An Improved Logarithmic Multiplier for Energy-Efficient Neural Computing

Vedant Mangrulkar (IMT2022519) Shannon Muthanna (IMT2022552) Sreyas Janamachi (IMT2022554) Ajitesh Singh (IMT2022559) Chinmay Krishna (IMT2022561) Chirag MV (IMT2022583)

April 24, 2025

All Code, Validation and Synthesis results are uploaded in this GitHub repository.

Abstract

Multiplication is the most resource-hungry operation in neural networks (NNs). Logarithmic multipliers (LMs) simplify multiplication to shift and addition operations and thus reduce the energy consumption. Since implementing the logarithm in a compact circuit often introduces approximation, some accuracy loss is inevitable in LMs. However, this inaccuracy accords with the inherent error tolerance of NNs and their associated applications. This article proposes an improved logarithmic multiplier (ILM) that, unlike existing designs, rounds both inputs to their nearest powers of two by using a proposed nearest-one detector (NOD) circuit. Considering that the output of the NOD uses a one-hot representation, some entries in the truth table of a conventional adder cannot occur. Hence, a compact adder is designed for the reduced truth table. The 88 ILM achieves up to 17.48 percent saving in power consumption compared to a recent LM in the literature while being almost 8 percent more accurate. Moreover, the evaluation of the ILM for two benchmark NN workloads shows up to 21.85 percent reduction in energy consumption compared to the NNs implemented with other LMs. Interestingly, using the ILM increases the classification accuracy of the considered NNs by up to 1.4 percent compared to a NN implementation that uses exact multipliers.

1 Introduction

The growing demand for faster, more power-efficient hardware has fueled interest in approximate computing, which sacrifices some numerical exactness to reduce power, area, and delay—especially in error-tolerant domains like multimedia, machine learning, and image filtering. Since binary multipliers are among the most power-hungry components in digital systems, numerous approximation strategies (truncation, simplified compressors, recursive schemes) have been explored to streamline their design. In this work, we present two novel 4×4 approximate multipliers—derived via controlled carry manipulation—and show how they can be assembled into 8×8 versions offering different accuracy–efficiency trade-offs. When synthesized in a 14 nm FinFET process, our best design reduces power and silicon area by roughly 46%, cuts delay by 21%, and still delivers competitive error performance compared to prior art. Finally, we demonstrate their practical value in image-processing and convolutional-neural-network applications.

2 Prerequisites

2.1 Genus Workflow

The process starts by parsing Verilog, standard cell libraries (.lib), and constraints (.sdc) for correctness, followed by setting up the technology library with functional, timing, power, and area data. Constraints are applied, and the design is converted into a technology-independent RTL netlist. Logic synthesis maps modules to cells, optimizes logic, and ensures power efficiency while meeting timing. Placement and wire estimation adjust cell strengths and estimate delays and power. Critical path and power analyses ensure constraints are met. A gate-level netlist and schematic are generated, followed by verification through equivalence checking, STA, and power analysis. The netlist is then handed off to a place-and-route tool for physical design, with key artifacts including Verilog, .lib, .sdc files, netlists, reports, and schematic.

2.2 Standard Cell Library

A Standard Cell Library (SCL) is a collection of pre-designed, characterized building blocks used in digital circuit design, comprising various cell types such as logic gates (AND, OR, NOT), flip-flops, multiplexers, and buffers. It includes characterization data such as timing (delay values for different input/output conditions), power (leakage, internal, and switching power consumption), and area (physical footprint on silicon). The library is optimized for specific technology nodes (e.g., 45nm, 14nm) and conditions like voltage and temperature, ensuring predictable and efficient behavior across different design scenarios.

2.3 How is Standard Library used by Genus

Genus uses the Standard Cell Library (SCL) to map high-level logic to standard cells through several steps. During RTL synthesis, it converts RTL (Verilog/VHDL) to a gate-level netlist using the library cells. Technology mapping then matches high-level logic to corresponding cells, optimizing for area, power, and timing. For timing analysis, Genus utilizes the cell timing data from the library, including delay tables to calculate signal propagation delays and ensures flip-flops and latches meet setup/hold timing requirements. For power analysis, Genus uses power tables that include internal, leakage, and switching power data for each cell, and dynamic calculations account for signal toggles and net capacitance. In area estimation, it sums the physical footprint of each cell to estimate total design area. For optimization, Genus selects cells with the best balance of timing, power, and area, and performs cell downsizing/upsizing to meet timing constraints while minimizing power and area. Lastly, Genus incorporates wireload models to estimate wire capacitance and, after layout, parasitic extraction combines these with cell data for accurate analysis.

2.4 Power Analysis in Genus

Synthesis to convert the design into a gate-level netlist. Power components are categorized into:

- Leakage power: Power consumed due to idle leakage current.
- Internal power: Power dissipated from short-circuit currents and capacitive charging.
- Switching power: Power consumed by charging and discharging load capacitance during signal transitions.

Activity factors are calculated based on transition probabilities, and power is computed using equations for leakage, internal, and switching power. Genus also uses advanced modeling for accurate power estimation under various conditions. The total power is derived from these components, and Genus generates a power report detailing consumption by components such as registers, logic, and the clock tree. Additionally, optimization features help reduce power consumption while maintaining design Performance.

2.5 Logarithmic Multipliers

Let;

$$Z = z_{n-1} z_{n-2} \dots z_1 z_0$$

be the n-bit binary representation of a positive integer N. Without loss of generality, let z_k, where $k < n$, be the most significant "1" in Z. Hence, N can be represented as

$$N = 2^k(1 + x), \quad 0 \leq x < 1 \tag{1}$$

Let A and B be the multiplicand and multiplier, respectively. Hence, we can write A and B as:-

$$A = 2^{k_1}(1 + x_1), \qquad B = 2^{k_2}(1 + x_2).$$

Accordingly, their base-2 logarithms are

$$\log_2 A = k_1 + \log_2(1 + x_1), \tag{2}$$
$$\log_2 B = k_2 + \log_2(1 + x_2), \tag{3}$$

and hence their product is

$$A \times B = 2^{k_1+k_2}\,(1 + x_1)\,(1 + x_2). \tag{4}$$

Depending on the computation process, different values for log2A and log2B and, consequently, different approximate products can be obtained.

3 Existing Multipliers used in the Paper

The paper compares the proposed Multiplier features with existing Multipliers. The below is the list of the existing Multipliers:-

3.1 Array Multiplier

Array multiplier is similar to how we perform multiplication with pen and paper that is finding a partial product and adding them together. It is a simple architecture for implementation.

Lets consider multiplication between 2 4-bits numbers as shown in the figure1:-

Here, A – Multiplicand and B – Multiplier.

p00 – a0*b0, p10 – a1*b0, p20 – a2*b0 and so on..

$$\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & \times & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & p_{30} & p_{20} & p_{10} & p_{00} \\
 & & & p_{31} & p_{21} & p_{11} & p_{01} & x \\
 & & p_{32} & p_{22} & p_{12} & p_{02} & x & x \\
 & p_{33} & p_{23} & p_{13} & p_{03} & x & x & x \\
\hline
z_7 & z_6 & z_5 & z_4 & z_3 & z_2 & z_1 & z_0
\end{array}$$

Figure 1: Array multiplication 4-bit example

The figure-2 shows the complete architecture of the array-multiplier. Note that the each element of the array-z is the final output of the multiplication. The first element of the array is the LSB and the last element is the MSB.

