FHEON: A Configurable Framework for Developing Privacy-Preserving Encrypted Neural Networks

Nges Brian Njungle
STAM Center, Arizona State University
nnjungle@asu.edu

Eric Jahns
STAM Center, Arizona State University
jjahns@asu.edu

Michel A. Kinsy
STAM Center, Arizona State University
mkinsy@asu.edu

Abstract

The widespread adoption of Machine Learning as a Service raises critical privacy and security concerns, particularly about data confidentiality and trust in both cloud providers and the machine learning models provided. Homomorphic Encryption (HE) has emerged as a promising solution to these problems, allowing computations on encrypted data without decryption. Despite its potential, existing works that integrate HE into neural networks are often limited to specific architectures or classes. This leaves a wide gap in providing a framework for easy development of HE-friendly privacy-preserving neural network models similar to what we have in the broader field of machine learning.

In this paper, we present FHEON, an open-source configurable framework for developing privacy-preserving neural network models for inference using the CKKS scheme of HE. FHEON introduces optimized and configurable implementations of privacy-preserving neural network layers, including convolution layers, average pooling layers, ReLU activation functions, and fully connected layers. These layers are configured using standard parameters such as input channels, output channels, kernel size, stride, and padding to support arbitrary convolution neural network (CNN) architectures. Furthermore, FHEON provides utility functions that ease usage and adoption. We assess the performance of FHEON using several CNN architectures, including LeNet-5, VGG-11, VGG-16, ResNet-20, and ResNet-34. FHEON maintains encrypted-domain accuracies within ±1% of their plaintext counterparts for ResNet-20 and LeNet-5 models. Notably, on a consumer-grade CPU, the models built on FHEON achieved 98.5% accuracy with a latency of 13 seconds on MNIST using LeNet-5, and 92.2% accuracy with a latency of 403 seconds on CIFAR-10 using ResNet-20. Though configurable, FHEON outperform all state-of-the-art HE inference works in both latency and memory utilization. Additionally, FHEON operates within a practical memory budget requiring not more than 42.3 GB for VGG-16.

Keywords

Homomorphic Encryption, Privacy-Preserving Machine Learning

1 Introduction

Machine Learning as a Service (MLaaS) has emerged as a cornerstone for deploying machine learning (ML) models at scale, leveraging cloud platforms' computational power and storage capabilities [50]. Model providers typically use cloud infrastructures to train models on diverse datasets and then commercialize the resulting models by offering them as services also hosted on the cloud. However, the adoption of MLaaS introduces critical privacy and security challenges. When training datasets originate from multiple private sources, data contributors may hesitate to share their data due to trust concerns in the model owner or cloud provider. Similarly, in scenarios where pre-trained models are offered as services, the model's end users may be reluctant to submit sensitive data to the model provider or to cloud. In some cases, models rely on sensitive data, such as private health records, must strictly adhere to data protection regulations and laws, like HIPAA and GDPR [37] [41].

These challenges highlight a pressing need for mechanisms that provide protection to both data and models throughout the MLaaS pipeline. Homomorphic Encryption (HE) has emerged as a promising solution by enabling computations to be performed directly on encrypted data, thereby offering strong privacy with cryptographic security guarantees rooted in hard mathematical assumptions with minimal communication overhead [42, 46]. In the context of MLaaS, HE provides a natural defense against untrusted service providers, addressing the trust concerns of both data owners and end users. However, the practical deployment of HE in MLaaS remains severely constrained by its high computational and memory costs. Neural networks pose a unique challenge due to their reliance on a combination of diverse and computationally complex linear and nonlinear layers [1]. These constraints make it difficult to directly adapt existing implementation methodologies, forcing researchers to come up with new strategies and optimizations for their efficient implementation under HE constraints.

Prior efforts have proposed multiple optimizations, but their designs and implementations are generally focused on a narrow space, such as a specific class of convolution neural networks (CNNS) or even a single architecture (mainly the ResNet-20 architecture) [35, 48]. Such solutions are not guaranteed to generalize across the diverse neural network classes and architecture widely used in practice thus, limit applicability in real-world MLaaS pipelines. What is missing is a unified, reusable, and efficient framework that provides efficient general-purpose encrypted neural network building blocks just like in the broader ML field. The framework has to be developed with usability, efficiency, and scalability in mind. This framework will allow practitioners to develop privacy-preserving neural networks under HE constraints without reinventing nor re-implementing most of the core HE-friendly techniques already proposed in literature. The development of HE-friendly neural network layers is very complex and time-consuming. It also requires a great mastery of both HE and ML foundations and concepts. By

Proceedings on Privacy Enhancing Technologies 2026(4), 526–541

https://doi.org/10.56553/popets-2026-0133

abstracting away these implementation complexities, such a framework will allows developers and researchers to concentrate on their specific applications and research goals.

We present FHEON, an open-source, efficient, and configurable framework for building privacy-preserving neural networks for encrypted inference using the CKKS scheme [13]. Unlike most prior works which often propose optimizations for specific CNN layers, FHEON provides reusable and parameterized HE-friendly layers and a set of optimizations that collectively make encrypted neural networks inference development and deployment easy even on consumer-grade hardware. At the core of FHEON are highly optimized and configurable encrypted CNN building blocks, including convolution, average pooling, ReLU activation, and fully connected layers. These building blocks are designed to support the standard parameter settings for all CNN models, such as stride, padding, kernel size, input channels, and output channels. We further introduce utility functions that provide all the auxiliary operations necessary for the easy development of HE-friendly models. For example, FHEON supports different methods for importing model weights across heterogeneous training pipelines, as well as HE evaluation key reuse strategies that provide a trade-off between model latency and memory overhead for inference on encrypted data.

To demonstrate the effectiveness of FHEON, we instantiate several widely-used CNN models, including LeNet-5, VGG-11, VGG-16, ResNet-20, and ResNet-34, and evaluate them on MNIST, CIFAR-10, and CIFAR-100. Our models preserve high accuracy (e.g., 98.5% on MNIST with LeNet-5 and 92.2% on CIFAR-10 with ResNet-20) while operating within practical resource limits. Notably, FHEON achieves a very low memory footprint for encrypted ResNet-20 inference (20.4 GB) and latency (403 seconds) on a consumer-grade CPU. FHEON also ensures that even deeper networks like VGG-16 and ResNet-34 remain feasible within a practical memory footprint of 42.3 GB, thus substantially lowering the barrier for real-world adoption. Concretely, our main contributions in this work are:

- We design and develop FHEON, an open-source framework that provides optimized and parameterized HE-friendly neural network layers, enabling easy development of end-to-end encrypted models for inference across diverse architectures and classes. Our HE-compatible layers support generic configurations, including stride, padding, kernel sizes, and input and output channels. We also introduce novel algorithms to improve encrypted models inference.
- We design and support multiple helper functions that facilitate the development of encrypted models. Our helper functions include but are not limited to; abstraction of the HE cryptographic context and flexible pipelines for importing model weights independent of their training pipeline.
- We propose and implement multiple evaluation-key generation and reuse mechanisms for different pipelines. These pipelines trade-off the memory footprint and computational overhead requirements under different inference scenarios.
- We instantiate and benchmark multiple representative CNN models across multiple datasets, showing that FHEON consistently preserves accuracy while achieving the lowest reported memory footprint and latency for encrypted CNN inference on consumer-grade CPUs.

1.1 Summary of Technical Contributions

FHEON supports core neural network layers using carefully optimized HE-friendly algorithms, combining techniques from prior work with several novel system-level optimizations and algorithmic contributions. While FHEON relies on the widely adopted SIMD-based vector encoding for data representation and polynomial approximations for non-linear operations, its primary contribution lies in providing a *fully configurable, latency-aware, and memory-efficient framework* for encrypted neural networks development. Algorithmically, FHEON introduces novel HE-friendly algorithms, new mechanisms for rotation-key minimization, and parameterized computation pipelines. Below, we summarize the main technical contributions of FHEON across the different neural network layers.

Convolution Layer. FHEON generalizes previous approaches into a fully parameterized layer that supports arbitrary input/output channels, kernel sizes, padding, and stride configurations [29, 31]. Furthermore, FHEON introduces a rotation-key reuse strategy that minimizes memory overhead by reusing a single rotation key to align all channels. This technique provides a channel-agnostic convolution abstraction with a bounded number of rotation keys.

Padding. FHEON introduces an algorithm that exploits the flattened ciphertext layout to efficiently insert padding values. The method aligns adjacent rows through structured rotations and injects zero-padding in $O(1)$ operations per row pair. At the channel level, the design also reduces the number of required rotation keys. To the best of our knowledge, this is the first HE-friendly padding algorithm that supports arbitrary padding configurations while maintaining feasible computation and memory overheads.

Striding. FHEON implements two HE-friendly striding algorithms with complementary trade-offs. (i) a noval rotation-based extraction method that selectively retrieves stride-aligned outputs while preserving the ciphertext noise budget, and (ii) a generalized latency-sensitive multiplicative masking method [48]. In addition, FHEON introduces a configurable multi-channel striding layer, where multiple configurable channels are jointly processed.

Special 3×3 *Convolution (padding =1, stride=1).* Prior works observed that this configuration can avoid explicit expensive padding thus, introduce optimized algorithms. FHEON formalizes the observation into a dedicated, configurable layer with a complete algorithmic pipeline [48]. It introduces a dynamically generated masking to eliminate boundary artifacts without explicit padding or extraction thus, enabling re-usability irrespective of the input dimensions.

Pooling. FHEON introduces a dedicated HE-friendly algorithm for global average pooling, which aggregates all spatial elements within each channel into a single value. This approach minimizes overhead for architectures that employ global pooling.

Fully Connected Layer. Introduces a parameterized strategy that reduces the number of rotation keys required by grouping and merging output neurons based on a user-defined parameter. Instead of performing one rotation per output, the neurons are rotated as groups resulting in significant memory footprint reduction, particularly for models with a large number of output neurons.

Activation (ReLU) Layer. FHEON introduces a scaling-aware approximation pipeline, where inputs are first normalized using a configurable scaling parameter prior to the polynomial approximation. This design makes the approximation adaptive to varying input distributions and also enable more accurate approximations.

2 Related Works

Since the introduction of CryptoNets in 2016 [23], HE-friendly neural networks have advanced significantly. These developments can be broadly categorized into two generations, characterized by (i) improvements in computational efficiency and (ii) the ability to evaluate deeper and more expressive neural network models [48].

2.1 First Generation

The first generation of HE-based neural networks established the feasibility of performing inference directly over encrypted data using leveled HE schemes. These works relied on two key simplifications. First, non-linear activation functions were replaced with low-degree polynomial approximations, most commonly the square function, during both training and inference. Second, custom shallow network architectures were designed to remain within the noise and depth constraints of the leveled HE schemes used.

