POLITECNICO DI TORINO

Master's Degree in Data Science and Engineering

Master's Degree Thesis

Pruning-Aware Analysis of Accumulator Bit-Width in TFHE-Based CNN Inference

Supervisors

Prof. Andrea CALIMERA

Prof. Valentino PELUSO

Candidate

Giovanni PELLEGRINO

JULY 2026

Summary

This thesis investigates techniques for improving the efficiency of neural network inference under Fully Homomorphic Encryption (FHE), with a particular focus on quantization, pruning, and accumulator bit-width optimization. While FHE enables privacy-preserving inference by operating directly on encrypted data, its computational overhead remains a major obstacle to practical deployment, especially for deep convolutional neural networks.

The work first reviews the foundations of homomorphic encryption, neural network inference under FHE, and model compression techniques including quantization and pruning. Special attention is given to the role of accumulator bit width in integer arithmetic, as it significantly impacts both performance and hardware efficiency in encrypted and low-precision inference.

The core contribution of the thesis is an experimental study of the interaction between pruning strategies and quantization in the context of FHE-oriented neural networks. A layer-wise analysis is conducted on selected convolutional layers of ResNet architectures to evaluate how sparsity and quantization affect accumulator requirements and inference cost. This analysis is then extended to an end-to-end exploration on a compact ResNet model, using systematic experiments to identify configurations that balance accuracy, sparsity, and accumulator bit width.

Results show that carefully designed pruning and quantization pipelines can substantially reduce accumulator requirements while preserving acceptable model accuracy. The study highlights the existence of bottleneck layers that dominate accumulator constraints and demonstrates how targeted optimization can improve the overall efficiency of encrypted inference.

The findings contribute to a deeper understanding of joint optimization strategies for privacy-preserving neural network inference and provide practical guidelines for designing efficient FHE-compatible models.

Acknowledgements

To those who cannot find it in a messy room.

To those who lost it along the way.

To those who keep it in their pocket yet still stand still before the door.

To those who never had one.

A key.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
artificial intelligence

A2Q
Accumulator-Aware Quantization

AXE
Accumulator-Aware eXtension

BFV
Brakerski–Fan–Vaikuntanathan

BGV
Brakerski–Gentry–Vaikuntanathan

CKKS
Cheon–Kim–Kim–Song

CNN
Convolutional Neural Network

CPU
Central Processing Unit

FHE
Fully Homomorphic Encryption

GLWE
Generalized Learning With Errors

GPU

Graphics Processing Unit

ISQ

Intermediate-Value Slip Quantization

LUT

Look-Up Table

LWE

Learning With Errors

MAC

Multiply–Accumulate

ML

Machine Learning

NN

Neural Network

PBS

Programmable Bootstrapping

PHE

Partially Homomorphic Encryption

PTQ

Post-Training Quantization

QAT

Quantization-Aware Training

ReLU

Rectified Linear Unit

ResNet

Residual Network

SHE

Somewhat Homomorphic Encryption

STE

Straight-Through Estimator

TFHE

Torus Fully Homomorphic Encryption

Chapter 1

Introduction

In many modern applications of machine learning, the data used for inference is highly sensitive, raising serious privacy and security concerns. For example, healthcare and finance domains involve personal data that must remain confidential even while being processed by machine learning models. Conventional inference approaches require sending raw data to a server or cloud, creating potential vulnerabilities for data leaks or unauthorized access. This motivates the need for privacy-preserving inference techniques that can protect sensitive data during neural network processing.

Fully Homomorphic Encryption (FHE) has emerged as a promising solution to this challenge, as it allows computations to be performed directly on encrypted data without ever decrypting it. The results of the computation remain encrypted and only the data owner can decrypt the final output, ensuring end-to-end data confidentiality. This strong security guarantee, however, comes at the cost of significant computational overhead. FHE operations are several orders of magnitude slower than standard arithmetic, making naive encrypted inference extremely slow. The latency challenge is especially severe for deep convolutional neural networks (CNNs), which consist of many layers and arithmetic operations. Complex CNN models on large data can be prohibitively slow under FHE processing. Overcoming the latency and efficiency bottleneck of FHE-based CNN inference is therefore a critical problem.

Recent advances in cryptography have started to make FHE more practical for machine learning. In particular, the TFHE scheme (Torus Fully Homomorphic Encryption) introduced a fast bootstrapping technique that refreshes ciphertexts efficiently, allowing arbitrary-depth circuits to be evaluated homomorphically. TFHE supports *programmable bootstrapping* (PBS), a method that applies an arbitrary look-up table to a ciphertext during bootstrapping. This capability enables the evaluation of non-linear functions (such as neural network activation functions like ReLU or Sigmoid) on encrypted data, which was a major hurdle for

earlier FHE schemes. In essence, PBS allows each encrypted neuron output to pass through an activation function by homomorphically evaluating a pre-defined LUT, thereby making fully encrypted deep learning inference feasible. Thanks to these developments, several frameworks (e.g., Concrete-ML by Zama) can compile quantized neural networks into TFHE circuits that support basic operations and activations on ciphertexts. Nonetheless, even with modern schemes like TFHE, the computational cost of each operation remains high. It is therefore crucial to optimize neural network implementations for FHE, so as to reduce the number and complexity of homomorphic operations required.

A key observation is that the precision and size of the numbers being processed homomorphically have a direct impact on FHE computation cost. In particular, the *accumulator bit-width*—the number of bits needed to represent the intermediate sums in neural network layers—strongly influences the runtime and parameter settings for TFHE inference. For example, current TFHE-based compilers impose a limit on accumulator values to ensure they fit within a single programmable bootstrapping operation. In Concrete-ML, the sum computed by each neuron (the accumulator $v = \sum_i w_i x_i$ for weights w_i and inputs x_i) is constrained to at most 16 bits, i.e., $0 \leq v < 2^{16}$. This constraint comes from the fact that the PBS look-up table can only handle inputs up to a certain bit-width (around 16 bits with standard parameters). If the accumulator grows larger, it would either overflow the encrypted domain or require more expensive multi-step processing. Thus, controlling the accumulator bit-width is essential for FHE compatibility and efficiency. Smaller accumulators not only avoid overflow but also lead to faster homomorphic operations. In practice, quantization is applied to reduce the numeric precision of network parameters and activations, ensuring that all values (weights, activations, and sums) lie within a low-bit-width range (e.g. 8-bit or 4-bit integers). Indeed, in the context of TFHE (with a $\sim$16-bit ciphertext limit), quantizing neural networks to use small integers dramatically improves performance by keeping computations within the supported precision. Quantization alone, however, may not be sufficient for deeper layers that accumulate many terms; even low-bit inputs can sum to a large value if a neuron has a high fan-in (many inputs).

This thesis addresses the research problem of how to further reduce the required accumulator bit-width in quantized neural networks through *pruning*, without significantly compromising model accuracy. Pruning is a model compression technique that removes (sets to zero) a subset of the network's weights, effectively sparsifying the neural network. By eliminating certain weights, especially those that contribute little to the model's predictions, we can potentially control the growth of the sum in each neuron. Intuitively, if fewer weights are active (non-zero), then for a given set of inputs the maximum possible accumulated sum will be lower. Pruning has been proposed as a way to ensure FHE compatibility by avoiding accumulator overflow. By reducing the number of non-zero terms

in the summation, pruning limits the range of the accumulator and helps keep it within the bit-width budget. In other words, an *accumulator-aware* pruning strategy can bridge the gap between the complexity of modern neural networks and the strict precision constraints of FHE inference. The challenge, however, is that pruning (and aggressive quantization) can degrade the accuracy of the neural network. Removing too many weights or using too low precision might undermine the model's predictive performance. Therefore, a careful balance must be struck: we seek to prune and quantize the network just enough to minimize accumulator size, but not so much that accuracy falls below acceptable levels.

The goal of this thesis is to systematically explore the relationship between weight sparsity and accumulator bit-width in the context of FHE-compiled CNNs under the TFHE scheme. We focus on convolutional neural networks that have been quantized to low bit-width, and we investigate how different degrees of pruning affect the accumulator requirements at both the layer level and the network level. The overarching question is: *Can we significantly reduce the homomorphic computation cost by pruning weights (thereby reducing accumulator bit-width), while maintaining high model accuracy?* By answering this question, we aim to develop guidelines for designing FHE-friendly neural networks that meet both security and performance demands.

To this end, we present an in-depth study comprising both analytical and experimental components. First, we perform a fine-grained, layer-wise analysis on a standard deep network to isolate how pruning impacts the accumulator in each individual layer. Second, we extend our investigation to the entire network level, using a smaller CNN, to evaluate the combined effects of quantization and pruning under realistic accuracy constraints. Through these studies, we derive insights into when and where pruning is most beneficial for reducing FHE computation costs.

The main contributions of this thesis are as follows:

1. **Layer-wise sparsity analysis on ResNet-18:** We conduct a detailed analysis of the ResNet-18 convolutional neural network, pruning different layers in isolation to quantify the effect of sparsity on the accumulator bit-width. This layer-by-layer approach allows us to pinpoint which layers are most sensitive to accumulator growth and how much each layer's precision requirements can be relaxed through pruning, independent of other layers' effects.

2. **Full-network pruning exploration on ResNet-8 under accuracy constraints:** We implement and evaluate a smaller CNN (ResNet-8) with various combinations of quantization and pruning applied across the entire network. By imposing a target accuracy constraint, we explore the trade-offs between model sparsity and predictive performance. This experiment reveals how far one can prune and down-quantize a network as a whole while still meeting

acceptable accuracy, and how those compression techniques translate into reduced accumulator bit-width requirements for FHE inference.

