Training TFHE-Based Neural Networks with Approximated Floating-Point Arithmetic

Emanuele Nicoletti
Politecnico di Milano
Milan, Italy
emanuele.nicoletti@polimi.it

Fabrizio Pittorino
Politecnico di Milano
Milan, Italy
fabrizio.pittorino@polimi.it

Alessandro Falcetta
Politecnico di Milano
Milan, Italy
alessandro.falcetta@polimi.it

Luca Colombo
Politecnico di Milano
Milan, Italy
luca2.colombo@polimi.it

Manuel Roveri
Politecnico di Milano
Milan, Italy
manuel.roveri@polimi.it

Abstract

Training neural networks under Torus Fully Homomorphic Encryption (TFHE) is severely constrained by the native restriction of the scheme to boolean and integer arithmetic, forcing prior work to rely on quantized integer pipelines limited to shallow MLPs. We present a framework that enables approximate floating-point training within TFHE by reinterpreting IEEE 754 representations as encrypted integers and operating on them with redesigned arithmetic. Our core contribution adapts L-mul, an approximate integer-based multiplication, to the encrypted setting, introducing numerical safeguards that prevent catastrophic wrap-around errors near zero and in subnormal ranges. We extend this approach to approximate division and implement exact addition and square root, yielding a sufficient arithmetic for backpropagation with SGD. For CPU-based FP32 encrypted operations, our approach achieves up to 2.1× faster multiplication and 29.3× faster division compared to state-of-the-art exact TFHE floating-point arithmetic, alongside up to 57× lower peak memory. Using these primitives, we perform, to our knowledge, the first fully encrypted training of a small-scale CNN under TFHE. To bypass the prohibitive overhead of training deep networks in this encrypted environment, we utilize plain-text emulation. We ensure strict output alignment by verifying that the network parameters generated during emulation are identical to those produced by the encrypted execution. Leveraging this equivalence, we use emulation to evaluate convergence on larger architectures (LeNet-5, VGG-style CNNs, and ResNet-20) across six benchmarks from MNIST variants to CIFAR-10 and medical imaging, matching exact-arithmetic baselines within 1% accuracy. Fully encrypted training of these deeper models remains beyond current hardware; we provide wall-clock projections quantifying this gap.

Keywords

fully homomorphic encryption, TFHE, encrypted training, neural network training, floating-point, L-mul

Proceedings on Privacy Enhancing Technologies 2026(4), 376–395

https://doi.org/10.56553/popets-2026-0126

1 Introduction

Fully Homomorphic Encryption under the TFHE scheme enables arbitrary computation on encrypted data, making it a natural candidate for privacy-preserving machine learning. This potential is further strengthened by the fact that TFHE is also Turing-complete; in principle, this provides the highest possible level of expressiveness, allowing the scheme to evaluate any computable function or complex algorithmic logic without being restricted by circuit depth or complexity. However, TFHE operates natively on booleans and integers only. This restriction creates a fundamental tension with deep learning practice, where training relies on floating-point arithmetic for gradient computation, weight updates, and normalization operations that involve multiplications and divisions pervasively throughout both forward and backward passes. Prior work on encrypted TFHE training has navigated this limitation through two strategies: quantized integer backpropagation [24], which suffers increasing accuracy degradation on complex tasks, and Direct Feedback Alignment [8], which avoids backpropagation entirely but cannot scale to convolutional architectures. Both approaches remain restricted to small MLPs. Meanwhile, exact floating-point arithmetic has recently been demonstrated in TFHE [4], but a single FP32 multiplication requires nearly 3 seconds and a division over 25 seconds, making training, which demands millions of such operations per epoch, prohibitively expensive. We present a framework that bridges this gap by enabling approximate floating-point arithmetic within TFHE. Our approach reinterprets IEEE 754 floating-point numbers [1–3] as fixed-size encrypted integers and defines arithmetic operations that follow floating-point logic while executing entirely within the integer domain. The core idea redesigns L-mul [21], an approximate multiplication that replaces the mantissa product with an integer addition and a constant offset. In more detail, we introduced L-mul for the encrypted training setting by designing dual numerical safeguards that prevent catastrophic wrap-around errors when operands are zero or results fall into subnormal ranges, failure modes that, left unaddressed, immediately destabilize training. We extend the same principle to construct an approximate division, and implement exact addition, subtraction, and for the first time in TFHE, square root, enabling components such as batch normalization. Since the entire framework operates through integer reinterpretation, it remains fully compliant with TFHE without requiring modifications to the underlying cryptographic primitives. Consequently, it is independent

of any specific TFHE implementation and can directly benefit from future advances in bootstrapping speed and hardware acceleration. We validate the framework in two stages. First, we perform fully encrypted training of both an MLP and a CNN on a small-scale benchmark, establishing the first fully encrypted training of a CNN of this scale under TFHE. Encrypted and emulated executions produce numerically identical network parameters, establishing that plain-text emulation faithfully reproduces encrypted behavior. Second, through this equivalence, we use emulation to demonstrate convergence on deeper neural networks with batch normalization and residual connections across five datasets including Fashion-MNIST, CIFAR-10, and medical imaging benchmarks. In addition, high computational overhead prevents fully encrypted training for deep networks, thus we provide wall-clock estimations for encrypted training of emulated experiments.

Our main contributions are as follows:

(1) Approximate floating-point arithmetic for TFHE. We redesign L-mul with numerical safeguards for encrypted training stability, introduce an approximate division based on the same principle, and implement exact square root, collectively yielding a sufficient arithmetic for backpropagation with SGD. Consequently, for CPU-based FP32 operations, our approach achieves up to 2.1× faster multiplication and 29.3× faster division compared to the state-of-the-art (SotA) exact TFHE floating-point arithmetic [4], with sub-linear bit-width scaling; peak memory is reduced by up to 57×.
(2) First fully encrypted small-scale CNN training in TFHE. We train a convolutional network entirely in the encrypted domain, and validate zero numerical divergence between encrypted and emulated execution.
(3) Scalability demonstration via validated emulation. We show that approximate floating-point backpropagation preserves convergence on architectures up to ResNet-20 on CIFAR-10, matching exact-arithmetic baselines within 1% accuracy.

To facilitate reproducibility, we will release the full codebase upon acceptance.

2 Preliminaries

2.1 Torus Fully Homomorphic Encryption

We briefly recall the aspects of TFHE relevant to our construction; for comprehensive treatment we refer to Chillotti et al. [6] and Clet et al. [7]. TFHE is a Fully Homomorphic Encryption scheme built on the hardness of the Learning With Errors (LWE) [29] and Ring-LWE [22] problems, adapted to operate over the real torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$. It encrypts plaintexts using three layered structures: TLWE, TRLWE, and TRGSW samples, that support homomorphic addition and multiplication on encrypted data. Refer to Appendix A for a definition list.

The key property distinguishing TFHE from leveled or somewhat homomorphic schemes is *bootstrapping*: a procedure that refreshes ciphertexts by reducing accumulated noise, enabling an unlimited number of sequential homomorphic operations. TFHE provides two bootstrapping variants: *gate bootstrapping*, which reduces noise after boolean gate evaluation, and *circuit bootstrapping*, which converts TLWE samples into TRGSW samples suitable for further homomorphic computation. Bootstrapping is the dominant cost in any TFHE computation and the primary driver of the overhead our framework seeks to reduce. Crucially for this work, TFHE natively supports only boolean and integer operands. In practice, we interface with TFHE through the FheUint API provided by the TFHE-rs library [34], which exposes unsigned integers of configurable bit-width as homomorphic data types with GPU-accelerated operations. All experiments use parameter sets guaranteeing a minimum security level of 128 bits. The absence of native floating-point support motivates the approximate arithmetic we develop in Section 4.

2.2 IEEE 754 Floating-Point Representation

Under the IEEE 754 standard [1–3], a floating-point number x is encoded as three fields: a sign bit x_s, a biased exponent x_e of n_e bits, and a mantissa x_m of n_m bits. The value is reconstructed as:

$$x = (-1)^{x_s} \times 1.x_m \times 2^{x_e - b},$$

where b is a format-dependent bias and the leading 1 before the mantissa is implicit. We denote the concatenation of x_e and x_m, that is the full bit-field excluding the sign, as x_d. This quantity plays a central role in our arithmetic, since adding two such fields $x_d + y_d$ simultaneously adds the exponents and mantissas as a single integer operation.

The standard also defines *subnormal* numbers to represent values near zero. Subnormals replace the implicit leading 1 with 0 and use the minimum exponent, allowing gradual underflow rather than abrupt truncation to zero. This distinction is critical in our setting: as we show in Section 4, the L-mul approximation interacts pathologically with subnormal values, producing catastrophic wrap-around errors that must be explicitly guarded against.

The proposed framework supports different floating-point configurations; however, in the final experiments in Section 5, we adopt the standard FP32 format ($n_e = 8, n_m = 23, b = 127$).

2.3 L-mul

Standard floating-point multiplication decomposes as:

$$x \cdot y = (-1)^{x_s \oplus y_s} \left(1 + x_m + y_m + x_m \cdot y_m\right) \cdot 2^{x_e + y_e - 2b}. \quad (1)$$

The mantissa product $x_m \cdot y_m$ is the most expensive term. L-mul [21] replaces it with a fixed offset $l = 2^{-4}$, yielding the approximation:

$$x \cdot y \approx (-1)^{x_s \oplus y_s} \left(1 + x_m + y_m + l\right) \cdot 2^{x_e + y_e - 2b}. \quad (2)$$

The key observation is that this entire computation reduces to integer arithmetic on the concatenated fields x_d and y_d. Since $x_d + y_d$ simultaneously adds the exponent and mantissa fields, the approximate product can be computed as shown in Algorithm 1: a single integer addition, a bias correction, an offset addition, a sign computation via XOR and final concatenation using a bitwise or.

This transforms a floating-point multiplication dominated by an $O(n_m^2)$ mantissa product, into an algorithm with complexity $O(n_e + n_m)$. The approximation error arises from replacing $x_m \cdot y_m$ with the constant l; the resulting maximum relative error is bounded at approximately 7.3% for normalized operands (see Section 4 for detailed analysis). Crucially for our application, the operation involves no multiplication at any level, making it directly implementable over TFHE's encrypted integers.

Algorithm 1 L-mul [21]

Require: x, y, l, b
1: $z = x_d + y_d$
2: $z = z - b$
3: $z = z + l$
4: $z_s = x_s \oplus y_s$ ▷ sign via XOR
5: $z = z_s \mid z$ ▷ prepend sign bit
6: **return** z

However, Algorithm 1 assumes both operands are normalized and non-zero. When either condition is violated, as frequently occurs during neural network training where weights and gradients cluster near zero, the bias subtraction in line 2 triggers integer wrap-around, producing catastrophic errors. We address these failure modes in Section 4.

3 Related Work

3.1 Encrypted Neural Network Training

Early work on privacy-preserving deep learning focused exclusively on encrypted inference. CryptoNets [9] demonstrated that a neural network trained in plaintext could perform predictions on encrypted inputs, and subsequent work [5] reduced the associated latency through architecture search targeting fewer nonlinear activations. Recent research indicates that encrypted inference represents a more mature and viable application compared to fully encrypted training. A notable example is DCT-CryptoNets [30], which achieves encrypted inference within hundreds of seconds, significantly reducing the required latency. However, this approach relies on models pre-trained on plaintext and subsequently utilizes integer-quantized versions exclusively for the inference phase. Despite employing ResNet architectures, DCT-CryptoNets circumvents the need for encrypted square root operations, as Batch Normalization statistics remain frozen during the encrypted execution stage. Encrypted training of simpler models followed: Park et al. [27] showed that logistic regression and SVMs could be trained under HE, though these models lack the expressiveness required for complex tasks.

For neural networks, encrypted training remains considerably more challenging. Existing approaches differ in the HE scheme employed, the degree of end-to-end encryption, and the supported architectures. ReBoot [28] performs end-to-end encrypted MLP training using bootstrapped CKKS, exploiting its SIMD capabilities. Glyph [20] proposes a hybrid TFHE-BGV/BFV pipeline that trains CNNs on medical imaging data. However, this approach necessitates scheme conversions between layers and relies on transfer learning from a plaintext-pretrained model. Furthermore, it compromises cryptographic robustness by operating at an 80-bit security level, falling well short of modern 128-bit standards. Following a similar approach to Glyph, HE-SecureNet [31] employs a hybrid scheme-switching framework combining CKKS for linear layers and FHEW for non-linear activations. While it currently stands as one of the fastest fully encrypted training frameworks, it achieves this performance by operating at a reduced security level of 119 bits. Full details on the comparison are shown in Table 4. The two works most directly comparable to ours both operate within pure TFHE. Montero et al. [24] uses offline quantization calibration to perform integer-based encrypted training, exploiting low bit-widths to reduce bootstrapping cost, but they do not extend their work to convolutional architectures. [8] trains MLPs using 22-bit integers and Direct Feedback Alignment (DFA) [18] instead of backpropagation, avoiding the need for symmetric gradient flow. While effective on MNIST [15] and Fashion-MNIST [37], DFA is structurally unsuitable for CNNs [33].

