BitLogic: Training Framework for Gradient-Based FPGA-Native Neural Networks

Anonymous authors
Paper under double-blind review

Abstract

The energy and latency costs of deep neural network inference are increasingly driven by deployment rather than training, motivating hardware-specialized alternatives to arithmetic-heavy models. Field-Programmable Gate Arrays (FPGAs) provide an attractive substrate for such specialization, yet existing FPGA-based neural approaches are fragmented and difficult to compare. We present BitLogic, a fully gradient-based, end-to-end trainable framework for FPGA-native neural networks built around Lookup Table (LUT) computation. BitLogic replaces multiply–accumulate operations with differentiable LUT nodes that map directly to FPGA primitives, enabling native binary computation, sparse connectivity, and efficient hardware realization. The framework offers a modular functional API supporting diverse architectures, along with learned encoders, hardware-aware heads, and multiple boundary-consistent LUT relaxations. An automated Register Transfer Level (RTL) export pipeline translates trained PyTorch models into synthesizable HDL, ensuring equivalence between software and hardware inference. Experiments across standard vision benchmarks and heterogeneous hardware platforms demonstrate competitive accuracy and substantial gains in FPGA efficiency, including 72.3% test accuracy on CIFAR-10 achieved with fewer than 0.3 M logic gates, while attaining sub-20 ns single-sample inference using only LUT resources. [1]

1 Introduction

Deep Neural Networks (DNNs) have transformed applications ranging from autonomous driving to large-scale language modeling, yet their growing size and complexity increasingly strain energy and hardware resources. In production environments, *inference* now dominates machine-learning energy consumption: Yang et al. (2024) report that ML workloads accounted for 10–15% of Google's total energy use from 2019 to 2021, with inference responsible for roughly 60%, while Meta observed a 10:20:70 split across experimentation, training, and inference. These trends underscore that the energy footprint of AI is increasingly driven by deployment rather than model development.

While traditional Graphics Processing Units (GPUs) provide high compute throughput, they often lack efficiency for large-scale inference or power-constrained edge deployment. FPGAs, by contrast, offers programmable specialization, enabling custom datapaths, optimized memory access, and lower power consumption. Differentiable, LUT-based neural architectures exploit these strengths by replacing arithmetic-heavy operations with compact lookup-table computations, paving the way for high-performance, energy-efficient inference on FPGAs.

A major source of inference inefficiency on general-purpose accelerators is the reliance on floating-point or high-precision integer arithmetic. Digital hardware, however, natively operates on bit-level logic, where Boolean functions can be realized directly as combinational circuits or lookup tables. LUT-based neural networks align with this hardware's "native language," replacing arithmetic-heavy multiply–accumulate operations with simple table lookups and bitwise logic. Importantly, LUT-based representations are strictly more general: given sufficient capacity, they can approximate or even exactly implement floating-point or

[1] The code will be made public upon acceptance.

integer operations when beneficial, rather than being limited to a fixed arithmetic abstraction. On FPGAs, this enables efficient mapping to on-chip resources, reduced data movement, and lower energy consumption compared to floating-point or integer-based designs.

A growing body of recent work explores LUT-centric and hardware-native neural architectures, demonstrating promising gains in latency, energy efficiency, and resource utilization Guo (2025). However, a meaningful comparison of these approaches remains challenging. Existing studies evaluate on heterogeneous hardware platforms, adopt differing assumptions about pipelining and clocking, and are often opaque about how latency and throughput are measured. Moreover, several works investigate orthogonal design dimensions that could, in principle, be combined, yet are typically studied in isolation.

The goal of this work is to consolidate these efforts by providing a unified framework that summarizes and systematizes existing approaches, enables fair and reproducible comparison under consistent assumptions, and facilitates exploration of interactions between their core ideas. By establishing conceptual and empirical links between prior methods, our framework supports rapid development of new LUT-based architectures and aims to make future research in this space more accessible, comparable, and cumulative.