This generation predominantly utilized early HE constructions such as BGV [2] and BFV [19]. Representative works include CryptoNets [23], which demonstrated encrypted inference on MNIST, CryptoDL [26], which explored improved polynomial approximations of activation functions, and the work of Al Badawi et al. [3], which optimized encrypted CNN execution. Additional contributions such as FHE-DiNN [10] introduced discretized neural networks with sign activations evaluated during bootstrapping, while later implementations like HCNN [3] leveraged GPU acceleration to improve performance. Despite these advances, first-generation approaches imposed significant architectural constraints. The reliance on low-degree polynomial activations limited representational capacity, while the absence of efficient bootstrapping restricted circuit depth. As a result, these models were primarily effective on simple datasets such as MNIST, achieving reasonable accuracy but failing to generalize to more complex datasets or deeper architectures.

2.2 Second Generation

The second generation of HE-friendly neural networks focused on enabling deeper models for more complex tasks through improved algorithms and HE scheme capabilities. These works employ advanced techniques such as bootstrapping, improved ciphertext packing, and more accurate polynomial approximations to support the evaluation of deep CNNs. Two dominant HE schemes are used in this setting: TFHE and CKKS. TFHE enables efficient gate-level bootstrapping, allowing effectively unbounded-depth computation by refreshing ciphertexts after each operation [14]. This makes it well suited for evaluating non-linear functions with high precision. However, TFHE operates on bit-wise representations and does not support SIMD-style parallelism, resulting in significantly higher latency for large-scale data processing [28, 40, 44]. Consequently, TFHE-based CNNs typically employ small architectures with only a few layers. A recent example is the work of Benamira et al. [7], which evaluates CIFAR-10 using a shallow CNN in 2256 seconds.

In contrast, CKKS has emerged as the most suitable HE scheme for ML workloads due to its support for approximate floating-point arithmetic and efficient SIMD-based parallelism [34]. These properties enable a more natural mapping from plaintext models to their encrypted counterparts. During training, standard activation functions are used, while their polynomial approximations are applied during encrypted inference. Significant progress has been made in CKKS-based CNNs. Lee et al. [34] implemented the first encrypted ResNet-20 achieving 91.31% accuracy on CIFAR-10 with a runtime of 2271 seconds. Kim et al. [31] improved convolution efficiency, achieving 92.04% accuracy with a reduced inference time of 255 seconds per image. However, these works showed a substantial memory footprint during inference. The work of Lee et al. reported over 512 GB with Kim et al. reducing this to about 100 GB of RAM. More recently, Rovida et al. [48] optimized these designs, achieving 91.53% accuracy on CIFAR-10 in 331 seconds while reducing memory usage to 15.1 GB. Notably, most of these optimizations are centered around a ResNet-20, highlighting the limited diversity of neural network models explored under HE constraints.

2.3 HE-based PPML Frameworks

While the aforementioned works focus on optimizing specific CNN architectures for encrypted inference, they do not provide a unified framework for developing diverse HE-friendly models. To address this, several frameworks have been proposed. ConcreteML [54] provides a high-level interface for building TFHE-compatible ML models using a PyTorch-like workflow. While it performs well for simpler ML tasks such as regression, neural networks remain slow and often impractical due to the underlying TFHE constraints.

Other approaches adopt a compilation-based strategy. Systems such as Fhelipe [32] and Orion [18] translate high-level neural network descriptions into optimized HE circuits. Similarly, earlier efforts such as nGraph-HE [8, 9] and SEALion [51] explored integrating HE into existing deep learning frameworks through compilation and abstraction layers. These approaches improve usability and accessibility, particularly for developers with limited HE expertise. However, they reduce flexibility by limiting fine-grained control over model design and optimization.

In contrast, our approach shifts the focus from optimizing individual CNN instances or relying on compilation pipelines. We propose an efficient, scalable, and configurable CKKS-based framework that enables developers to construct arbitrary HE-friendly CNNs directly. Our system provides a collection of well-optimized and configurable layers, along with supporting utilities that simplify the development process while preserving flexibility for custom optimizations. Our results show that FHEON achieves superior performance and resource efficiency compared to existing approaches.

2.4 Other Research Works

Additional research on HE-friendly CNNs has primarily focused on two directions: high-throughput model design and activation function optimization. High-throughput approaches leverage batching techniques to amortize computation across multiple inputs and often rely on powerful hardware to reduce latency. Notable examples include [6, 15, 52]. For instance, Baruch et al. [6] demonstrated encrypted ResNet-50 inference using high-end hardware, including

an NVIDIA A100 GPU, an AMD EPYC CPU, and over 200 GB of RAM. In parallel, other works aim to improve accuracy and efficiency by optimizing non-linear function evaluation. These include hybrid approaches that combine CKKS with TFHE for activation functions [4, 43], as well as alternative paradigms such as secure multi-party computation [16]. While these directions address complementary challenges, FHEON simplifies both by providing highly configurable HE-friendly layers and reusable model components. This design enables seamless integration of new optimizations, lowering the barrier for rapid experimentation and development.

3 Background

3.1 Homomorphic Encryption

Homomorphic Encryption (HE) enables computations to be performed directly on encrypted data, preserving privacy without requiring decryption. This property ensures that operations on ciphertexts produce encrypted results that, once decrypted, match the outcome of equivalent plaintext operations. Fully Homomorphic Encryption (FHE), first realized by Gentry in 2009 [21], achieved this capability for arbitrary computations by introducing the concept of bootstrapping. Bootstrapping is a noise-refreshing technique that securely re-encrypts ciphertexts while reducing accumilated noise to maintain correctness and security over unbounded operations. Subsequent FHE constructions have significantly improved HE's efficiency and applicability in real-world scenarios.

3.2 Cheon-Kim-Kim-Song (CKKS) Scheme

Among the various FHE schemes that haven been introduced, the CKKS scheme stands out as the most suitable for ML workloads as it is the only mainstream scheme supporting approximate arithmetic. Its security is rooted in the non-probabilistic Ring Learning with Errors (RLWE) problem, an extension of the Learning with Errors (LWE) problem introduced by Regev [47] and further formalized for polynomial rings by Lyubashevsky et al. [36]. At its core, the LWE problem involves reconstructing secret information from noisy linear equations generated using random samples. Mathematically, let $\mathbb{Z}$, $\mathbb{R}$, and $\mathbb{C}$ denote the sets of integers, real numbers, and complex numbers, respectively. Given a vector $b \in \mathbb{Z}_q^m$ and a matrix $A \in \mathbb{Z}_q^{m\times n}$, the standard LWE problem seeks an unknown vector $s \in \mathbb{Z}_q^n$ such that:

$$As + e = b \bmod q, \tag{1}$$

where e is a noise vector sampled from a specific error distribution, and q is a large prime modulus.

The LWE problem is extended to polynomial rings over finite fields, enabling the CKKS scheme to leverage the algebraic structure of polynomials for computational efficiency. This allows support for "slot packing" where multiple plaintext data elements are encoded into the coefficients of a single polynomial. This packing enables the Single Instruction Multiple Data (SIMD) parallelism paradigm in CKKS. Let $R = \mathbb{Z}[X]/(X^N + 1)$ represent a ring of polynomials modulo $X^N + 1$, where N is a power of 2. Given a message vector $v \in \mathbb{C}^{N/2}$, CKKS maps v into the ring R, utilizing at most half of the ring's coefficients to store data. This encoding ensures that polynomial multiplication in the ciphertext space corresponds to element-wise multiplication in the plaintext space, making the scheme highly efficient for parallel computations.

CKKS defines addition, multiplication, rotation, and bootstrapping operations for the manipulation of encrypted data. The SIMD paradigm enables parallel data processing by simultaneously applying a single operation across multiple data elements, all encoded within a single ciphertext. Homomorphic operations are performed in parallel across all slots, greatly enhancing computational throughput. However, due to CKKS's reliance on complex number encoding, at most half of the available slots of a ciphertext can be independently utilized. This limitation arises from the need to maintain conjugate symmetry in encoding to ensure correct decryption. Despite this limitation, the SIMD feature boosts performance gains through parallel processing of large-scale data, making it practical and scalable for real-world applications. In this work, we leverage the OpenFHE library's implementation of CKKS [5] which is based on the residual number systems version introduced in [12]. OpenFHE provides all the CKKS operations outlined in this literature as well as other advanced optimizations such as Fast Rotations [11], further boosting the operational efficiency and performance.

4 Threat Model

The threat model assumed by FHEON is consistent with most prior works leveraging HE for privacy and security guarantees. Specifically, we adopt a *semi-honest* (honest-but-curious) adversary model. Under this model, the client is assumed to be a trusted entity and the server is assumed to faithfully follow the prescribed protocol without deviation. However, the server do not launch active attacks such as tampering or denial-of-service but they may attempt to infer additional information about the encrypted data by analyzing the ciphertext or encrypted computation.

In our setting, we assume that the server holds the model weights in plaintext, as is typical in MLaaS deployments where the model is owned or maintained by the service provider. The client encrypts its input locally and transmits only ciphertexts to the server. The server then performs inference over these encrypted inputs using its plaintext model parameters and returns the encrypted prediction to the client. Since only the client possesses the decryption key, only the client can recover the inference result. This setting protects the confidentiality of client data while allowing the server to leverage its proprietary models without modification.

Although FHEON can be extended to support encrypted model weights, thereby also protecting model confidentiality, we do not pursue this setting in the present work. Our focus is on ensuring strong privacy for client data in a semi-honest setting, while maintaining practical performance and memory efficiency for encrypted inference. Furthermore, FHEON, like most works that rely on HE security assumptions, does not defend against active adversaries or side-channel attacks, nor does it address availability attacks such as denial-of-service. These types of attacks also remain out of scope.

5 Secure Neural Network Layers

This section presents an overview of the core CNN layers and operations supported by FHEON. We first examine the fundamental components of CNNs and review prior works to adopt and generalize the methodologies proposed. We further introduce additional optimizations across the different layers to improve efficiency. The selected layers presented in this work represent the essential set

of layers required to implement any CNN inference model. Other important layers such as Batch Normalization and SoftMax are primarily relevant during training, thus are considered as future extensions. Conversely, layers like MaxPooling that involve non-linear operations that are computationally expensive under the CKKS scheme are also considered as future research. Overall, our objective is to enable ML practitioners to easily develop efficient and privacy-preserving CNN inference models by employing FHEON.