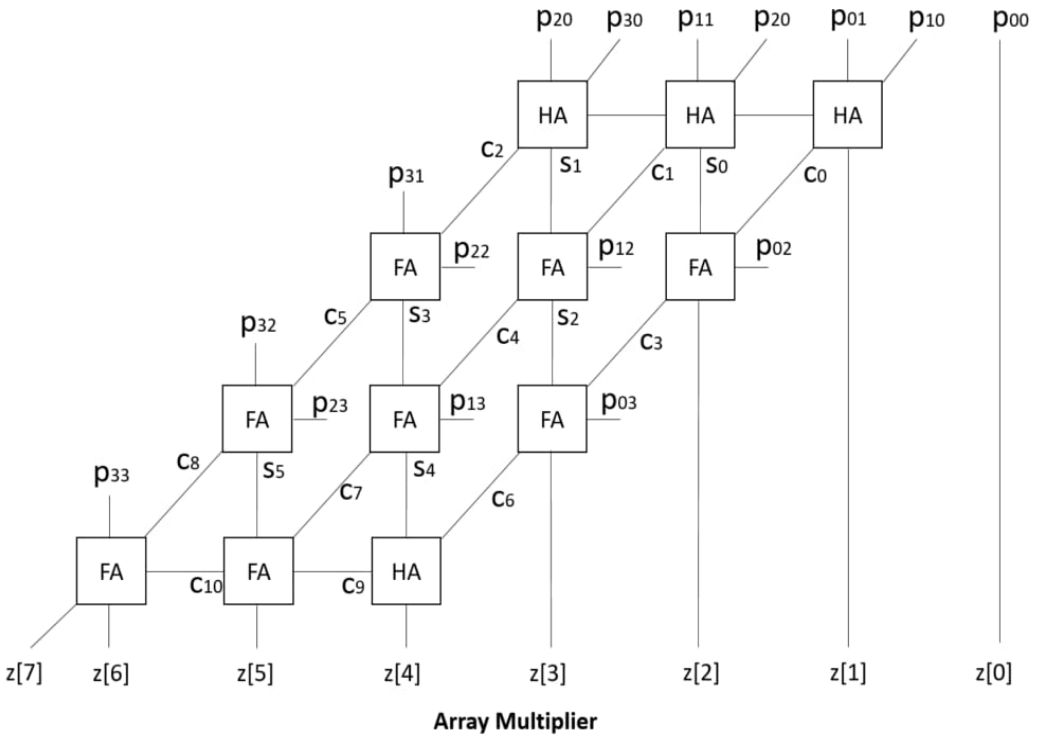


Figure 2: Array multiplier architecture

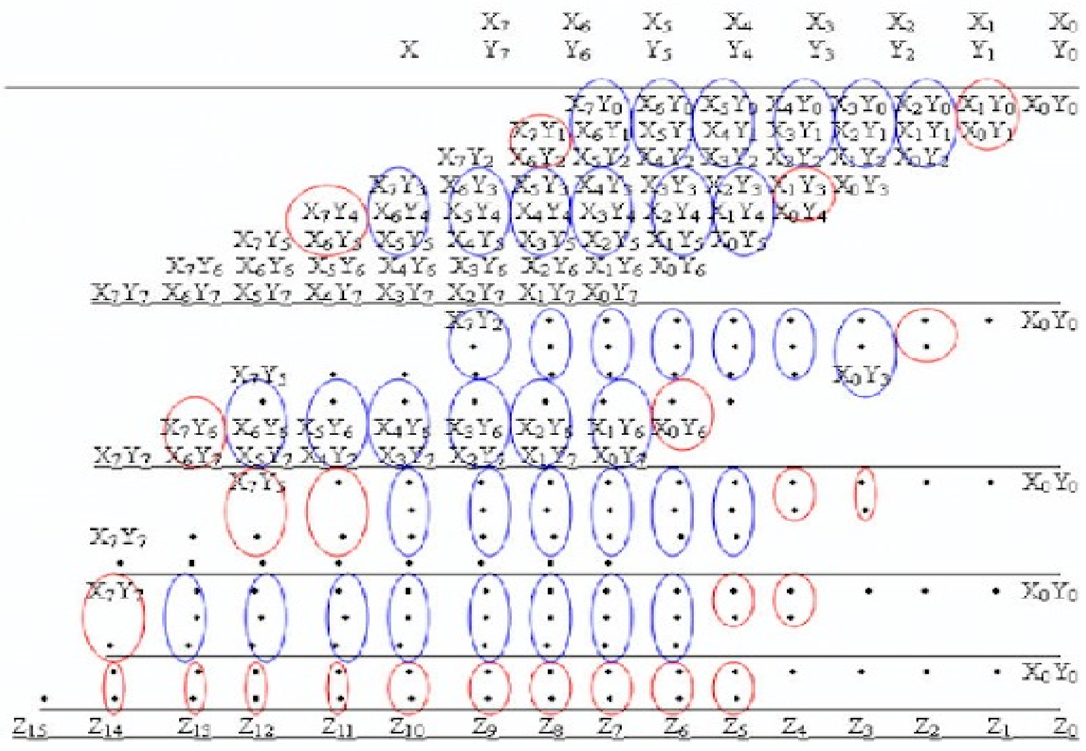
3.2 Wallace Multiplier

The Wallace Tree Multiplier (WTA) is a parallel multiplication technique that utilizes the Wallace Tree algorithm to efficiently perform integer multiplication. In this approach:

- Three wires of equal weight are fed into a full adder, producing one output wire of the same weight and another with a higher weight.

- If only two wires of the same weight remain, they are processed using a half adder.
- A single remaining wire is directly passed to the next layer.

The below figure-3 shows the architecture of a 8x8-bit Wallace Multiplier producing a 16-bit output.



X
X_7 Y_7
X_6 Y_6
X_5 Y_5
X_4 Y_4
X_3 Y_3
X_2 Y_2
X_1 Y_1
X_0 Y_0
X_0Y_0
Z_{15} Z_{14} Z_{13} Z_{12} Z_{11} Z_{10} Z_9 Z_8 Z_7 Z_6 Z_5 Z_4 Z_3 Z_2 Z_1 Z_0

Figure 3: Wallace multiplier architecture

3.3 Mitchell Multiplier

The Mitchell multiplier simplifies binary multiplication using logarithms. Instead of direct multiplication, it converts numbers to logarithmic form, adds them, and converts back using antilogarithms to get an approximate product.

This method provides a practical approach for hardware multiplication, though with some accuracy loss due to the approximations used. The Mitchell multiplier works on the below key steps:-

1. **Normalization**
   - Every positive binary number is written as $N = 2^k \cdot (1 + x)$
   - k is the position of the leftmost 1 (the *characteristic*)
   - x is the fractional part, where $0 \leq x < 1$

   Example: $13_{10} = 1101_2 = 2^3 \cdot (1 + 0.625)$

2. **Logarithm Approximation**
   - Exact: $\log_2 N = k + \log_2(1 + x)$
   - Mitchell's Approximation: $\log_2 N \approx k + x$

3. **Multiplication Process**

   For $A = 2^{k_1} \cdot (1 + x_1)$ and $B = 2^{k_2} \cdot (1 + x_2)$:
   - Add logarithms: $\log_2 P \approx (k_1 + k_2) + (x_1 + x_2)$
   - Take antilogarithm with two cases:
     - **Case 1:** $x_1 + x_2 < 1 \Rightarrow P' = 2^{k_1+k_2} \cdot (1 + x_1 + x_2)$
     - **Case 2:** $x_1 + x_2 \geq 1 \Rightarrow P' = 2^{k_1+k_2+1} \cdot (x_1 + x_2)$

The figure-4 shows the architecture of the Mitchell multiplier.