3. **Practical insights for FHE-friendly model design:** From our results, we distill key insights about when pruning is most effective in reducing accumulator requirements in quantized CNNs. We identify patterns such as which network layers or structures benefit most from pruning, the interplay between quantization level and pruning percentage, and the diminishing returns of pruning in certain scenarios. These findings provide guidance on designing and optimizing neural networks for efficient homomorphic encryption deployment, indicating under what conditions sparsity can meaningfully ease FHE computational costs without sacrificing accuracy.

In summary, this work advances the understanding of how model compression techniques—specifically weight pruning in conjunction with quantization—can facilitate efficient CNN inference on encrypted data. By elucidating the balance between sparsity, precision, and accuracy, we contribute practical strategies for mitigating the latency of FHE-based deep learning while preserving the confidentiality of sensitive data.

Chapter 2

Background

2.1 Cryptography

Cryptography is the discipline that embodies the principles, means, and methods for the transformation of data in order to hide their semantic content, prevent their unauthorized use, or prevent their undetected modification [1]. Over the centuries, cryptography has evolved from simple classical ciphers, such as the Caesar cipher, to complex modern algorithms that underpin digital security today (post-quantum cryptography).

A cryptographic scheme can be described by the following fundamental elements:

- **Plaintext**: the original readable data that needs to be protected.
- **Ciphertext**: the transformed version of the plaintext after encryption.
- **Encryption**: the process of converting plaintext into ciphertext using a specific algorithm and key.
- **Decryption**: the reverse process of converting ciphertext back into plaintext using the appropriate key.
- **Key**: a piece of information (a string of bits) that determines the output of the encryption and decryption algorithms. Only authorized parties possessing the correct key should be able to perform decryption.

Cryptography can be divided into **symmetric** and **asymmetric** schemes.

- Symmetric cryptography uses the same **key** for both *encryption* and *decryption*.

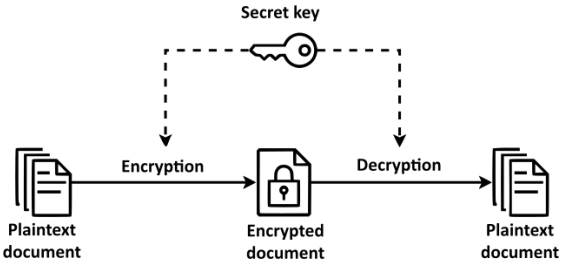


Figure 2.1: Simple diagram of a symmetric encryption scheme [2].

- Asymmetric cryptography employs a pair of *public* and *private* keys, without the need for a shared secret.

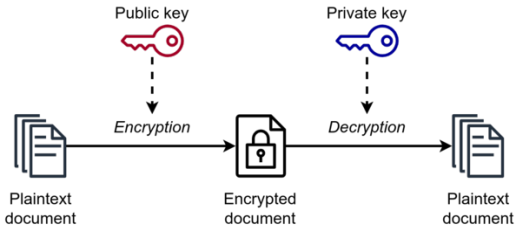


Figure 2.2: Simple diagram of an asymmetric encryption scheme [3].

Despite their strengths, traditional cryptographic schemes require data to be decrypted before it can be processed or analyzed. This limitation poses significant privacy risks, especially in scenarios where sensitive data is outsourced to third-party providers.

2.2 Homomorphic Encryption

Homomorphic encryption addresses the fundamental challenge of enabling computations on encrypted data without ever exposing the plaintext. This property is particularly valuable in scenarios where privacy is paramount, such as cloud computing, secure data outsourcing, encrypted search, and privacy-preserving machine learning.

For example, if two values are encrypted separately, it is possible to compute the encryption of their sum or product without ever decrypting them. Formally, given two elements e_1 and e_2, and an encryption function Enc, a scheme is homomorphic if operations like $Enc(e_1)+Enc(e_2) = Enc(e_1+e_2)$ or $Enc(e_1)*Enc(e_2) = Enc(e_1*e_2)$ hold.

Homomorphic encryption schemes are classified as:

- **Partially Homomorphic Encryption (PHE)**: supports only one operation (either addition or multiplication) an unlimited number of times.
- **Somewhat Homomorphic Encryption (SHE)**: supports both addition and multiplication, but only a limited number of operations.
- **Fully Homomorphic Encryption (FHE)**: supports an unlimited number of both additions and multiplications on ciphertexts. The concept of FHE was first realized by Craig Gentry in 2009, marking a breakthrough in cryptographic research [4].

2.2.1 Main Families of FHE Schemes

Over the years, several families of fully homomorphic encryption schemes have been developed, each with distinct characteristics and application domains:

- **BGV** [5]: The Brakerski-Gentry-Vaikuntanathan scheme is well-suited for integer arithmetic and supports batching, making it efficient for operations on vectors of data.
- **BFV** [6]: The Brakerski-Fan-Vaikuntanathan scheme is also designed for exact integer arithmetic and is widely used in privacy-preserving computations on encrypted databases.
- **CKKS** [7]: The Cheon-Kim-Kim-Song scheme enables approximate arithmetic over real or complex numbers, making it particularly suitable for privacy-preserving machine learning and signal processing applications where some loss of precision is acceptable.
- **TFHE** [8]: The Torus Fully Homomorphic Encryption scheme leverages the mathematical structure of the *torus* to enable fast gate-by-gate (bitwise) operations and highly efficient **programmable bootstrapping**, making it ideal for logic circuits and neural networks with binary or quantized activations.

The security of these (*asymmetric*) schemes relies on the hardness of lattice-based problems, such as the **Learning With Errors** (**LWE**) and **Generalized Learning With Errors** (**GLWE**) problems, which are considered computationally infeasible for both quantum and classical adversaries [9].

2.2.2 Bootstrapping, Programmable Bootstrapping, and Computational Overhead

A key challenge in FHE is the growth of noise in ciphertexts after each operation. If the noise exceeds a certain threshold, decryption fails. To overcome this, Gentry

introduced the concept of **bootstrapping**, a technique that refreshes ciphertexts by *homomorphically* evaluating the decryption circuit, thus reducing noise and enabling unlimited computations.

Programmable Bootstrapping (PBS) is used in TFHE, allowing the evaluation of *arbitrary functions* during the bootstrapping process, further enhancing the flexibility of FHE [10]. These functions are typically represented as Look-Up Tables (LUTs), which map input values to output values, enabling complex operations on encrypted data.

However, bootstrapping and PBS are **computationally expensive** and represent the main source of overhead in FHE schemes. Despite significant improvements, FHE remains much slower than traditional encryption, and reducing this overhead is a major focus of current research.

2.3 Neural Networks

Artificial **Neural Networks (NNs)** are computational models inspired by the structure and functioning of the *human brain*. They consist of interconnected processing units, known as **neurons**, which transform input signals through a combination of *linear operations* and *non-linear activation functions*. When multiple layers are stacked together, these networks acquire the ability to approximate highly complex functions, making them a fundamental component of modern **machine learning** [11].

A neural network is typically organized into three main parts: an **input layer**, one or more **hidden layers**, and an **output layer**. Each layer applies a learned transformation to its inputs, allowing the extraction of progressively more abstract features from raw data. In **fully connected (dense) layers**, each neuron is connected to all neurons in the previous layer, whereas in **convolutional layers** the network applies learnable filters using sliding-windows. **Convolutional Neural Networks** (**CNNs**) have become the dominant architecture for computer vision tasks due to their ability to exploit *spatial locality* and *parameter sharing* [12].

The **backpropagation algorithm**[**rumelhart1986learning**] computes the gradients of the loss with respect to the network parameters, and it is updated iteratively over multiple **epochs**. The learning rate controls the magnitude of each update and its value affects the convergence. Training often requires large datasets and hardware accelerators such as **GPUs**, which are suited for parallel tensor operations [13].

The power of neural networks comes from their ability to learn **hierarchical representations**. Early layers detect low-level patterns (e.g., edges, textures), while deeper layers capture high-level semantic features. Now, tasks such as classification, segmentation, and generation are easily implemented through NNs[11].

Increasing network depth, however, introduces challenges such as *vanishing or exploding gradients*, *overfitting*, and higher computational cost. Techniques such as **batch normalization**, **skip connections**, **dropout**, and **weight initialization** have been developed to mitigate these issues.

One of the most famous families of CNNs is the ResNet family, and it was taken into account during this work to bring some concrete applications.

2.3.1 Residual Networks (ResNet)

In classical CNNs, adding layers to create *deeper* (and potentially more powerful) networks often led to a frustrating technical problem: the **vanishing gradient**. During training, the gradients would become so weak as they traveled back through the layers that the early layers could barely learn: in this way, deep networks perform worse than shallower ones.

The solution, proposed by He et al. in 2016 [14], was to introduce a *network skip*. Instead of making each block of layers learn to map the input directly to the desired output, we let them learn the *difference* (the **residual**) between the input and the output. This architectural change allows gradients to "skip" one or more layers, flowing better through the network. As shown in Figure 2.3, the residual block introduces a skip connection that facilitates gradient flow during training.

ResNets thus became the state-of-the-art for computer vision. The notation **ResNet-N** (for example, ResNet-18, ResNet-50) conventionally indicates the total number **N** of parameterized layers. In our study, we chose two variants:

- **ResNet-18**: A deep but still manageable network, perfect for *layer-wise* analysis of the effect of quantization and pruning on very different types of layers (for example, an early layer and a deep one).

- **ResNet-8**: A much more compact and faster-to-train version, ideal for *full-network* experiments where we need to iterate quickly on different aggressive compression strategies. It is used for embedded-oriented tasks.