5.1 Secure Convolution Layer

The convolution operation is a data processing technique used to extract features from input data [39]. In deep learning, the convolutions play a key role in automatically and adaptively learning spatial hierarchies of features by extracting them from its input data. A convolution utilizes filters, also known as kernels, which are applied to the input data by sliding them across the spatial dimensions of the layer's input data. At each position, the convolution operation, which is an element-wise multiplication, is performed between the kernel and the corresponding region of the input data, followed by a summation of the results to produce a single output value [30]. Mathematically let the input tensor be represented by X, with dimensions (C, H, W), where C corresponds to the number of input channels, H is the height, and W is the width of the input matrix. The kernel K has dimensions (F, C, k_h, k_w), where F represents the number of output channels, C is the number of input channels, k_h is the height of the kernel, and k_w is the width of the kernel. The convolution operation can be expressed mathematically as shown in Equation 2.

$$Y_{f,h_{\text{out}},w_{\text{out}}} = \sum_{c=1}^{C} \sum_{i=1}^{k_h} \sum_{j=1}^{k_w} X_{c,h_{\text{out}}+i-1,w_{\text{out}}+j-1} K_{f,c,i,j} + b_f \tag{2}$$

$X_{c,h_{\text{out}}+i-1,w_{\text{out}}+j-1}$ is the value of the input tensor at the corresponding input channel, height, and width indices. $K_{f,c,i,j}$ is the value of the kernel associated with the output channel f, input channel c, and kernel indices i and j. The term b_f is the bias added to the result for the output channel f. The convolution operation produces the output feature map Y, with dimensions $(F, H_{\text{out}}, W_{\text{out}})$ where the height and width of the output tensor are defined by:

$$H_{\text{out}} = H - k_h + 1, \tag{3}$$

$$W_{\text{out}} = W - k_w + 1. \tag{4}$$

Performing convolution operations naively within HE infrastructure poses significant challenges due to the substantial memory and computational resources required, since every value in the output tensor will require a distinct computation. To address this, Juvekar et al. introduced an optimized method known as Vector Encoding [29]. This technique exploits SIMD capabilities to perform convolution operations more efficiently, reducing memory overhead and improving overall performance. Vector Encoding has since become a standard in second-generation HE-based CNNs, as seen in works such as [31], [34], and [48]. A common limitation in the literature is that encodings are often optimized for the specific model being evaluated. To address this limitation, this work generalizes this technique and implements a configurable convolution layer that supports arbitrary input and output channels and also incorporates different padding and striding techniques. Our approach provides an adaptable, configurable HE-friendly convolutional layer similar to what we see in the broader ML field thus, applicable to all CNNs.

The convolution layer receives a flattened input tensor, represented as a single SIMD ciphertext. It also receives, the input width, kernel width, input channels, and output channels configurations. Following this, $k^2 - 1$ rotations of the input ciphertext are generated, where k denotes the kernel width. These rotations align all the elements of the input tensor that correspond to the first convolution operation as the first elements of the rotated ciphertexts. Each rotated ciphertext is then multiplied by a corresponding vector, which contains a repeated kernel value of the corresponding rotated ciphertext. The original ciphertext is also multiplied by a vector that contains the repeated value of the first element of the kernel matrix. The results of these multiplications are summed to produce the convolution output for the entire input.

In the second step, each rotated ciphertext is multiplied by the equivalent repeated kernel element vectors, and the resulting ciphertexts are summed to produce a ciphertext that contains the convolution result of the input ciphertext, denoted as A, as shown in Algorithm 1. Just like the input channels are flattened and aligned channel by channel, the repeated kernel value vectors of input channels are also flattened and aligned in a similar fashion. This ensures that kernel data for all input channels matches the appropriate data in the input tensor. This alignment allows the convolution operation to be applied for all elements across all input channels simultaneously, further improving our convolution layer's efficiency. This Stage requires a single multiplication for every kernel element and $k^2 - 1$ additions to sum the results into a new ciphertext.

After summing the ciphertexts, for each output channel, we construct C ciphertexts from A, where each ciphertext corresponds to the convolution result of a single input channel. This step require distinct rotation keys for each channel offset. Instead, FHEON introduces a key-reuse optimization that leverages a single rotation key of W^2 to iteratively generate all required channel-aligned ciphertexts. Specifically, for each output channel, we construct C ciphertexts from A, where each ciphertext corresponds to the convolution result of a single input channel. Rather than independently rotating A with different keys, we perform this process iteratively: starting from the initial ciphertext, each subsequent ciphertext is obtained by applying the same rotation of W^2 to the previous one. This effectively shifts the data across channel boundaries in a structured manner. As a result, the number of required rotation keys is reduced from $O(C)$ to $O(1)$ for this step.

The resulting C ciphertexts are then aggregated using homomorphic addition to produce a single ciphertext that contains the combined convolution values for the corresponding output channel. However, this intermediate ciphertext still contains redundant values across channel-aligned slots. To eliminate these, we apply a binary masking operation using a vector composed of w^2 ones followed by $C - 1$ zeros. This masking step selectively retains only the valid output positions while zeroing out irrelevant elements, yielding a cleaned ciphertext denoted as A^*.

To obtain the result, W_{out}, additional rotations of the resultant ciphertext are performed to extract all elements of $A_{\text{out,out}}$. To further optimize this process, we also reuse a single rotation key of W_{out} to extract all the information across all output widths. This process is repeated for all output channels. For the remaining $F - 1$ output channels, negative rotations of W_{out}^2 are applied to their ciphertexts, and the results are added into a single ciphertext to

Algorithm 1 Algorithm for Secure Convolution

Input: Encrypted input ciphertext c, kernel weights W, bias b, input width w_{in}, input channels C_{in}, output channels C_{out}, kernel width w_k, padding p, stride s
Output: Encrypted output ciphertext c_{out}
$w_{in} \leftarrow w_{in} + 2p$
$w_{out} \leftarrow \frac{w_{in} - w_k}{s} + 1$
$N_{in} \leftarrow w_{in}^2, \quad N_{out} \leftarrow w_{out}^2, \quad N_k \leftarrow w_k^2$
Initialize list $R \leftarrow [\,]$
for $i = 0$ **to** $w_k - 1$ **do**
  $c \leftarrow \text{Rotate}(c, w_{in})$
  Append c to R
  for $j = 1$ **to** $w_k - 1$ **do**
    Append $\text{Rotate}(c, j)$ to R
  end for
end for
Initialize list $F \leftarrow [\,]$
for $out\_ch = 0$ **to** $C_{out} - 1$ **do**
  Initialize list $V \leftarrow [\,]$
  for $k = 0$ **to** $N_k - 1$ **do**
    $V \leftarrow V \cup \{\text{Mult}(R[k], W[out\_ch][k])\}$
  end for
  $conv\_sum \leftarrow \text{AddMany}(V)$
  if $C_{in} > 1$ **then**
    Initialize list $S \leftarrow [conv\_sum]$
    for $ch = 1$ **to** $C_{in} - 1$ **do**
      $conv\_sum \leftarrow \text{Rotate}(conv\_sum, N_{in})$
      $S \leftarrow S \cup \{conv\_sum\}$
    end for
    $conv\_sum \leftarrow \text{AddMany}(S)$
  end if
  Append $conv\_sum$ to F
end for
$c_{out} \leftarrow \text{AddMany}(F)$
return c_{out}

construct the final feature map ciphertext Y. An equivalent bias vector is then added to form the final ciphertext. As shown in Algorithm 2, A generates $C - 1$ rotated ciphertexts of all input channels. This ciphertext is summed into A^*. Then, the extraction process is conducted to produce $A_{out,out}$, and the Bias vector $B_{out,out}$ is added to produce the final results $C_{out,out}$.

5.2 Secure Advanced Convolution Layer

The convolution layers perform the essential task of feature extraction. With additional parameters such as padding and striding, the convolution layer controls the spatial dimensions of the output. Padding is a technique used to preserve the spatial dimensions of the input tensor. By adding extra rows and columns of values (typically zeros) around the input tensor, padding ensures that the convolution operation can extract more information from edge regions to contribute to the feature map and prevents premature spatial downsampling, which can lead to information loss. On the other hand, striding refers to the step size at which the kernel is moved across the input tensor to perform the convolution operation. A stride of 1 moves the kernel one unit at a time. Stride values

Algorithm 2 Cleaning Secure Convolution and adding Bias

Input: Ciphertext $conv\_sum$, input width w_{in}, output width w_{out}, output size N_{out}, bias b, cleaning mask P_{out}, number of output channels C_{out}
Output: Cleaned output ciphertext c_{out}
Initialize list $F \leftarrow [\,]$
for $out\_ch = 0$ **to** $C_{out} - 1$ **do**
  Initialize list $V \leftarrow [\,]$
  for $l = 0$ **to** $w_{out} - 1$ **do**
    if $l == 0$ **then**
      $cleaned \leftarrow \text{Mult}(conv\_sum, P_{out})$
    else
      $conv\_sum \leftarrow \text{Rotate}(conv\_sum, w_{in})$
      $cleaned \leftarrow \text{Rotate}(\text{Mult}(conv\_sum, P_{out}), -(w_{out} \times l))$
    end if
    Append $cleaned$ to V
  end for
  $strided \leftarrow \text{AddMany}(V)$
  Append $\text{Rotate}(strided, -(out\_ch \times N_{out}))$ to F
end for
$c_{out} \leftarrow \text{Add}(\text{AddMany}(F), b)$
return c_{out}

from 2, lead to a downsampled output by skipping intermediate computations, reducing the dimensions of the feature map.

The choice of striding and padding values is crucial in CNNs as they directly affect the performance and feature preservation in models. The convolution layer with stride value S and padding value P produces the output feature map Y, with dimensions $(F, H_{\text{out}}, W_{\text{out}})$. The height and width are given by Equations 5 and 6, respectively.

$$H_{\text{out}} = ((H + 2P - k_h)/S) + 1, \tag{5}$$

$$W_{\text{out}} = ((W + 2P - k_w)/S) + 1. \tag{6}$$

5.2.1 Secure Advanced Convolution With Padding. Padding modifies the dimensions of the input tensor by introducing additional rows and columns around the input feature map. The padded tensor, denoted as X_p, has dimensions (C, H_p, W_p), where:

$$H_p = H + 2P, \tag{7}$$

$$W_p = W + 2P. \tag{8}$$

In the HE setting, padding introduces non-trivial computational and memory overhead due to the need for data movement across ciphertext slots. To address this challenge, FHEON introduces a novel SIMD-aware padding algorithm that exploits the structural properties of the flattened input tensor. Specifically, we leverage the observation that in the flattened SIMD representation, the first element of each row is adjacent to the last element of the previous row. Using this property, we efficiently introduce both horizontal and vertical padding through structured ciphertext rotations. We first rotate the ciphertext by W to align adjacent rows and inject $2P$ zero values in the appropriate positions. This operation enables padding insertion using a single rotation per pair of rows, effectively achieving $O(1)$ padding operations per row pair.