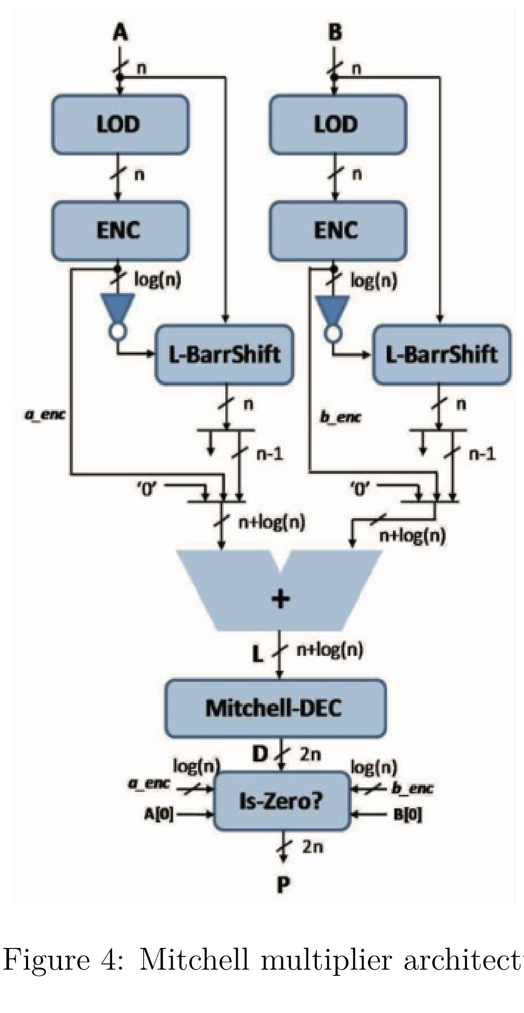


Figure 4: Mitchell multiplier architecture

3.4 ALM-SOA Mulitplier

The ALM-SOA multiplier is designed to overcome the complexity which is required in the Mitchell adder. It can also be said that the ALM-SOA is a modified version of the

Mitchell adder. The entire process of the ALM-SOA is similar to the Mitchell adder, but the way the addition is performed is a bit different. Rather than simply adding the log output of the BLC , we also add the Cin which is generated from the Estimator.

As shown in the figure-5, the ALM-SOA (Approximate Logarithmic Multiplier using Sum-of-Operands Algorithm) uses both the *Set-One Adder* as the mantissa adder and a truncated Binary Logarithmic Converter (TLBC). An n-bit SOA consists of two parts:

- an m-bit **inexact adder**, and
- an $(n-m)$-bit **exact adder**.

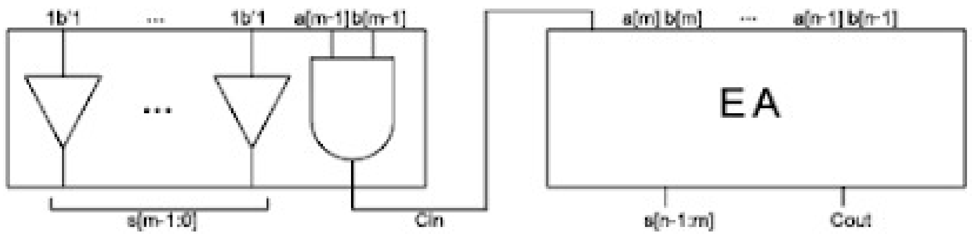


Fig. 5. SOA with m inexact bits (modified from [37]).

Figure 5: ALM-SOA multiplier architecture

The sum of the m-bit adder is forced to logic one. The logic expressions for the lower m-bit adder are as follows:

$$\text{Sum}[m-1:0] = 1 \tag{5}$$

$$\text{Cin} = a[m-1] \cdot b[m-1] \tag{6}$$

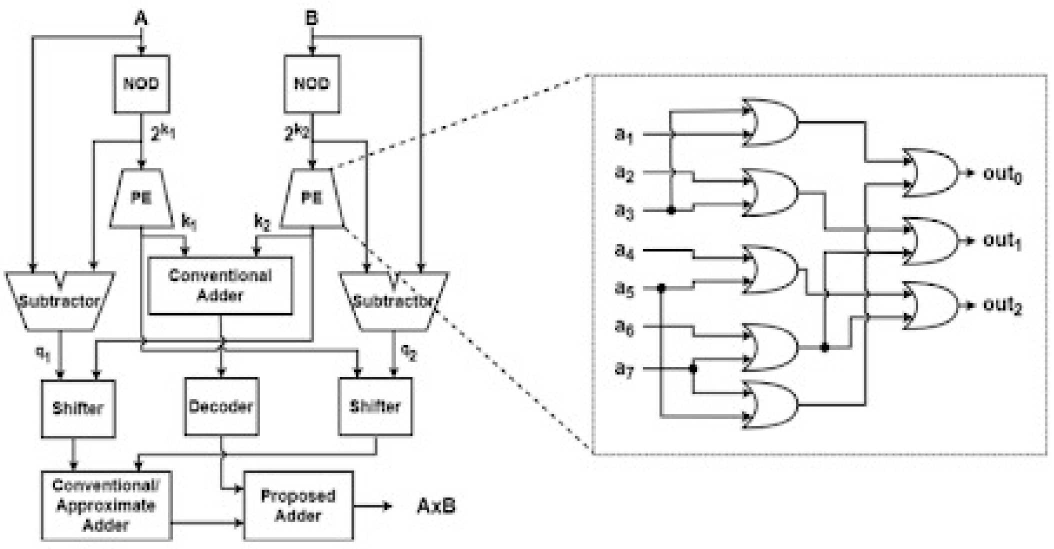
As the lower significant sum bits of SOA are set to one already, it is not necessary to perform an exact binary-logarithm conversion. Therefore, TBLC is used in the ALM-SOA to further reduce its complexity. The number of truncated bits is the same as the number of inexact bits in SOA which is = m.

4 The proposed ILM Multiplier in the paper

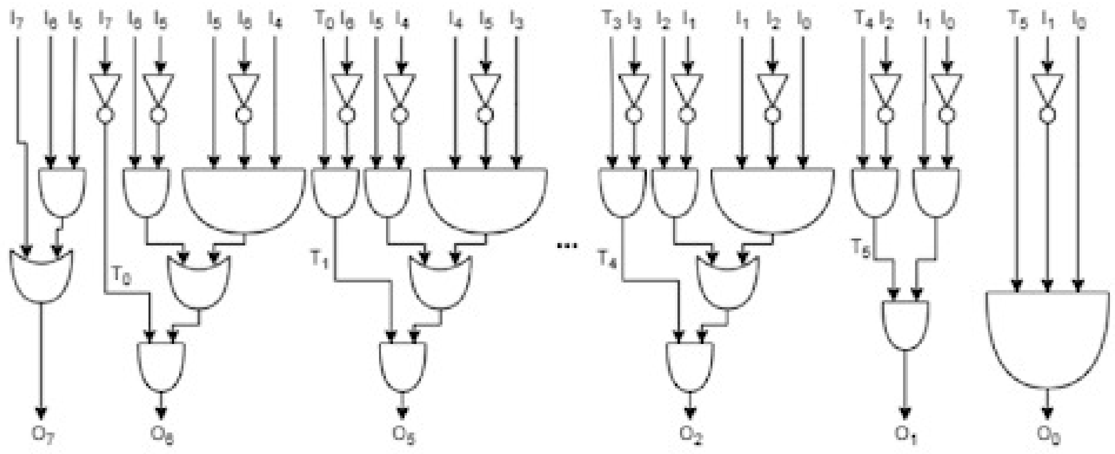
The **ILM circuit** (Integer Logarithmic Multiplier) implements multiplicative operations using logarithmic principles. Instead of directly multiplying two numbers A and B, it converts them to the logarithmic domain, adds the logarithms, and then converts back to obtain the product. This approach is based on the mathematical identity:

$$A \times B = 2^{\log_2 A + \log_2 B} \tag{7}$$

The key advantage of this method is that it can potentially reduce hardware complexity for multiplication operations. Now, the multiplication operation is reduced to simple *shift* and *add* operations.



