring any loss in accuracy compared to networks trained in floating point precision. Following [38], for each convolutional layer, we quantize floating point weight variables w to k bits numbers w_q using simple uniform scalar quantizer shown below:

$$w_q = \frac{1}{2^k - 1} round(w * (2^k - 1))$$

This equation is non-differentiable function, we use Straight Through Estimator (STE) [7] to enable the back-propagation. We trained the 5-layer network on MNIST training set with precision of weights at 2, 4, 8 and 32 bits, and evaluated on MNIST test set with reported accuracy 96%, 99%, 99% and 99% respectively. In view of this, we choose the 4-bit network for the following experiments. It's worth noting that CryptoNets [13, 14] requires 5 to 10 bits of precision on weights to hit 99% accuracy on MNIST test set, while our approach further reduces it to 4 bits and still maintain the same accuracy.

The second part is more involved since it requires running the network (with the pre-learned model) on encrypted data. First, we need to fix HE parameters to accommodate for both the network multiplicative depth and precision. We optimized the scaling factor in all aspects of the HCNN. Inputs were normalized to $[0, 1]$, scaled by 4 and then rounded to their nearest integers. With the low-precision network trained from scratch, we convert the weights of the convolution-type layers to short 4-bit integers, using a small scaling factor of 15; no bias was used in the convolutions. Next, We implement the network using NTL [32] (a multi-precision number theory C++ library). NTL is used to facilitate the treatment of the scaled inputs and accommodate for precision expansion of the intermediate values during the computation. We found that the largest precision needed is less than (2^{43}). This is low enough to fit in a single word on 64-bit platforms without overflow. By estimating the maximum precision required by the network, we can estimate the HE parameters required by HCNN.

The next step is to implement the network using a HE library. We implement HCNN using two HE libraries: SEAL and our GPU-accelerated BFV (A*FV) [2]. The purpose of implementing the network in SEAL is to facilitate a more unified comparison under the same system parameters. In addition, we would like to highlight a limitation in the Residue Number Systems (RNS) variant that is currently implemented in SEAL. Before delving into the details of our implementation, we introduce an approach that is commonly followed to choose the FHE parameters.

Table 2: HE parameters for 1- and 2-CRT channels designs. λ denotes the security level in bits.

Design	Parameter Set	N	$\log Q$	t	Depth	λ
1-CRT channel	1	2^{13}	330	$\{5522259017729\}$	4	82
	2	2^{13}	360	$\{5522259017729\}$	5	76
	3	2^{14}	330	$\{5522259017729\}$	4	175
	4	2^{14}	360	$\{5522259017729\}$	5	159
2-CRT channels	5	2^{14}	240	$\{2424833, 2654209\}$	4	252

4.1 Choice of Parameters

Similar to other cryptographic schemes, one needs to select FHE parameters to bound the known attacks computationally infeasible. We denote to the desired security parameter by λ measured in bits. This means that an adversary needs to perform 2^{λ} elementary operations to break the scheme with probability one. A widely acceptable estimate for λ in the literature is ≥ 128 bits [35], which is used here to generate the BFV parameters. We also show parameters for $\lambda = 80$-bit for comparison with previous works.

In this work, we used a levelled BFV scheme that can be configured to support a known multiplicative depth L. L can be controlled by three parameters: Q, t and noise growth. Q and t are problem dependent whereas noise growth is scheme dependent. As mentioned in the previous section, we found that t should be at least a 43-bit integer to accommodate the precision expansion in HCNN evaluation.

For our HCNN, five multiplication operations are required: 2 ciphertext by ciphertext (in the square layer) and 3 ciphertext by plaintext (in convolution and fully connected layers) operations. It is known that the latter has less effect on noise growth. This means that L needs not be set to 5. We found that $L = 4$ is sufficient to run HCNN in A*FV. However, SEAL required higher depth ($L = 5$) to run our HCNN. The reason behind this is that SEAL implements the BEHZ [6] RNS variant of the BFV scheme that slightly increases the noise growth. Whereas in A*FV, we implement the HPS [19] RNS variant that has lower effect on the noise growth. For a detailed comparison of these two RNS variants, we refer the reader to [1].