In this work, we present a *fully gradient-based*, end-to-end trainable framework for hardware-native neural network design. The main contributions of this paper are as follows:

- **Modular and extensible architecture.** We introduce a highly modular framework that enables the construction of arbitrary models across diverse problem domains, with built-in support for automatic RTL design generation. The architecture is designed for extensibility, allowing new modules and components to be integrated with minimal effort.
- **Novel architectural and training components.** We propose several new building blocks, including a GroupedDSP head, transposed convolution blocks, attention mechanisms, residual connections, and probabilistic nodes. In addition, we introduce gradient stabilization techniques, in-layer bit-flip operations, and novel regularization and initialization strategies to improve training stability and robustness.
- **Comprehensive empirical evaluation.** We systematically evaluate the performance of individual components within a unified experimental setting, enabling fair and controlled comparisons. Furthermore, we assess the framework across multiple hardware platforms (GPU, CPU, and FPGA), reporting inference throughput, latency, and hardware resource utilization. On FPGA, the framework achieves inference times under 20 ns while maintaining the following test accuracies: CIFAR-10: 72.3%, CIFAR-100: 23.4%, Fashion-MNIST: 93.8%, and MNIST: 99.1%.

2 Related Work

2.1 Mapping Conventional Models to FPGA Resources

Early approaches for FPGA-based neural network inference focused on adapting conventional models via quantization to fixed-point arithmetic and mapping computations to existing hardware resources. For instance, Farabet et al. (2009) trained a CNN on a CPU, then compiled it into instructions for an FPGA-based processor that executed convolutions using hardwired DSP multipliers and a soft CPU for control. To further reduce computational complexity, binarization methods like BinaryConnect (Courbariaux et al., 2016b;a) constrained weights to binary values during forward/backward propagation (while maintaining full-precision updates), effectively replacing multiplications with bitwise operations. More recent work by Gerlinghoff et al. (2024) exploited weight redundancy in quantized networks by encoding weights directly into LUTs, implementing Multiply-Accumulate (MAC) operations through table lookups and clustering algorithms to minimize resource usage.

2.2 LUT-Centric and Differentiable Architectures

LUT-centric and differentiable architectures depart from mapping conventional neural networks onto hardware and instead design the network directly in terms of lookup tables. Optimizing LUT contents and

connectivity is an exponentially hard combinatorial problem; to make it tractable, these methods introduce continuous relaxations that render LUT parameters differentiable. Gradient-based optimization is then used to learn both functions and/or connections, after which the relaxed representations are collapsed back to exact Boolean functions or finite truth tables, yielding hardware-ready, fully discrete implementations. This paradigm enables end-to-end learning while preserving a tight correspondence between the trained model and its final FPGA realization.

The shift toward FPGA-native computation began with LUTNet (Wang et al., 2019), which replaced binary exclusive NOR (XNOR) operations in Binarized Neural Networks (BNNs) with arbitrary K-input LUTs to increase logic density and enable aggressive pruning. LogicNets (Umuroglu et al., 2020) extended this by co-designing sparse, quantized networks where neurons with limited fan-in map directly to truth tables, producing deeply pipelined circuits for extreme-throughput applications.

Building on learned LUTs, several architectures have emerged. PolyLUT embeds multivariate polynomial functions and applies hardware-aware structured pruning for ultra-low-latency inference (Andronic et al., 2025). NeuraLUT implements each logical neuron as a small Multilayer Perceptron (MLP) with skip connections, compiled into LUT structures (Andronic & Constantinides, 2025). WARP-LUTs use Walsh-based probabilistic relaxations to better match discrete inference (Gerlach et al., 2025), while Differentiable Weightless Neural Networks learn both address mapping and reduction over symbolic inputs, generalizing classical weightless and LUT-based models (Bacellar et al., 2025). For a comprehensive overview, Guo (2025) survey LUT-based FPGA DNNs, covering training schemes from gradient-based to combinatorial and hybrid approaches.