Both approaches are constrained by integer-only arithmetic, which limits their extensibility: quantized backpropagation requires extensive epochs to match floating-point accuracy on non-trivial tasks [35], and DFA cannot propagate gradients through convolutional or residual structures effectively.

3.2 Floating-Point Arithmetic in TFHE

TFHE does not natively support floating-point representations, and only three prior approaches have addressed this gap. Lee and Shin [17] implement floating-point operations using boolean gates exclusively, resulting in prohibitive time complexity. Moon and Lee [25] use three RLWE ciphertexts to represent sign, exponent, and mantissa respectively, but again suffer from high latency. The current SotA is Bergerat et al. [4], who represent each floating-point component with separate LWE ciphertexts and leverage circuit bootstrapping to reduce operational cost. Their implementation provides exact arithmetic with a failure probability $\leq 2^{-40}$ and scales efficiently with bit-width, representing the fastest available TFHE floating-point operations. However, even under their approach, a single FP32 multiplication requires approximately 2.9 seconds and a division over 25 seconds (Table 3). This highlights that training in this environment is exceptionally difficult, as current performance limits still preclude the practical use of deeper network structures.

No prior work has applied floating-point arithmetic of any kind to neural network training within TFHE.

3.3 Approximate Multiplication

A separate line of research seeks to reduce the cost of multiplication in neural networks through approximation. Mogami [23] proposes multiplication-free training using additive operations, while Kosson and Jaggi [13] develop piecewise affine approximations that eliminate multiplications from transformer training entirely. Most relevant to our work, Luo and Sun [21] introduce L-mul, which approximates the floating-point mantissa product $x_m \cdot y_m$ with a constant offset, reducing each multiplication to integer additions (Section 2.3). Their evaluation focuses on LLM inference in low-bitwidth formats, where the approximation error is negligible in attention mechanisms.

However, L-mul was evaluated on attention mechanisms using a limited bit-width (e.g., FP8). It does not handle the edge cases (zero-valued operands, subnormal products) that arise pervasively during training and that become catastrophic failure modes when executed on encrypted integers (Section 4.4). Moreover, no approximate division counterpart has been proposed. Our work adapts and extends L-mul to the encrypted training setting, addressing both limitations.

4 Methodology

4.1 Threat Model

We consider a standard cloud-based training-as-a-service setting with two parties: a *client* who holds the data and the TFHE secret key, and a *cloud provider* who performs homomorphic computation using only the public evaluation key. The provider is assumed semi-honest (honest-but-curious): it follows the training protocol correctly but may attempt to infer information about the plaintext data or model outputs by inspecting ciphertexts and intermediate results.

4.1.1 Leakage profile. To provide a formal treatment of security, we define the explicit leakage profile, denoted as $\mathcal{L}$. During the execution of the protocol, the semi-honest provider only learns $\mathcal{L}$, which consists of:

- The network architecture, including layer types and dimensionalities.
- The total number of training steps and epochs.

This leakage profile is standard among homomorphic encryption-based machine learning frameworks (e.g. HE-Securenet [31]) and is necessary for the server to allocate the correct sequence of homomorphic operations.

4.1.2 Security Guarantees. We can frame our security guarantees within the standard real/ideal simulation paradigm.

- **Ideal World:** An ideal functionality receives the plaintext and labels from the client, performs the training based on the architecture specified in $\mathcal{L}$, and returns the final encrypted weights to the client, leaking only $\mathcal{L}$.
- **Real World:** The client encrypts the data, initial weights and activations using TFHE with 128-bit security parameters and sends it to the provider, which performs the homomorphic computations and returns the encrypted model.

4.1.3 IND-CPA property. Our protocol securely realizes the ideal functionality against a semi-honest adversary. The security reduces directly to the IND-CPA (indistinguishability under chosen-plaintext attack) security of the underlying TFHE scheme adopting 128-bit of security level. Under this guarantee, all data, inputs, weights, activations, gradients, and loss values remain computationally indistinguishable from random noise, ensuring the server learns nothing beyond $\mathcal{L}$.

4.1.4 Data-independent execution. Crucially, our redesigned approximate arithmetic does not introduce new attack vectors or side-channel leakage. The approximation and numerical safeguards (e.g., flushing to zero) are implemented entirely through data-independent control flow. Specifically, the arithmetic is implemented entirely through integer operations on standard TFHE ciphertexts (FheUint types), using the same bootstrapping procedures and parameter sets as integer arithmetic. Furthermore, safeguard conditions are evaluated via encrypted multiplexing: both branches of a condition are computed unconditionally, and the selection is performed homomorphically. Consequently, no branching, timing variations, or memory access patterns depend on the underlying plaintext values.

4.2 Floating-Point Emulation over Encrypted Integers

TFHE operates natively on encrypted integers. To perform floating-point arithmetic, we adopt a software emulation strategy analogous to how microcontrollers without a hardware floating-point unit handle real-valued computation through libraries such as SoftFloat [10]: the bit-pattern of an IEEE 754 number is stored and manipulated as an integer, with arithmetic logic implemented algorithmically.

Concretely, given an FP32 number x, we treat its 32-bit representation as an unsigned integer and encrypt it as a single TFHE ciphertext (an `FheUint32` in the TFHE-rs library). The sign bit x_s, exponent field x_e, mantissa field x_m, and their concatenation x_d are accessed and manipulated through encrypted bitwise and arithmetic operations (shifts, masks, additions, and comparisons), all of which are natively supported by TFHE on integer ciphertexts.

The arithmetic operations we defined (multiplication, division, addition, subtraction, and square root) are deterministic functions from k encrypted integers to j encrypted integers that implement the corresponding floating-point logic:

$$f : \texttt{FheUint}^k \rightarrow \texttt{FheUint}^j.$$

This design has two practical consequences. First, it is *scheme-agnostic at the primitive level*: our operations compose standard TFHE integer primitives and will automatically benefit from any future improvements in bootstrapping speed or hardware acceleration, without requiring reimplementation. Second, it is *self-contained*: unlike approaches that design custom low-level cryptographic routines for each floating-point operation [4], our framework can be implemented by any practitioner familiar with the high-level TFHE-rs API, lowering the barrier to adoption and extension.

4.3 Arithmetic Operations

4.3.1 Multiplication. Computing the mantissa product $x_m \cdot y_m$ in Equation 1 is the dominant cost in encrypted floating-point multiplication, and multiplications are pervasive throughout both forward and backward passes of neural network training. We address this by extending L-mul (Section 2.3) to the encrypted training setting.

The original L-mul algorithm assumes normalized, non-zero operands. This assumption is violated routinely during training, where weights, gradients, and activations frequently take values at or near zero. Two failure modes arise:

(1) **Zero-valued operands.** If either $x = 0$ or $y = 0$, their exponent fields are zero. The addition $x_d + y_d$ produces a small value from which the bias b is subtracted (line 2 of Algorithm 1), causing unsigned integer underflow. The result wraps around to a large exponent, yielding an output near the maximum representable value instead of zero.
(2) **Subnormal products.** When two small but non-zero operands are multiplied, the sum of their biased exponents may satisfy $x_e + y_e < b$. The bias subtraction again triggers wrap-around, mapping what should be a near-zero result to an astronomically large value.

Both failure modes produce errors that immediately destabilize training through explosive gradient growth. To address them, we

introduce a dual safeguard condition that detects both cases and forces the output to zero. Algorithm 2 shows the redesigned method.

Algorithm 2 Redesigned L-mul with numerical safeguards

Require: x, y, l, b
1: $z = x_d + y_d$
2: $z = z - b$
3: $z = z + l$
4: $z_s = x_s \oplus y_s$ ▷ sign via XOR
5: $z = z_s \mid z$ ▷ prepend sign bit
6: $c \leftarrow (x_e + y_e < b) \vee (x = 0) \vee (y = 0)$ ▷ safeguard condition
7: **return** $c \; ? \; 0 : z$ ▷ encrypted multiplexing

Implementation in TFHE. Since TFHE cannot branch on encrypted values, the conditional in line 7 is implemented via encrypted multiplexing: both the zero output and the computed z are evaluated unconditionally, and the safeguard condition c (itself an encrypted boolean) selects the correct result homomorphically. This ensures that the execution path is data-independent and reveals no information to the server.

Approximation error. Following what introduced in [13], we quantify accuracy using the absolute relative error:

$$E(x, y) = \frac{|(x \cdot_{\spadesuit} y) - (x \cdot y)|}{|x \cdot y|} \tag{3}$$

where $\cdot_{\spadesuit}$ denotes the approximate multiplication. Figure 1 shows the error distribution for operands in $[1, 2]$. The error pattern is self-similar across powers of two: the heatmap in any range $[2^k, 2^{k+1}]$ (with $k \in \mathbb{Z}$) is a rescaled copy of Figure 1 with identical error values. Consequently, the maximum relative error of 7.28% holds regardless of operand magnitude. For an in-depth analysis, please refer to Appendix B.

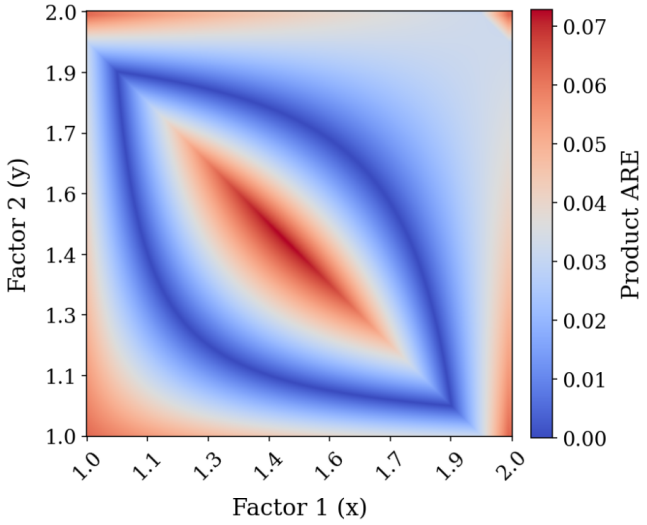

Figure 1: Heatmap of the absolute relative error between approximate and exact multiplication for operands in $[1, 2]$. The pattern is self-similar across all $[2^k, 2^{k+1}]$ intervals.

4.3.2 Division. Exact encrypted division is substantially more expensive than multiplication: a single FP32 division takes approximately 25.6 seconds under the SotA [4], roughly 9× slower than multiplication (Table 3). Although divisions are less frequent than multiplications during training, they arise in loss computation, normalization layers, and optimizer updates, making their cost non-negligible.

The original L-mul paper does not address the division. We derive an approximate division by inverting the L-mul logic: subtracting the concatenated fields rather than adding them, adding the bias rather than subtracting it, and subtracting the offset rather than adding it. This mirrors the approach suggested by [13] for plaintext piecewise-affine operations, adapted here to the encrypted integer setting. As with multiplication, unguarded execution produces catastrophic errors. Two safeguards are required:

(1) **Exponent overflow.** When dividing a large number by a small one, the result exponent $x_e - y_e + b$ may exceed the maximum representable exponent m, causing overflow. The safeguard detects this and forces the output to zero.
(2) **Zero dividend.** If $x = 0$, the subtraction $x_d - y_d$ produces a meaningless result. We explicitly check for this case. Note that a zero divisor ($y = 0$) need not be handled, as deep learning routines conventionally add a small constant ϵ to denominators.

Algorithm 3 shows the complete procedure, again using encrypted multiplexing for the conditional.

The approximation error follows the same self-similar pattern as multiplication. The maximum relative error for division is 7.86%, as shown in Figure 2.

4.3.3 Addition and Subtraction. Unlike multiplication and division, addition and subtraction are implemented exactly. The primary

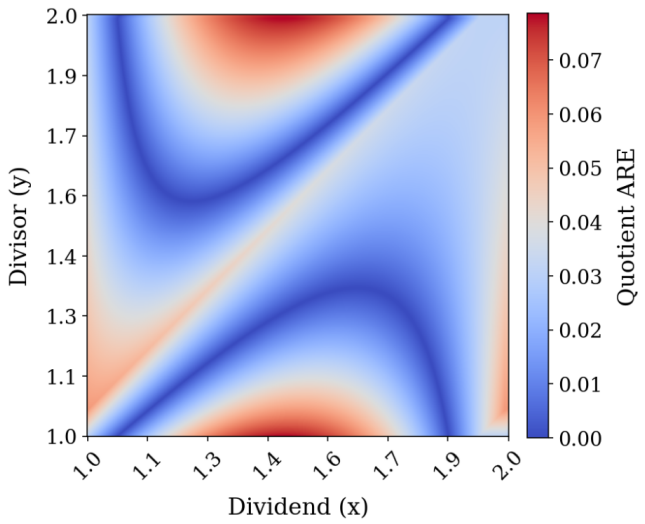

Figure 2: Heatmap of the absolute relative error between approximate and exact division for operands in $[1, 2]$. As with multiplication, the pattern is self-similar across all $[2^k, 2^{k+1}]$ intervals.