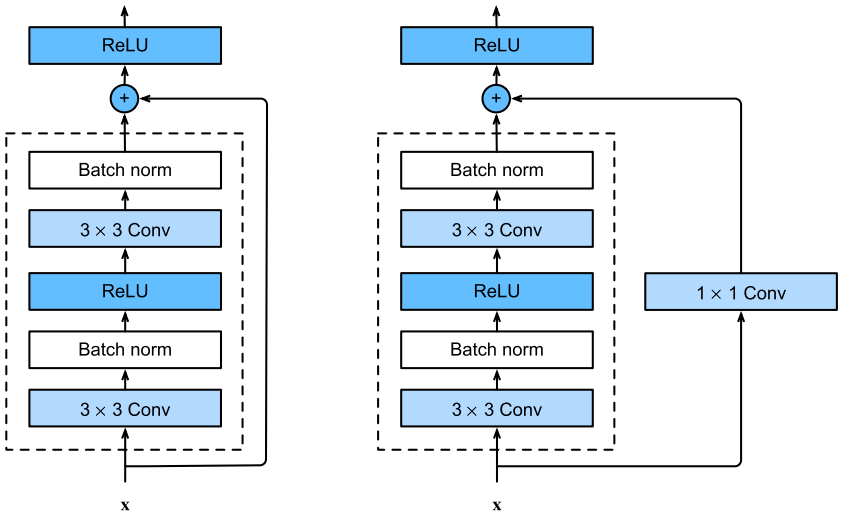


Figure 2.3: Typical ResNet block architecture. The 1×1 Convolution block helps the gradient to flow through the network without vanishing. Source: Zhang et al., *ResNet block*, Wikimedia Commons, licensed under CC BY-SA 4.0.

Despite their strengths, inference requires plaintext access to input data, which raises concerns regarding privacy and confidentiality in sensitive deployments. FHE enables neural network inference directly on encrypted data [15], removing the need for clients to reveal their raw inputs.

Convolutions

Convolutional layers are the fundamental building blocks of CNNs. Unlike fully connected layers, which connect every input neuron to every output neuron, convolutions exploit the spatial structure of the data by applying a small learnable kernel (filter) that slides across the input feature map.

Formally, given an input tensor $X \in \mathbb{R}^{C_{in} \times H \times W}$ and a convolutional kernel $W \in \mathbb{R}^{C_{out} \times C_{in} \times K \times K}$, the output feature map $Y \in \mathbb{R}^{C_{out} \times H' \times W'}$ is obtained by computing, for each output channel c and spatial position (i, j):

$$Y_{c,i,j} = \sum_{k=1}^{C_{in}} \sum_{u=1}^{K} \sum_{v=1}^{K} W_{c,k,u,v} \cdot X_{k,\, i+u,\, j+v}.$$

Each output value is therefore the result of a *multiply–accumulate* (MAC) operation between a local receptive field of the input and the convolutional kernel. The kernel is shared across all spatial locations, enabling parameter reuse and translation invariance. Hyperparameters such as stride and padding control how

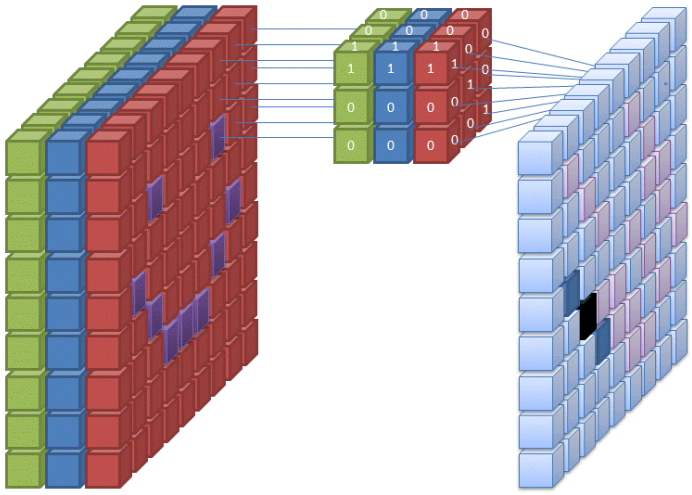


Figure 2.4: A filter (=kernel, neuron) in a convolutional artificial neural network. The filter takes a 9 pixel input for each color.[16].

the kernel is positioned over the input and determine the spatial resolution of the output.

Figure 2.4 illustrates the basic convolution process, where a small kernel scans the input image and produces a new feature map by computing local dot products. By stacking multiple convolutional layers, CNNs can progressively extract higher-level features, from simple edges and textures to complex semantic patterns.

From a computational perspective, convolutions are dominated by repeated MAC operations. For each output activation, a potentially large number of products must be accumulated, and the resulting partial sums determine the dynamic range required by the underlying arithmetic. This aspect becomes particularly critical in the context of FHE, where the size of these intermediate accumulations directly impacts ciphertext parameters, noise growth, and execution latency. Consequently, understanding the numerical behavior of convolutional accumulators is essential for designing FHE-efficient neural networks.

2.3.2 FHE for Neural Network Inference

Performing neural network inference under FHE introduces significant computational challenges. Non-linear activation functions—such as **ReLU**, **sigmoid**, and

softmax are not natively supported by encrypted arithmetic and must be approximated via **polynomials** or **Look-Up Tables (LUTs)**. Furthermore, encrypted operations are more expensive than plaintext ones, and network depth affects latency due to **noise accumulation** and **bootstrapping** requirements. Moreover, when neural networks are executed under FHE, the use of GPU acceleration does not guarantee faster execution. In fact, according to the official documentation of Concrete-ML, GPU-based FHE may even be slower than CPU execution depending on the hardware and the cryptographic parameters available, and in some cases the model cannot be compiled for GPU at all due to strict parameter constraints [17]. Consequently, FHE-aware neural network designs often employ **quantization** and **pruning** to remain computationally feasible [18].

2.3.3 Accumulator

In convolutional layers, each output feature is obtained by summing multiple products between input activations and kernel weights. These partial results are stored in an **accumulator**, a component that is always present in neural network implementations. In plaintext computation, the accumulator width is typically fixed to **32 bits**.

Under FHE, however, the situation is different. The **accumulator bit-width** determines the maximum dynamic range of the intermediate sums and therefore is the main character of the required ciphertext modulus, the rate of noise growth, and the cost of **PBS**. A larger accumulator demands larger ciphertexts, higher polynomial degrees, and more expensive bootstrapping operations, ultimately increasing the latency of encrypted convolution.

For this reason, the accumulator is a **primary computational bottleneck** in FHE-based convolutional neural networks. Reducing its bit-width means smaller ciphertexts, lower noise amplification, and faster bootstrapping. As result, we obtain the searched speed-ups in encrypted inference.

Techniques such as **quantization** and **pruning** are often employed to decrease the dynamic range of activations and weights, indirectly reducing the required accumulator width. However, their effectiveness varies across layers, and some layers remain bottlenecks even after aggressive quantization or sparsification.

2.4 Quantization

Quantization is the process of mapping *floating-point values* to a discrete set of *integer levels*, leading to a more efficient memory usage and less complex operations. quantization operates by approximating floating-point tensors using integer representations together with an appropriate scaling factor.

Most practical approaches rely on **uniform linear quantization**:

$$x_q = \text{round}\Big(q_x x_f - zp_x\Big),$$

where the scale factor q_x and the zero-point zp_x are defined as:

$$q_x = \frac{2^n - 1}{\max(x_f) - \min(x_f)}, \qquad zp_x = q_x \ \min(x_f).$$

The scale factor determines the resolution of the quantized values, while the zero-point shifts the integer domain to correctly represent distributions that are not centered around zero [19].

Uniform quantization can be implemented in two main variants: Asymmetric and Symmetric.

In **asymmetric quantization**, the minimum and maximum of the tensor are mapped exactly to the limits of the quantized range, allowing full utilization of the available integer levels but requiring zero-point corrections in convolutional and matrix-multiplication operations.

In **symmetric quantization**, the maximum absolute value is used to determine the representable range, eliminating the need for a zero-point and simplifying arithmetic, but potentially wasting precision when the distribution is heavily skewed (e.g., after ReLU activation).

The approximation introduced by quantization can be modeled as additive noise:

$$x_f = \frac{x_q + zp_x}{q_x} + \varepsilon,$$

where ε represents the **quantization error**. Minimizing this error is essential for preserving model accuracy and it can be reached through appropriate scaling strategies, clipping mechanisms or training.

In the context of **Fully Homomorphic Encryption (FHE)**, quantization becomes even more critical. Reducing the dynamic range of weights and activations directly affects ciphertext modulus, noise growth and the computational depth of encrypted operations. As a result, the choice of quantization strategy plays a central role in the practical feasibility of encrypted neural network inference.

2.4.1 Post-Training Quantization (PTQ)

Post-Training Quantization (PTQ) is a method applied *after* a neural network has been fully trained, where model weights and activations are converted from floating-point to integer representations without retraining the model. This approach is computationally efficient and requires no additional training data beyond a small calibration set used to determine the quantization parameters (scale and zero-point). PTQ typically involves analyzing the statistical distribution of activations

across a representative dataset to set appropriate dynamic ranges for each layer [20]. Common techniques include **per-channel** and **per-tensor** quantization, with symmetric or asymmetric scaling schemes. While PTQ is straightforward to implement and fast to apply, it often results in higher accuracy degradation compared to training-aware methods, especially for low-bit quantization (e.g., below 8 bits). In general model deployment, PTQ serves as a baseline method for reducing memory footprint and accelerating inference, but its limitations become evident when aggressive quantization is required without significant accuracy loss.