In parallel, a distinct paradigm has emerged with *differentiable logic gate networks*, which use **binary logic gates** as fundamental neurons instead of LUTs. The foundation was established by Petersen et al. (2022), who introduced Deep Differentiable Logic Gate Networks (DDLGNs) by relaxing two-input Boolean functions to enable gradient-based training of sparse, gate-level networks. This work was later extended to convolutional architectures by Petersen et al. (2024), who achieved 86.29% accuracy on CIFAR-10 using logic gate tree convolutions and OR pooling, while reducing the gate count by 29×. Subsequent research has focused on improving scalability and training. Rüttgers et al. (2025) introduced a parameter reparameterization that reduces complexity from $O(2^{2^n})$ to $O(2^n)$, forming the theoretical basis for our probabilistic node design. Yousefi et al. (2025) employed Gumbel noise to reduce discretization gap and enhance model robustness. Furthermore, Bührer et al. (2025) demonstrated the applicability of this paradigm to sequence modeling with recurrent architectures. In contrast to LUT-based methods, this line of work treats binary gates as native computational units, resulting in extreme sparsity and direct compatibility with digital circuit synthesis.

2.3 Hybrid Models and Emerging Research Directions

LUT-based and logic operatored based components are increasingly integrated into larger architectures and non-neural models. Nag et al. (2025) introduce LL-ViT, which replaces channel-mixing MLPs in vision transformers with LUT-neuron operators alongside an FPGA accelerator, targeting edge deployment with fewer multiplications and reduced latency and energy consumption. TreeLUTquantizes gradient-boosted decision trees into fully unrolled, pipelined LUT-only designs, achieving competitive accuracy and favorable area–delay product compared to both LUT-based neural networks and prior GBDT accelerators (Khataei & Bazargan, 2025). Complementary work on interconnect learning develops scalable wiring parametrizations whose size does not grow with input width, highlighting routing, sparsity, and learning rules as key open dimensions for LUT-based and Boolean networks (Kresse et al., 2025; Fojcik et al., 2025).

3 Method

3.1 LUT Nodes and a Differentiable Training Relaxation

The main building block of our networks is a *LUT node*. A LUT node implements an n-input Boolean function using a truth table with 2^n entries:

$$\mathrm{y} = g(\mathbf{x}; \boldsymbol{\theta}), \quad \mathbf{x} \in \{0,1\}^n,\ \mathrm{y} \in \{0,1\},\ \boldsymbol{\theta} \in \{0,1\}^{2^n}. \tag{1}$$

This maps directly to FPGA LUTs, which makes deployment efficient. Compared to standard neural network neurons, LUT nodes have a fixed, small fan-in n (sparse connectivity) and operate on binary values (discrete computation). This lets us consider accuracy and hardware cost already during training.

Because g is discrete, we use a differentiable surrogate function during training:

$$\mathrm{y} = f(\mathbf{x}; \boldsymbol{\theta}), \quad \mathbf{x} \in [0,1]^n,\ \mathrm{y} \in [0,1],\ \boldsymbol{\theta} \in \mathbb{R}^d. \tag{2}$$

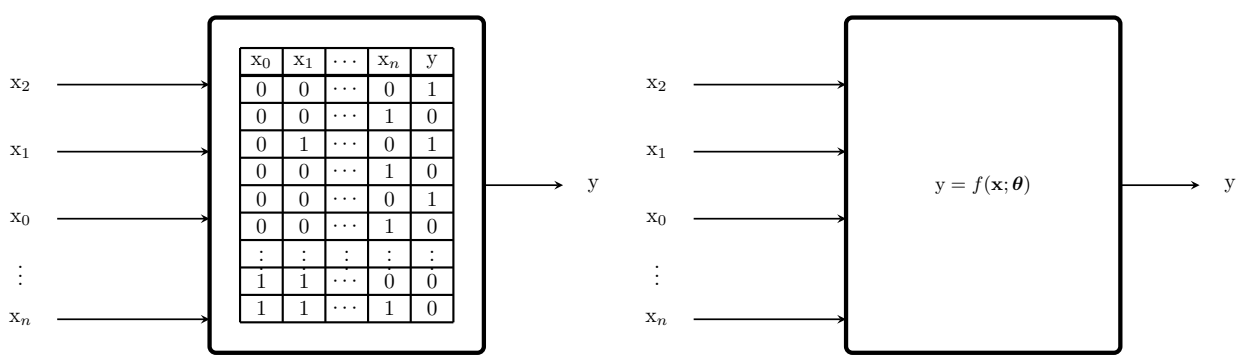
In the forward pass we compute f; in the backward pass we use either exact or surrogate gradients (see Section A for the implemented options). After training, we discretize inputs and outputs again to recover a Boolean LUT.