Algorithm 3 Approximate division with numerical safeguards

Require: x, y, l, b, m
1: $z = x_d - y_d$
2: $z = z + b$
3: $z = z - l$
4: $z_s = x_s \oplus y_s$ ▷ sign via XOR
5: $z = z_s \mid z$ ▷ prepend sign bit
6: $c \leftarrow (x_e - y_e + b > m) \vee (x = 0)$ ▷ safeguard condition
7: **return** $c \ ? \ 0 : z$ ▷ encrypted multiplexing

challenge is exponent alignment: to add two floating-point numbers with different exponents, the mantissa of the smaller operand must be right-shifted by the exponent difference before the mantissas can be summed. In the encrypted setting, this requires an encrypted comparison to determine the shift direction and amount, followed by an encrypted variable-length shift, both of which involve multiple bootstrapping operations.

We chose exact implementation over approximation for two reasons. First, addition accumulates values across neurons and batch elements; any approximation error in addition would compound across layers in ways that multiplication error does not, since the latter is bounded per-operation by the self-similar property described above. Second, the cost of exact encrypted addition (1.38s on GPU for FP32, Table 3) is already moderate relative to approximate multiplication (0.33s), making the accuracy speed trade-off less favorable for approximation.

Subtraction requires no separate implementation: it is performed by homomorphically flipping the sign bit of the second operand and invoking the addition routine.

4.3.4 Square Root. We implement, to the best of our knowledge, for the first time in TFHE, an exact encrypted square root. This operation is essential for batch normalization, which divides by the standard deviation of activations and thus requires computing $\sqrt{\sigma^2 + \epsilon}$. Without square root support, batch normalization, and by extension deep architectures such as ResNet [11] that depend on it, cannot be implemented.

We adopt the digit-by-digit restoring shift-subtract algorithm [26], which computes the result one bit at a time through shifts, subtractions, and comparisons. This method is particularly well-suited to TFHE for one reason. Since each iteration uses only operations that are inexpensive in TFHE (shifts, additions, comparisons), it avoids the repeated floating-point divisions that would be required by iterative methods such as Newton-Raphson. For FP32, Newton-Raphson would typically require 4–5 iterations, each involving division costing over 25 seconds encrypted; the digit-by-digit method avoids this entirely.

The algorithm works by maintaining a partial remainder and a trial root. In each cycle, the current remainder is shifted left, and a candidate bit is tested by subtracting the trial root (shifted) from the remainder. If the subtraction result is positive, the next bit of the root is set to 1, and the remainder is updated. If the result is negative, the bit is set to 0, and the previous remainder is restored. In our TFHE context, this decision is implemented as a homomorphic multiplexer, which allows the algorithm to proceed evaluating both branches without revealing information.

To ensure maximum numerical fidelity, we maintain a total of n_m iterations, matching the full bit-width of the mantissa. While some implementations use only half the bit-width to generate a rough root (exploiting the fact that $\sqrt{x}$ carries half the precision of x^2), we intentionally compute the full significand. A more detailed explanation is provided in the Appendix C.

4.4 Empirical Validation of Numerical Safeguards

Section 4.3.1 introduced dual safeguards to prevent wrap-around errors in our approximate multiplication. Here we demonstrate their necessity through concrete failure cases and an ablation study.

Failure cases under unguarded L-mul. Table 1 shows three pathological cases where standard L-mul produces catastrophic errors in FP32. Cases A and B arise from zero-valued operands: the bias subtraction on near-zero exponent sums causes unsigned integer underflow, wrapping the result to the maximum exponent range. Case C is more insidious, since neither operand is zero, but their exponent sum falls below the bias, triggering the same wrap-around and inflating a near-zero product to $\sim 10^{34}$.

Table 1: Failure cases of unguarded L-mul and their resolution under our dual safeguard. "Pure L-mul" denotes Algorithm 1 without safeguards; "Ours" denotes Algorithm 2.

Case	Operation	Pure L-mul	Ours
A	0×0	4.25	0
B	0×0.8	NaN	0
B	0×1.0	7.35×10^{-40}	0
C	$10^{-38} \times 5 \times 10^{-5}$	6.45×10^{34}	0

Insufficiency of a single safeguard. One might consider checking only for zero-valued operands, omitting the exponent-sum condition. This leaves Case C entirely unresolved: since neither 10^{-38} nor 5×10^{-5} is exactly zero, the zero-check passes and the wrap-around executes unchecked. The dual condition (checking both for zero operands *and* for $x_e + y_e < b$) is therefore essential. We note that this design intentionally sacrifices subnormal precision (all near-subnormal products are flushed to zero) in favor of global numerical stability, a trade-off that is well-suited to neural network training where values in the subnormal range carry negligible gradient signal.

Impact on training. Table 6 quantifies the effect on end-to-end training. Without safeguards, pure L-mul produces immediate numerical divergence: weights immediately explode to $\sim 10^{37}$, and thus accuracy collapses. With our dual safeguard, layer-wise weight statistics (μ, σ^2) remain closely aligned with exact arithmetic throughout training, and final accuracy is preserved (see Section 5 for detailed results).

4.5 Supported Neural Network Layers

A key advantage of floating-point emulation over integer-only arithmetic is that standard neural network layers can be implemented with their original plaintext logic: no quantization-aware redesign,

scaling factor management, or accumulator overflow handling is required. Table 2 summarizes the components supported by our framework, along with the arithmetic operations each requires.

Convolutional layers, in particular, require no adaptation beyond what is needed for fully connected layers: both reduce to multiply-accumulate (MAC) operations over encrypted floating-point values, with the convolution simply applying MACs in a sliding-window pattern. This stands in contrast to integer-based approaches, where convolutions require careful management of quantization scales across channels and accumulator bit-widths [35].

Batch normalization is the most arithmetically demanding component, requiring all four operations including square root for the standard deviation computation. Residual connections, by contrast, are trivial; they consist solely of element-wise exact addition. By combining these components, we can construct architectures ranging from LeNet-5 to ResNet-20, as demonstrated in Section 5.

4.6 Training Procedure

Our framework implements standard backpropagation with floating-point gradient computation, matching the training procedure used in plaintext deep learning. This contrasts with prior TFHE training approaches that rely on integer backpropagation [24] or alternative algorithms such as DFA [8], both of which face fundamental extensibility limitations for convolutional and residual architectures (see Section 3).

4.6.1 Loss Function. We employ Mean Squared Error (MSE) as the loss function. While cross-entropy is standard for classification, it requires exponential and logarithm operations that are not yet available in our framework. MSE requires only subtraction and multiplication, both of which are supported:

$$L_{\spadesuit} = \frac{1}{B_s} \sum_{i=1}^{B_s} (f(x_i) - y_i) \cdot_{\spadesuit} (f(x_i) - y_i) \tag{4}$$

where $f(x_i)$ is the network output, y_i is the target, B_s is the batch size, and $\cdot_{\spadesuit}$ denotes our approximate multiplication. The gradient with respect to the output is:

$$\frac{\partial L_{\spadesuit}}{\partial f(x_i)} = \frac{2}{B_s}(f(x_i) - y_i) \tag{5}$$

which involves only subtraction and a division by the constant B_s, both computed homomorphically. As an additional note, while exact arithmetic allows division to be substituted with multiplication by the inverse, this property does not hold in our implementation. Because our multiplication and division operations are not strict mathematical inverses, replacing division with multiplication would introduce an additional source of approximation error. Extending the framework to cross-entropy loss, the standard and more suitable metric for classification tasks compared to MSE, would require implementing encrypted exponential and logarithm operations, which we identify as a direction for future work.

4.6.2 Optimizer. We implement SGD with momentum and weight decay. The encrypted parameter update is:

$$v_{t+1} = \gamma \cdot_{\spadesuit} v_t + \eta \cdot_{\spadesuit} (\nabla L_{\spadesuit}(\theta_t) + \lambda \cdot_{\spadesuit} \theta_t) \tag{6}$$

$$\theta_{t+1} = \theta_t - v_{t+1} \tag{7}$$

where θ_t denotes the encrypted parameters at step t, v is the momentum buffer, η the learning rate, γ the momentum coefficient, and λ the weight decay coefficient. All multiplications are approximated except for the momentum term, which warrants selective analysis based on the specific task. The rationale behind this selective use of exact arithmetic, along with its impact on time, is elaborated in the subsequent sections.

Selective precision for momentum. In deeper architectures (Experiments 3–4 in Section 5), we observe that momentum-driven updates, both in the SGD optimizer and in the moving average/variance of batch normalization, are sensitive to multiplicative error accumulation. The momentum coefficient γ is applied recursively at every step, causing approximate multiplication errors to compound over time in the velocity buffer. For these components, we use exact multiplication. This highlights a limitation in our current method that we intend to investigate further as future work. Crucially, these exact floating-point encrypted multiplications constitute a negligible fraction of total operations (0.003% of all multiplications per batch in ResNet-20), so the overall performance advantage is preserved. We discuss this selective precision strategy in detail in Section 5.

Extensibility to adaptive optimizers. The availability of square root within our framework makes it straightforward to implement Adam [12] or AdamW [19], which require $\sqrt{\hat{v}_t}+\epsilon$ in the denominator of the update rule. These optimizers were previously infeasible in TFHE due to the lack of square root support.

5 Experiments

5.1 Experimental Setup

All experiments were conducted on a workstation with an Intel Xeon Gold 5318S CPU (24 cores, 48 threads), 384 GB RAM, and an NVIDIA A40 GPU, running Ubuntu 24.04 LTS. The framework is implemented in Rust, using the TFHE-rs v1.3 library [34] with its high-level `FheUint` API and native GPU acceleration. All cryptographic parameter sets guarantee a minimum security level of 128 bits.

Our framework supports two execution modes: *fully encrypted*, where all data and parameters are TFHE ciphertexts, and *plain emulation*, which executes identical bit-level arithmetic logic on unencrypted integers. As validated in Experiment 1 (Section 5.4.1), the two modes produce numerically identical results, establishing emulation as a faithful proxy for encrypted execution. We use fully encrypted training for small-scale experiments and emulation for architectures where encrypted training is currently infeasible (see Section 5.5 for wall-clock projections).

5.2 Operation-Level Benchmarks

Two related works, Lee and Shin [17] and Moon and Lee [25], do not provide publicly available or reproducible code; therefore, we report only the execution times stated in the respective papers and do not include them in our direct comparison, as both are superseded by [4]. Lee and Shin conducted their experiments on an Intel Xeon Silver 4210, reporting execution times of 181s and 169s for 32-bit addition and multiplication, respectively, and 256s and 293s for the corresponding 64-bit operations. Moon and Lee performed their evaluation on an Intel Core i7-6700. For 32-bit operands, they

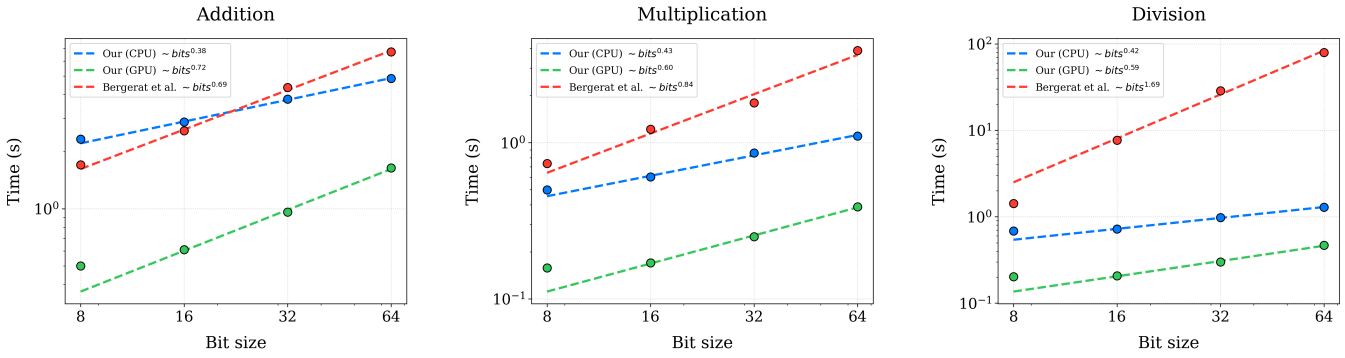

Figure 3: Operation scaling with respect to bit-width, comparing our implementation (CPU and GPU) with Bergerat et al. [4] on the same hardware. Fitting is performed on the last three data points (excluding FP8). The notation *bits^x* indicates how runtime scales with the bit-width.