2.4.2 Quantization-Aware Training (QAT)

Quantization-Aware Training (QAT) integrates the quantization process directly into the training loop, allowing the model to adapt to the quantization error during backpropagation. During QAT, forward passes simulate quantization using *fake quantization* operations that emulate integer arithmetic while preserving gradient flow through **Straight-Through Estimators** (STE) [21]. This enables the network to learn weights that are more robust to quantization, often achieving better accuracy than PTQ at the same bit-width. QAT typically involves three phases:

1. training a full-precision model
2. fine-tuning with quantization simulation
3. deploying the quantized integer model.

In applications targeting resource-constrained hardware and edge devices, QAT is particularly valuable because it allows for more aggressive bit-width reduction while maintaining model performance. However, QAT requires *additional* computational resources and training time, and the quantization simulation must accurately reflect the constraints of the target hardware to ensure efficient inference.

2.5 Pruning

Pruning is a model compression technique that reduces the computational and memory footprint of neural networks by removing parameters that contribute less to the final output. Setting a subset of weights to zero (**sparsity**) decreases the number of non-zero multiplications during inference, which in turn can reduce the bit-width required in the accumulator of convolutional and fully connected layers.

The pruning process typically follows an iterative procedure: training a dense model, removing weights below a certain threshold, fine-tuning the remaining weights, and repeating until a target sparsity is met. More advanced approaches

incorporate **pruning-aware training**, where sparsity is gradually introduced during training to allow the network to adapt and recover accuracy [22]. Nevertheless, aggressive pruning, especially beyond 90% sparsity, often leads to irreversible **accuracy loss**, even when combined with QAT.

Pruning methods can be broadly classified into two categories: **unstructured** and **structured** pruning.

2.5.1 Unstructured Pruning

Unstructured pruning removes individual weights irrespective of their position in the network, resulting in a sparse, irregular connectivity pattern. This approach is flexible and can achieve high sparsity ratios without significant accuracy loss when combined with proper fine-tuning.

2.5.2 Structured Pruning

Structured pruning removes *entire groups* of weights, such as channels, filters, or blocks. The main drawback of structured pruning is its *higher impact on model accuracy* for a given sparsity level, since removing entire structural units may discard important features. Techniques like **channel pruning** and **filter pruning** require careful selection criteria (e.g., based on L1-norm, activation importance or gradient signals) to minimize accuracy degradation.

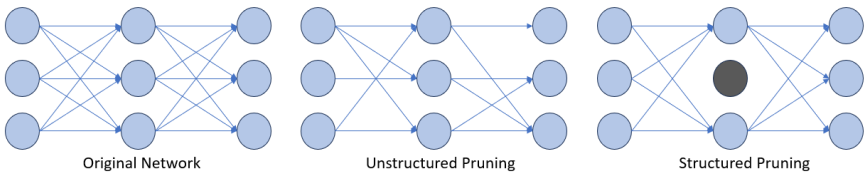


Figure 2.5: Comparison between an original neural network and the effects of unstructured and structured pruning. Unstructured pruning removes individual connections while preserving all neurons, whereas structured pruning removes entire neurons and their associated connections, resulting in a smaller but regular architecture[23].

Chapter 3

Related Works

This work is positioned at the intersection of three main research directions: *programmable bootstrapping for homomorphic neural network inference*, *compiler-based toolchains for TFHE deployment*, and *integer-only inference as a plaintext baseline motivating quantization.*

3.1 Programmable Bootstrapping in TFHE for Deep Neural Network Inference

The **TFHE** (*Fast Fully Homomorphic Encryption over the Torus*) cryptosystem introduces a key technique known as **Programmable Bootstrapping** (PBS) [24]. PBS allows a ciphertext to be refreshed while simultaneously evaluating an *arbitrary function* on the encrypted plaintext, typically represented as a programmable lookup table (LUT). This capability enables the homomorphic evaluation of **non-linear functions**, such as neural network activation functions, while keeping noise growth under control [25].

This approach represents a major breakthrough for privacy-preserving inference. Recent works have demonstrated, for the first time, that **pre-trained deep neural networks can be executed end-to-end under fully homomorphic encryption** without modifying or retraining the original model [25]. This stands in contrast with earlier approaches, which relied on polynomial approximations of activation functions. For example, CryptoNets replaced the ReLU activation with a quadratic polynomial and required training a model specifically tailored to encrypted execution [26].

By leveraging PBS, TFHE preserves the original network structure and supports efficient evaluation of operations such as *ReLU*, *sigmoid*, and *max-pooling* directly in the encrypted domain.

3.2 Concrete-ML Toolchain for Quantized Models under TFHE

Alongside theoretical advances in FHE schemes, software frameworks have emerged to make encrypted neural network inference practical and automated. In particular, the **Concrete** library [27] provides a compiler toolchain that translates *integer arithmetic programs* into homomorphic circuits executable with TFHE. Concrete automatically manages cryptographic aspects such as key generation, noise tracking, and bootstrapping, offering an end-to-end compilation flow from a quantized model to an executable FHE circuit.

Built on top of this infrastructure, **Concrete-ML** extends the toolchain with machine-learning-oriented abstractions, integrating **model quantization** and **FHE compilation** into a unified workflow. A key feature of Concrete-ML is its *compiler-driven numerical analysis*: during compilation, the tool inspects the integer arithmetic of each layer to determine the required **accumulator bit-width**. This quantity represents the number of bits needed to store the maximum intermediate sum produced by operations such as multiply–accumulate (MAC) in convolutional and fully connected layers.

The accumulator bit-width is a critical parameter in TFHE, as each additional bit leads to a substantial increase in ciphertext size, noise growth, and programmable bootstrapping cost. Concrete-ML mitigates this overhead by automatically inserting homomorphic rounding operations—implemented via PBS—when necessary, reducing the required precision while preserving correctness. Thanks to this compilation framework, a practitioner can start from a floating-point trained model, quantize it (typically to 8 or 16 bits), and obtain a functionally equivalent TFHE-compatible circuit with optimized cryptographic parameters [28]. This toolchain represents a fundamental step toward the practical deployment of deep neural networks in privacy-preserving settings.

3.3 Integer-Only Inference as a Plaintext Baseline

The execution of neural network inference using exclusively **integer-only arithmetic** was originally introduced in the plaintext domain to improve efficiency on resource-constrained hardware. A substantial body of work has shown that deep neural networks can operate with *low-precision quantized weights and activations*without significant loss in accuracy [19] (typically 8 bits or fewer). In particular, Jacob *et al.* propose **quantization-aware training** techniques that enable fully integer inference on CPUs and mobile accelerators, achieving substantial improvements in latency, energy efficiency, and memory footprint compared to

floating-point execution [19].

Similarly, Wu *et al.* demonstrate that both training and inference can be performed using integer arithmetic while maintaining competitive performance, confirming the robustness of low-precision representations even for complex architectures [29]. The underlying motivation is hardware efficiency: integer operations require less memory bandwidth, better exploit SIMD units, and significantly reduce computational cost.

In the context of **fully homomorphic encryption**, integer-only inference assumes an additional and fundamental role. Since TFHE natively operates over discrete integer domains, quantized integer models constitute a natural *plaintext baseline* for encrypted inference. By aligning neural network computations with modular integer arithmetic, quantization becomes a prerequisite for TFHE deployment rather than a mere optimization. Consequently, prior work on integer-only inference provides both a performance reference and a conceptual foundation for encrypted neural network execution, bridging the gap between efficient plaintext inference and practical FHE-based deployment [29].

3.4 Accumulator-Aware Quantization and Bit-Width Optimization

Recent work has investigated the problem of reducing accumulator bit width in quantized neural networks while preserving model accuracy. This problem is particularly relevant for hardware-efficient inference and for settings that require strict numerical correctness, such as encrypted computation.

Zhang et al. proposed *Intermediate-Value Slip Quantization* (ISQ), a quantization-aware training method that explicitly constrains intermediate weight values to suppress accumulator overflow. ISQ introduces accumulator-aware regularization and integrates bias and batch normalization quantization into a unified fixed-point training framework, enabling accurate inference with low-bit accumulators [30].

Colbert et al. introduced *Accumulator-Aware Quantization* (A2Q), a principled approach that derives theoretical bounds on accumulator bit width and enforces ℓ_1-norm constraints on weights during training to guarantee overflow avoidance [31]. A2Q inherently promotes sparsity and enables low-precision accumulation without arithmetic errors. Building on this idea, A2Q+ refines the original formulation by relaxing overly conservative constraints and improving weight initialization, resulting in a better trade-off between accumulator bit width and accuracy [32].

While these approaches focus on quantization-aware training, Colbert et al. later extended accumulator-aware techniques to the post-training quantization setting through the AXE framework. AXE augments layer-wise PTQ algorithms with theoretical overflow guarantees and supports multi-stage accumulation, enabling

accumulator-aware optimization for large-scale models without full retraining [33].

Together, these works establish accumulator-aware quantization as a key design dimension for efficient neural network inference, motivating the exploration of joint optimization strategies involving quantization, pruning, and accumulator bit-width control.

Chapter 4

Method

This chapter presents a general methodology to study the interaction between **quantization**, **pruning**, and **accumulator bit-width requirements** in neural networks. The framework is designed to be *architecture-agnostic* and applicable to different convolutional networks and deployment scenarios, including FHE-oriented inference pipelines.

4.1 Layer-wise Analysis Framework

Objective. The layer-wise framework is designed to isolate how sparsity and quantization affect accumulator growth in individual convolutional layers. The goal is to determine whether *sparsity alone* can reduce accumulator requirements and how accumulator bit-width relates to execution cost in integer convolution circuits.