To complete the padding process, we introduce channel-level padding by appending additional zeros at the beginning and end of each input channel. This is achieved by rotating each channel ciphertext using a single rotation of W^2, followed by a single rotation

with key W_p+P to insert the required boundary zeros corresponding to the padded rows and columns. Notably, the SIMD encoding naturally provides zero-filled slots at the tail of the ciphertext, which are reused to avoid additional padding operations for the final channel.

Overall, this process is repeated independently across all C input channels. The novelty of this approach lies in enabling arbitrary padding configurations under HE while requiring only C+3 rotation keys, significantly reducing the memory overhead compared to naive padding implementations. To the best of our knowledge, this is the first HE-friendly padding algorithm that achieves both full configurability and low rotation-key complexity. The resulting padded ciphertext X_p is then passed to the convolution layer with updated dimensions defined in Equations 7 and 8.

5.2.2 Secure Advanced Convolution With Striding. Striding is a technique used to reduce the feature map's spatial dimensions by controlling the kernel's step size as it moves over the input. The striding value, denoted as S, determines the number of units the kernel shifts horizontally and vertically during the convolution process. When $S > 1$, the convolution operation skips $S - 1$ units both horizontally and vertically. This reduces the number of computed output values, effectively down-sampling the feature map. In the SIMD-based convolution process described earlier, the convolution operation for all elements in the input tensor is calculated at once; thus, striding introduces an additional processing overhead in selecting the right output values. Initially, all convolution-resulting values for every output channel are calculated with a single multiplication operation and additions resulting in the intermediate ciphertext A^*, which contains the convolution results for a striding value of $S = 1$. FHEON offers two methods for striding with different advantages and computational overheads.

The first method minimize the noise level in the ciphertext, making it easier to evaluate deeper models with fewer bootstrappings. Here, we perform an extraction process to isolate only the convolution values corresponding to the desired striding value $S > 1$. The striding process involves skipping values both vertically and horizontally in the ciphertext. Starting with the vertical striding process, for each vertical index i, if $i \mod S = 0$ and $i < H_{\text{out}}$, we extract the required convolution values by rotating A^* by $W \times i$, where W is the width of the input tensor. This operation aligns the convolution values for the i-th row into an intermediate ciphertext T_i. In the Horizontal Striding process, for each horizontal index j, we extract the first j-th values in T_i. By applying additional rotations of $S \times j$, where $S \times j < W_{\text{out}}$, we extract all right convolution values on the ciphertext. The extracted values are then merged to form the horizontally strided convolution values for the i-th row of the output feature map. After extracting the horizontal values for each row, we apply negative rotations of W_{out} to align the needed values and add them to produce the final output ciphertext for that corresponding channel output $A_{\text{out,out}}$. This process is repeated for all output channels, ensuring that the resulting feature map accurately reflects the specified strided values. This approach to striding uses W_{out} + 2 rotation keys while maintaining the integrity and functionality of the convolution with striding within the HE infrastructure, even for large and complex input tensors and kernels.

The second method implement an optimized striding method that significantly reduces the number of ciphertext rotations by strategically using multiplicative masks. The use of multiplicative masks drastically use up the ciphertext noise budget compared to the first approach but lead to lower runtime and reduced memory overhead since it requires fewer rotation keys. The method begins by generating a binary mask of size W^2. This mask contains ones in the positions corresponding to stride-aligned elements and zeros elsewhere. By multiplying the input ciphertext with this mask, we effectively eliminate all non-stride positions from the ciphertext, reducing unnecessary data early in the process. Following this initial pruning, we perform a sequence of $\log_2(W_{\text{out}})$ ciphertext rotations. After each rotation, we apply an additional multiplicative mask to remove redundant or overlapping values. These masks are constructed to preserve only the valid components introduced by each rotation. The resulting ciphertexts are then aggregated to gradually form a compact representation of the strided output. Finally, a set of W_{out} row-specific masks is applied in a loop, each isolating a single row in the output grid. These masked ciphertexts are accumulated to create the final compacted ciphertext that contains the correctly downsampled result. This final stage ensures that all retained values are aligned into contiguous slots in the ciphertext. Overall, this approach requires $\log_2(W_{\text{out}}) + W_{\text{out}}$ ciphertext multiplications and rotations, and uses only $\log_2(W_{\text{out}}) + 1$ unique rotation keys.

To further optimize the second striding approach, we introduce a novel configurable pipeline that performs striding across multiple output channels simultaneously, without incurring additional memory overhead. Specifically, instead of applying convolution and striding channel by channel, we first compute the convolution for a group of g output channels, where g is set equal to the number of input channels of that layer (The value of g is parameterized to support other possible pipelines and usages). The resulting intermediate outputs are then concatenated, and striding is applied jointly across all g output channels. This design yields several advantages in the HE setting. First, it eliminates the need for generating rotation keys that would otherwise be used only once, thereby further reducing the memory overhead. Second, it allows the the ciphertext ring dimension and number of slots to be constrained to a smaller dimension which is exactly equal to the maximum number of elements required for data movement within the CNN model.

5.3 Special Secure Convolution Layer

The computational overhead introduced by conventional padding and striding operations in HE-friendly neural networks makes their adoption and implementation highly challenging. Consequently, most existing works rely on a specialized convolution configurations that simplify implementation and also improve performance. However, this specialized design is restrictive, as it is only applicable to a limited subset of CNN configurations thus architectures. In many modern neural network architectures such as ResNet and VGG, convolution layers are typically configured with a padding of 1, a stride of 1, and a kernel size of 3×3. These settings preserve the spatial dimensions of the input and output tensors. This structural property creates a special case where the HE-friendly convolution operation can be efficiently implemented by exploiting the fixed relationship between the input and output tensors. Instead of explicitly padding the input tensor, a specialized binary cleaning mask of 1s and 0s is used to handle boundaries allowing convolution layers with these configurations to be performed more efficiently.

Rovida et al. [48] demonstrated the effectiveness of this optimization in their ResNet-20 state-of-the-art implementation. Building on this insight, FHEON offers a special secure convolution that generalizes this approach by fully parameterizing it. Since the input and output tensor dimensions remain unchanged, FHEON also offers a special configurable convolution layer that avoids both input padding and the extraction phase typically required in standard convolution, thus reducing computational cost. Instead, we perform a series of structured ciphertext rotations on the input ciphertext c, producing a set of rotated ciphertexts r_i, as outlined in Algorithm 3. Each r_i is then multiplied by its corresponding repeated kernel vector, forming the basis for efficient convolution.

Algorithm 3 Rotated Ciphertexts for Convolution

Require: Ciphertext c, input width w
Ensure: Rotated ciphertexts $r_0, \ldots, r_8$
$r_4 \leftarrow c$
$r_1 \leftarrow \text{Rot}_{-w}(r_4)$
$r_7 \leftarrow \text{Rot}_{w}(r_4)$
$r_3 \leftarrow \text{Rot}_{-1}(r_4)$
$r_5 \leftarrow \text{Rot}_{+1}(r_4)$
$r_0 \leftarrow \text{Rot}_{-1}(r_1)$
$r_2 \leftarrow \text{Rot}_{+1}(r_1)$
$r_6 \leftarrow \text{Rot}_{-1}(r_7)$
$r_8 \leftarrow \text{Rot}_{+1}(r_7)$
return $r_0, r_1, \ldots, r_8$

Algorithm 4 Generation of a mask repeated for all input channels in a function called build_mask

Input: *sp* (starting padding), *ep* (ending padding), w (window width), m (W^2), C (input channels)
Output: *tmask* (all channels mask)
Initialization: Initialize an empty list *mask*
for $i = 0$ **to** $sp - 1$ **do**
  Append 0.0 to *mask*
end for
while size of *mask* $< (m - ep)$ **do**
  for $j = 0$ **to** $w - 1$ **do**
    Append 1.0 to *mask*
  end for
  Append 0.0 to *mask*
end while
while size of *mask* $> m$ **do**
  Remove last element from *mask*
end while
while size of *mask* $< m$ **do**
  Append 0.0 to *mask*
end while
for $i = 0$ **to** $ep - 1$ **do**
  Set $mask[m - i - 1] \leftarrow 0.0$
end for
Initialize an empty list *tmask*
for $i = 0$ **to** $t - 1$ **do**
  Append *mask* to *tmask*
end for
return *tmask*

To ensure correctness of the convolution output without explicitly performing padding and extraction, FHEON introduces a mask-generation mechanism that operates directly on the kernel representations. Instead of modifying the input ciphertext, we dynamically construct binary masks that inject zeros into the repeated kernel vectors, thereby eliminating all invalid or out-of-bound contributions during the convolution. As shown in Algorithm 4, we first generate channel-aligned binary masks composed of 1s and 0s. These masks are designed such that the zero entries correspond exactly to positions that would otherwise require padding or would introduce incorrect boundary interactions. Algorithm 5 then systematically produces the complete set of masks required for all kernel positions. Each mask is applied to the corresponding repeated kernel vector of size W^2, resulting in a set of optimized kernel vectors that inherently encode the necessary boundary corrections. After applying the structured rotations and multiplying with the optimized kernel vectors, the convolution results across all input channels are aligned and aggregated using only a single reusable rotation key of W^2 irrespective of the input width. Finally, a bias vector is added to produce the output ciphertext.