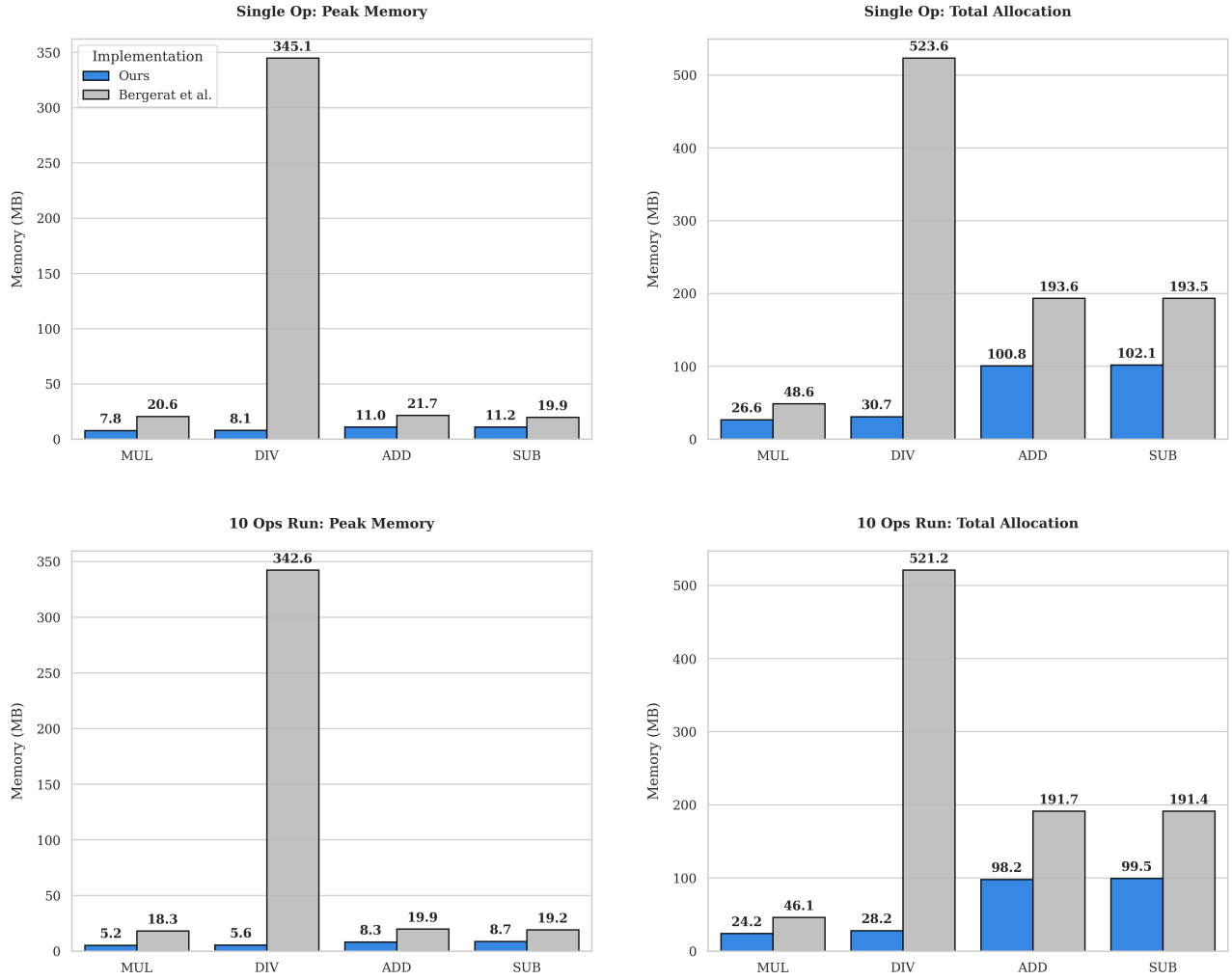

Figure 4: Memory requirements per FP32 operation: comparison between our implementation and Bergerat et al. [4]. Values are averaged across multiple single-operation and 10-operation sequences for statistical consistency.

Table 2: Neural network components supported by our framework. *Approx* denotes operations using our approximate arithmetic; *Exact* denotes operations using exact encrypted arithmetic.

Component	Required operations	Enabled by
Fully Connected	mul, add	Approx mul + Exact add
Convolution	mul, add	Approx mul + Exact add
ReLU	comparison	Encrypted compare
Tanh	mul, add	PLA approximation
Max Pooling	comparison	Encrypted compare
Average Pooling	add, div	Exact add + Approx div
Global Average Pooling	add, div	Exact add + Approx div
Batch Normalization	mul, add, div, sqrt	All four operations
Residual (skip) connection	add	Exact add

Table 3: Execution time (seconds) for a single encrypted floating-point operation. Bold indicates the fastest time for each operation and bit-width. Ref denotes Bergerat et al. [4]. The last row shows the Multiply-Accumulate (MAC) operation.

	8-bit			16-bit			32-bit			64-bit		
	GPU	**CPU**	**Ref**	**GPU**	**CPU**	**Ref**	**GPU**	**CPU**	**Ref**	**GPU**	**CPU**	**Ref**
Mul	**0.158**	0.499	0.735	**0.170**	0.603	1.220	**0.250**	0.859	1.798	**0.389**	1.101	3.886
Div	**0.202**	0.684	1.422	**0.207**	0.721	7.662	**0.300**	0.978	28.701	**0.467**	1.289	79.495
Add	**0.500**	2.328	1.705	**0.609**	2.868	2.575	**0.961**	3.784	4.354	**1.641**	4.860	6.710
Sub	**0.506**	2.250	1.771	**0.618**	2.877	2.707	**0.962**	3.861	4.438	**1.662**	4.991	7.063
Sqrt	–	–	–	**1.452**	6.888	–	**4.823**	18.586	–	–	–	–
MAC	**0.658**	2.827	2.440	**0.779**	3.471	3.795	**1.211**	4.643	6.151	**2.030**	5.961	10.596

report 490s, 162s, and 686s for addition, multiplication, and division, respectively. The 64-bit counterparts require substantially more time, amounting to 1207s, 686s, and 3559s for the same three operations.

Thus, we compare our operations against the SotA exact TFHE floating-point arithmetic of [4], evaluated on the same hardware. Their approach designs specialized low-level cryptographic routines, operating directly on ciphertext structures to achieve exact results. Our implementation operates at a higher abstraction level, composing standard `FheUint` primitives, the encrypted integer API provided by TFHE-rs, rather than redefining low-level ciphertext operations. Both approaches inherit the same failure probability ($\leq 2^{-40}$) from the cryptographic parameter sets. In our case, we utilized one of the pre-defined parameter sets provided by Zama for key generation. Crucially, our high-level approach enables GPU acceleration, which is not available for the baseline implementation.

Execution time. Table 3 reports per-operation timings across four bit-widths. The gains are concentrated in multiplication and division, where our approximate algorithms avoid the expensive mantissa-level computation. For FP32 multiplication, we achieve a 7.2× speedup on GPU and 2.1× on CPU over the baseline. For FP32 division, the improvement is 95.7× on GPU and 29.3× on CPU, reflecting the particularly high cost of exact encrypted division.

By sacrificing exactness for multiplication and division, we achieve significant speedups; nonetheless, for 32-bit operations, fully exact addition and subtraction remain 1.15× faster on CPU and 4.5× faster on GPU compared to the state of the art.

The square root operation has no baseline comparison. To our knowledge, this is the first TFHE implementation. At 4.823s on GPU for FP32, it is comparable in cost to approximately five additions.

Multiply-Accumulate (MAC). The MAC operation (one multiplication followed by one addition) is the fundamental building block of neural network layers. For FP32, our GPU implementation completes a MAC in 1.211s versus the baseline's 6.151s, a 5× speedup. Even on CPU alone (4.643s), we outperform the baseline. Since the addition is exact, the MAC result carries only the relative error from the multiplication step ($\leq 7.28\%$).

Bit-width scaling. Figure 3 shows how operation cost scales with bit-width. Our approximate multiplication and division scale sub-linearly ($\sim \text{bits}^{0.43}$ and $\sim \text{bits}^{0.42}$ respectively), while the exact baseline scales almost linearly ($\sim \text{bits}^{0.84}$) for multiplication, and super-linearly for division ($\sim \text{bits}^{1.69}$). This means the advantage of our approach grows with precision; a property that becomes increasingly relevant as higher-precision encrypted computation is explored.

Memory. Figure 4 compares memory consumption for FP32 operations. The most dramatic improvement is in division, where our implementation achieves a 57× reduction in peak memory and an 18.6× reduction in total allocation. Across all other operations, peak memory is reduced by 2–3× and total allocation is approximately halved. These reductions are critical for scalability, as encrypted neural network training must maintain many ciphertexts simultaneously across layers and batch elements. Regarding the square

Table 4: Comparison of state-of-the-art fully encrypted training frameworks. LHE denotes Leveled Homomorphic Encryption operating without bootstrapping. (✓) indicates native support, while (×) indicates a lack thereof.

Work	Linear	Non-linear	Turing Compl.	Precision	BatchNorm	Security
Colombo et al. [8]	TFHE	TFHE	✓	22-bit Int	×	128-bit
Glyph [20]	BGV/BFV (LHE)	TFHE	×	8-bit Int	✓	80-bit
HE-SecureNet [31]	CKKS (LHE)	FHEW	×	32-bit FP	×	119-bit
Ours	TFHE	TFHE	✓	32-bit FP	✓	128-bit

root operation, our implementation exhibits a peak memory usage of 8.8 MB; no comparison with prior work is possible, as no other proposal defines this operation.

Layer-level speedups. Translating operation-level gains to network layers, we estimate speedups by computing the operation mix for each layer type under FP32. For a single convolutional or fully connected layer (dominated by MACs), the estimated speedup is 26.4% on CPU and 65.5% on GPU. For batch normalization layers, which require a high proportion of divisions, the speedup reaches 58.0% on CPU and 80.3% on GPU, excluding the square root computation which has no baseline equivalent.

5.3 Encrypted vs. Emulated Equivalence

To ensure strict equivalence between the plain emulated and encrypted operations, we validated our approach both theoretically and empirically.

Theoretical guarantees. Our framework does not introduce new cryptographic primitives, nor does it modify existing ones. Because it is built entirely upon standard TFHE-rs unsigned integers (FheUint), core mechanisms such as noise management, key switching, and blind rotations apply identically, regardless of the fact that the underlying integer encodes an IEEE 754 floating-point value. Consequently, our theoretical guarantees of correctness and security are directly inherited from the established TFHE-rs library.

Empirical evaluation. Given the theoretical soundness of the underlying cryptography, the primary objective is to verify the semantic faithfulness of the implemented operations. To address this, we conducted an empirical validation by executing complex sequences of chained calculations that encompass our entire supported operation set. Specifically, the framework was tested across 50,000 chained random operations, an operation count that substantially exceeds a single forward+backward pass through ResNet-20. To prevent premature divergence to $\pm\infty$ during these chained executions, the input floating-point numbers were randomly sampled within the restricted range $[-10.0, 10.0]$, an interval that is notably broader than the typical distribution of weights and activations encountered in standard neural networks. In other words, the same chained operations and initial inputs were supplied to both the encrypted functions and their plaintext counterparts. The difference between the two executions was exactly zero throughout every iteration, demonstrating perfect semantic equivalence across the entire supported instruction set. A detailed analysis can be found in Appendix D. Furthermore, to evaluate the framework within its intended application domain, we directly compared the network's weights and activations obtained during the encrypted execution with those generated by the plaintext emulation. A comprehensive analysis of this comparison is detailed in Section 5.4.1.

5.4 Learning Experiments

We evaluated our framework through six experiments of varying complexity, summarized in Table 5. As experimental baselines, we compared against Colombo et al. [8], Glyph [20], and HE-SecureNet [31]. Colombo et al. is the most structurally similar to our work, utilizing a pure TFHE pipeline to maintain the native Turing-completeness of the scheme.

Conversely, Glyph and HE-SecureNet adopt a scheme-switching approach, combining word-wise LHE schemes (like CKKS or BGV) for linear layers with gate-based schemes (like TFHE or FHEW) for non-linearities. While this mitigates the latency of linear operations, it breaks the Turing-completeness of the pipeline and necessitates rigid architectural adaptations. Notably, the lack of an encrypted square root in HE-SecureNet prevents it from supporting Batch Normalization, a limitation overcome by both Glyph and our framework.

Beyond architectural differences, our approach provides stronger operational and security guarantees. We compute gradients using 32-bit floating-point arithmetic (matching HE-SecureNet, but surpassing the 22-bit limit of Colombo et al. and the 8-bit quantization of Glyph). Cryptographically, we strictly enforce a modern 128-bit security level, whereas HE-SecureNet operates at 119 bits and Glyph falls to an insecure 80-bit standard. Furthermore, our framework ensures future scalability by avoiding the deprecated cryptographic backends relied upon by previous works, such as NuFHE (Colombo et al.) and PEGASUS (HE-SecureNet). A summary of this comparison is provided in Table 4.

Only Experiment 1 uses fully encrypted training. Indeed, it further validates the framework and establish the exact equivalence between encrypted and emulated execution. Experiments 2–6 use validated emulation to demonstrate convergence on progressively deeper architectures and more challenging datasets. This represents a limitation caused by the prohibitive overhead of the scheme, which prevents us from executing fully encrypted training runs.