Example: Probabilistic relaxation. A simple relaxation interprets each input $x_j \in [0,1]$ as the probability of a Bernoulli variable being 1. The LUT output is then the expected value over all binary input patterns Rüttgers et al. (2025):

$$f(\mathbf{x}; \boldsymbol{\theta}) = \sum_{\boldsymbol{a} \in \{0,1\}^n} \sigma(\theta_{\iota(\boldsymbol{a})}) \prod_{j=1}^{n} x_j^{a_j} (1 - x_j)^{1-a_j}, \tag{3}$$

where $\boldsymbol{\theta} \in \mathbb{R}^{2^n}$ are trainable logits, $\sigma(\cdot)$ is the hard sigmoid function, and $\iota(\boldsymbol{a}) = \sum_{j=1}^{n} a_j 2^{j-1}$ maps a bit pattern to its truth table index. For binary inputs, exactly one term in the sum remains, which is equivalent to a normal LUT lookup. For inference we discretize with a 0.5 threshold:

$$g(\mathbf{x}; \boldsymbol{\theta}) = \mathbf{1}[f(\mathbf{x}; \boldsymbol{\theta}) \geq 0.5].$$



(a) Discrete LUT as a truth table with 2^n entries.

(b) Differentiable relaxation $\mathrm{y} = f(\mathbf{x}; \boldsymbol{\theta})$.

Figure 1: LUT representations. **Left:** discrete lookup table mapping $\mathbf{x} \in \{0,1\}^n$ to $\mathrm{y} \in \{0,1\}$. **Right:** continuous relaxation with $\mathbf{x} \in [0,1]^n$ and $\mathrm{y} \in [0,1]$ for gradient-based training.

3.2 Layers and Blocks

Layers. To build larger networks, we group LUT nodes into *layers*. A layer $\mathcal{L}$ contains w nodes, each computing

$$y_j = f\big((x_{\mathcal{M}_j(1)}, \ldots, x_{\mathcal{M}_j(n)}); \boldsymbol{\theta}^{(j)}\big), \quad j = 1, \ldots, w, \tag{4}$$

where the *connection mapping* $\mathcal{M}_j : \{1, \ldots, n\} \to \{1, \ldots, w_{\text{in}}\}$ selects which n inputs feed into node j. This mapping is the main design choice for a layer. It can be set randomly, follow a structure (e.g., local neighborhoods), or be learned during training. For FPGA deployment, the final mapping must be sparse and fixed. Training may start with richer connectivity as long as it can be discretized to a valid sparse mapping in the end. All implemented layer variants are listed in Section B.

Blocks. *Blocks* apply a layer repeatedly to different parts of the input while reusing the same parameters (parameter sharing):

$$\mathcal{B}(\boldsymbol{X};\boldsymbol{\Theta}) = \big[\mathcal{L}(\boldsymbol{X}_1;\boldsymbol{\Theta}),\ldots,\mathcal{L}(\boldsymbol{X}_m;\boldsymbol{\Theta})\big], \tag{5}$$

where $\boldsymbol{X}_1,\ldots,\boldsymbol{X}_m$ are partitions of the input (e.g., sliding windows). Sharing parameters reduces FPGA resource usage and matches common hardware-friendly patterns (see Section 3.5).

Example: Convolutional block. For images, we extract small patches (windows) and apply the same LUT layer to every patch, similar to a Convolutional Neural Network (CNN). For example, for a 32×32 image, a 3×3 convolutional block slides a 3×3 window across the image and produces feature maps. Additional blocks (e.g., residual and attention-style) are described in Section C.

3.3 Encoders and Heads

Encoders. LUT nodes operate on binary inputs, but real-world data is often continuous or integer-valued. We therefore use an *encoder* to convert each input dimension into a binary representation:

$$\mathcal{E} : \mathbb{R}^d \to \{0,1\}^{d\cdot b}, \tag{6}$$

where b is the number of encoding bits per input dimension. Encoders are fitted on training data and then applied deterministically at inference time. This keeps the LUT-based core computation purely binary while still supporting different input modalities.