Layer selection principle. The methodology requires selecting representative convolutional layers that operate in distinct regimes, for example shallow layers processing high-resolution feature maps and deeper layers with higher channel counts. This allows evaluating whether accumulator behavior is layer-dependent or governed by universal structural properties.

Experimental variables. Each configuration is defined by a tuple

$$(\text{layer}, n\_\text{bit}, \text{pruning scheme}, \text{sparsity})$$

where n_{bit} controls integer precision and sparsity determines the fraction of weights set to zero. Multiple pruning schemes may be considered to decouple sparsity effects from magnitude scaling.

Controlled inputs and weights. Synthetic inputs and weights are generated to ensure reproducibility and to remove dataset-dependent effects. Since quantization maps values to bounded integer ranges, accumulator growth is primarily driven by precision and sparsity rather than initialization variance.

Measured outputs. For each configuration, the framework extracts:

- accumulator bit-width from compiled integer circuits
- execution-cost estimates from the compilation toolchain

Repeated measurements are used to reduce runtime noise.

4.2 End-to-End Exploration Framework

Objective. The end-to-end framework evaluates whether layer-wise sparsity effects translate into network-level improvements and whether they yield global reductions in accumulator requirements.

Design space exploration. The framework explores configurations defined by

$$(n\_\text{bit},\ \text{sparsity},\ \text{pruning method})$$

applied to full neural networks. Both global and per-layer pruning strategies can be considered.

Compression pipeline. The methodology follows a staged pipeline:

1. train a full-precision baseline model
2. apply quantization-aware training to target precisions
3. perform iterative pruning with post-pruning fine-tuning

Acceptance criterion. Only configurations satisfying a predefined accuracy threshold are retained for accumulator analysis. This ensures that the study focuses on practically usable models.

Accumulator profiling. For valid configurations, the framework extracts per-layer accumulator bit-width profiles

$$B_\ell * \ell \in \mathcal{L} * \text{conv}$$

to identify bottleneck layers and characterize global trade-offs between sparsity, precision, and accumulator growth.

Chapter 5

Experimental Setup

This chapter instantiates the proposed methodology using concrete models, datasets, and implementation choices.

5.1 Preliminary Layer-wise Study

Layer selection. We select two convolutional layers from ResNet-18 representing distinct operating regimes: a shallow high-resolution layer (`conv1`) and a deeper high-channel layer (`layer3.1.conv1`). Table 5.1 reports their parameters.

Table 5.1: Parameters of the two convolutional layers used in the preliminary study.

Layer	K	S	P	C_{in}	C_{out}
`conv1`	7	2	3	3	64
`layer3.1.conv1`	3	1	1	256	256

Inputs and weights generation. Synthetic CIFAR-like inputs (3,32,32) are sampled from $\mathcal{N}(0,1)$ and convolution kernels from $\mathcal{N}(0,10)$ with fixed seeds.

Measurement protocol. Each configuration is compiled with Concrete-ML. Accumulator bit-width and execution time are measured over 100 repeated runs.

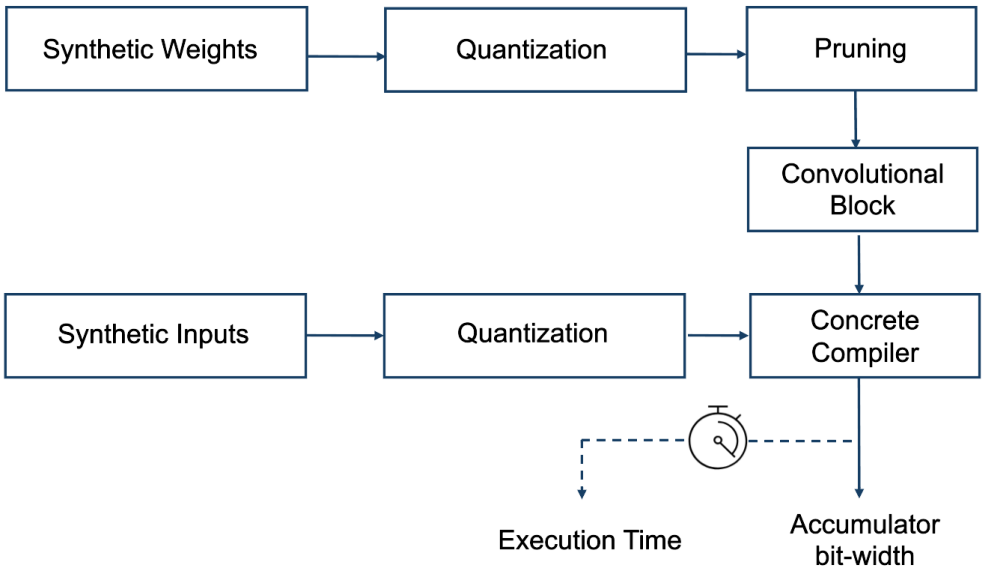


Figure 5.1: Preliminary study workflow.

5.2 Systematic Exploration on ResNet-8

Model and dataset. Experiments use ResNet-8 on CIFAR-10 with standard normalization.

Training configuration. The training and pruning hyperparameters used in the experiments are summarized in Table 5.2.

Table 5.2: Training and pruning hyperparameters used in the ResNet-8 experiments.

Parameter	Value
Dataset	CIFAR-10
Batch size	128
Total epochs	60
Optimizer	Adam
Initial learning rate	1×10^{-3}
Scheduler	StepLR (step=25, $\gamma = 0.5$)
Loss	CrossEntropyLoss
Target bit-widths	4,6,8
Pruning start/end epoch	0 / 40
Post-pruning fine-tuning	20 epochs

Acceptance threshold. Only models achieving >85% test accuracy are considered valid, following MLPerf Tiny criteria [34, 35].

Accumulator extraction. Valid models are compiled with Concrete-ML and per-layer accumulator bit-widths are extracted.

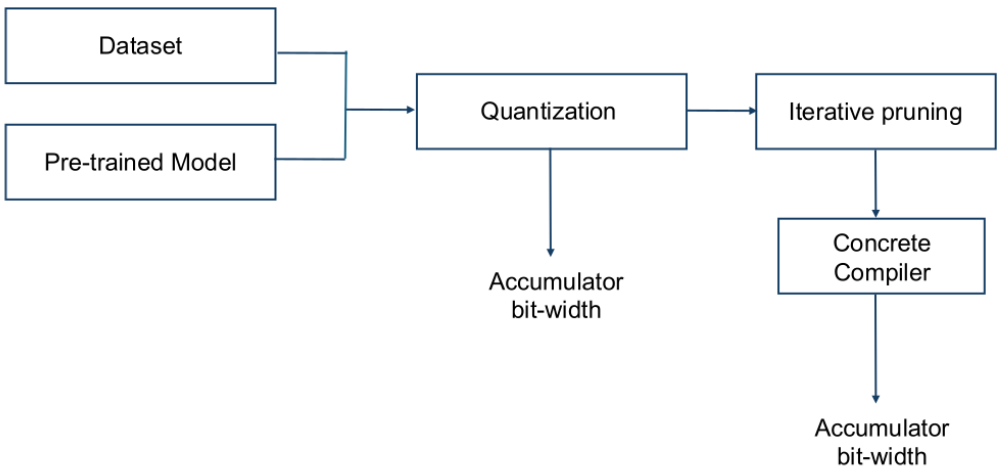


Figure 5.2: Overview of the ResNet-8 experimental workflow.

Chapter 6

Results

This chapter reports the experimental results obtained in this thesis. Following the methodology introduced in Chapter 4, we present:

- a controlled layer-wise study on selected ResNet-18 convolutions, aimed at understanding the relationship between sparsity, accumulator bit-width, and execution cost under FHE constraints;
- an end-to-end exploration on ResNet-8, where quantization and pruning are applied under an explicit accuracy constraint, and the resulting accumulator requirements are analyzed to identify network-level bottlenecks.

6.1 Layer-wise Results on ResNet-18

6.1.1 Evaluation metrics

The preliminary study isolates individual convolutional layers and evaluates two FHE-oriented quantities:

- **Accumulator bit-width**, extracted from the Concrete-ML compilation of the integer convolution circuit;
- **Inference time**, estimated by executing the compiled circuit.

Each experiment is defined by

$$(\text{layer}, n_{\text{bit}}, \text{pruning scheme}, \text{sparsity}),$$

with $n_{\text{bit}} \in \{4,6,8\}$. Two sparsity injection schemes are compared:

- **random pruning**;

- **magnitude-based retention**, where only the largest-magnitude weights are retained.

Unless otherwise specified, the tables reported in this section summarize the *best-performing configurations* observed in the explored design space.

6.1.2 Results for `conv1`

Table 6.1 summarizes the most relevant configurations obtained for the `conv1` layer under different quantization precisions. For each value of n_{bit}, we report the configurations providing the most favorable trade-off between accumulator bit-width and inference time.

Figure 6.1 illustrates the dependence of the measured inference latency on both sparsity and accumulator requirements.

The results show that, although reductions in accumulator bit-width can be achieved, especially under magnitude-based pruning, the corresponding speed-up remains limited. This suggests that, for shallow layers of this size, execution cost is largely dominated by structural factors and quantization precision, while sparsity alone provides only a marginal contribution to latency reduction.

Method	n_{bit}	**Sparsity**	**Accumulator BW**	**Avg. time (s)**	**Std (s)**	**Speed-up (%)**
BASELINE	4	0.23	10	2.36	0.04	/
Random	4	0.95	8	2.32	0.05	+1.69
Magnitude-based	4	0.90	7	2.06	0.06	+12.79
BASELINE	6	0.05	14	2.96	0.05	/
Random	6	0.95	12	2.93	0.10	+1.01
Magnitude-based	6	0.95	9	2.44	0.11	+17.73
BASELINE	8	0.01	18	3.61	0.07	/
Random	8	0.95	16	3.60	0.09	+0.28
Magnitude-based	8	0.90	13	3.05	0.10	+15.53

Table 6.1: ResNet-18 `conv1`: best observed configurations across sparsity and pruning schemes.