All reported speedups are estimated relative to a hypothetical baseline using the exact operations of [4], assuming their fastest theoretical implementation for each layer. Since the operation mix is similar across convolutional and fully connected layers (dominated by MACs), the estimated speedups are consistent at approximately 26–27% on CPU and 65–66% on GPU across most experiments. We therefore focus the discussion of each experiment on its unique findings rather than repeating these figures. Detailed architectures and hyperparameters are provided in Appendix E.

Table 5: Summary of learning experiments. *Enc*: fully encrypted training. *Emu*: validated plain-text emulation. ΔAcc denotes the accuracy difference relative to exact arithmetic (positive = our method is higher). Speedups are estimated from operation-level benchmarks (Section 5.2) at approximately 26–27% on CPU and 65–66% on GPU across all experiments. Exact mul. shows the fraction of multiplications using exact rather than approximate arithmetic.

Exp	Dataset	Architecture	Mode	Accuracy	Acc. Exact	ΔAcc	Exact mul.	Prior work
1	Ternary-MNIST	MLP [87]	Enc	92.47%	91.64%	+0.83%	0%	91.90%[a]
1	Ternary-MNIST	CNN [109]	Enc	96.03%	96.09%	−0.06%	0%	—
2	Fashion-MNIST	MLP [159k]	Emu	85.86%	86.73%	−0.87%	0%	86.00%[a]
2	Fashion-MNIST	LeNet-5 [62k]	Emu	93.29%	92.86%	+0.43%	0%	—
3	Derma-MNIST	CNN [366k]	Emu	75.76%	75.36%	+0.40%	0.00002%	73.20%[b]
3	Blood-MNIST	CNN [366k]	Emu	94.68%	94.12%	+0.56%	0.00002%	—
4	CIFAR-10	ResNet-20 [273k]	Emu	87.79%	87.92%	−0.13%	0.003%	—
5	Breast Cancer	MLP [650]	Emu	97.37%	97.37%	0.00%	0%	—
6	MNIST	MLP [118k]	Emu	97.60%	97.43%	+0.17%	0%	97.80%[c]
6	MNIST	CNN [127k]	Emu	97.91%	97.79%	+0.12%	0%	98.50%[c]

[a]Colombo et al. [8] [b]Lou et al. [20] (with transfer learning) [c]Schneider et al. [31]

Table 6: Layer-wise weight statistics (μ, σ^2) and final accuracy under three multiplication strategies for Experiment 1 architectures. Pure L-mul denotes Algorithm 1 without safeguards.

Exp	Arch.	Layer	Exact (Baseline)			Ours (Proposed)			L-mul (Pure)		
			μ	σ^2	Acc	μ	σ^2	Acc	μ	σ^2	Acc
1	MLP[87]	FC1	0.0152	0.0247	91.64%	0.0148	0.0243	92.47%	1.06×10^{37}	2.53×10^{74}	32.79%
		FC2	−0.1494	0.1018		−0.1489	0.0959		-1.16×10^{36}	1.51×10^{74}	
		FC3	−0.2104	0.1985		−0.2096	0.1948		−0.1977	0.0738	
1	CNN[109]	Conv	−0.2636	0.0865	96.09%	−0.2658	0.0846	96.03%	NaN	NaN	19.9%
		FC	0.0066	0.0275		0.0074	0.0285		NaN	NaN	

5.4.1 *Experiment 1: Encrypted Training Validation.*

Setup. Following Colombo et al. [8], we train a small MLP (87 parameters) on the Ternary MNIST dataset (50 images, 16×16) for 3 epochs. We additionally train a shallow CNN (109 parameters) on the same dataset, to our knowledge, the first fully encrypted small scale CNN training under TFHE.

Results. Both models converge successfully in the fully encrypted setting. The MLP achieves 92.47% accuracy, consistent with Colombo et al.'s reported 91.90%. The CNN reaches 96.03%, demonstrating that convolutional architectures substantially outperform MLPs even at this small scale.

Encrypted vs. emulated equivalence. The central result of this experiment is that the numerical difference between encrypted parameters and emulated parameters is *exactly zero* across all layers and both architectures. This confirms that our plain emulation reproduces encrypted training with bit-level fidelity, validating its use as a proxy for all subsequent experiments.

Wall-clock comparison. Fully encrypted MLP training completed in 604 minutes on our hardware. Colombo et al. report 13,140 minutes for the same task. However, this comparison is confounded by differences in hardware (different GPU) and a different library. The CNN required 1,774 minutes for encrypted training.

Safeguard ablation. Training without the dual safeguard (Sec. 4.4) produces immediate numerical collapse: weights diverge and NaN values appear within the first epoch. Table 6 provides a layer-wise comparison of weight statistics (μ, σ^2) across three configurations: L-mul [21], our safeguarded variant, and exact arithmetic. Pure L-mul diverges catastrophically, while our variant closely tracks exact arithmetic.

5.4.2 *Experiment 2: Fashion-MNIST.*

MLP. Building on the setup of Colombo et al. [8], we train an MLP with 159k parameters on Fashion-MNIST for 20 epochs. The achieved accuracy (85.86%) is within 1% of the exact arithmetic baseline (86.73%), confirming that the approximate arithmetic does not degrade learning on a moderately complex task.

LeNet-5. We introduce a LeNet-5 architecture [16] (62k parameters) trainable within the TFHE framework. Despite having 2.6× fewer parameters than the MLP, LeNet-5 achieves 93.29% accuracy, a 7.4 percentage point improvement, demonstrating the substantial benefit of CNNs for structured inputs. The accuracy is within

0.43% of the exact-operation baseline, consistent with the bounded per-operation error of our approximate multiplication.

5.4.3 Experiment 3: Medical Imaging. We evaluate a VGG-style CNN [32] (366k parameters) with batch normalization on two medical imaging datasets from MedMNIST [38].

Derma-MNIST. We compare against Glyph [20], who uses a hybrid TFHE-BGV/BFV pipeline with transfer learning. Following their setup (average pooling, no class rebalancing on this imbalanced dataset), our single-scheme TFHE approach trained from scratch achieves 75.76%, outperforming their baseline (69.20%) by 6.6 percentage points and their transfer-learning variant (73.20%) by 2.6 points, while eliminating the need for scheme conversions between layers. Direct runtime comparisons with Glyph are methodologically problematic and therefore omitted. In addition to the unavailability of Glyph source code, a equitable benchmark is precluded by fundamental differences in cryptographic robustness (an 80-bit security level versus our 128-bit standard) and numerical precision (8-bit quantized integers versus 32-bit floating-point arithmetic).

Blood-MNIST. On this more balanced dataset, we replace average pooling with max pooling and introduce learning rate decay at epoch 15. The final accuracy of 94.68% is consistent with the exact arithmetic baseline (94.12%).

Selective precision for batch normalization. This architecture systematically uses batch normalization, requiring the selective precision strategy described in Section 4.6.2: exact multiplication is used solely for updating moving averages and variances via the momentum factor. These exact operations constitute approximately 0.00002% of all multiplications per batch, having negligible impact on overall performance.

5.4.4 Experiment 4: ResNet-20 on CIFAR-10. We construct and train a ResNet-20 [11] (273k parameters), comprising 9 residual blocks with stacked convolutions, batch normalization, and skip connections, on CIFAR-10 [14] for 50 epochs with basic augmentation. This is, to our knowledge, the deepest architecture for which TFHE-compatible training has been demonstrated.

The model converges to 87.79% accuracy (exact baseline: 87.92%), provided that the selective precision strategy is extended beyond batch normalization to include momentum updates in the SGD optimizer for convolutional and batch normalization layers. Without this extension, training diverges after several epochs due to compounding multiplicative error in the velocity buffer (Section 4.6.2). The exact multiplications required for this strategy account for 0.003% of total multiplications per batch. We acknowledge this selective requirement as a limitation of our current approximation method.

The 87.79% accuracy is below SotA plaintext ResNet-20 results (~92% with data augmentation), due to three factors: the light data augmentation in our setup, the use of MSE loss rather than cross-entropy and the limited number of epochs used for training. The gap with our own exact-arithmetic baseline (0.13%) is the more relevant comparison, as it isolates the effect of approximate arithmetic.

5.4.5 Experiment 5: Tabular Data. To demonstrate versatility beyond image data, we train a small MLP (650 parameters) on the Breast Cancer Wisconsin dataset [36], a binary classification task with 30 tabular features. The framework achieves 97.37% accuracy, matching the exact baseline exactly. Notably, all multiplications (including momentum update) use approximate arithmetic without any selective precision. This contrasts with Experiments 3–4, and is consistent with the shallow depth of the network: with only three layers, multiplicative errors in the momentum buffer do not accumulate sufficiently to cause divergence.

5.4.6 Experiment 6: Comparison with HE-SecureNet. In this evaluation, we compare our work against HE-SecureNet [31], one of the fastest existing frameworks. HE-SecureNet employs a scheme-switching mechanism, utilizing CKKS in a Leveled Homomorphic Encryption (LHE) mode for linear operations, while delegating non-linearities to the FHEW scheme. Although not natively Turing-complete, this approach effectively demonstrates the primary advantage of scheme-switching: significantly reduced runtime. As shown in Table 4, HE-SecureNet matches our 32-bit floating-point (FP) precision; however, its security is limited to a sub-standard level of 119 bits.

HE-SecureNet achieves a tremendous speedup, largely due to the capacity of CKKS to perform highly efficient batched matrix multiplications. Any comparison with our work must acknowledge this theoretical performance advantage inherent to scheme-switching architectures. Despite the runtime differences, accuracy remains consistent between the two experiments, demonstrating that our approach can achieve equivalent predictive performance. However, a direct comparison of convergence capabilities is not possible. The original HE-SecureNet study [31] relied on Logistic Regression, which requires some epochs to reach plaintext accuracy, leaving its convergence behavior on more complex architectures unexplored.

Regarding communication overhead, our approach significantly benefits from using compressed encrypted unsigned integers (CompressedFheUint32). With this configuration, an encrypted FP32 number requires only 1,556 bytes instead of the original 263,440 bytes. Consequently, the most demanding communication requirement is the transmission of the encrypted training dataset, which totals 73.2 GB. The remaining communication overhead is minimal, consisting only of the initial weights and biases exchanged between the client and the server, and at the end of the training process the ones exchanged from the server to the client. Refer to Table 7 for the detailed comparison. Due to reproduction challenges (details in Appendix F), the total times in Table 7 are projected. HE-SecureNet runtimes were obtained by measuring a single training epoch on our hardware and extrapolating linearly to the full training duration. Our runtimes are projected estimates derived from the operation-level benchmarks of Section 5.2 on the same hardware. Our measured HE-SecureNet runtime on our hardware differs substantially from the values reported in their paper (their Table 8 reports ~115 minutes total for MNIST MLP at batch 128, compared to our extrapolated 6,744 minutes). We attribute this gap to hardware differences between the Threadripper 3990X used in their evaluation and the Xeon Gold 5318S available to us. In fact, we do not claim that measurements in Table 7 reflect a precise end-to-end runtime benchmark of HE-SecureNet optimal achievable performance, but only an approximate comparison obtained

Table 7: Comparison of our proposed framework against HE-SecureNet [31]. HE-SecureNet runtimes were obtained by measuring a single training epoch on our hardware (NVIDIA A40 GPU, Xeon Gold 5318S CPU) and extrapolating linearly to the full training duration. Per-epoch runtimes measured on our platform differ from those implied by the totals in [31]; Appendix F documents our reproduction attempts and the platform differences that may account for this. Our runtimes are projected estimates derived from the operation-level benchmarks of Section 5.2 on the same hardware. The final two columns report the total communication overhead required by each framework during training.

Exp	Dataset	Architecture	Our Runtime	[31] Runtime	Our Comm.	[31] Comm.
6	MNIST	MLP[118k]	1.07×10^{10} min ‡	6,744 min †	73.56 GB	149.4 GB
6	MNIST	CNN[127k]	1.37×10^{10} min ‡	8,792 min †	73.59 GB	132.2 GB

† Measured value on a single epoch and then projected. ‡ Estimated projection.

under imperfect reproduction conditions. We provide complete methodology in Appendix F to support independent reproduction.

5.5 Wall-Clock Projections

Table 8 provides estimated fully encrypted training times for all architectures, derived from the operation counts per epoch and the per-operation GPU benchmarks in Table 3. For the two Experiment 1 architectures, estimated and measured times are consistent, validating the projection methodology.

Table 8: Estimated and measured wall-clock time for fully encrypted training (GPU). Measured values shown where available. Projections are computed from per-operation benchmarks and operation counts per epoch.

Exp	Architecture	Params	Epochs	Training Time
1	MLP	87	3	604 min †
1	CNN	109	3	1,774 min †
2	MLP	159k	20	9.64×10^{9} min ‡
2	LeNet-5	62k	20	2.54×10^{10} min ‡
3	CNN-VGG	366k	20	4.73×10^{11} min ‡
4	ResNet-20	273k	50	1.25×10^{13} min‡
5	MLP	650	100	1.81×10^{6} min ‡
6	MLP	118k	25	1.07×10^{10} min ‡
6	CNN	127k	25	1.37×10^{10} min ‡

† Measured value. ‡ Estimated projection.