Algorithm 5 Generate Masks for Special Convolution Layer

Input: m (W^2), C
Output: *all_masks* (set of 9 masks)
Initialize *all_masks* as an empty list
Append the following masks to *all_masks*:
  $build\_mask(W + 1, 0, W - 1, m, C)$
  $build\_mask(W, 0, m, m, C)$
  $build\_mask(W, 0, W - 1, m, C)$
  $build\_mask(1, 0, W - 1, m, C)$
  $build\_mask(0, 0, m, m, C)$
  $build\_mask(0, 1, W - 1, m, C)$
  $build\_mask(1, W - 1, W - 1, m, C)$
  $build\_mask(0, W, m, m, C)$
  $build\_mask(0, W + 1, W - 1, m, C)$
return *all_masks*

5.4 Secure Average Pooling Layer

The pooling layer is primarily used for dimensionality reduction [22]. It is conceptually similar to convolution with striding but differs in that its filters are uniform. Two widely used pooling types in neural networks are *Maximum Pooling* and *Average Pooling*. Maximum Pooling selects the maximum value from each region that the pooling kernel covers. On the other hand, Average Pooling computes the average of all values within the region covered by the pooling kernel [45]. Pooling layers generally use a stride of at least 2, allowing the filter to skip certain input regions, effectively reducing the data size in computation for subsequent network layers. In our context with CKKS, *Maximum Pooling* is computationally expensive as it involves evaluating a non-linear greater-than function. In contrast, *Average Pooling* is well-suited for encrypted computations as it can be performed using a single multiplication and addition, as shown in Equation 9. k is the kernel width, k^2 is the kernel size.

$$\text{Output}(i, j) = \frac{1}{k^2} \sum_{x=0}^{k-1} \sum_{y=0}^{k-1} \text{Input}(i + x, j + y), \tag{9}$$

To implement average pooling, FHEON uses the same algorithm as in the convolution layer to generate the different rotations required from the encrypted input. We then sum the rotated ciphertexts and multiply the resulting ciphertext by the precomputed value of $\frac{1}{k^2}$. This pre-computation value of $\frac{1}{k^2}$ efficiently introduces a division of k^2, ensuring that the average pooling values for the entire input tensor are calculated. We then follow the same algorithm as in the convolution layer with striding.

Additionally, FHEON supports a special form of pooling known as Global Average Pooling (GAP), which is widely used in modern architectures such as ResNet. Unlike standard average pooling, which operates over local regions, GAP aggregates all spatial elements within each feature map, producing a single value per channel thus, it is pooling with kernel size equals to dimensions of inputs. FHEON introduces a dedicated HE-friendly algorithm for global average pooling. Specifically, we perform a full reduction over the ciphertext by summing all elements within each channel across the W^2 spatial positions. These per-channel sums are then merged into a single ciphertext and scaled by a precomputed repeated vector of $\frac{1}{W^2}$, yielding the correct averaged output. This design enables an efficient realization of a full spatial reduction under HE.

Furthermore, when the pooling kernel size satisfies $k^2 \leq W^2$, we introduce an additional optimization. In this case, we multiply the input ciphertext by a precomputed repeated vector of $\frac{1}{k^2}$ and iteratively extract the required channel outputs using a single reusable rotation key. This is achieved by rotating the ciphertext by W^2 to align channel-wise values and merging the results.

5.5 Secure Fully Connected Layer

A fully connected layer maps all input data to an output space by learning a set of weights that define the relationships between every input and output neuron [20]. In this layer, each input neuron is connected to every output neuron, enabling the network to capture complex global patterns. Mathematically, given an input vector x of size n, weights W of size $m \times n$, biases b of size m, and output vector y of size m, it's computation is expressed as shown in Equation 10:

$$y_k = \sum_{i=1}^{n} W_{ki} x_i + b_k, \quad \text{for } k = 1, 2, \ldots, m. \tag{10}$$

In FHEON, the use of SIMD ciphertext makes the implementation of this layer highly efficient, requiring only a single homomorphic multiplication and n additions for each output neuron. For each output neuron, a SIMD multiplication is performed as $W_i \times x_i$, where W represents the weights vector associated with the output neuron, x is the input neurons vector, and i denotes the index of the input neuron. This is followed by n homomorphic additions to calculate the final value for the output neuron. This process is repeated m times to compute all the output neurons in the layer.

To consolidate the results, a straightforward implementation requires m ciphertext rotations, one per output neuron, along with m distinct rotation keys. This results in a rotation-key complexity of $O(m)$, which can lead to substantial memory overhead for layers with a large number of output neurons, such as those found in VGG-style architectures. To address this limitation, FHEON introduces a parameterized rotation-merging strategy that reduces the number of required rotations and rotation keys. Instead of performing one rotation per output neuron, the output neurons are partitioned into groups of size r, where r is a configurable parameter determined by the available contoinous rotation-keys in memory. Within each group, intermediate results are aggregated and merged using a contiguous set of rotation keys. Here, merging refers to applying negative rotations to each intermediate ciphertext based on its index in the group vector, such that the first slot of each ciphertext is aligned to a common position. The aligned values are then added together to produce a single output ciphertext. This design reduces the total number of rotation keys from $O(m)$ to $O\left(\frac{m}{r}\right) + r$.

By exposing the grouping factor as a configurable parameter, FHEON enables a flexible trade-off between memory consumption and computational cost, making the fully connected layer more practical for large-scale encrypted neural network inference.

Algorithm 6 Secure Fully-Connected Linear Layer

Input: Encrypted input ciphertext $\mathbf{c}_{in}$, weight matrix $\mathbf{W}$, bias $\mathbf{b}$, input size n, output size m, rotation step r
Output: Encrypted output ciphertext $\mathbf{c}_{out}$
Initialize list $\mathcal{R} \leftarrow [\,]$, $\mathcal{I} \leftarrow [\,]$
Set $j \leftarrow 0$, $k \leftarrow 0$
for $i = 0$ **to** $m - 1$ **do**
  Append Sum(Mult($\mathbf{c}_{in}, \mathbf{W}[i]$), n) to $\mathcal{I}$
  $j \leftarrow j + 1$
  if $j = r$ **or** $i = m - 1$ **then**
    Append Rotate(Merge($\mathcal{I}$), $-k$) to $\mathcal{R}$
    Clear $\mathcal{I}$
    $k \leftarrow k + r$
    $j \leftarrow 0$
  end if
end for
$\mathbf{c}_{out} \leftarrow$ Add(AddMany($\mathcal{R}, b$)
return $\mathbf{c}_{out}$

5.6 Activation Function (Secure ReLU)

Non linear layers also known as activation functions, play a crucial role in neural networks by allowing them to capture and represent complex, non linear patterns in the data [17]. Among the various activation functions used, the Rectified Linear Unit (ReLU) is one of the most widely adopted due to its simplicity and effectiveness [24]. It applies a threshold operation and sets all negative values to 0s, defined as shown in Equation 11. This non-linearity accelerates convergence and also helps reduce the vanishing gradient problem during training. Its approximation using Chebyshev polynomials has been widely adopted for use by HE-friendly works.

$$ReLU(x) = max(0, x) \tag{11}$$

One notable property of Chebyshev polynomials is that a polynomial of degree D has exactly d roots, all lying within the interval $[-1, 1]$. These roots are obtained as defined by Equation 12:

$$x_k = \cos\left(\frac{k\pi}{d}\right), \quad \text{where } 0 \leq k < d. \tag{12}$$

This structured arrangement of roots allows Chebyshev polynomials to achieve optimal uniform approximation for continuous

functions over the interval $[-1, 1]$. To approximate a non-linear function $f(x)$ over the interval $[-1, 1]$, it can be expressed as a truncated series of Chebyshev polynomials:

$$f(x) \approx \sum_{d=0}^{D} c_d T_d(x), \tag{13}$$

where c_d are the coefficients derived from $f(x)$, and D is the degree.

To apply Chebyshev approximations correctly, we normalize the input data to $[-1, 1]$. We introduce a scaling factor β that adjusts the input vector accordingly by multiplying it with $\frac{1}{\beta}$. Our design allows users to configure β based on the characteristics of their dataset. This scaling step provides several advantages. First, it ensures that the input lies within the approximation domain of the Chebyshev polynomial, thereby improving approximation accuracy for a fixed polynomial degree and setting a fix upper and lower bound for the approximation. Secondly, it offers flexibility to balance accuracy as different choices of β allow practitioners to adapt the approximation to varying input distributions. After scaling, the adjusted ciphertext is passed to the Chebyshev approximation function as shown in Algorithm 7, where n denotes the input size and D represents the degree of the Chebyshev polynomial.

Algorithm 7 Secure ReLU Using Homomorphic Encryption

1: **Input:** x, β, n, D
2: **Output:** y
3: **Initialization:** $v \leftarrow x$
4: **if** $\beta > 1$ **then**
5: $\quad p \leftarrow \text{GenerateScaleMask}(\beta, n)$
6: $\quad v \leftarrow \text{EvalMult}(x, p)$
7: **end if**
8: Define $f(z) : f(z) \leftarrow 0$ if $z < 0$; $f(z) \leftarrow \beta \cdot z$ otherwise
9: $y \leftarrow \text{EvalChebyshevFunction}(f(z), v, D)$
10: **return** y

In this section, we discuss the different algorithms implemented in FHEON. The computational cost and memory footprint of each layer are primarily determined by the number of homomorphic multiplications, rotations, and required rotation keys. Table 1 summarizes these operations using asymptotic notation. Unlike most prior works, which typically focus on optimizing specific algorithms, our analysis provides a unified view of the operational characteristics across the different layers supported in FHEON. As such, the table and analysis are intended to serve as a practical reference for understanding the trade-offs between primitives and guiding their selection for encrypted neural networks development.

5.7 FHEON'S Support Functions

To support the development of complete HE-friendly CNN models, FHEON provides numerous support functions. In this paper, we have described those we find to be the most useful subset, but a complete list shall be made available as part of FHEON's documentation.

5.7.1 Context Generation. The context generation layer is responsible for initializing the HE cryptographic environment required for secure computation. This layer abstracts the complexity of CKKS parameter tuning, making the framework accessible to both novice users with little to no background in cryptography and advanced users who require fine-grained control over their encryption settings. At its core, the layer accepts essential user-defined parameters such a, such as the polynomial ring dimension, the number of slots, and multiplicative depth. These parameters are internally mapped to the appropriate settings in the OpenFHE library to construct a valid HE cryptographic context for the model.

Once the context is initialized, the layer automatically generates the associated cryptographic keys. The generated keys can be serialized and stored locally in a user-specified location for ease of use and reproducibility. In addition to the required inputs, users may optionally specify advanced parameters such as the scaling mode, scaling factor, and the rescaling strategy used during homomorphic operations. If any of these parameters are omitted, FHEON falls back on a set of default values aim to provide optimal computational performance, memory usage, accuracy, and security.

5.7.2 Encryption Layer. The encryption layer is responsible for securely encrypting user data using the previously generated HE context and public key. This layer acts as a bridge between plaintext data and its encrypted counterparts It takes a matrix as input, flattens it, and constructs a vector. It then encodes the vector into a CKKS polynomial and encrypts it using the user's public key, allowing the user to infer it using a FHEON's model without revealing any information about the input to the server during computations.

5.7.3 Weights Loading. Model weights in FHEON are processed per layer. Model weights are ingested in CSV format, making FHEON models independent of the underlying training methodology. Exporting model weights from frameworks such as PyTorch or TensorFlow for FHEON is straightforward, as no additional preprocessing or reformatting is required. FHEON provides two primary mechanisms for integrating model weights, designed to balance development-time efficiency with runtime flexibility. The first mechanism uses a preprocessing strategy, where all model weights are loaded into the system before inference. To support this, we provide utility functions that accept CSV files along with the corresponding kernel shapes and channel configurations, automatically reshaping and formatting the data to match the model's internal structure. This enables seamless weight initialization and is well-suited for scenarios where performance is critical, as well as for continuous inference. The trade-off is increased memory usage, since all weights must be loaded into memory.