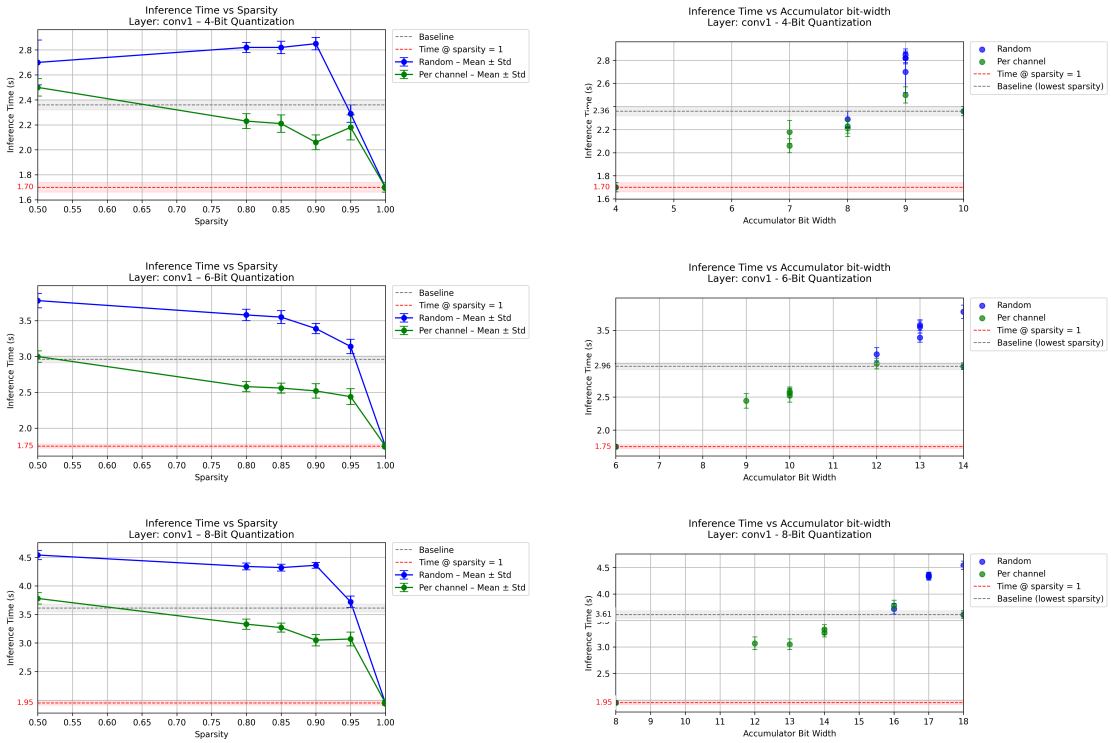


Figure 6.1: ResNet-18 `conv1`: inference time as a function of sparsity (left) and accumulator bit-width (right), evaluated at $n_{\text{bit}} = 4,6,8$.

6.1.3 Results for `layer3.1.conv1`

Table 6.2 summarizes the most relevant configurations obtained for the deeper convolutional layer `layer3.1.conv1`. For each quantization precision, we report the configurations providing the most favorable trade-off between accumulator bit-width and inference time.

Figure 6.2 illustrates the corresponding trends, highlighting how inference latency varies with sparsity and with the extracted accumulator requirement.

The results indicate that, for deeper convolutions, reductions in accumulator bit-width correlate more clearly with tangible improvements in inference time. In particular, reductions of several bits can yield speed-ups on the order of minutes, confirming the critical role of accumulator sizing in FHE-oriented execution.

Method	n_{bit}	Sparsity	Accumulator BW	Avg. time (s)	Std (s)	Speed-up (%)
BASELINE	4	0.27	11	186.47	18.85	/
Random	4	0.85	10	183.41	15.45	+1.64
Magnitude-based	4	0.95	8	160.85	18.33	+13.74
BASELINE	6	0.06	15	243.82	19.29	/
Random	6	0.80	14	213.60	16.28	+12.39
Magnitude-based	6	0.95	10	180.32	20.58	+26.05
BASELINE	8	0.02	19	276.74	25.38	/
Random	8	0.80	18	254.36	14.24	+8.09
Magnitude-based	8	0.95	13	214.15	23.09	+22.61

Table 6.2: ResNet-18 `layer3.1.conv1`: best observed configurations across sparsity and pruning schemes.

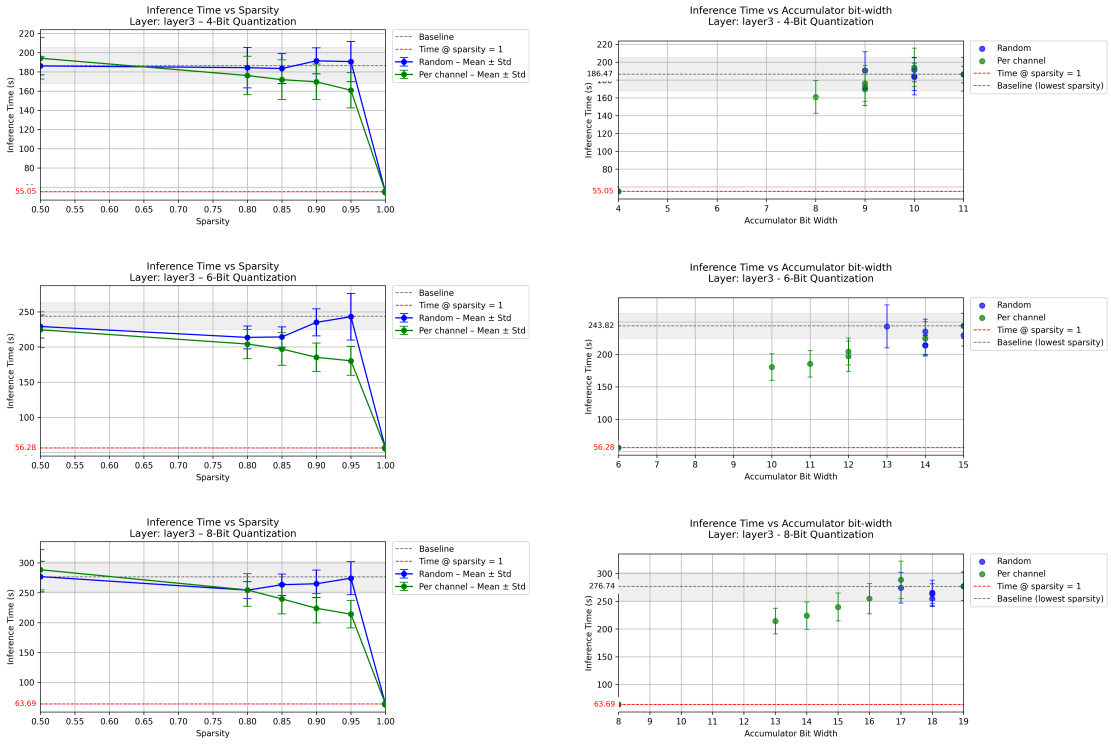


Figure 6.2: ResNet-18 `layer3.1.conv1`: inference time as a function of sparsity (left) and accumulator bit-width (right), evaluated at $n_{\text{bit}} = 4,6,8$.

6.1.4 Layer-wise conclusion

Across both analyzed layers, the results confirm that **sparsity alone is not sufficient** to guarantee performance gains. Meaningful improvements arise when pruning and quantization jointly reduce the dynamic range of partial sums, thereby

decreasing the accumulator bit-width required by the Concrete compiler.

6.2 End-to-end Results on ResNet-8

6.2.1 Accuracy-constrained evaluation

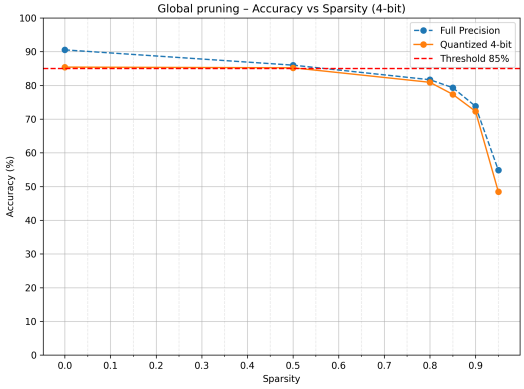
In order to restrict the analysis to configurations that remain meaningful in practice, we enforce an explicit accuracy constraint. Only models achieving a test accuracy above **85%** are considered valid and are carried forward to the accumulator bit-width investigation.

Figure 6.3 reports the test accuracy as a function of sparsity for different quantization precisions and pruning strategies, while Table 6.3 summarizes the subset of configurations that satisfy the imposed accuracy threshold.

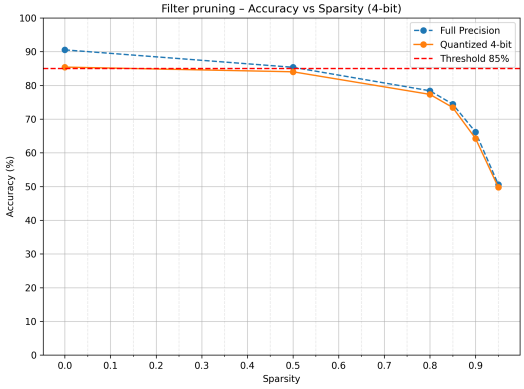
Table 6.3: Accuracy of valid ResNet-8 configurations under different quantization and pruning settings. Only models above the acceptance threshold are reported.