The projections make clear that fully encrypted training of deep architectures remains beyond current hardware capabilities. However, our algorithmic contribution (the approximate floating-point primitives) is independent of the underlying TFHE implementation speed. Every future improvement in bootstrapping latency, hardware acceleration, or parameter optimization will directly reduce these wall-clock times while preserving the convergence properties demonstrated through emulation. The operation-level speedups reported in Section 5.2 represent a permanent algorithmic advantage that compounds with such infrastructure advances.

6 Conclusions

We have presented a framework for approximate floating-point arithmetic within TFHE, enabling neural network training on a scheme that natively supports only integer and boolean operations. The core contributions are a redesigned L-mul multiplication with numerical safeguards for encrypted training stability, a new approximate division derived from the same principle, and the first TFHE implementation of exact square root, collectively yielding a sufficient arithmetic for backpropagation with SGD.

At the operation level, the GPU-based approximate arithmetic achieves up to 7.2× faster multiplication and 95.7× faster division compared to exact TFHE floating-point arithmetic, with sub-linear bit-width scaling that makes the advantage grow with precision; peak memory is reduced by up to 57×. Using these primitives, we performed the first fully encrypted training of a small-scale CNN under TFHE, and confirmed zero numerical divergence between encrypted and emulated execution, validating emulation as a faithful proxy.

Through validated emulation, we demonstrated that approximate floating-point backpropagation preserves convergence on architectures from LeNet-5 to ResNet-20 with batch normalization and residual connections, matching exact-arithmetic baselines within 1% accuracy across six benchmarks. A key finding is that while most multiplications tolerate approximation, momentum-driven updates in deep architectures require exact arithmetic to prevent error accumulation, though these constitute less than 0.01% of total multiplications.

Limitations. Fully encrypted training of deep architectures remains computationally infeasible with current TFHE implementations, as our wall-clock projections make explicit. The framework is currently limited to MSE loss due to the absence of encrypted exponential and logarithm operations, and the selective precision requirement for momentum, while negligible in cost, indicates that the approximation is not universally applicable to all components of the training pipeline.

Future work. Natural extensions include implementing exponential and logarithm operations to support cross-entropy loss, extending the optimizer to Adam or AdamW (enabled by our square root), and exploring lower-precision formats (FP16, FP8) where the sub-linear scaling of our arithmetic would still yield consistent speedups. More broadly, since our framework operates entirely through standard TFHE integer primitives, it will directly benefit from ongoing advances in bootstrapping acceleration, ASIC-based TFHE hardware, and improved parameter sets, narrowing the gap between emulated and practical encrypted training.

Acknowledgments

This paper is supported by Dhiria SRL. AI-based tools have been used to proofread and improve the text.

References

[1] 1985. IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985* (1985), 1–20. https://doi.org/10.1109/IEEESTD.1985.82928
[2] 2008. IEEE Standard for Floating-Point Arithmetic. https://doi.org/10.1109/IEEESTD.2008.4610935
[3] 2019. IEEE Standard for Floating-Point Arithmetic. https://doi.org/10.1109/IEEESTD.2019.8766229
[4] Loris Bergerat, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. 2025. TFHE Gets Real: an Efficient and Flexible Homomorphic Floating-Point Arithmetic. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2025, 2 (March 2025), 126–162. https://doi.org/10.46586/tches.v2025.i2.126-162 Artifact available at https://artifacts.iacr.org/tches/2025/a16.
[5] Ke Cheng, Ning Xi, Ximeng Liu, Xinghui Zhu, Haichang Gao, Zhiwei Zhang, and Yulong Shen. 2023. Private Inference for Deep Neural Networks: A Secure, Adaptive, and Efficient Realization. *IEEE Trans. Comput.* 72, 12 (2023), 3519–3531. https://doi.org/10.1109/TC.2023.3305754
[6] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *J. Cryptol.* 33, 1 (Jan. 2020), 34–91. https://doi.org/10.1007/s00145-019-09319-x
[7] Pierre-Emmanuel Clet, Martin Zuber, Aymen Boudguiga, Renaud Sirdey, and Cédric Gouy-Pailler. 2022. Putting up the swiss army knife of homomorphic calculations by means of TFHE functional bootstrapping. Cryptology ePrint Archive, Paper 2022/149. https://eprint.iacr.org/2022/149
[8] Luca Colombo, Alessandro Falcetta, and Manuel Roveri. 2024. Training Encrypted Neural Networks on Encrypted Data with Fully Homomorphic Encryption. In *Proceedings of the 12th Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (Salt Lake City, UT, USA) *(WAHC '24)*. Association for Computing Machinery, New York, NY, USA, 64–75. https://doi.org/10.1145/3689945.3694802
[9] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48* (New York, NY, USA) *(ICML'16)*. JMLR.org, 201–210.
[10] John R. Hauser. 2024. Berkeley SoftFloat. https://www.jhauser.us/arithmetic/SoftFloat.html.
[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90
[12] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1412.6980
[13] Atli Kosson and Martin Jaggi. 2023. Multiplication-Free Transformer Training via Piecewise Affine Operations. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 8208–8223. https://proceedings.neurips.cc/paper_files/paper/2023/file/19df21cd4931bd0caaa4d8480e9a59cd-Paper-Conference.pdf
[14] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. (2009), 32–33. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf
[15] Yann LeCun. 2018. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/
[16] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[17] Seunghwan Lee and Dong-Joon Shin. 2024. Overflow-Detectable Floating-Point Fully Homomorphic Encryption. *IEEE Access* 12 (2024), 6160–6180. https://doi.org/10.1109/ACCESS.2024.3351738
[18] Timothy P. Lillicrap, Daniel Cownden, Douglas B. Tweed, and Colin J. Akerman. 2016. Random Feedback Weights Support Learning in Deep Neural Networks. *Nature Communications* 7 (2016), 13276. https://doi.org/10.1038/ncomms13276
[19] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Bkg6RiCqY7
[20] Qian Lou, Bo Feng, Geoffrey Charles Fox, and Lei Jiang. 2020. Glyph: Fast and Accurately Training Deep Neural Networks on Encrypted Data. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 9193–9202. https://proceedings.neurips.cc/paper_files/paper/2020/file/685ac8cadc1be5ac98da9556bc1c8d9e-Paper.pdf
[21] Hongyin Luo and Wei Sun. 2024. Addition is All You Need for Energy-efficient Language Models. https://openreview.net/forum?id=nXV3C8aKxZ
[22] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2013. On Ideal Lattices and Learning with Errors over Rings. *J. ACM* 60, 6, Article 43 (Nov. 2013), 35 pages. https://doi.org/10.1145/2535925
[23] Tsuguo Mogami. 2020. Deep Neural Network Training without Multiplications. arXiv:2012.03458 [cs.LG]
[24] Luis Montero, Jordan Frery, Celia Kherfallah, Roman Bredehoft, and Andrei Stoian. 2024. Machine Learning Training on Encrypted Data with TFHE. In *Proceedings of the 10th ACM International Workshop on Security and Privacy Analytics* (Porto, Portugal) *(IWSPA '24)*. Association for Computing Machinery, New York, NY, USA, 71–76. https://doi.org/10.1145/3643651.3659891
[25] Subin Moon and Younho Lee. 2020. An Efficient Encrypted Floating-Point Representation Using HEAAN and TFHE. *Security and Communication Networks* 2020 (03 2020), 1–18. https://doi.org/10.1155/2020/1250295
[26] Behrooz Parhami. 2010. *Computer Arithmetic: Algorithms and Hardware Designs* (2nd ed.). Oxford University Press.
[27] Saerom Park, Junyoung Byun, Joohee Lee, Jung Hee Cheon, and Jaewook Lee. 2020. HE-Friendly Algorithm for Privacy-Preserving SVM Training. *IEEE Access* 8 (2020), 57414–57425. https://doi.org/10.1109/ACCESS.2020.2981818
[28] Alberto Pirillo and Luca Colombo. 2025. ReBoot: Encrypted Training of Deep Neural Networks with CKKS Bootstrapping. arXiv:2506.19693 [cs.LG]
[29] Oded Regev. 2005. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC '05)*, Harold N. Gabow and Ronald Fagin (Eds.). ACM, New York, NY, USA, 84–93. https://doi.org/10.1145/1060590.1060603
[30] Arjun Roy and Kaushik Roy. 2025. DCT-CryptoNets: Scaling Private Inference in the Frequency Domain. In *International Conference on Learning Representations*, Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (Eds.), Vol. 2025. 14690–14707. https://proceedings.iclr.cc/paper_files/paper/2025/file/26407588dfb08e48c459c074ab6adb7d-Paper-Conference.pdf
[31] Thomas Schneider, Huan-Chih Wang, and Hossein Yalame. 2025. HE-SecureNet: An Efficient and Usable Framework for Model Training via Homomorphic Encryption. In *Proceedings of the 24th Workshop on Privacy in the Electronic Society (WPES '25)*. Association for Computing Machinery, New York, NY, USA, 104–115. https://doi.org/10.1145/3733802.3764063
[32] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Association for Computing Machinery.
[33] Jaewoo Song and Fangzhen Lin. 2022. PocketNN: Integer-only Training and Inference of Neural Networks via Direct Feedback Alignment and Pocket Activations in Pure C++. arXiv:2201.02863 [cs.LG]
[34] Zama team. 2022. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data. Version 1.0.0 (and later); available at https://github.com/zama-ai/tfhe-rs.
[35] Maolin Wang, Seyedramin Rasoulinezhad, Philip H. W. Leong, and Hayden K.-H. So. 2022. NITI: Training Integer Neural Networks Using Integer-Only Arithmetic. *IEEE Transactions on Parallel and Distributed Systems* 33, 11 (Nov. 2022), 3249–3261. https://doi.org/10.1109/tpds.2022.3149787
[36] Wolberg William, Mangasarian Olvi, Street Nick, and Street W. 1993. Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5DW2B.
[37] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747* (2017).
[38] Jiancheng Yang, Rui Shi, and Bingbing Ni. 2023. MedMNIST v2-A large-scale lightweight benchmark for 2D and 3D biomedical image classification. *Scientific Data* 10, 1 (2023), 41.

A TFHE Ciphertext Definitions

To supplement the background in Section 2.1, we briefly define the three ciphertext structures utilized within the TFHE-rs library and our framework:

- **TLWE (Torus Learning With Errors):** The most basic encryption format, typically used to encrypt single scalars (e.g., individual bits or small integers). It consists of a vector of mask elements and a body containing the noisy message.
- **TRLWE (Torus Ring-LWE):** A polynomial-based extension of TLWE. It encrypts a vector of values into a single polynomial, allowing for efficient SIMD-like additions and rotations. This structure is the foundation for the TFHE *Blind Check* procedure.

- **TRGSW (Torus Ring Gentry-Sahai-Waters):** A complex structure composed of multiple TRLWE samples. While computationally heavier, TRGSW samples support external multiplications (TRGSW × TRLWE), acting as the "selector" or control bit in homomorphic multiplexers and bootstrapping.

B Relative-error analysis

Approximation error between multiple intervals. As discussed in Section 4.3.1, the approximation error for our multiplication exhibits a self-similar pattern across intervals bounded by powers of two. Specifically, the error distribution repeats identically within every $[2^k, 2^{k+1}]$ range, with the heatmap simply scaling in magnitude and mirroring for negative operands. However, as illustrated in Figure 5, the maximum absolute error scales proportionally with the maximum operand values within any given interval. Same applies for our approximation division (Figure 6).

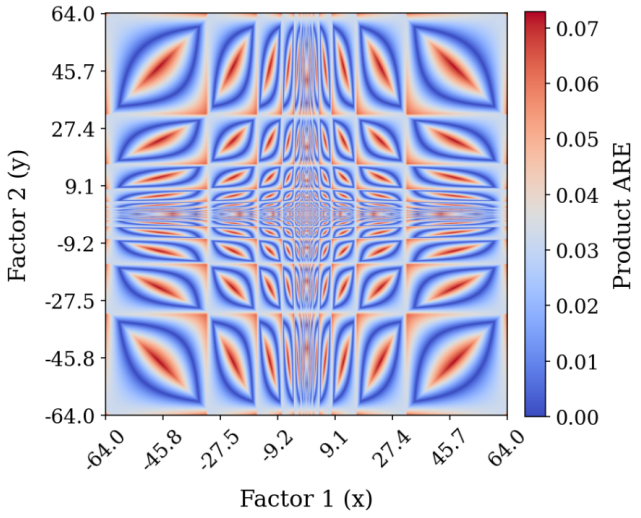

Figure 5: Heatmap of the absolute relative error between approximate and exact multiplication for operands in $[-64, 64]$.