Example: Thermometer encoding. Thermometer encoding compares an input value to b thresholds $\boldsymbol{t} = (t_1,\ldots,t_b)$:

$$\mathcal{E}(x;\boldsymbol{t}) = \big(\mathbf{1}_{x>t_1},\ldots,\mathbf{1}_{x>t_b}\big). \tag{7}$$

Thresholds can be uniform, based on Gaussian quantiles, or chosen from empirical data quantiles. See Section D for all encoders.

Heads. The final layer outputs a binary feature vector, but tasks like multi-class classification or regression need real-valued outputs. A *head* aggregates the binary vector into the desired output:

$$\mathcal{H} : \{0,1\}^w \to \mathbb{R}^c, \tag{8}$$

where w is the final width and c is the number of output classes (or targets). A simple head is *group-sum* (popcount): it splits the w bits into c groups and counts the number of active bits per group. We also tested weighted variants with learnable coefficients. See Section E for details.

Hardware mapping. Encoders and heads can be implemented efficiently on FPGAs, for example using Digital Signal Processing (DSP) blocks for arithmetic and Block Random Access Memories (BRAMs) for storing thresholds (see Section 3.5).

3.4 Models and Functional API

BitLogic provides a functional, configuration-driven API to build complete models by composing encoders, layers/blocks, and heads. The key idea is that model structure is defined in a configuration, while component implementations live in a registry.

Configuration-driven composition. A model is specified as an ordered sequence of registered components:

$$\text{Model} = \text{Encoders}(\mathcal{E}_1,\ldots,\mathcal{E}_m) \;\to\; \text{Blocks/Layers}(\mathcal{L}_1,\ldots,\mathcal{L}_k) \;\to\; \text{Heads}(\mathcal{H}_1,\ldots,\mathcal{H}_p).$$

All architectural choices (widths, kernels, encoding type, connectivity, node type, etc.) are set in the configuration. This makes it easy to swap components and compare variants without rewriting code.

With this approach, different architectures (feedforward, CNN-style, autoencoder, multi-input/output) can be expressed by changing the configuration only. Figure 2 shows an MNIST CNN example: a thermometer

encoder, two TopK-sparse convolutional blocks, flattening, a TopK-sparse lookup layer, and a GroupSum head.

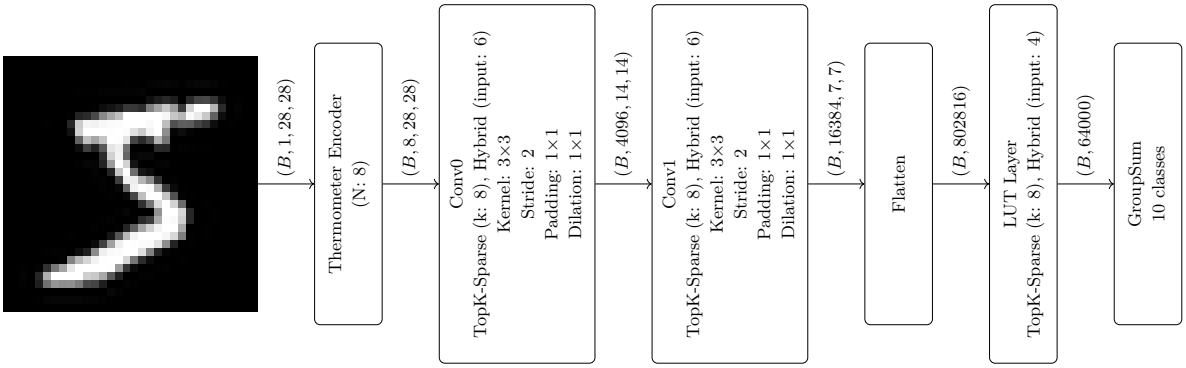


Figure 2: Declarative MNIST CNN: Thermometer encoder (N: 8) feeds two TopK-Sparse convolutional layers (k: 8, Hybrid nodes with input dimension 6) that reduce spatial dimensions via stride-2 convolutions. Features are flattened and processed by a TopK-Sparse lookup layer (input: 4, k: 8), then aggregated by a GroupSum head into 10 class predictions. Tensor shapes annotated on edges.