The second mechanism enables dynamic weight loading, where weights are formatted and injected into the model at runtime. This approach is more memory-efficient and offers greater flexibility, making it suitable for resource-constrained systems or applications requiring on-the-fly model adaptability. However, it introduces a computational trade-off, as model weights must be continuously loaded and unloaded from memory during inference.

5.7.4 Evaluation Keys Management. These helper functions are responsible for orchestrating the generation, serialization, deserialization, and efficient usage of evaluation keys required during encrypted inference. Specifically, the evaluation keys include rotation keys, which enable slot-wise data movement in ciphertexts and bootstrapping keys, which are required to refresh ciphertexts and extend computation depth during model evaluation. We also offer helper functions for all the CNN layers discussed in Section 5

Table 1: HE Cost per Neural Network Layer in FHEON

Layer	Multiplications	Rotations	Rotation Keys
Convolution	$(k^2 \cdot F)$	$k^2 - 1 + F \cdot (C_{\text{in}} - 1 + 2 \cdot W_{\text{out}})$	$K + F + 1$
Padding	0	$C_{\text{in}} \cdot W \cdot (W + P - 1)$	$C_{\text{in}} + 3$
Striding (Rotation-based)	0	W_{out}^2	$W_{\text{out}} + 2$
Striding (Mask-based)	$\log W_{\text{out}} + W_{\text{out}}$	$\log(W_{\text{out}}) + W_{\text{out}}$	$\log W_{\text{out}} + 1$
Special Convolution	$9 \cdot F$	$9 + (F \cdot C_{\text{in}})$	5
Average Pooling	$C \cdot striding + 1$	$k^2 - 1 + striding$	$k^2 - 1 + striding$
Global Pooling	1	C_{in}	C_{in}
Fully Connected	m	m	$\frac{m}{r} + r$
ReLU (degree D polynomial)	$O(D) + 1$	0	0

that receive layer configurations and automatically determine and generate the unique rotation indices needed for that layer. The equivalent rotation keys for the indices can then be generated.

Just as weights are loaded, keys can be loaded into a model during preprocessing or using a dynamic approach. In the preprocessing approach, FHEON analyzes the entire CNN architecture to identify the complete set of unique rotation indices and bootstrapping configurations required for all layers. It then generates a unified set of evaluation keys, ensuring no redundant keys are created for the application. This method is ideal for performance-critical scenarios as all necessary keys are preloaded into memory. It is also the easiest way for users to load keys and outsourced key management within their model. However, this approach introduces higher memory usage. In the dynamic approach, model keys are changed during runtime. We provide helper functions to support on-demand evaluation key loading and offloading. This mode is useful for users who understand the structure and execution flow of their model and can strategically manage its keys into blocks that reduce memory overhead. It can be very beneficial in inferring on large models. By supporting both preprocessing and dynamic key strategies, the Evaluation Keys Management layer provides flexibility to balance performance, memory efficiency, and user control over the HE-friendly privacy-preserving models built on FHEON.

5.7.5 Bootstrapping Layer. Bootstrapping is an important concept in HE because it enables arbitrary computations. We provide a helper function that maps the CKKS bootstrapping implementation in OpenFHE to the user. This layer can be called on any ciphertext within FHEON networks, enabling arbitrary models to be evaluated.

5.8 Model Design

The design and implementation of encrypted neural networks using FHEON follow a structured pipeline that maps conventional ML workflows to the homomorphic domain. The process begins with cryptographic context initialization which stage the encryption enviroment. The encrypted inference is executed through a sequence of modular building blocks using FHEON layers. To sustain deep circuit evaluation, FHEON's bootstrap layer can be used within the model design. The pipeline concludes with an encrypted output, ensuring that only the client can decrypt for the final prediction. Figure 1 illustrates how arbitrary models are developed on FHEON.

5.9 Extensibility

A central design principle of FHEON is extensibility, enabling seamless integration of new neural network operations without requiring

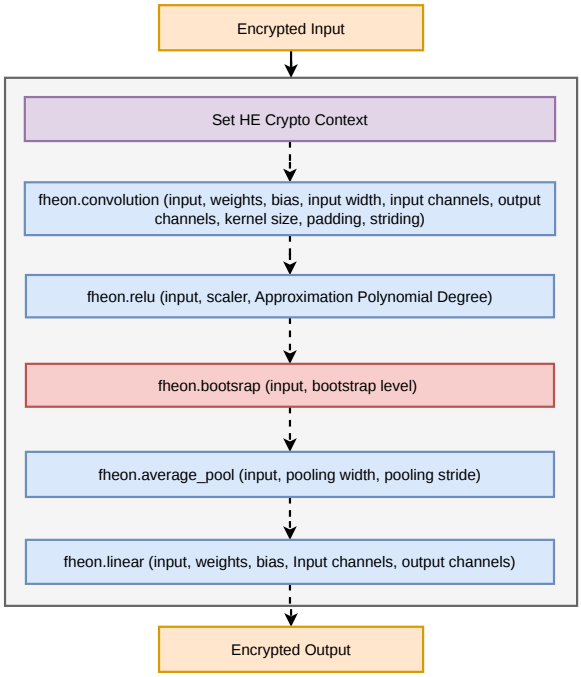

Figure 1: An architecture for developing an encrypted model using FHEON's layers. The model interconnected linear and non-linear layers, taking an encrypted user input and producing encrypted predictions

modifications to the underlying homomorphic execution pipeline and existing operations. This design is motivated by the practical need for rapid prototyping of encrypted deep learning models, new algorithms, and the addition of new layers such as new activation functions and normalization layers. In FHEON, extending functionality is intentionally lightweight. To add a new operation (e.g., Softmax, GeLU, Swish, or domain-specific transformations), a developer only needs to implement the corresponding function within the neural network C++ class and declare its interface in the associated header file. Once defined, the operation becomes immediately available for use within the model construction workflow.

At the model level, newly added operations can be invoked directly alongside existing components. Importantly, FHEON exposes a unified set of reusable homomorphic building blocks, allowing new operations to be composed from or interleaved with existing primitives without requiring changes to ciphertext representation, evaluation context, or execution logic. As a result, FHEON provides

a clean separation between cryptographic execution and model design. This allows researchers and practitioners to focus on algorithmic innovation while relying on a stable and reusable encrypted framework with a suit of configurable efficient established primitives, extensive examples, and a detailed documentation thereby accelerating experimentation and deployment of novel systems.

6 Evaluation and Results

All experiments in this study were performed on a consumer-grade CPU (AMD Ryzen 9 5900X 12-core processor with 128 GB of RAM). Unlike many existing works that rely on high-cost hardware, this work is also a significant step toward making HE-friendly, privacy-preserving ML feasible for everyday systems thus the reason for our experimentation setup choices. FHEON was developed using OpenFHE v1.2.4 (released March 2025) and have also been tested on v1.2.3, v1.3.0, v1.3.1, v1.4.0, and v1.4.2. FHEON will be actively maintained, with continuous support for new features, compatibility updates with future OpenFHE releases, and ongoing contributions to serve the user community. In addition to open-sourcing the code, we will host a comprehensive documentation portal that describes all available neural network layers and operations, configurations, comprehensive usage examples, and integration guidelines.

6.1 HE Security Parameters

For all experiments, we picked the CKKS parameters that enable efficient computations across all HE-friendly CNNs models. Balancing the cryptographic parameters is essential to achieve both runtime performance and security. To ensure fair and consistent comparisons, we adopted a uniform set of configurations, except for a lower polynomial degree and fewer slots for the LeNet-5 to better match its architecture. Specifically, we used a multiplicative depth of 25, a first modulus size of 50, a rescaling factor of 46, 4 digits for key switching, and the `flexibleauto` rescaling strategy. Table 2 summarizes the remaining parameters.

Table 2: FHE parameter sets for encrypted inference.

Parameter	LeNet-5	ResNets + VGGs
Polynomial Degree	16,384	32,768
Number of SIMD Slots	8,192	16,384

6.2 Models Evaluated

We validate our work across a wide range of CNN architectures, covering the most commonly used CNN classes. These include networks from the LeNet, VGG, and ResNet classes. Each class represents a distinct level of complexity and is generally applied to different problem domains, making them ideal for validating FHEON's effectiveness and efficiency in diverse usage scenarios.

LeNets are characterized by their relatively simple and shallow architectures. They were designed for grayscale images and are commonly used for tasks involving simpler datasets like the MNIST dataset [33]. LeNet-5 is the most widely used variant in CNN literature; thus, We developed and validated it on encrypted MNIST.

The VGG family was introduced in 2015 and has also been extensively used in the CNN literature due to its effectiveness on large-scale image tasks [49]. These networks employ deeper architectures with uniform convolutional layers. VGG-11 and VGG-16 are among the most commonly implemented models from this class. While VGG-11 has 11 layers, VGG-16 is deeper with similar characteristics, having 16 layers. We implemented these two VGG models on FHEON and validated them on an encrypted CIFAR-10 dataset.

The ResNet family is the most recent of the CNN classes [25]. It introduces the concept of residual connections to improve the training of deep networks by addressing the vanishing gradient problem. In previous HE-friendly CNN works, ResNet-20 has been pervasively used due to its efficient balance of model depth and performance. For this work, we also implemented ResNet-20 and ResNet-34 models. We evaluate the ResNet-20 model using the encrypted CIFAR-10 dataset, while the ResNet-34 model was validated on the more complex encrypted CIFAR-100 dataset.

By selecting representative architectures and datasets across varying levels of complexity, we ensure our methods are validated across a broad spectrum of tasks. Our evaluation across datasets, ranging from the simple MNIST to the complex CIFAR-100, also allows us to assess the robustness and efficiency of FHEON's models. Table 7 and 8 in the Appendix show all our CNN architectures, along with their configurations used in this study. The basic block structure in ResNets contains two convolution layers with an additional third convolution layer only used in down-sampling blocks. [38].

For this evaluation, we implemented our HE-friendly models using some of the building blocks provided by FHEON. For the LeNet-5 model, all model weights and evaluation keys were preloaded during an initial preprocessing phase. In contrast, for the deeper ResNet and VGG architectures, we adopted the dynamic loading strategies to balance memory efficiency with runtime performance. Specifically, these models were divided into blocks based on their downsampling stages, and for each block, the corresponding evaluation keys were serialized and loaded on demand while keys from the previous layers are cleared from memory. We further tailored our implementations to each model's architectural characteristics. For example, ResNet models used specialized secure convolution layers, whereas LeNet-5 employed generic secure convolution layers.