(a) Block diagram of the ILM and the priority encoder [35].



(b) Proposed nearest-one detector (NOD) Circuit.

Figure 6: ILM multiplier architecture

The circuit in figure-6 is a hardware implementation of the algorithm described in figure-7, which outlines logarithmic multiplication. The correspondence between each hardware component and the algorithmic steps is described below.

Nearest-One Detector (NOD)

The NOD blocks in the circuit identify the position of the most significant '1' bit in the input numbers A and B. This corresponds to steps 3 and 6 in Algorithm 2:

$$2^{k_1} \leftarrow \mathrm{NOD}(A) \qquad 2^{k_2} \leftarrow \mathrm{NOD}(B)$$

The NOD circuit, shown in Image 1(b), uses a logic gate network to detect the highest '1' bit position. This effectively finds the largest power of 2 less than or equal to the input. If the input exceeds 128, the NOD downscales it to 128; it does not upscale beyond 128.

Priority Encoder (PE)

The PE blocks convert the power-of-two outputs 2^{k_1} and 2^{k_2} into their corresponding exponents k_1 and k_2. These correspond to steps 4 and 7:

$$k_1 \leftarrow \mathrm{PE}(2^{k_1}) \qquad k_2 \leftarrow \mathrm{PE}(2^{k_2})$$

This operation effectively computes the binary logarithm (base-2 exponent) of the identified highest powers of 2.

Subtractors

The subtractors compute the residual (fractional) parts of the inputs:

$$q_1 = A - 2^{k_1} \qquad q_2 = B - 2^{k_2}$$

These implement steps 5 and 8 of the algorithm and represent how much A and B exceed their respective nearest powers of 2.

Shifters

The shifter blocks carry out the following bit-shift operations from steps 9 and 10:

$$q_1 \cdot 2^{k_2} = q_1 \ll k_2 \qquad q_2 \cdot 2^{k_1} = q_2 \ll k_1$$

These left-shift operations prepare intermediate terms for the final multiplication approximation.

Decoder

The decoder performs:

$$2^{k_1+k_2} \leftarrow \mathrm{DEC}(k_1 + k_2)$$

This operation, corresponding to step 11, reconstructs the main term by converting the summed exponents back to the linear domain.

Adders

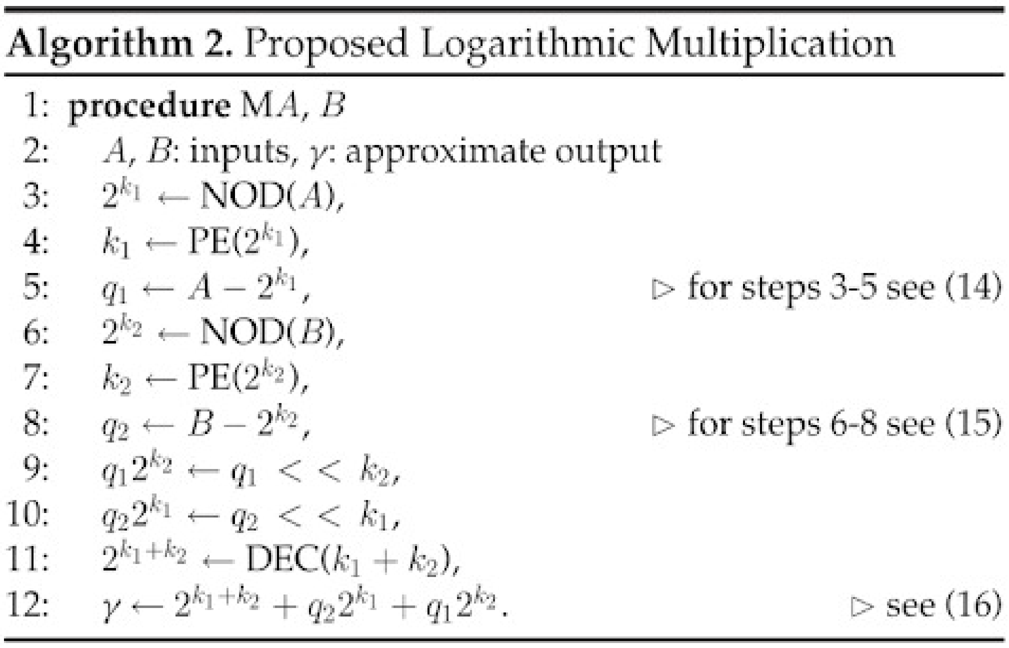
The final step (step 12) uses conventional or approximate adders to compute:

$$\gamma = 2^{k_1+k_2} + q_2 \cdot 2^{k_1} + q_1 \cdot 2^{k_2}$$

This results in an approximate product $\gamma \approx A \times B$. The calculation consists of:

- The main term: $2^{k_1+k_2}$
- First correction term: $q_2 \cdot 2^{k_1}$
- Second correction term: $q_1 \cdot 2^{k_2}$

Together, these form the final output of the logarithmic multiplication circuit.



Algorithm 2. Proposed Logarithmic Multiplication

```
1:  procedure M$A$, $B$
2:      $A$, $B$: inputs, $\gamma$: approximate output
3:      $2^{k_1} \leftarrow \mathrm{NOD}(A)$,
4:      $k_1 \leftarrow \mathrm{PE}(2^{k_1})$,
5:      $q_1 \leftarrow A - 2^{k_1}$,                    ▷ for steps 3-5 see (14)
6:      $2^{k_2} \leftarrow \mathrm{NOD}(B)$,
7:      $k_2 \leftarrow \mathrm{PE}(2^{k_2})$,
8:      $q_2 \leftarrow B - 2^{k_2}$,                    ▷ for steps 6-8 see (15)
9:      $q_1 2^{k_2} \leftarrow q_1 << k_2$,
10:     $q_2 2^{k_1} \leftarrow q_2 << k_1$,
11:     $2^{k_1+k_2} \leftarrow \mathrm{DEC}(k_1 + k_2)$,
12:     $\gamma \leftarrow 2^{k_1+k_2} + q_2 2^{k_1} + q_1 2^{k_2}$.          ▷ see (16)
```

Figure 7: ILM multiplier algorithm2

Connection to Algorithm 1

Algorithm 1 (shown in figure-8) outlines a piecewise approximation method for calculating $\log_2 N$, which is a fundamental part of the logarithmic multiplication process.