3.5 FPGA Export

BitLogic includes an export pipeline that converts a trained PyTorch model into synthesizable RTL for FPGA deployment. The model is exported *hierarchically*: each component exports itself, and the full design is built by composing these exported modules.

Hierarchical RTL extraction. Each component provides a `to_hdl()` method that generates Hardware Description Language (HDL) modules. These modules are composed recursively and keep the same hierarchy as the PyTorch model. Learned parameters (e.g., LUT contents, thresholds) are embedded directly into the generated logic, so no external configuration is needed. A helper script generates a full Vivado project structure (see Figure 3).

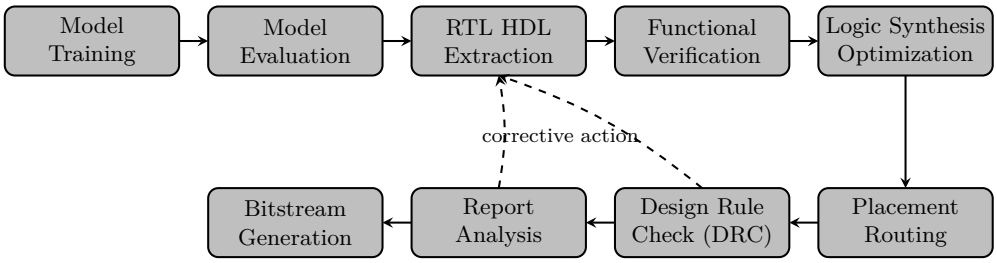


Figure 3: End-to-end FPGA deployment methodology. The pipeline transforms a trained PyTorch model into deployable hardware through automated RTL generation, synthesis and implementation, verification (including Design Rule Check (DRC)), and bitstream generation.

Combinational optimization. The exported compute path is fully combinational. Vivado can optimize the logic by removing redundancies, factoring common subexpressions, and packing logic efficiently into LUTs. In practice, synthesis can sometimes reduce the number of required LUTs compared to the unoptimized trained structure. After synthesis and implementation, timing and resource reports allow checking feasibility for target clock rates and FPGA capacity.

Sequential optimization. The export step also supports resource/latency trade-offs via configuration options: *pipelining* (insert registers for higher throughput), *replication* (duplicate modules for parallelism),

and *iterative decomposition* (process over multiple cycles to save area). These options are chosen at export time to match different hardware targets and throughput/latency requirements.

Verification and deployment. The deployment flow is automated using Tool Command Language (TCL) scripts: **Stage 1:** run testbenches for functional verification before synthesis; **Stage 2:** run Vivado synthesis, placement/routing, DRC, and collect timing/resource reports; **Stage 3:** generate the final bitstream if all checks pass.

4 Experiments

4.1 Benchmark Accuracy

We evaluate *BitLogic* on four standard image-classification benchmarks: MNIST LeCun et al. (2010), Fashion-MNIST Xiao et al. (2017), CIFAR-10, and CIFAR-100 Krizhevsky (2009). Test accuracies are reported in Table 1. Full training protocols and hyperparameters are provided in Section H.

Comparing logic-based neural models by parameter count can be misleading because architectures differ in their primitive operations (e.g., node input sizes, internal representations, and parameterization). To enable a more hardware-relevant comparison, we therefore report *equivalent binary gate counts* as a proxy for computational size. For BitLogic, the reported gate counts include only the computational logic layers and exclude the encoder and decoder/classification head. We conservatively upper-bound an n-input LUT by $2^n - 1$ binary gates; this bound ignores possible reductions from synthesis and optimization, which we analyze separately in Section 4.3. For prior work, gate counts are taken from the original publications when available. Since reporting conventions vary (e.g., whether auxiliary modules are included and whether numbers are pre- or post-optimization), absolute comparisons should be interpreted with care.