Having L and t fixed, we can estimate Q using the noise growth bounds enclosed with the BFV scheme. Next, we try to estimate n to ensure a certain security level. To calculate the security level, we used the LWE hardness estimator in [4] (commit `76d05ee`).

The above discussion suggests that the design space of HCNN is not limited depending on the choice of the plaintext coefficient modulus t. We identify a set of possible designs that fit different requirements. The designs vary in the number of factors in t (i.e., number of CRT channels) and the provided security level. Note that, in the 1-CRT channel, we set t as a 43-bit prime number, whereas in the 2-CRT channels, we use 2 22-bit prime numbers whose product is a 43-bit number. Table 2 shows the system parameters used for each design with the associated security level.

4.2 HCNN Inference Library

As most deep learning frameworks do not use functions that fit the restrictions of HE schemes, we designed an inference library using standard C++ libraries that implements some of the CNN layers using only additions and multiplications. Support for arbitrary scaling factors per layer is included for

flexibility and allows us to easily define neural network layers for HCNN inference. We give a brief summary of the scaling factor growth of the layers we used in Table 3.

Table 3: Scaling Factor Growth by Layer

Layer Type	Output Scaling Factor
Convolution-Type ($\sum_{i=1}^{n} w_i z_i$)	$\Delta_o = \Delta_w \Delta_i \cdot 2^{c\lceil \log n \rceil}$, for some $0 < c < 1$.
Square Activation ($f(z) = z^2$)	$\Delta_o = \Delta_i^2$.

Δ_i and Δ_w are the input and weight scaling factors respectively.

In Section 2.2, we introduced several types of layers that are commonly used in designing neural networks, namely activation, convolution-type and pooling. Now, we briefly describe how our library realizes these layers. For convolution-type layers, they are typically expressed with matrix operations but only require scalar additions and multiplications. Our inference library implements them using the basic form, $b + \sum_{i=1}^{n} w_i \cdot z_i$, for input $\mathbf{z} = (z_1, \ldots, z_n)$ and weights $\mathbf{w} = (w_1, \ldots, w_n)$.

For the other two, activation and pooling, some modifications had to be done for compatibility with HE schemes. In activation layers, the most commonly used functions are ReLU, sigmoid ($f(z) = \frac{1}{1+e^{-z}}$) and softplus ($f(z) = \log(1 + e^z)$). These are non-polynomial functions and thus cannot be directly evaluated over HE encrypted data. Our library uses integral polynomials to approximate these functions; particularly for our HCNN for MNIST data, we used the square function, $f(z) = z^2$, as a low-complexity approximation of ReLU. Pooling layers, as mentioned in Section 3.2, are not straightforward to implement with what HE offers. For this work, we choose to avoid pooling layers entirely, in contrast to CryptoNets [13, 14] which uses them in the training phase.

4.3 GPU-Accelerated Homomorphic Encryption

The HE engine includes an implementation of an RNS variant of the BFV scheme [20] that we implemented in previous works [2, 1]. The BFV scheme is considered among the most promising HE schemes due to its simple structure and low overhead primitives compared to other schemes. Moreover, it is a scale-invariant scheme where the ciphertext coefficient modulus is fixed throughout the entire computation. This contrasts to other scale-variant schemes that keep a chain of moduli and switch between them during computation. We use our GPU-based BFV implementation as an underlying HE engine to perform the core HE primitives: key generation, encryption, decryption and homomorphic operations such as addition and multiplication.

Our HE engine (shown in Figure 3) is comprised of three main components:

1. Polynomial Arithmetic Unit (PAU): performs basic polynomial arithmetic such as addition and multiplication.
2. Residue Number System Unit (RNSU): provides additional RNS tools for efficient polynomial scaling required by BFV homomorphic multiplication and decryption.
3. Random Number Generator Unit (RNG): used to generate random polynomials required by BFV key generation and encryption.

In addition, the HE engine includes a set of Look-Up Tables (LUTs) that are used for fast modular arithmetic and number theoretic transforms required by both PAU and RNSU. For further details on the GPU implementation of BFV, we refer the reader to the aforementioned works.