Model	**# Bits**	**Sparsity**	**Test Accuracy (%)**
Dense	4	0.00	85.42
Global	4	0.50	85.22
Dense	6	0.00	86.22
Filter	6	0.50	85.28
Global	6	0.50	85.54
Dense	8	0.00	86.50
Filter	8	0.50	85.72
Global	8	0.50	85.95

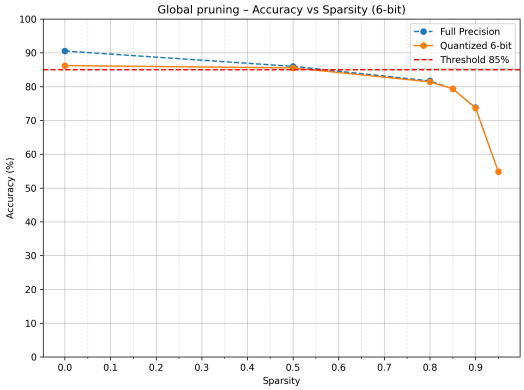
The results show that accuracy degrades rapidly beyond moderate sparsity levels, and that only a limited region of the explored design space remains feasible under this constraint. This motivates focusing the subsequent accumulator analysis exclusively on these accuracy-compliant configurations.



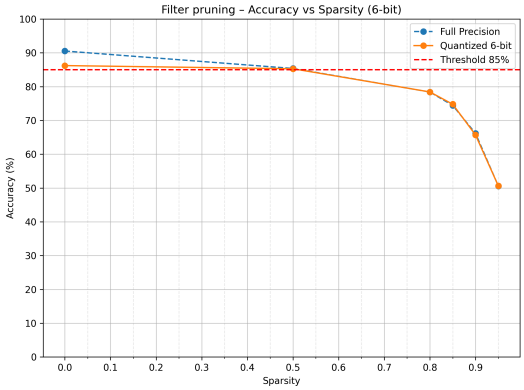
(a) Global pruning – 4 bit



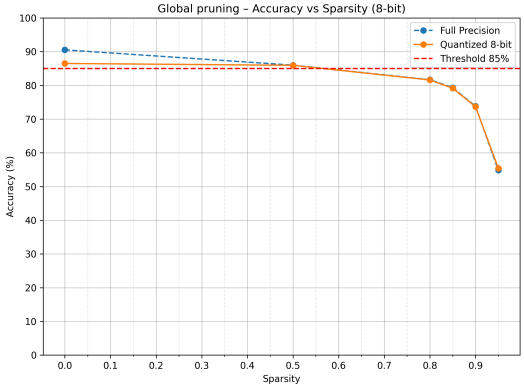
(b) Filter pruning – 4 bit



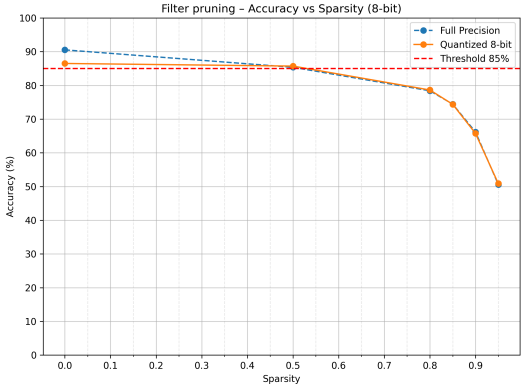
(c) Global pruning – 6 bit



(d) Filter pruning – 6 bit



(e) Global pruning – 8 bit



(f) Filter pruning – 8 bit

Figure 6.3: Test accuracy as a function of sparsity for **global pruning** (left column) and **filter pruning** (right column), evaluated at $n_{\text{bit}} = 4,6,8$. The dashed horizontal line denotes the **85% accuracy threshold**. Only configurations above this threshold are considered valid for the subsequent accumulator analysis.

6.2.2 Accumulator bit-width profiles

Having identified the configurations that satisfy the 85% accuracy threshold (Table 6.3), we now analyze their implications in terms of **accumulator bit-width**. For each valid model, the corresponding quantized network is compiled and, for every convolutional layer ℓ, the required accumulator width B_ℓ is extracted directly from the Concrete compiler.

In the following, we focus exclusively on these compiler-derived values, as they represent the effective numerical requirements for FHE execution and hardware sizing. Therefore, only one accumulator bit-width value per layer is reported in the tables below.

Throughout this section, we report only convolutional layers, since they dominate accumulator growth and represent the primary cost driver in FHE-oriented execution.

6.2.3 Bottleneck layers and lower-bound experiment

To expose a lower bound on accumulator requirements, we apply an aggressive sparsity setting (95%) and measure the layer-wise accumulator bit-width after pruning. In this diagnostic regime, accuracy is not considered, since the goal is to study the numerical effect of extreme sparsification on the accumulator range.

Unlike the accuracy-constrained section, here we report **both pruning strategies**: **global** and **per-filter** pruning, each evaluated at 95% sparsity. For each layer ℓ, we report the accumulator bit-width extracted from the Concrete compiler, and the reduction with respect to the dense model:

$$\Delta_\ell = B_\ell^{\text{Dense}} - B_\ell^{\text{Pruned}(0.95)}.$$

Key observation (bottlenecks). The reduction is not uniform across layers: some layers decrease by multiple bits, while others remain comparatively stable and therefore determine the network-wide upper bound. This confirms that global metrics can be misleading, and that a layer-wise analysis is required to identify persistent bottlenecks.

Table 6.4: Bottleneck analysis at 95% sparsity: 4-bit **Global** pruning (Concrete compiler).

Layer	Dense	Pruned (Global 0.95)	Delta
`stem.0`	11	10	1
`first_stack.block.0`	11	10	1
`first_stack.block.3`	11	10	1
`second_stack.block.0`	11	10	1
`second_stack.block.3`	10	10	0
`second_stack.residual`	10	10	0
`third_stack.block.0`	11	9	2
`third_stack.block.3`	11	9	2
`third_stack.residual`	11	9	2
`fc`	9	9	0

Table 6.5: Bottleneck analysis at 95% sparsity: 4-bit **Filter** pruning (Concrete compiler).

Layer	Dense	Pruned (Filter 0.95)	Delta
`stem.0`	11	10	1
`first_stack.block.0`	11	9	2
`first_stack.block.3`	11	9	2
`second_stack.block.0`	11	9	2
`second_stack.block.3`	10	9	1
`second_stack.residual`	10	9	1
`third_stack.block.0`	11	10	1
`third_stack.block.3`	11	10	1
`third_stack.residual`	11	10	1
`fc`	9	9	0

Table 6.6: Bottleneck analysis at 95% sparsity: 6-bit **Global** pruning (Concrete compiler).

Layer	Dense	Pruned (Global 0.95)	Delta
`stem.0`	14	13	1
`first_stack.block.0`	14	14	0
`first_stack.block.3`	13	13	0
`second_stack.block.0`	13	13	0
`second_stack.block.3`	14	13	1
`second_stack.residual`	14	13	1
`third_stack.block.0`	14	13	1
`third_stack.block.3`	14	13	1
`third_stack.residual`	14	12	2
`fc`	12	12	0

Table 6.7: Bottleneck analysis at 95% sparsity: 6-bit **Filter** pruning (Concrete compiler).

Layer	Dense	Pruned (Filter 0.95)	Delta
`stem.0`	14	13	1
`first_stack.block.0`	14	12	2
`first_stack.block.3`	13	13	0
`second_stack.block.0`	13	13	0
`second_stack.block.3`	14	12	2
`second_stack.residual`	14	12	2
`third_stack.block.0`	14	13	1
`third_stack.block.3`	14	13	1
`third_stack.residual`	14	12	2
`fc`	12	14	-2

Table 6.8: Bottleneck analysis at 95% sparsity: 8-bit **Global** pruning (Concrete compiler).

Layer	Dense	Pruned (Global 0.95)	Delta
`stem.0`	18	17	1
`first_stack.block.0`	18	18	0
`first_stack.block.3`	17	16	1
`second_stack.block.0`	16	16	0
`second_stack.block.3`	18	16	2
`second_stack.residual`	18	16	2
`third_stack.block.0`	18	16	2
`third_stack.block.3`	18	16	2
`third_stack.residual`	18	16	2
`fc`	17	16	1

Table 6.9: Bottleneck analysis at 95% sparsity: 8-bit **Filter** pruning (Concrete compiler).

Layer	Dense	Pruned (Filter 0.95)	Delta
`stem.0`	18	16	2
`first_stack.block.0`	18	16	2
`first_stack.block.3`	17	16	1
`second_stack.block.0`	16	15	1
`second_stack.block.3`	18	17	1
`second_stack.residual`	18	17	1
`third_stack.block.0`	18	17	1
`third_stack.block.3`	18	17	1
`third_stack.residual`	18	17	1
`fc`	17	16	1

6.3 Summary of results

The experimental evidence supports three consolidated findings:

1. **Sparsity alone is not a reliable driver of FHE gains.** Random pruning does not consistently reduce accumulator requirements nor inference time.

2. **Accumulator reduction is layer-dependent.** Deeper convolutions exhibit clearer correlations between accumulator bit-width and execution cost, whereas shallow layers show limited speed-up.

3. **Accuracy constraints strongly restrict pruning regimes.** Under an 85% accuracy threshold, only moderate sparsity levels remain feasible, and bottleneck layers still dominate the network-wide accumulator upper bound.