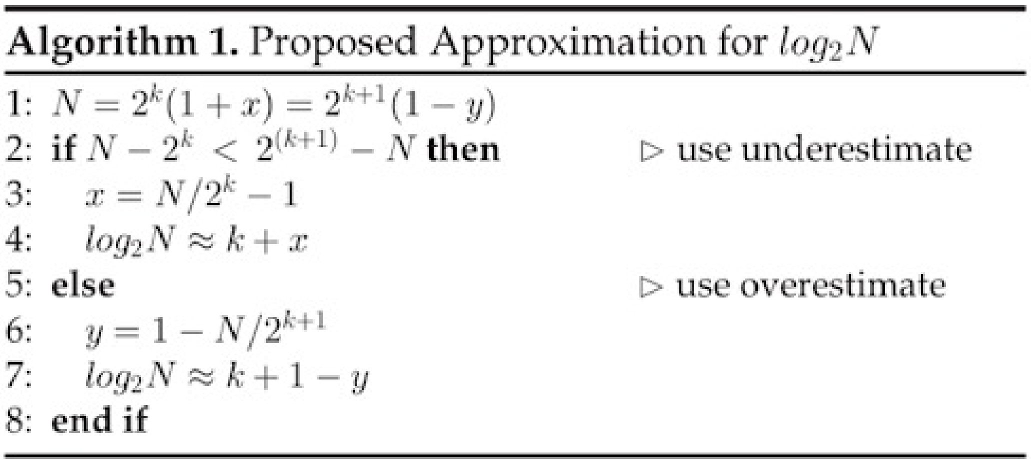
The approximation works as follows:

- If $N - 2^k < 2^{k+1} - N$, then the algorithm uses the *underestimate*:

$$\log_2 N \approx k + x, \quad \text{where } x = \frac{N}{2^k} - 1$$

- Otherwise, it uses the *overestimate*:

$$\log_2 N \approx k + 1 - y, \quad \text{where } y = 1 - \frac{N}{2^{k+1}}$$



Algorithm 1. Proposed Approximation for $log_2 N$

```
1: $N = 2^k(1+x) = 2^{k+1}(1-y)$
2: if $N - 2^k < 2^{(k+1)} - N$ then        ▷ use underestimate
3:     $x = N/2^k - 1$
4:     $log_2 N \approx k + x$
5: else                                      ▷ use overestimate
6:     $y = 1 - N/2^{k+1}$
7:     $log_2 N \approx k + 1 - y$
8: end if
```

Figure 8: ILM multiplier algorithm1

The overall operation of the ILM (Integer Logarithmic Multiplier) circuit can be summarized as follows:

1. **Decompose** inputs A and B into their logarithmic components using the Nearest-One Detector (NOD) and Priority Encoder (PE).
2. **Calculate** the error terms q_1 and q_2, which represent the fractional parts beyond the nearest power of 2.
3. **Perform logarithmic addition:** compute $k_1 + k_2$, the sum of the logarithmic exponents.
4. **Convert back to the linear domain** using decoding and add the correction terms.
5. **Output** the final product approximation $\gamma \approx A \times B$.

The key insight is that by operating in the *logarithmic domain*, multiplication is reduced to addition, which is typically more efficient to implement in hardware. The correction terms $q_1 \cdot 2^{k_2}$ and $q_2 \cdot 2^{k_1}$ help refine the final result, improving the accuracy of the approximation.