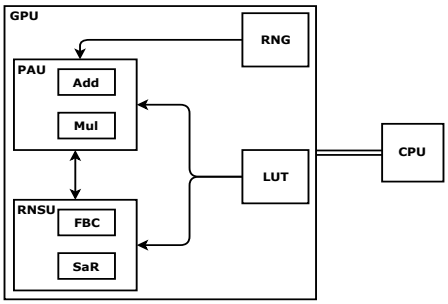

Figure 3: Top-Level Structure of Our GPU-Accelerated BFV Crypto-Processor.

We note that further task parallelism can be extracted from HCNN by decomposing the computation into smaller independent parts that can run in parallel. For instance, in the 2-CRT design, each channel can be executed on a separate GPU. In this scenario, the computation is completely separable requiring communication only at the beginning and end of computation for CRT calculations. Nevertheless, our implementation executes the channels sequentially on a single GPU.

5 Experiments

In this section, we describe our experiments to evaluate the performance of HCNN using the aforementioned mentioned designs. We start by describing the hardware configuration. Next, we present the results together with discussion and remarks on the performance.

5.1 Hardware Configuration

The experiments were performed on a server with an Intel® Xeon® Platinum 8170 CPU @ 2.10 GHz with 26 cores, 188 GB RAM and an NVIDIA® Tesla V100 GPU card with 16 GB on-board memory.

5.2 Methodology

We run our HCNN under the aforementioned designs using both SEAL, version 2.3.1 [33] and our A*FV library [2] on CPU and GPU, respectively. Note that our HCNN implementations execute the 2-CRT channels sequentially on a single GPU card. Timing results can be reduced into half if the network is run simultaneously on two GPUs. This also applies for SEAL as well.

Dataset. The MNIST dataset [27] consists of 60,000 images (50,000 in training dataset and 10,000 in testing dataset) of hand-written digits, each is a 28×28 array of values between 0 and 255, corresponding to the gray level of a pixel in the image.

5.3 Results

Table 4 shows the runtime of evaluating our HCNN using SEAL and A*FV on CPU and GPU, respectively. We include the timing of all the aforementioned parameter sets. It can be clearly seen that A*FV outperforms SEAL significantly in all instances. In particular, the speedup factors achieved are 61.68× (1-CRT at 76-bit security), 108.20× (1-CRT at 159-bit security) and 80.57× (2-CRT at 252-bit security). The results show that A*FV is superior at handling large FHE parameters where the maximum speedup is recorded. The amortized time represents the per-image inference time. Note that in parameter sets (3,4 and 5) we can classify the entire testing dataset of MNIST in a single network evaluation.

Table 4: Latency (in seconds) of running the HCNN on using SEAL and AFV on multi-core CPU and GPU, respectively.

Design	**Parameter Set**	**multi-core CPU**		**GPU**		**Speedup**
		SEAL	Amortized time	A*FV	Amortized time	Speedup
1-CRT channel	1	Failure	–	11.286	1.378×10^{-3}	–
	2	739.908	90.321×10^{-3}	11.996	1.464×10^{-3}	61.68×
	3	Failure	–	14.105	1.411×10^{-3}	–
	4	1563.852	156.385×10^{-3}	14.454	1.445×10^{-3}	108.20×
2-CRT channels	5	1860.922	0.186	23.098	2.310×10^{-3}	80.57×

The results also show the importance of low-precision training which reduced the required precision to represent the network output. This allows running a single instance of the network without plaintext decomposition (1-CRT channel). We remark that CryptoNets used higher precision training and required plaintext modulus of higher precision (2^{80}). Therefore, they had to run the network twice using 2-CRT channels. Moreover, our low-precision training did not affect the accuracy of the inference as we managed to achieve 99% accuracy.

We also note that our timing results shown here for SEAL are much higher than those reported in CryptoNets (570 seconds at 80-bit security). This can be attributed to the following reasons: 1) CryptoNets used the YASHE′ levelled FHE scheme which is known to be less computationally intensive compared to BFV that is currently implemented in SEAL [28]. It should be remarked that YASHE′ is no more considered secure due to the subfield lattice attacks [3], and 2) CryptoNets used much lower system parameters that guarantee only 80-bit security level whereas our implementation ensures much higher security level (>128-bit).

Lastly, we compare our best results with the currently available solutions in the literature. Table 5 shows the reported results of two previous works that utilized FHE to evaluate HCNNs. As we can see, our solution outperforms both solutions in total and amortized time. For instance, A*FV is 50.51× and 2.53× faster than CryptoNets and E2DM, respectively in classifying the entire MNIST dataset. Note that E2DM classifies 64 images in a single evaluation. This means that to classify the entire dataset, one would need more than 1 hour.