These outcomes motivate the final discussion in Chapter 7, where we interpret the implications for practical FHE deployment and outline accumulator-aware optimization directions.

Chapter 7

Conclusions

FHE represents one of the strongest paradigms for privacy-preserving machine learning, as it enables neural network inference directly on encrypted data without exposing sensitive inputs. However, despite major cryptographic advances such as TFHE and programmable bootstrapping, the computational overhead of encrypted inference remains a fundamental obstacle to practical deployment.

A key limiting factor is the **accumulator bit-width** required in convolutional layers: intermediate sums determine the numerical range that must be supported by the encrypted circuit, strongly impacting ciphertext parameters, bootstrapping cost, and ultimately latency. This thesis investigated whether two widely adopted compression techniques, **quantization** and **weight pruning**, can effectively reduce accumulator requirements and enable meaningful speed-ups in FHE-oriented neural networks.

The experimental study presented in this work combined both controlled layer-wise analysis and full-network exploration.

Introducing sparsity does not consistently translate into reduced accumulator bit-width or improved execution cost. While some reductions can be observed under specific pruning strategies, the relationship between sparsity and latency is weak when sparsity does not directly reduce the dynamic range of partial sums.

In particular, inference cost is often dominated by architectural factors and quantization precision rather than by the fraction of weights set to zero. Therefore, pruning cannot be considered a reliable driver of FHE acceleration when applied in a sparsity-only fashion.

The full-network ResNet-8 exploration under accuracy constraint highlighted that accumulator reduction is not uniform across layers. Even when many convolutions benefit from quantization or pruning, a subset of critical layers remains stable and continues to impose the maximum accumulator bit-width required by the network.

As a consequence, the overall encrypted inference cost is dictated by these **bottleneck layers**, meaning that improvements in non-critical layers may not

yield proportional end-to-end speed-ups.

Furthermore, under realistic quality constraints, only moderate sparsity levels remain feasible, and these do not decisively reduce the worst-case accumulator bound.

In conclusion, this thesis advanced the understanding of the relationship between quantization, pruning, and accumulator growth in convolutional neural networks compiled for TFHE-based inference. While quantization provides consistent reductions, pruning alone does not guarantee meaningful accumulator improvements under accuracy constraints. The persistence of bottleneck layers indicates that future progress in practical FHE deployment will require accumulator-centric optimization strategies rather than standard sparsity-based compression.

Bibliography

[1] National Institute of Standards and Technology. *Cryptography — NIST Computer Security Resource Center*. Accessed: 2025-01-26. 2023. URL: https://csrc.nist.gov/glossary/term/cryptography (cit. on p. 5).

[2] Hedinn. *Simple symmetric encryption*. https://commons.wikimedia.org/wiki/File:Simple_symmetric_encryption.png. Licensed under CC BY-SA 3.0. 2012 (cit. on p. 6).

[3] MarcTOK and JGraph. *Asymmetric encryption scheme*. https://commons.wikimedia.org/wiki/File:Asymmetric_encryption_scheme.png. Licensed under CC BY 4.0. 2024 (cit. on p. 6).

[4] Craig Gentry. «A fully homomorphic encryption scheme». PhD thesis. Stanford University, 2009 (cit. on p. 7).

[5] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. «(Leveled) Fully Homomorphic Encryption without Bootstrapping». In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ACM. 2012, pp. 309–325 (cit. on p. 7).

[6] Zvika Brakerski and Vinod Vaikuntanathan. «Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages». In: *Annual Cryptology Conference*. Springer. 2011, pp. 505–524 (cit. on p. 7).

[7] Rajeev Agrawal and Abhishek Joshi. «The CKKS FHE Scheme». In: *On Architecting Fully Homomorphic Encryption-based Computing Systems*. Synthesis Lectures on Computer Architecture. Cham: Springer, 2023. DOI: 10.1007/978-3-031-31754-5_2. URL: https://doi.org/10.1007/978-3-031-31754-5_2 (cit. on p. 7).

[8] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. «TFHE: Fast Fully Homomorphic Encryption Over the Torus». In: *Journal of Cryptology* 33 (2020), pp. 34–91. DOI: 10.1007/s00145-019-09319-x. URL: https://doi.org/10.1007/s00145-019-09319-x (cit. on p. 7).

[9] Oded Regev. «On lattices, learning with errors, random linear codes, and cryptography». In: *Journal of the ACM (JACM)* 56.6 (2009), pp. 1–40 (cit. on p. 7).

[10] Marc Joye and Michael Walter. «Liberating TFHE: Programmable Bootstrapping with General Quotient Polynomials». In: *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC'22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 1–11. ISBN: 9781450398770. DOI: 10.1145/3560827.3563376. URL: https://doi.org/10.1145/3560827.3563376 (cit. on p. 8).

[11] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. «Deep learning». In: *Nature* 521.7553 (2015), pp. 436–444 (cit. on p. 8).

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. «ImageNet classification with deep convolutional neural networks». In: *Advances in neural information processing systems*. 2012, pp. 1097–1105 (cit. on p. 8).

[13] Adam Paszke et al. «PyTorch: An imperative style, high-performance deep learning library». In: *Advances in neural information processing systems*. 2019, pp. 8024–8035 (cit. on p. 8).

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep Residual Learning for Image Recognition». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. URL: https://ieeexplore.ieee.org/document/7780459 (cit. on p. 9).

[15] 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*. 2016, pp. 201–210 (cit. on p. 10).

[16] Wikimedia Commons contributors. *Convolutional Neural Network with Color Image Filter*. https://commons.wikimedia.org/wiki/File:Convolutional_Neural_Network_with_Color_Image_Filter.gif. Licensed under CC BY-SA 4.0. Accessed: 2026-02-07. 2016 (cit. on p. 11).

[17] Zama. *Using GPU Execution in Concrete-ML*. https://docs.zama.ai/concrete-ml/guides/using_gpu/. 2024 (cit. on p. 12).

[18] Florian Bourse, Marco Minelli, Michael Minihold, and Pascal Paillier. «Programmable bootstrapping enables efficient homomorphic evaluation of deep neural networks». In: *EUROCRYPT*. 2021 (cit. on p. 12).

[19] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. «Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference». In: *CoRR* abs/1712.05877 (2017). arXiv: 1712.05877. URL: http://arxiv.org/abs/1712.05877 (cit. on pp. 13, 17, 18).

[20] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. «SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models». In: *Proceedings of the 2023 International Conference on Machine Learning (ICML)*. 2023 (cit. on p. 14).

[21] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. «Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference». In: *CoRR* abs/1712.05877 (2017). arXiv: 1712.05877. URL: http://arxiv.org/abs/1712.05877 (cit. on p. 14).

[22] Song Han, Huizi Mao, and William J Dally. «Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding». In: *arXiv preprint arXiv:1510.00149* (2015) (cit. on p. 15).

[23] Ayush Goyal. *Pruning Neural Networks*. Accessed: 2026-02-08. Jan. 2023. URL: https://medium.com/@goyal.ayush55/pruning-neural-networks-1f3a3be79c7d (cit. on p. 15).

[24] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. «TFHE: Fast Fully Homomorphic Encryption over the Torus». In: *Journal of Cryptology* 33.1 (2020), pp. 34–91 (cit. on p. 16).

[25] Ilaria Chillotti, Marc Joye, and Pascal Paillier. «Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks». In: *Proc. 5th Intl. Symp. on Cyber Security Cryptography and Machine Learning (CSCML)*. Vol. 12716. LNCS. Springer, 2021, pp. 1–19 (cit. on p. 16).

[26] 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: *Proc. 33rd Int. Conf. on Machine Learning (ICML)*. Vol. 48. Proceedings of Machine Learning Research. 2016, pp. 201–210 (cit. on p. 16).

[27] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. «CONCRETE: Concrete Operates oN Ciphertexts Rapidly by Extending TFHE». In: *8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC)*. Virtual Event: ACM, Dec. 2020 (cit. on p. 17).

[28] Zama. *Introducing the Concrete Framework*. https://www.zama.ai/post/introducing-the-concrete-framework. July 2022 (cit. on p. 17).

[29] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. «Training and Inference with Integers in Deep Neural Networks». In: *International Conference on Learning Representations (ICLR)*. 2018 (cit. on p. 18).

[30] Chi Zhang, Xu Yang, Shuangming Yu, Runjiang Dou, and Liyuan Liu. «ISQ: Intermediate-Value Slip Quantization for Accumulator-Aware Training». In: *IEEE Signal Processing Letters* (2025). DOI: 10.1109/LSP.2025.3539579 (cit. on p. 18).

[31] Ian Colbert, Alessandro Pappalardo, and Jakoba Petri-Koenig. «Accumulator-Aware Quantization with Guaranteed Overflow Avoidance». In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2023 (cit. on p. 18).

[32] Ian Colbert, Alessandro Pappalardo, Jakoba Petri-Koenig, and Yaman Umuroglu. «A2Q+: Improving Accumulator-Aware Weight Quantization». In: *arXiv preprint arXiv:2401.10432* (2024) (cit. on p. 18).

[33] Ian Colbert, Fabian Grob, Giuseppe Franco, Jinjie Zhang, and Rayan Saab. «Accumulator-Aware Post-Training Quantization». In: *arXiv preprint arXiv:2409.17092* (2024) (cit. on p. 19).

[34] Colby Banbury et al. «MLPerf Tiny Benchmark». In: *arXiv preprint arXiv:2106.07597* (2021) (cit. on p. 24).

[35] MLCommons. *MLPerf Tiny Benchmark: Reference Implementations and Rules*. https://github.com/mlcommons/tiny/tree/master/benchmark. Accessed: 2026-01-04 (cit. on p. 24).