All our plaintext models were developed and trained in Python using PyTorch. During training, we used batch normalization after each convolutional layer for all models except LeNet-5. Batch normalization is used because it improves network training and keeps the data range within the network relatively small and stable [27]. After training, the learned batch normalization weights were folded back into their respective convolutional kernel weights before exporting them [53]. All weights were exported as CSV files and later imported into their HE-friendly equivalent model.

6.2.1 Bootstrap Placement. To mitigate ciphertext noise growth during inference, we strategically incorporate bootstrapping operations at carefully selected points within each network. Specifically, bootstrapping is applied prior to each ReLU activation (except for the first two instances), before the second pooling layer, and before the global pooling layer. These placements are guided by observed ciphertext noise accumulation during evaluation, ensuring that noise remains within acceptable bounds throughout the computation. The choice of bootstrapping locations is inherently dependent on both the architecture and the underlying HE parameter configuration. As such, different deployment scenarios may benefit from alternative bootstrapping strategies. In models employing multiplicative-intensive striding, we introduce additional

bootstrapping operations prior to downsampling. Table 3 summarizes the total number of bootstrapping operations required for each evaluated architecture. This design reflects the flexibility of FHEON, which provides fine-grained control over model construction.

Table 3: Bootstrap Operations for different Architectures.

Architecture	Number of Bootstraps
LeNet-5	3 or 4
ResNet-20	19 or 21
ResNet-34	31 or 34
VGG-11	11 or 15
VGG-16	15 or 18

6.3 Results

Table 4 illustrates the accuracy of the different CNN models evaluated under plaintext and encrypted settings. Notably, encrypted models developed with FHEON shows consistent performance with minimal degradation in accuracy. The table also shows the number of images inference in the encrypted domain for each model. The encrypted LeNet-5 and ResNet-20 models are within $\pm$ 1% of the corresponding plaintext accuracies of these models. Table 5 presents the memory utilization for all our models. FHEON, demonstrates balanced memory usage and inference speed across all models. The reported values are computed by selecting a random image from the validation set and averaging memory usage and inference time across 10 independent inference runs. These values were observed to be close range across all images in the full validation set.

Table 4: Accuracy of models in the Plaintext and FHE settings.

Architecture	Plaintext(%)	Encrypted(%)	Images
LeNet-5	98.8	98.5	1000
ResNet-20	92.8	92.2	560
ResNet-34	76.5	74.4	210
VGG-11	87.7	86.0	350
VGG-16	88.3	86.0	150

Table 5: Memory Usage and Inference Time

Architecture	Memory(GB)	Inference Time (s)
LeNet-5	4.2	13
ResNet-20	20.4	403
ResNet-34	25.6	1594
VGG-11	39.6	1647
VGG-16	42.3	2232

6.4 Comparison with Related Works

To contextualize our findings, we compare the ResNet-20 model implemented using FHEON with ResNet-20 models from state-of-the-art HE-based CNN works on the CIFAR-10 dataset, as summarized in Table 6. We select this architecture and dataset because they constitute the most widely adopted evaluation baseline in HE-friendly CNN literature and is also the only model studied in the state-of-the-art work of Rovida et al. [48]. All comparisons are conducted on the same AMD Ryzen CPU by benchmarking the publicly available implementations of Rovida et al. and Orion [18].

FHEON 's ResNet-20 model achieves a slightly lower memory footprint and improved inference latency compared to the design of Rovida et al. Specifically, Rovida et al. work recorded an inference time of 410 seconds and a memory footprint of 21.8 GB per image on our system. While their approach relies on a highly specialized and aggressively optimized design, including embedding preprocessed weights directly into the model and adopting a carefully tuned vector-encoding strategy with fixed rotation indices, it is tightly coupled to fixed deployment scenarios thus their work only supports a ResNet-20 model. In contrast, FHEON enables the construction of a wide range of HE-friendly CNN models through a configurable and reusable development pipeline that closely resembles conventional ML workflows yet able to match their performance.

We compare FHEON against Orion [18], a state-of-the-art compiler-based framework for HE-friendly model development. Under identical experimental settings, Orion requires 507 seconds of inference time and 119 GB of memory to process a single image. In contrast, FHEON achieves a 25.8% reduction in runtime and a 5.32× reduction in memory consumption. In terms of bootstrapping operations, our ResNet-20 model employed the latency-aware striding and used 21 bootstraps, whereas Orion requires 37 bootstraps [18]. These gains, together with the development flexibility position FHEON as a more practical and efficient alternative to building encrypted models.

Table 6: Comparison with Rovida et al. and Orion using the ResNet-20 Model

Proposal	Acc. (%)	Mem. (GB)	Lat. (s)
Ebel et al. (Orion) [18]	87.05	119	507
Rovida et al. [48]	91.53	21.8	410
FHEON	**92.2**	**20.4**	**403**

7 Conclusion

In this work, we introduced FHEON, a configurable and highly optimized framework for privacy-preserving neural network models development using HE. We used FHEON to develop various well-established CNN models, including LeNet-5, VGG-11, VGG-16, ResNet-20, and ResNet-34. Notably, our results show that even complex CNN architectures, such as VGG-16 and ResNet-34, can run on consumer-grade hardware with less than 64 GB of memory and achieve inference times within practical constraints. Furthermore, comparisons with prior works highlight substantial reductions in resource consumption and comparable performance with highly optimized state-of-the-art systems, underscoring the efficiency and scalability of FHEON. These findings establish FHEON as a comprehensive solution for developing efficient HE-friendly neural network models. By offering configurable CNN layers designed on CKKS, FHEON bridges the gap between theoretical advancements in HE and practical use in machine learning applications. Its compatibility with consumer-grade hardware strengthens its viability for widespread adoption in developing HE-friendly models.

While additional features such as encrypted weight processing are important, future research will also focus on optimizing FHEON to further improve computational efficiency and enhance its practicality in resource-constrained environments. Extending to high-throughput inference and incorporating automatic noise management strategies are also interesting future directions. Enabling FHEON to support applications beyond CNNs requires extending its functionality to additional ML operations, particularly those used in domains such as transformers. Finally, enabling privacy-preserving training through encrypted backpropagation and stochastic gradient descent also presents a compelling avenue for future research.

References

[1] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (Csur)*, 51(4):1–35, 2018.

[2] Nitesh Aggarwal, CP Gupta, and Iti Sharma. Fully homomorphic symmetric scheme without bootstrapping, 2014.

[3] Ahmad Al Badawi, Chao Jin, Jie Lin, Chan Fook Mun, Sim Jun Jie, Benjamin Hong Meng Tan, Xiao Nan, Khin Mi Mi Aung, and Vijay Ramaseshan Chandrasekhar. Towards the alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1330–1343, 2020.

[4] Wei Ao and Vishnu Naresh Boddeti. AutoFHE: Automated adaption of CNNs for efficient evaluation over FHE. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2173–2190, Philadelphia, PA, August 2024. USENIX Association.

[5] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. Openfhe: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022. https://eprint.iacr.org/2022/915.

[6] Moran Baruch, Nir Drucker, Gilad Ezov, Yoav Goldberg, Eyal Kushnir, Jenny Lerner, Omri Soceanu, and Itamar Zimerman. Polynomial adaptation of large-scale cnns for homomorphic encryption-based secure inference. In *International Symposium on Cyber Security, Cryptology, and Machine Learning*, pages 3–25. Springer, 2024.

[7] Adrien Benamira, Tristan Guérand, Thomas Peyrin, and Sayandeep Saha. Tt-tfhe: a torus fully homomorphic encryption-friendly neural network architecture. *arXiv preprint arXiv:2302.01584*, 2023.

[8] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. ngraph-he2: A high-throughput framework for neural network inference on encrypted data. In *Proceedings of the 7th ACM workshop on encrypted computing & applied homomorphic cryptography*, pages 45–56, 2019.

[9] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. ngraph-he: a graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM international conference on computing frontiers*, pages 3–13, 2019.

[10] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Annual International Cryptology Conference*, pages 483–512. Springer, 2018.

[11] Jung Hee Cheon, Wonhee Cho, Jaehyung Kim, and Damien Stehlé. Homomorphic multiple precision multiplication for CKKS and reduced modulus consumption. Cryptology ePrint Archive, Paper 2023/1788, 2023.

[12] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full rns variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*, pages 347–368. Springer, 2018.

[13] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. Cryptology ePrint Archive, Paper 2016/421, 2016.

[14] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: Fast fully homomorphic encryption over the torus. Cryptology ePrint Archive, Paper 2018/421, 2018. https://eprint.iacr.org/2018/421.

[15] Leo de Castro, Antigoni Polychroniadou, and Daniel Escudero. Privacy-preserving large language model inference via gpu-accelerated fully homomorphic encryption. In *Neurips Safe Generative AI Workshop 2024*, 2024.

[16] Abdulrahman Diaa, Lucas Fenaux, Thomas Humphries, Marian Dietz, Faezeh Ebrahimianghazani, Bailey Kacsmar, Xinda Li, Nils Lukas, Rasoul Akhavan Mahdavi, Simon Oya, et al. Fast and private inference of deep neural networks by co-designing activation functions. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2191–2208, 2024.

[17] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomputing*, 503:92–108, 2022.

[18] Austin Ebel, Karthik Garimella, and Brandon Reagen. Orion: A fully homomorphic encryption framework for deep learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 734–749, 2025.

[19] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012:144, 2012.

[20] geeksforgeeks. Introduction to convolution neural network, 2018.

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

[22] Hossein Gholamalinezhad and Hossein Khosravi. Pooling methods in deep neural networks, a review, 2020.

[23] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*, pages 201–210. PMLR, 2016.

[24] Juncai He, Lin Li, Jinchao Xu, and Chunyue Zheng. Relu deep neural networks and linear finite elements. *arXiv preprint arXiv:1807.03973*, 2018.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

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

[27] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[28] Lei Jiang and Lei Ju. Fhebench: Benchmarking fully homomorphic encryption schemes. *arXiv preprint arXiv:2203.00728*, 2022.

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

[30] Uday Kamath, John Liu, and James Whitaker. *Convolutional Neural Networks*, pages 263–314. Springer International Publishing, Cham, 2019.

[31] Dongwoo Kim and Cyril Guyot. Optimized privacy-preserving cnn inference with fully homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 18:2175–2187, 2023.