6 Conclusions

In this work, we presented a fully FHE-based CNN that is able to homomorphically classify the encrypted MNIST images with A*FV. The main motivation of this work was to show that privacy-preserving deep learning with FHE is dramatically accelerated with GPUs and offers a way towards efficient DLaaS.

Table 5: Comparison of runnning time (seconds) between prior FHE-based HCNN and A*FV HCNN.

Solution	**Runtime**		λ
	Total	Amortized time	
CryptoNets [13]	570	69.580×10^{-3}	80
E2DM [23]	28.590	450.0×10^{-3}	80
A*FV	11.286	1.378×10^{-3}	82

Our implementation included a set of techniques such as low-precision training, unified training and testing network, optimized FHE parameters and a very efficient GPU implementation to achieve high performance. We manged to evaluate our HCNN in 1-CRT setting in contrast to previous works that required at least 2-CRT. Our solution achieved high security level (> 128 bit) and high accuracy (99%). In terms of performance, our best results show that we could classify the entire testing dataset in 14.105 seconds, with per-image amortized time (1.411 milliseconds) $40.41\times$ faster than prior art.

In its current implementation, our HCNN have adopted the simple encoding method of packing the same pixel of multiple images into one ciphertext, as described in Section 3.1. This packing scheme is ideal for applications that require the inference of large batches of images which can be processed in parallel in a single HCNN evaluation. Other application may have different requirements such as classifying 1 or small number of images. For this particular case, other packing methods that pack more pixels of the same image in the ciphertext can be used. As future work, we will investigate other packing methods that can fit a wide-range of applications. Moreover, we will target more challenging problems with larger datasets and deeper networks.

References

[1] Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt Rohloff. *Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme.* In: *IACR Cryptology ePrint Archive* 2018 (2018), p. 589.

[2] Ahmad Al Badawi, Bharadwaj Veeravalli, Chan Fook Mun, and Khin Mi Mi Aung. *High-Performance FV Somewhat Homomorphic Encryption on GPUs: An Implementation using CUDA.* In: *IACR TCHES* 2018.2 (2018), pp. 70–95.

[3] Martin R. Albrecht, Shi Bai, and Léo Ducas. *A Subfield Lattice Attack on Overstretched NTRU Assumptions - Cryptanalysis of Some FHE and Graded Encoding Schemes.* In: *CRYPTO 2016, Part I.* Vol. 9814. LNCS. Springer, Heidelberg, 2016, pp. 153–178.

[4] Martin R Albrecht, Rachel Player, and Sam Scott. *On the concrete hardness of learning with errors.* In: *Journal of Mathematical Cryptology* 9.3 (2015), pp. 169–203.

[5] Louis J. M. Aslett, Pedro M. Esperança, and Chri. C. Holmes. *Encrypted statistical machine learning: new privacy preserving methods.* In: *ArXiv e-prints* (2015). arXiv: 1508.06845.

[6] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. *A full RNS variant of FV like somewhat homomorphic encryption schemes.* In: *International Conference on Selected Areas in Cryptography.* Springer. 2016, pp. 423–442.

[7] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation*. In: *CoRR* (2013).

[8] Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. *Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme*. In: *14th IMA International Conference on Cryptography and Coding*. Vol. 8308. LNCS. Springer, Heidelberg, 2013, pp. 45–64.

[9] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. *Fast Homomorphic Evaluation of Deep Discretized Neural Networks*. In: *CRYPTO 2018, Part III*. Vol. 10993. LNCS. Springer, Heidelberg, 2018, pp. 483–512.

[10] Zvika Brakerski. *Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP*. In: *CRYPTO 2012*. Vol. 7417. LNCS. Springer, Heidelberg, 2012, pp. 868–886.

[11] Zvika Brakerski and Vinod Vaikuntanathan. *Efficient Fully Homomorphic Encryption from (Standard) LWE*. In: *52nd FOCS*. IEEE Computer Society Press, 2011, pp. 97–106.

[12] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. *Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds*. In: *ASIACRYPT 2016, Part I*. Vol. 10031. LNCS. Springer, Heidelberg, 2016, pp. 3–33.