5 Multiplier Performance in Approximating logN

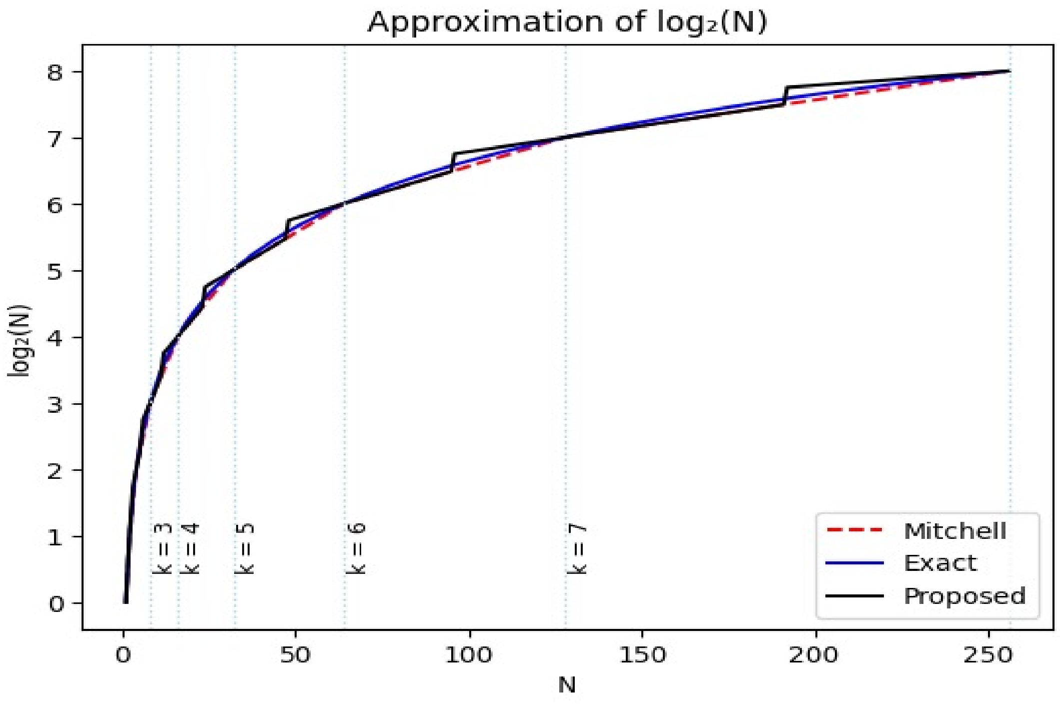


Figure 9: Approximation of log2N

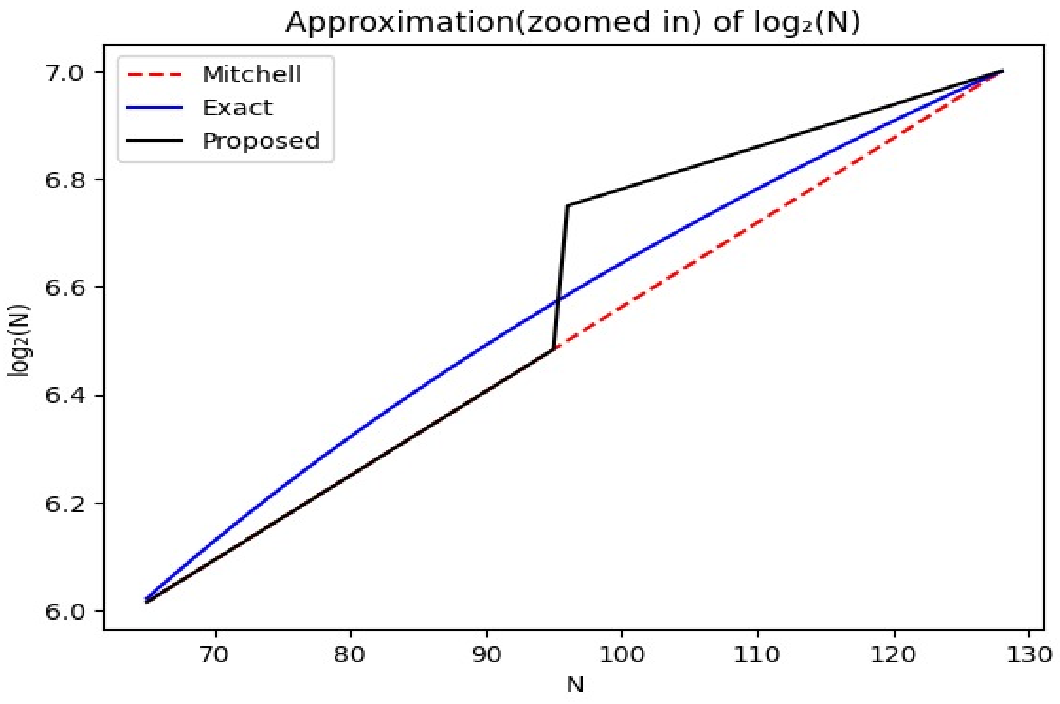


Figure 10: Zoomed version of figure 9

6 Performance evaluation of Logarithmic Multipliers

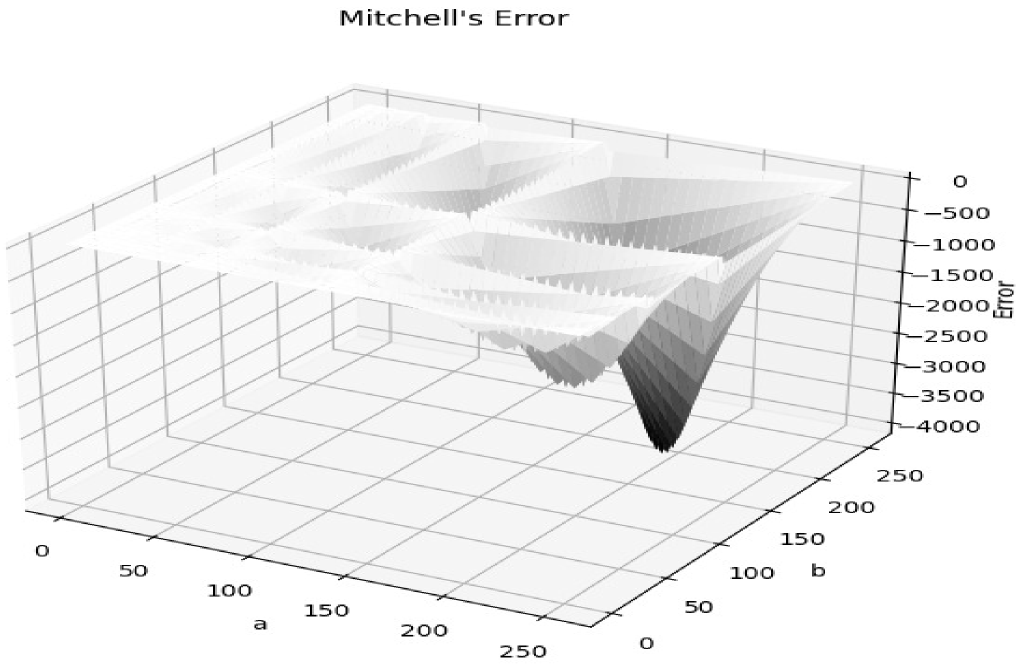


Figure 11: (a) Mitchell's multiplier, $error_{maximum} = -4096$.

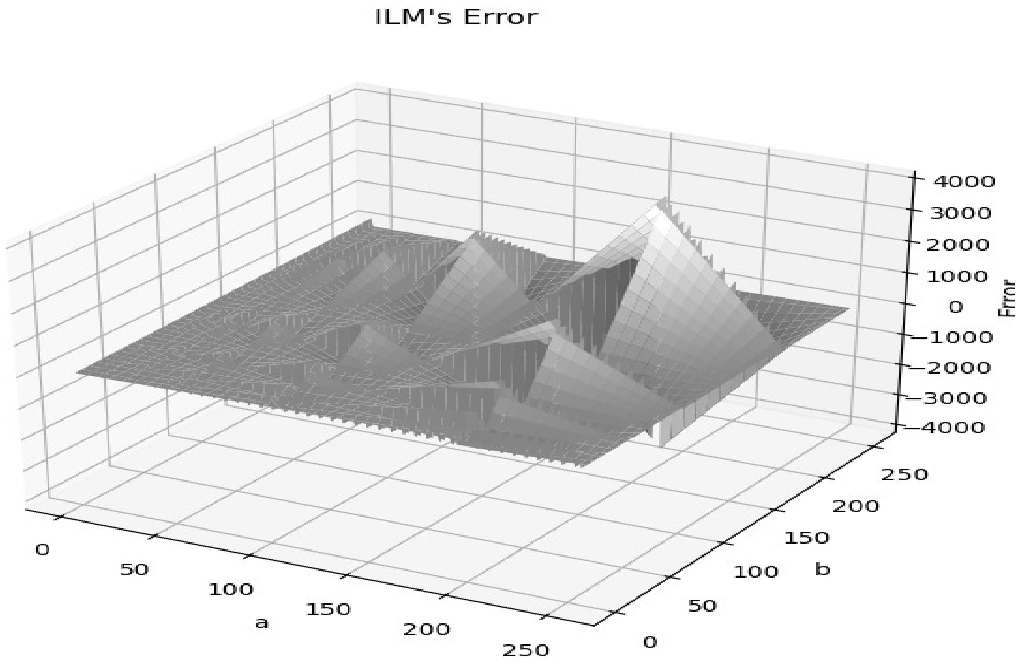


Figure 12: (a) ILM's multiplier, $error_{maximum} = -4096$.

Design	**Power (μW)**	**Delay (ns)**	**Area (μm^2)**	**PDP (fJ)**
Array	18.399	3.055	499.374	56.2089
Wallace	19.590	2.878	489.060	56.3800
Mitchell	12.960	3.570	431.138	46.2672
ALM-SOA-5	10.983	3.340	413.492	36.6832
ILM-0	9.982	3.640	492.315	36.3345
ILM-5	9.122	3.610	413.491	32.9304

Table 1: Hardware Metrics of the Logarithmic Multipliers

Distribution	**Multiplier Type**	**|AE|**	**MRED**	**NMED**
Uniform	Mitchell	607.15	0.0379	0.0093
	ALM-SOA-3	607.52	0.0377	0.0093
	ALM-SOA-5	599.94	0.0373	0.0092
	ILM-0	0.2662	0.0288	0.0065
	ILM-5	4.3083	0.0351	0.0066
	ILM-9	23.1226	0.1771	0.0353
Normal	Mitchell	96.04	0.0358	0.0015
	ALM-SOA-3	173.73	0.0360	0.0027
	ALM-SOA-5	171.91	0.0358	0.0026
	ILM-0	3.3567	0.0394	0.0012
	ILM-5	9.6572	0.1152	0.0018
	ILM-9	21.7361	0.0334	0.0049