Safeguard for Case A and B. As detailed in Section 4.4, multiplying by zero introduces critical vulnerabilities in the baseline approximation. Without intervention, the error is never strictly zero and varies heavily depending on the magnitude of the second operand. Specifically, in Case B, if the second factor y falls within the interval $(-1.0, 1.0)$, the error compounds until the result degrades to NaN. For $|y| \geq 1.0$, the error is negligible but remains non-zero. This behavior is illustrated in Figure 7 (left), where x is fixed to 0 and y varies in the range $[-64, 64]$. The baseline method exhibits massive error spikes exceeding 10^{13} within the $(-1.0, 1.0)$ interval, alongside a singularity at $y = 0$ (which yields an anomalous result of 4.25, as described in Case A). Conversely, Figure 7 (right) demonstrates the efficacy of our numerical safeguards: by explicitly flushing these boundary values to zero, the approximate multiplication matches the exact arithmetic perfectly, completely resolving Cases A and B with an error of exactly 0.0 across the entire range.

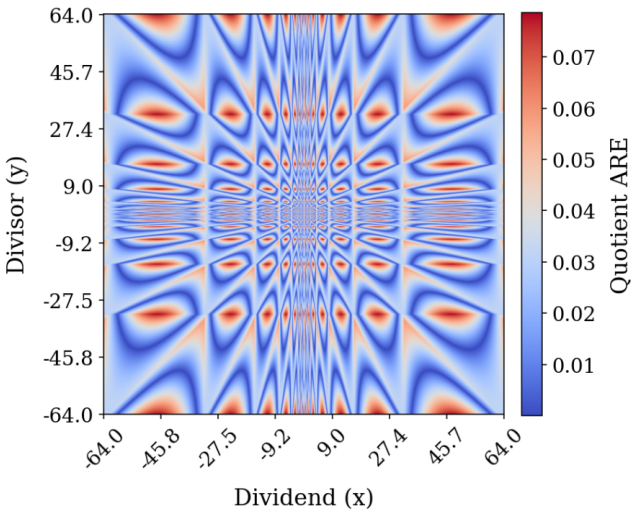

Figure 6: Heatmap of the absolute relative error between approximate and exact division for operands in $[-64, 64]$.

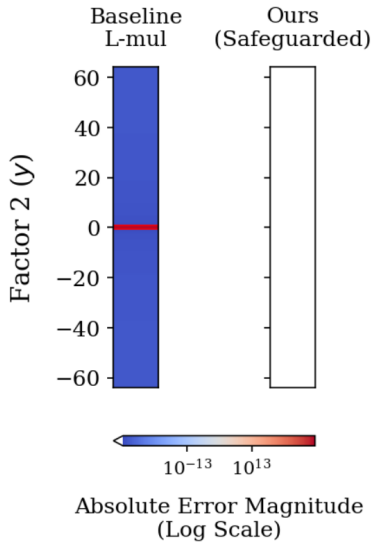

Figure 7: Heatmap of the absolute relative error for Case A and B, with and without safeguards.

Safeguard for Case C. Figure 8 evaluates Case C, which isolates the extreme subnormal range. Although this numerical interval is exceptionally narrow, the baseline method generates catastrophic errors consistently exceeding 10^{108}. By implementing our safeguard mechanism, any subnormal output is systematically flushed to zero, independent of the inputs. While this does not yield an error of exactly zero, since the true mathematical product is a non-zero subnormal, it successfully bounds the instability, reporting a constant and strictly controlled absolute relative error of 1.0 (Figure 9).

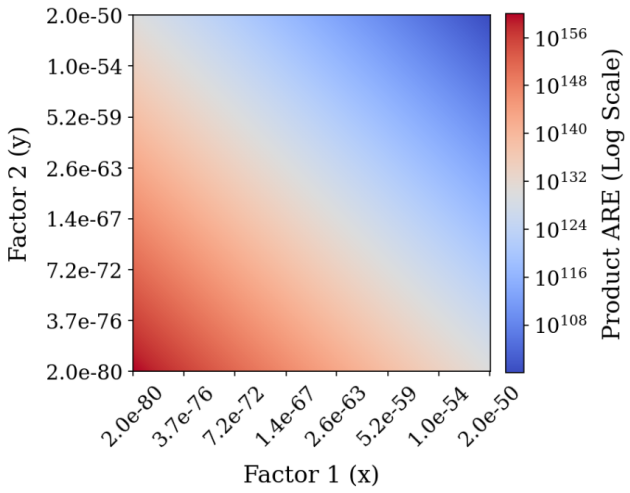

Figure 8: Heatmap of the absolute relative error in the subnormal range without safeguard

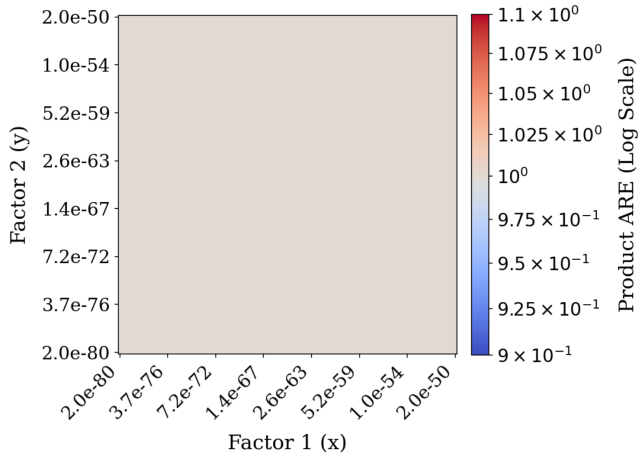

Figure 9: Heatmap of the absolute relative error in the subnormal range with safeguard

C Square Root Algorithm

Among several possible methods to compute the exact square root of a floating-point number in this work it is adopted the digit-by-digit restoring shift/subtract algorithm, as described in [26].
This algorithm is based on shift and subtraction operation. Iterative algorithms like the Newton–Raphson method are often preferred for calculating square roots in plaintext computations. However, these methods require repeated floating-point additions, multiplications, and divisions, which become costly and complex to perform efficiently when data is encrypted. This explains why the digit-by-digit algorithm is preferred in this context.
The full mathematical formulation and proof of correctness are given in Parhami's book [26]. Here, it is provided a more pseudocode style description adapted to this context.

The first step is to extract the exponent x_e and the mantissa x_m. As in addition, the mantissa must include the implicit leading 1, explicitly placed at position $n_m + 1$.
To recall, a floating-point number can be written as:

$$x = \pm 1.x_m \times 2^{x_e}.$$

Taking the square root gives:

$$\sqrt{x} = \sqrt{x_m \cdot 2^{x_e}} = \sqrt{x_m} \cdot 2^{x_e/2},$$

thus the exponent must be halved.
If x_e is even, then $x_e/2$ is valid. But if x_e is odd, halving it produces a fractional exponent, that cannot be directly represented. For example:

$$\sqrt{1.5 \cdot 2^5} = \sqrt{1.5} \cdot 2^{2.5}.$$

To fix this, it is required to:

(1) Reduce the exponent by 1, making it even:

$$1.5 \cdot 2^5 = (1.5 \cdot 2^1) \cdot 2^4 = 3.0 \cdot 2^4$$

(2) Compensate in the mantissa, multiplying the latter by 2:

$$\sqrt{1.5 \cdot 2^5} = \sqrt{3.0 \cdot 2^4} = \sqrt{3.0} \cdot 2^2$$

In practice, this means to check whether the exponent is odd. If so, left-shift the mantissa (multiply by 2), and right-shift the exponent (divide by 2). If even, simply right-shift the exponent.
Once these steps have been done, the digit-by-digit iteration begins. The loop runs for exactly $n_m + 1$ steps.
At each step j, the algorithm contains:

- a partial root $q^{(j-1)}$,
- a partial remainder $s^{(j-1)}$,
- and a trial value of the next root digit.

The recurrence relation is derived from the square-root identity:

$$s^{(j-1)} = z - (q^{(j-1)})^2.$$

The iteration proceeds as follows:

(1) Test whether adding the bit weight at position j to the root keeps the square below the radicand. It is equivalent to check:

$$s^{(j-1)} \geq 2q^{(j-1)} + 2^{-j}.$$

(2) If the condition holds set $q_{-j} = 1$, update the remainder:

$$s^{(j)} = s^{(j-1)} - (2q^{(j-1)} + 2^{-j})$$

and update the root by shifting and inserting the bit.
If the condition fails $q_{-j} = 0$ the remainder is restored, thus unchanged, the root is simply shifted without adding a value.

(3) Advance to the next iteration by consuming two radicand bits, i.e. shift right by two.

Additionally, it can be added a further check to verify if the input number is non-negative and return a standard value (e.g. zero), as it has been done in the actual implementation.

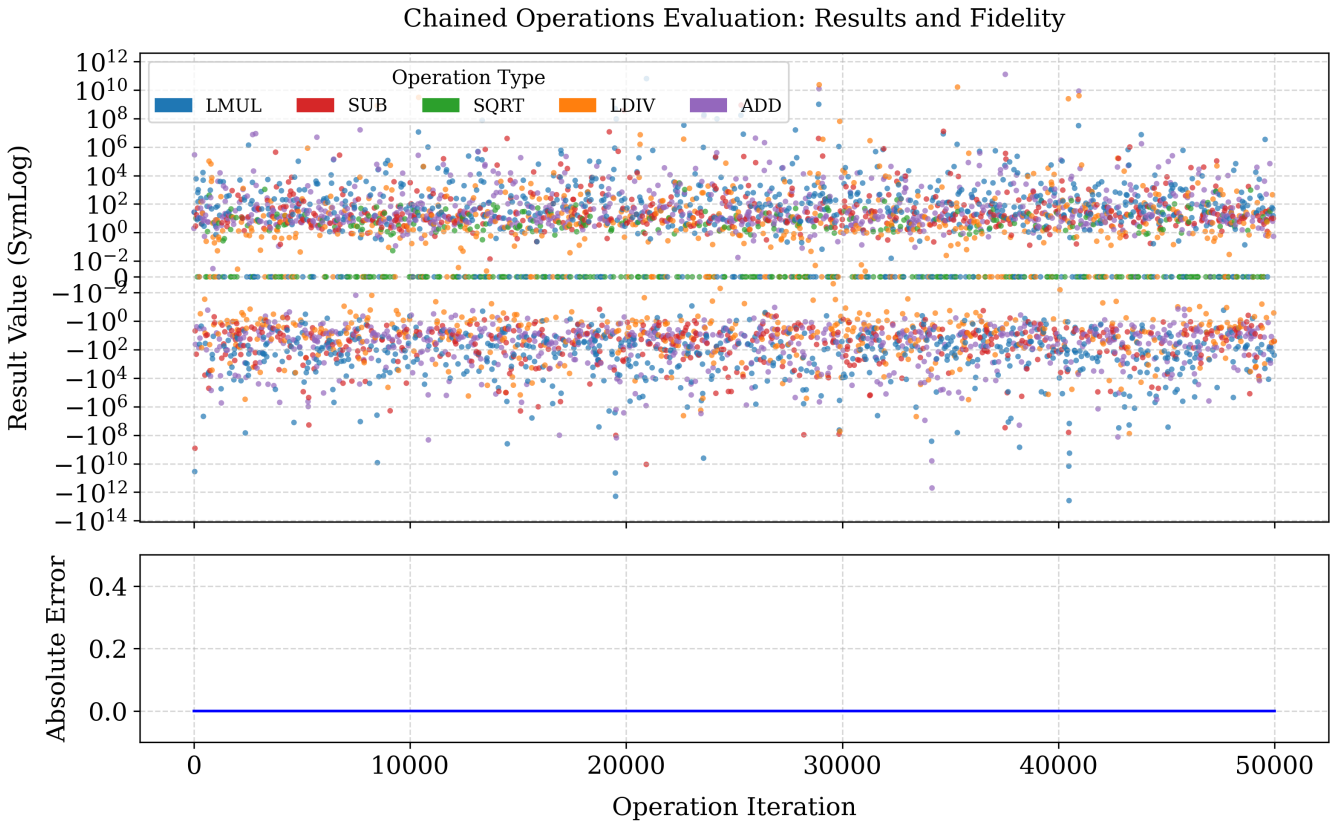

Figure 10: Analysis of operation equivalence: The upper plot displays the resulting values on a log scale categorized by operation type. The bottom plot illustrates the difference between encrypted and emulated operations, demonstrating consistent numerical alignment.

Algorithm 4 Square Root

Require: x

$(\_, x_e, x_m) \leftarrow unpack(x)$ ▷ extract exponent and mantissa
$odd \leftarrow x_e.is\_odd()$
$x_m \leftarrow odd\ ?\ (x_m \ll 1) : x_m$ ▷ adjust mantissa for odd exponent
$x_e \leftarrow x_e \gg 1$ ▷ halve exponent
$result \leftarrow 0$
$z \leftarrow x_m$
for $j = (n_m + 1)$ **downto** 0 **do**
  if $z \geq result + 2^j$ **then**
    $z \leftarrow z - (result + 2^j)$
    $result \leftarrow (result \gg 1) + 2^j$
  else
    $result \leftarrow result \gg 1$
  end if
  $j \leftarrow j - 2$
end for
return $pack(x_s, x_e, result)$ ▷ reconstruct result

D Encrypted vs. Emulated Equivalence Analysis

As detailed in Section 5.3, the empirical validation of the equivalence was conducted by executing a sequence of 50,000 randomly chained operations. To ensure the test reflected realistic conditions, each of the five operations was assigned a selection probability based on its typical frequency in neural network training. Specifically, addition and multiplication were assigned the highest probabilities, subtraction and division a moderate probability, and the square root the lowest.