[13] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. *CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy*. Tech. rep. Feb. 2016.

[14] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. *CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy*. In: *Proceedings of the 33rd International Conference on Machine Learning*. 2016.

[15] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. *Manual for Using Homomorphic Encryption for Bioinformatics*. In: *Proceedings of the IEEE* 105.3 (2017), pp. 552–567.

[16] Junfeng Fan and Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Report 2012/144. 2012.

[17] Craig Gentry. *Fully homomorphic encryption using ideal lattices*. In: *41st ACM STOC*. ACM Press, 2009, pp. 169–178.

[18] Thore Graepel, Kristin Lauter, and Michael Naehrig. *ML Confidential: Machine Learning on Encrypted Data*. In: *ICISC 12*. Vol. 7839. LNCS. Springer, Heidelberg, 2013, pp. 1–21.

[19] Shai Halevi, Yuriy Polyakov, and Victor Shoup. *An Improved RNS Variant of the BFV Homomorphic Encryption Scheme*. In: (2018).

[20] Shai Halevi, Yuriy Polyakov, and Victor Shoup. *An Improved RNS Variant of the BFV Homomorphic Encryption Scheme*. Cryptology ePrint Archive, Report 2018/117. 2018.

[21] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N. Wright. *Privacy-preserving Machine Learning as a Service*. In: *PoPETs* 2018.3 (2018), pp. 123–142.

[22] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. *Secure Outsourced Matrix Computation and Application to Neural Networks*. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. 2018.

[23] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. *Secure Outsourced Matrix Computation and Application to Neural Networks*. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 1209–1222.

[24] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. *GAZELLE: A Low Latency Framework for Secure Neural Network Inference*. In: *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018, pp. 1651–1669.

[25] Konstantinos Kamnitsas, Christian Ledig, Virginia F.J. Newcombe, Joanna P. Simpson, Andrew D. Kane, David K. Menon, Daniel Rueckert, and Ben Glocker. *Efficient multi-scale 3D CNN with fully connected CRF for accurate brain lesion segmentation*. In: *Medical Image Analysis* 36 (2017), pp. 61–78.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. *ImageNet Classification with Deep Convolutional Neural Networks*. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Curran Associates Inc., 2012, pp. 1097–1105.

[27] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. *The MNIST Database of Handwritten Digits*. 1998. URL: `http://yann.lecun.com/exdb/mnist/`.

[28] Tancrede Lepoint and Michael Naehrig. *A comparison of the homomorphic encryption schemes FV and YASHE*. In: *International Conference on Cryptology in Africa*. Springer. 2014, pp. 318–335.

[29] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. *Oblivious Neural Network Predictions via MiniONN Transformations*. In: *ACM CCS 17*. ACM Press, 2017, pp. 619–631.

[30] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. *On Ideal Lattices and Learning with Errors over Rings*. In: *EUROCRYPT 2010*. Vol. 6110. LNCS. Springer, Heidelberg, 2010, pp. 1–23.

[31] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. *On Data Banks and Privacy Homomorphisms*. In: *Foundations of Secure Computation* 4.11 (1978), pp. 169–180.

[32] Victor Shoup et al. *NTL, a library for doing number theory, version 5.4*. 2005.

[33] *Simple Encrypted Arithmetic Library (release 2.3.1)*. `http://sealcrypto.org`. Microsoft Research, Redmond, WA. 2017.

[34] N. P. Smart and F. Vercauteren. *Fully homomorphic SIMD operations*. In: *Designs, Codes and Cryptography* 71.1 (2014), pp. 57–81.

[35] Nigel P Smart, Vincent Rijmen, B Gierlichs, KG Paterson, M Stam, B Warinschi, and G Watson. *Algorithms, key size and parameters report*. In: *European Union Agency for Network and Information Security* (2014), pp. 0–95.

[36] J.T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. *Striving for Simplicity: The All Convolutional Net*. In: *ICLR (Workshop Track)*. 2015.

[37] Yuxin Wu et al. *Tensorpack*. `https://github.com/tensorpack/`. 2016.

[38] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. *DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients*. In: *CoRR* abs/1606.06160 (2016). URL: `http://arxiv.org/abs/1606.06160`.