Table 2: Error Metrics of the LMs for General Input Distributions

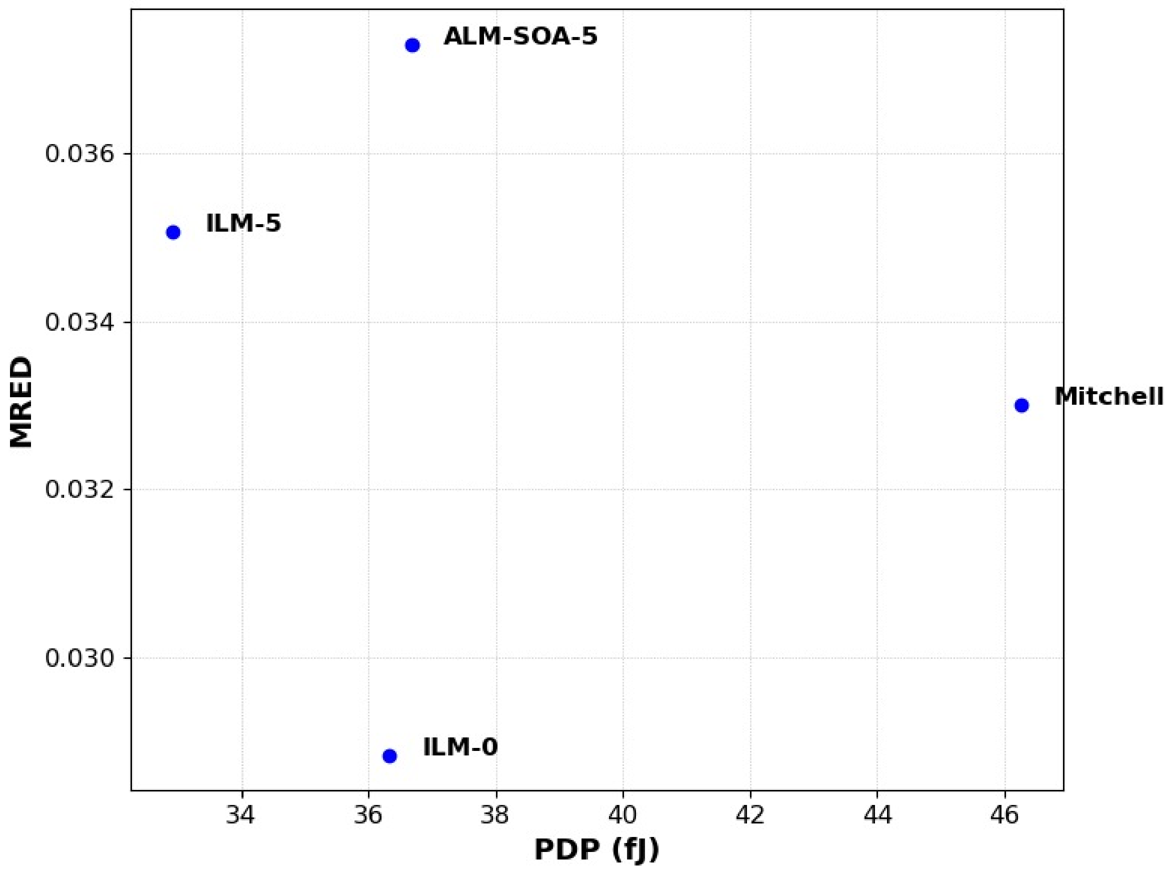


Figure 13: MRED versus PDP trade-off: the proposed ILM can achieve a higher accuracy (ILM-0) and a better trade-off in accuracy and hardware efficiency (ILM-5), compared to other LMs.

7 Neural Network Workloads

The paper used MLP and AlexNet to classify the MNIST dataset. Below shown are the figures figure-13 and figure-14 of trained weights for MLP and AlexNet respectively.

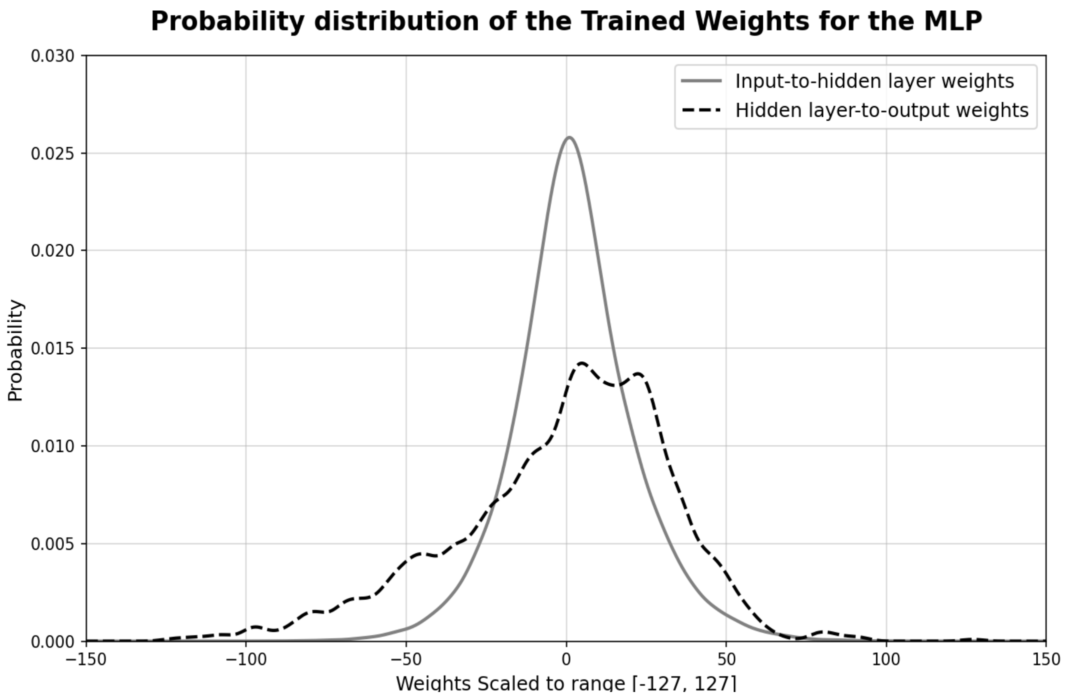


Figure 14: Probability distribution of the trained weights for the (784-128-10) MLP, mapped into the range [-127, 127].

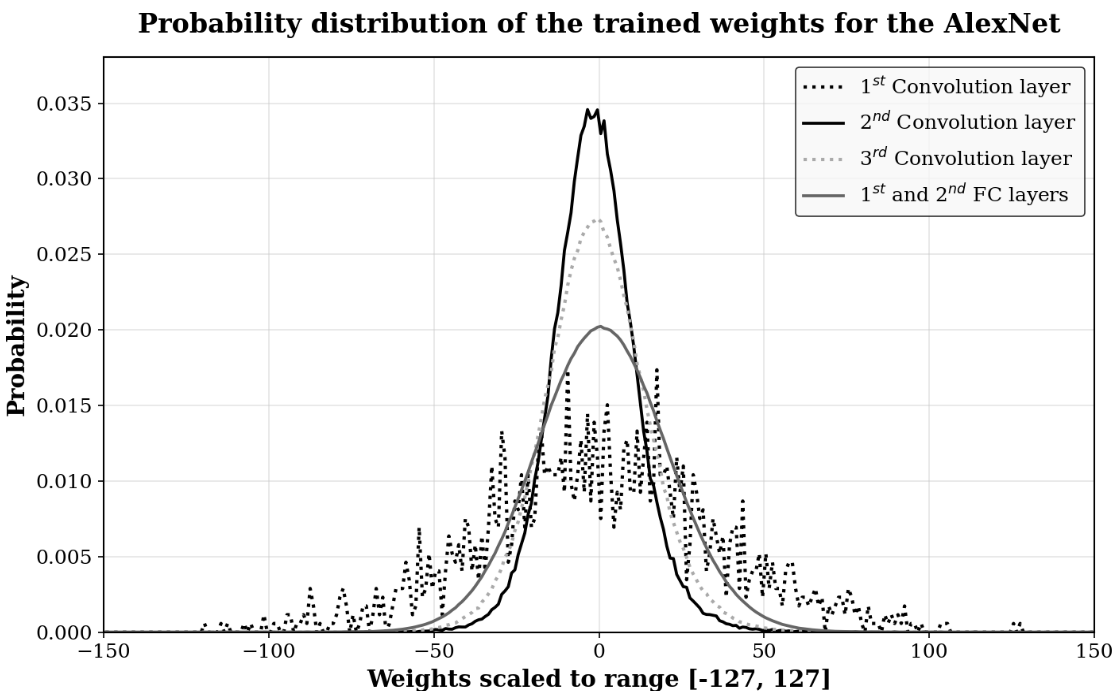


Figure 15: Probability distribution of the trained weights for Alexnet, mapped into the range [-127, 127].