The sampling range was restricted to $[-10.0, 10.0]$, which successfully prevented numerical divergence toward $\pm\infty$; as shown in Figure 10, the absolute values of the results were capped at 10^{12}. Regarding the square root computation (represented by the green dots in the plot), the values are frequently zero because the operation is undefined for negative inputs. Consequently, these results consistently remain non-negative.

Ultimately, the Absolute Error remained at zero across every iteration. This lack of deviation provides empirical confirmation of the functional equivalence between the encrypted and emulated implementations.

E Experimental Details

E.1 Network Architectures

The MLP architectures are straightforward fully connected networks. Tables 9–12 detail all architectures. The ResNet-20 follows the standard design of He et al. [11] with 9 residual blocks.

Table 9: Network architecture for Experiment 1: CNN on Ternary-MNIST.

Stage	Layer Type	Output size
Input	Image	16×16
MaxPool	Max Pooling	8×8
Conv1	Conv(3x3) + ReLU	$4 \times 4 \times 2$
Fc1	Fully Connected	3

Table 10: Network architecture for Experiment 2: LeNet-5 for Fashion-MNIST.

Stage	Layer Type	Output size
Input	Image	28×28
Conv1	Conv(5x5) + ReLU	$28 \times 28 \times 6$
AvgPool1	Avg Pooling	$14 \times 14 \times 6$
Conv2	Conv(5x5) + ReLU	$10 \times 10 \times 16$
AvgPool2	Avg Pooling	$5 \times 5 \times 16$
Fc1	FC + ReLU	120
Fc2	FC + ReLU	84
Fc3	Fully Connected	10

Table 11: Network architecture for Experiment 4: ResNet-20 on CIFAR-10.

Stage	Layer Type	Output size
Input	Image	$32 \times 32 \times 3$
Initial	Conv (3×3) Batch Norm	$32 \times 32 \times 16$
Stage 1 (RB 1-3)	$\mathbf{3 \times}$ **Residual Blocks** $\begin{bmatrix}\text{Conv } 3 \times 3, 16\\ \text{Conv } 3 \times 3, 16\end{bmatrix}$	$32 \times 32 \times 16$
Stage 2 (RB 4-6)	$\mathbf{3 \times}$ **Residual Blocks** $\begin{bmatrix}\text{Conv } 3 \times 3, 32\\ \text{Conv } 3 \times 3, 32\end{bmatrix}$	$16 \times 16 \times 32$
Stage 3 (RB 7-9)	$\mathbf{3 \times}$ **Residual Blocks** $\begin{bmatrix}\text{Conv } 3 \times 3, 64\\ \text{Conv } 3 \times 3, 64\end{bmatrix}$	$8 \times 8 \times 64$
Pooling Classifier	Global Avg Pooling Fully Connected	$1 \times 1 \times 64$ 10

Table 12: Network architecture for Experiment 5: MLP on Breast Cancer.

Stage	Layer Type	Output size
Input	Tabular	30
Fc1	FC + ReLU	16
Fc2	FC + ReLU	8
Fc3	FC + ReLU	2

E.2 Activation Functions

MLPs from experiments 1 and 2 use a piecewise linear approximation of the hyperbolic tangent:

$$\tanh(x) \approx \alpha \cdot_{\blacktriangle} x + \beta \tag{8}$$

where α and β are the slope and intercept of the linear segment containing x. All other experiments use ReLU, which requires only an encrypted comparison and is therefore exact.

E.3 Hyperparameters

Table 13 summarizes the training hyperparameters for all experiments. Where hyperparameters follow the setup of a prior work, this is indicated. All experiments use SGD with MSE loss.

E.4 Operation Counts per Epoch

Table 14 details the estimated number of operations required per training epoch for each architecture. The "Mul Ex." column shows the number of exact multiplications required by the selective precision strategy (Section 4.6.2), confirming that these constitute a negligible fraction of total multiplications in all cases. The "Add" column includes subtractions, as the underlying algorithm is identical apart from a sign-bit flip.

E.5 Learning Dynamics and Convergence

Figure 11 compares the learning curves of our approximate arithmetic against exact arithmetic across all experiments. In all cases, the approximate method converges to accuracy within 1% of the exact baseline, consistent with the bounded per-operation error of L-mul.

The widest gaps appear in Experiments 1 and 2, which use the piecewise linear tanh approximation. This additional source of error, independent of the L-mul approximation, compounds with the approximate multiplication to produce a slightly larger accuracy difference. From Experiment 3 onward, where ReLU replaces tanh, the gap narrows and the learning curves closely overlap.

F HE-SecureNet Reproduction

As noted in Section 5.4.6, several technical obstacles prevented a full reproduction of the HE-SecureNet [31] framework. Although the repository provided pre-compiled binaries, executing these files consistently resulted in a runtime crash after the completion of the first epoch. Both experimental models failed at the exact same execution point with the following error:

```
628 runtime_->LoadCtx(&W_cipher[kc][ic][i][j], osin)
LoadOneModuli: invalid ct header: pid
```

Table 13: Training hyperparameters for all experiments. LR: learning rate. BS: batch size. γ: momentum. λ: weight decay. A dash indicates the feature is not used.

Exp	Architecture	Dataset	Ep.	BS	LR	Notes	γ	λ
1	MLP [87]	Ternary-MNIST	3	5	0.1	Per [8]	—	—
1	CNN [109]	Ternary-MNIST	3	2	0.3	—	—	—
2	MLP [159k]	Fashion-MNIST	20	50	0.5	Per [8]	—	—
2	LeNet-5 [62k]	Fashion-MNIST	20	50	2.0	—	—	—
3	CNN [366k]	Derma-MNIST	15	60	0.05	Per [20]	0.9	0.0001
3	CNN [366k]	Blood-MNIST	20	64	$0.1 \rightarrow 0.01$	Decay at ep. 15	0.9	0.0001
4	ResNet-20 [273k]	CIFAR-10	50	64	0.1	$\times 0.1$ at ep. 15, 38, 45	0.9	0.0001
5	MLP [650]	Breast Cancer	100	16	0.005	—	0.9	0.0001
6	MLP[118k]	MNIST	25	128	0.5	Per [31]	-	-
6	CNN[127k]	MNIST	25	128	0.5	Per [31]	-	-

Table 14: Estimated operation counts per training epoch (forward + backward pass, full dataset).

Exp	Architecture	Mul	Mul Ex.	Add	Div	Sqrt
1	MLP [87]	16020	0	14040	10	0
1	CNN [109]	36625	0	30825	25	0
2	MLP [159k]	2.41×10^{10}	0	2.4×10^{10}	1000	0
2	LeNet-5 [62k]	6.26×10^{10}	0	6.29×10^{10}	7.88×10^{7}	0
3	VGG-CNN [366k]	1.7×10^{12}	273920	1.7×10^{12}	1.48×10^{8}	68480
4	ResNet-20 [273k]	1.24×10^{13}	4.29×10^{8}	1.24×10^{13}	1.1×10^{7}	1.2×10^{6}
5	MLP [650]	926620	0	894389	29	0
6	MLP[118k]	2.13×10^{10}	0	2.12×10^{10}	469	0
6	CNN[127k]	2.73×10^{10}	0	2.72×10^{10}	469	0

In an attempt to bypass this, we tried to rebuild the project using the provided Python-based model definitions. However, the transpiler from the Python model definitions to C++ produced code that failed to compile (an undeclared identifier 11 in the generated `model_b.cpp`). Resolving these issues would have required substantial modifications to the underlying framework. We therefore measured a single successful training epoch directly and extrapolated to the reported number of epochs.

We note that the per-epoch runtimes measured on our platform differ substantially from those implied by the total training times reported in Table 8 of [31]. For MNIST MLP at batch size 128 over 25 epochs, [31] reports a total training time of 1.91 hours, implying approximately 4.6 minutes per epoch. Our single-epoch measurement on the same configuration was approximately 270 minutes, a per-epoch difference of approximately 58×. For MNIST CNN, the analogous comparison is approximately 7.6 minutes per epoch reported versus approximately 350 minutes per epoch measured (approximately 46×). We attribute this gap primarily to hardware differences. [31] evaluates on an AMD Ryzen Threadripper 3990X (64 cores, 256 MB L3 cache, quad-channel DDR4), whereas our reproduction used an Intel Xeon Gold 5318S (24 cores, 36 MB L3 cache, eight-channel DDR4). The 2.7× difference in core count and the 7× difference in L3 cache size are both substantial, and HE-based workloads (dominated by polynomial-arithmetic and NTT operations) are highly sensitive to both. The technical obstacles described above prevented us from investigating further or recompiling under platform-specific optimization flags. We therefore report only what we measured under our specified conditions and make no claim about HE-SecureNet optimal performance on its intended hardware.

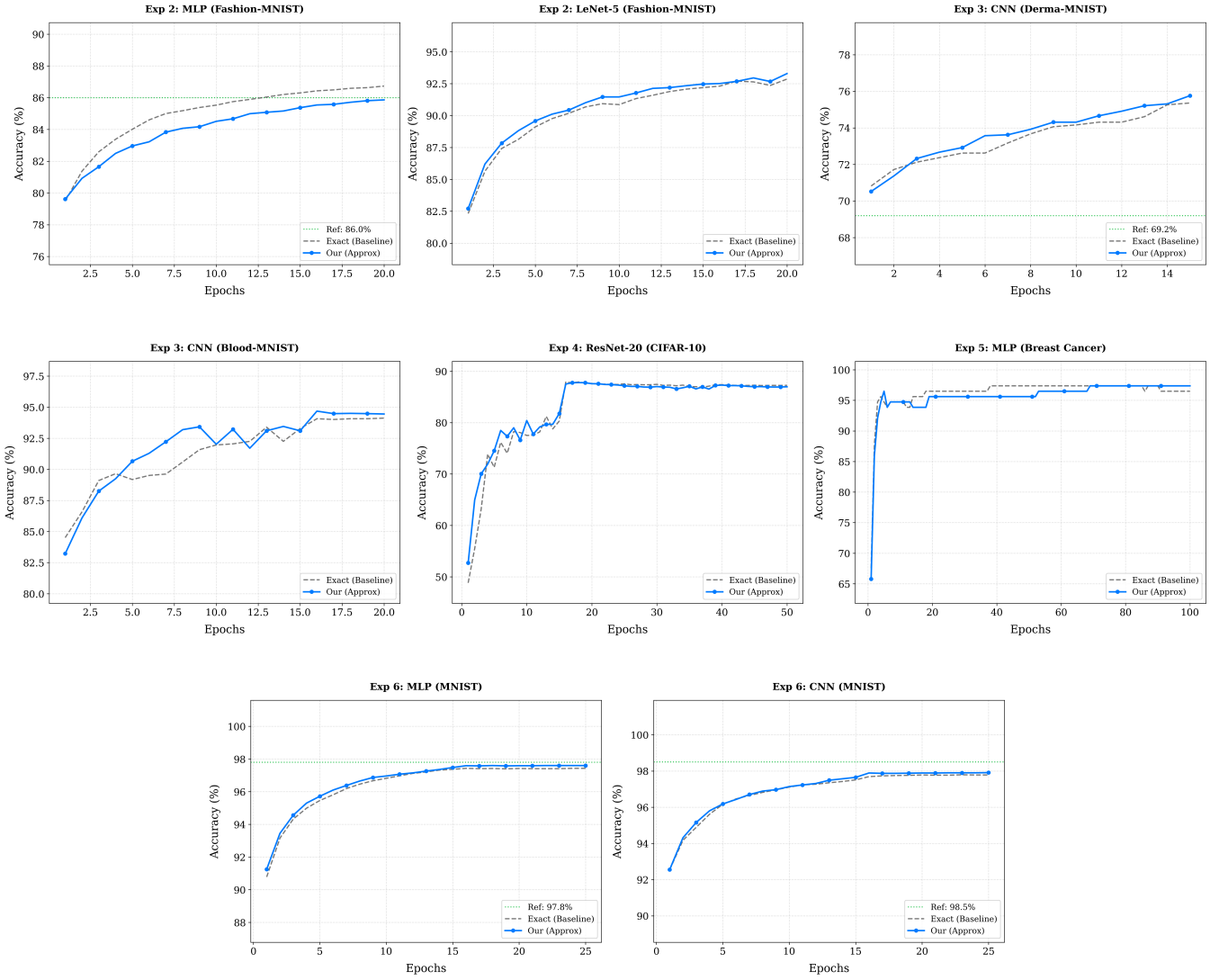

Figure 11: Learning curves across all experiments. Dotted gray: exact arithmetic baseline. Solid blue: proposed approximate arithmetic. Green horizontal lines: reference accuracy from prior work where available.