[32] Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. A tensor compiler with automatic data packing for simple and efficient fully homomorphic encryption. *Proceedings of the ACM on Programming Languages*, 8(PLDI):126–150, 2024.

[33] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[34] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *iEEE Access*, 10:30039–30054, 2022.

[35] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. Cryptology ePrint Archive, Paper 2021/783, 2021.

[36] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. Cryptology ePrint Archive, Paper 2012/230, 2012.

[37] Deven McGraw and Kenneth D Mandl. Privacy protections to encourage use of health-relevant digital data in a learning health system. *NPJ digital medicine*, 4(1):2, 2021.

[38] Piyush Nagpal, Shivani Bhinge, and Ajitkumar Shitole. A comparative analysis of resnet architectures. pages 1–8, 12 2022.

[39] Ivars Namatēvs. Deep convolutional neural networks: Structure, feature extraction and training. *Information Technology and Management Science*, 20(1):40–47, 2017.

[40] Harika Narumanchi, Dishant Goyal, Nitesh Emmadi, and Praveen Gauravaram. Performance analysis of sorting of fhe data: integer-wise comparison vs bit-wise comparison. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 902–908. IEEE, 2017.

[41] Claudia Negri-Ribalta, Marius Lombard-Platet, and Camille Salinesi. Understanding the gdpr from a requirements engineering perspective—a systematic mapping study on regulatory data protection requirements. *Requirements Engineering*, pages 1–27, 2024.

[42] Nges Brian Njungle, Eric Jahns, Zhenqi Wu, Luigi Mastromauro, Milan Stojkov, and Michel A. Kinsy. Guardianml: Anatomy of privacy-preserving machine learning techniques and frameworks. *IEEE Access*, 13:61483–61510, 2025.

[43] Nges Brian Njungle and Michel A Kinsy. Activate me!: Designing efficient activation functions for privacy-preserving machine learning with fully homomorphic encryption. In *International Conference on Cryptology in Africa*, pages 51–73. Springer, 2025.

[44] Nges Brian Njungle, Milan Stojkov, and Michel A Kinsy. A safety-centric analysis and benchmarks of modern open-source homomorphic encryption libraries. 2025.

[45] Vishal Passricha and Rajesh Kumar Aggarwal. Chapter 2 - end-to-end acoustic modeling using convolutional neural networks. In Nilanjan Dey, editor, *Intelligent Speech Signal Processing*, pages 5–37. Academic Press, 2019.

[46] Luis Bernardo Pulido-Gaytan, Andrei Tchernykh, Jorge M Cortés-Mendoza, Mikhail Babenko, and Gleb Radchenko. A survey on privacy-preserving machine learning with fully homomorphic encryption. In *Latin American High Performance Computing Conference*, pages 115–129. Springer, 2020.

[47] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Procedings of the thirty-seventh annual ACM symposium on Theory of Computing*, 2005.

[48] Lorenzo Rovida and Alberto Leporati. Encrypted image classification with low memory footprint using fully homomorphic encryption. Cryptology ePrint Archive, Paper 2024/460, 2024.

[49] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[50] Harry Chandra Tanuwidjaja, Rakyong Choi, Seunggeun Baek, and Kwangjo Kim. Privacy-preserving deep learning on machine learning as a service—a comprehensive survey. *IEEE Access*, 8:167425–167447, 2020.
[51] Tim van Elsloo, Giorgio Patrini, and Hamish Ivey-Law. Sealion: A framework for neural network inference on encrypted data. *arXiv preprint arXiv:1904.12840*, 2019.
[52] Yinghao Yang, Xicheng Xu, Haibin Zhang, Jie Song, Xin Tang, Hang Lu, and Xiaowei Li. Hydra: Scale-out fhe accelerator architecture for secure deep learning on fpga. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1174–1186, 2025.
[53] Edouard Yvinec, Arnaud Dapogny, and Kevin Bailly. To fold or not to fold: a necessary and sufficient condition on batch-normalization layers folding, 2022.
[54] Zama. Concrete ML: a privacy-preserving machine learning library using fully homomorphic encryption for data scientists, 2022. https://github.com/zama-ai/concrete-ml.

A APPENDIX

B Visualization of the Convolution Primitives

Figure 2 shows how an input tensor X with dimensions (C, H, W) is been flattened and aligned channel by channel as a SIMD Vector to be used as input to the first secure convolution layer in FHEON.

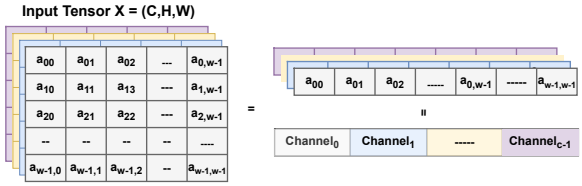

Figure 2: Flattening of an Input tensor into a SIMD Vector to be encrypted and used as an input of FHEON

Figure 3 shows FHEON's convolution on a 2x2 kernel.

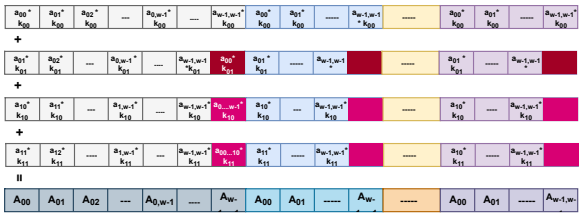

Figure 3: Rotated SIMD ciphertexts multiplied with their equivalent kernel vectors and summed to produce the A ciphertext.

Figure 3 shows the of output of the convolution layer in FHEON.

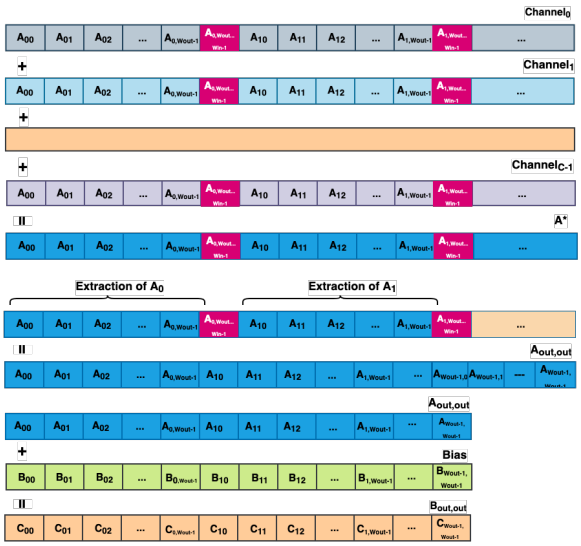

Figure 4: Secure convolution using SIMD showing the extraction process and addition of bias

Figure 5 shows the average pooling layer design of FHEON for a kernel size of 2x2.

Figure 5: Secure Average Pooling with kernel size of k^2. The resulting ciphertext is passed to a striding module.

C Model Architectures

Table 7: Architectural Comparison of ResNet-20 and ResNet-34. Downsampling convolutions with a 1×1 kernel, a stride of 2, and a padding of 1 are omitted from this table. Acronyms: Residual Blocks (RBs)

Network	Layer	Input Channels	Output Channels	Kernel Size, Stride, Padding
ResNet-20	Conv + ReLU	3	16	3×3, 1, 1
	3 RBs	16	16	3×3 , 1, 1
	3 RBs	16	32	3×3 , 1 , 1
	3 RBs	32	64	3×3 , 1 , 1
	GlobalAvgPool	64	64	Global , 1 , 0
	FC	64	10	N/A
ResNet-34	Conv + ReLU	3	64	7×7 , 2 , 3
	3 RBs	64	64	3×3 , 1 , 1
	4 RBs	64	128	3×3 , 1 , 1
	6 RBs	128	256	3×3 , 1 , 1
	3 RBs	256	512	3×3 , 1 , 1
	GlobalAvgPool	512	512	Global , 1 ,0
	FC	512	10	N/A

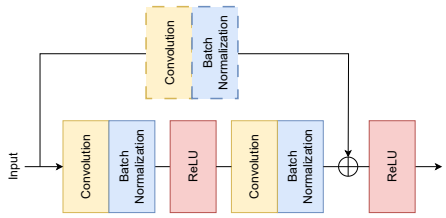

Figure 6: Basic Block Structure of ResNet. Dashed blocks signify that these operations only occur after downsampling.

Table 8: Architectural of LeNet-5, VGG-11, and VGG-16. All convolution (conv) and fully connected (FC) layers, except for the last layer of the network, are followed by a ReLU.

Network	Layer	Input Channels	Output Channels	Kernel Size, Stride, Padding
LeNet-5	Conv	1	6	5×5 , 1 , 0
	AvgPool	6	6	2×2 , 2 , 0
	Conv	6	16	5×5 , 1 , 0
	AvgPool	16	16	2×2 , 2 , 0
	FC	256	120	N/A
	FC	120	84	N/A
	FC	84	10	N/A
VGG-11	Conv	3	16	3×3 , 1 , 1
	AvgPool	16	16	2×2 , 2 , 0
	Conv	16	32	3×3 , 1 , 1
	AvgPool	32	32	2×2 , 2 , 0
	Conv	32	64	3×3 , 1 , 1
	Conv	64	64	3×3 , 1 , 1
	AvgPool	64	64	2×2 , 2 , 0
	Conv	64	128	3×3 , 1 , 1
	Conv	128	128	3×3 , 1 , 1
	AvgPool	128	128	2×2 , 2 , 0
	Conv	128	128	3×3 , 1 , 1
	Conv	128	128	3×3 , 1 , 1
	AvgPool	128	128	2×2 , 2 , 0
	FC	1024	1024	N/A
	FC	1024	1024	N/A
	FC	1024	10	N/A
VGG-16	Conv	3	16	3×3 , 1 , 1
	Conv	16	16	3×3 , 1 , 1
	AvgPool	16	16	2×2 , 2 , 0
	Conv	16	32	3×3 , 1 , 1
	Conv	32	32	3×3 , 1 , 1
	AvgPool	32	32	2×2 , 2 , 0
	Conv	32	64	3×3 , 1 , 1
	Conv	64	64	3×3 , 1 , 1
	Conv	64	64	3×3 , 1 , 1
	AvgPool	64	64	2×2 , 2 , 0
	Conv	64	128	3×3 , 1 , 1
	Conv	128	128	3×3 , 1 , 1
	Conv	128	128	3×3 , 1 , 1
	AvgPool	128	128	2×2 , 2 , 0
	Conv	128	128	3×3 , 1 , 1
	Conv	128	128	3×3 , 1 , 1
	Conv	128	128	3×3 , 1 , 1
	AvgPool	128	128	2×2 , 2 , 0
	FC	1024	1024	N/A
	FC	1024	1024	N/A
	FC	1024	10	N/A