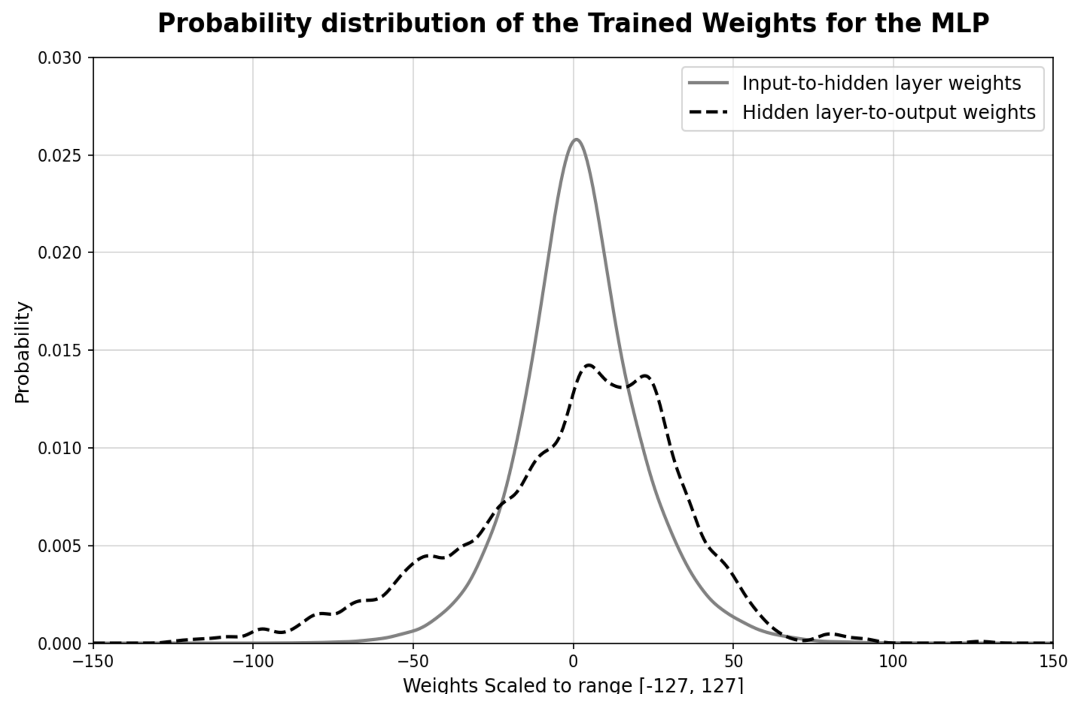


Figure 16: Probability distribution of the trained weights for the (784-128-10) MLP, mapped into the range [-127, 127].

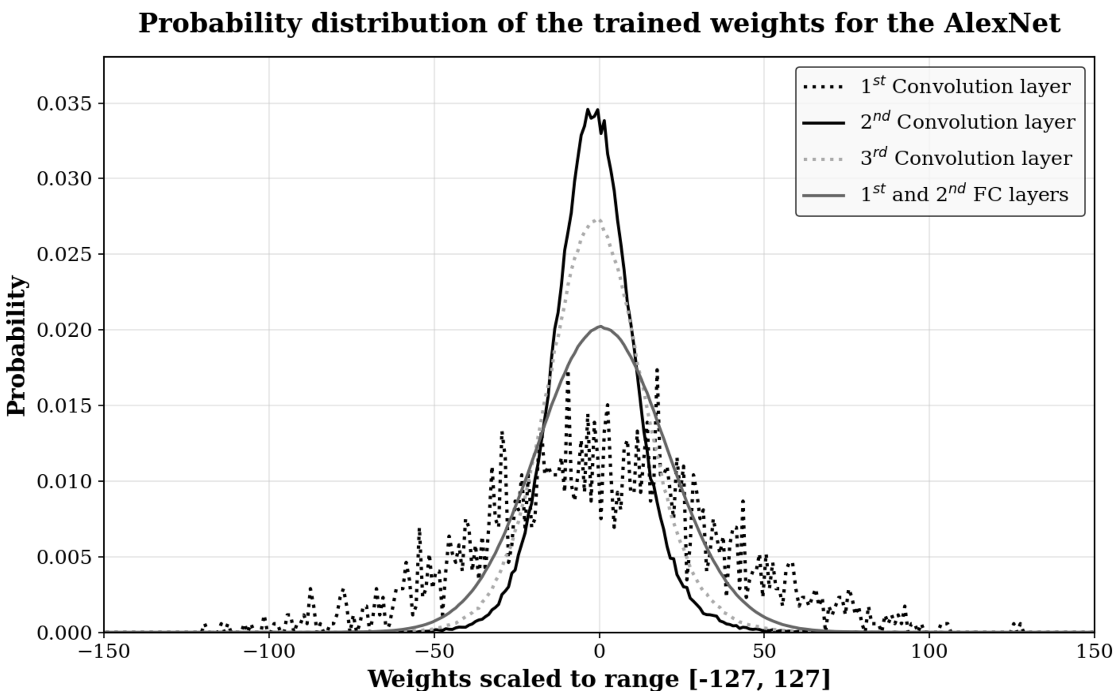


Figure 17: Probability distribution of the trained weights for Alexnet, mapped into the range [-127, 127].

Multiplier	**|AE|**	**MRED**	**NMED**
ILM0	0.1177	0.0870	0.2564
ALM3	0.7526	0.4803	0.3355
Mitchell	7.1602	3.6273	0.3619

Table 3: Error Metrics of the LMs for the NN Workloads

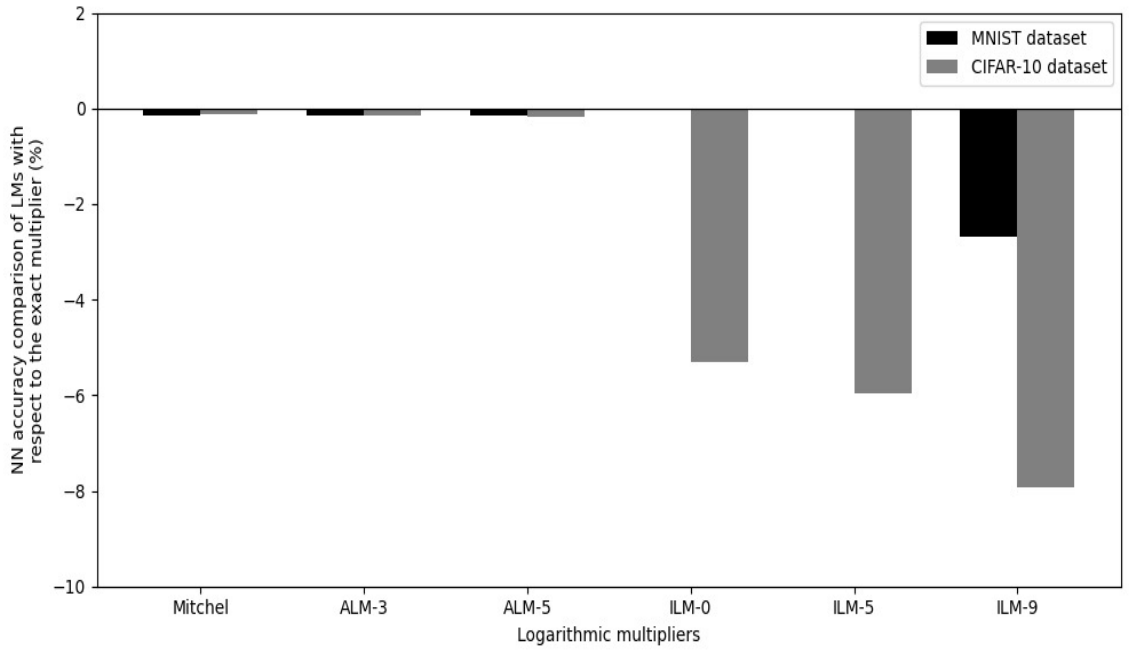


Figure 18: Comparison of the top-1 classification accuracy of the MNIST and CIFAR-10 datasets with LMs (decrease or increase in accuracy from using accurate multipliers)