Table 1: Benchmark comparison of BitLogic against logic-based neural network baselines. Our methods are highlighted. Gate counts denote equivalent binary gates and, for BitLogic, cover only the computational logic layers.

Dataset	Model	Method	Accuracy (%)	Gate count
MNIST	FFN	DiffLogic Net (small) Petersen et al. (2022)	97.69	48 K
		DiffLogic Net (largest) Petersen et al. (2022)	98.47	384 K
		LILogicNet-S Fojcik et al. (2025)	97.96	4 K
		LILogicNet-M Fojcik et al. (2025)	98.45	8 K
		LILogicNet-L Fojcik et al. (2025)	98.95	32 K
		Ours	**99.15**	384 K
	CNN	LogicTreeNet-S Petersen et al. (2024)	98.46	147 K
		LogicTreeNet-M Petersen et al. (2024)	99.23	566 K
		LogicTreeNet-L Petersen et al. (2024)	**99.35**	1.27 M
		Ours	95.72	253.4 K
Fashion-MNIST	FFN	DWN (n=2) Bacellar et al. (2025)	89.12	—
		DWN (n=6) Bacellar et al. (2025)	89.01	—
		Ours	**93.81**	384 K
	CNN	Ours	**81.07**	253.4 K
CIFAR-10	FFN	DiffLogic Net-S Petersen et al. (2022)	51.27	48 K
		DiffLogic Net-M Petersen et al. (2022)	57.39	512 K
		DiffLogic Net-L Petersen et al. (2022)	60.78	1.28 M
		LILogicNet-S Fojcik et al. (2025)	55.11	8 K
		LILogicNet-M Fojcik et al. (2025)	57.66	64 K
		LILogicNet-L Fojcik et al. (2025)	60.98	256 K
		Ours	**72.36**	384 K
	CNN	LogicTreeNet-S Petersen et al. (2024)	60.38	400 K
		LogicTreeNet-M Petersen et al. (2024)	71.01	3.08 M
		LogicTreeNet-G Petersen et al. (2024)	**86.29**	61.0 M
		Ours	50.53	253.4 K
CIFAR-100	FFN	Ours	**23.43**	384 K
	CNN	Ours	**10.18**	253.4 K

— Gate count not reported in the original work.

4.2 Component-wise Analysis

To identify which design choices contribute most to performance in the feedforward BitLogic architecture, we run a controlled ablation study over five component families: (A) input encoding, (B) layer type, (C) node type, (D) node input dimensionality, and (E) classification head. Results on Fashion-MNIST are summarized in Table 2.

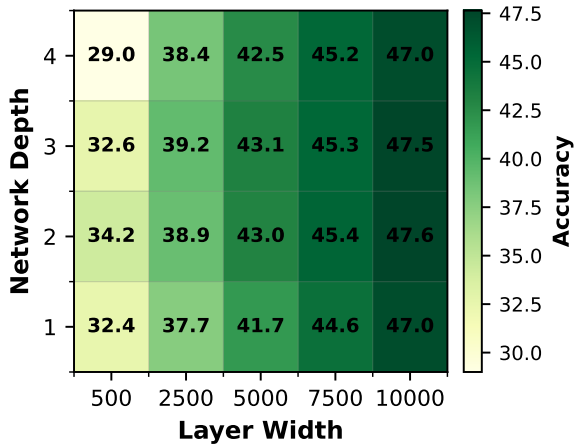
Each configuration changes exactly one component relative to a fixed base model (two-layer FFN, 4000 units per layer, 20 epochs, batch size 128). This design separates the influence of each individual component and prevents interfering interactions between them. Since every component family brings in its own set of hyperparameters, we avoid extensive retuning of each variant. Instead, we evaluate different options under the same training budget and within a shared architectural framework.

Two patterns stand out. First, accuracy is strongly affected by the *node input dimensionality* (Table 2 block D): in this setting, increasing the number of inputs per node consistently improves performance. Second, increasing *width* is more effective than increasing *depth*. This is illustrated in Figure 4a for CIFAR-10: the best result occurs with two layers and higher width, while deeper networks show diminishing returns. In practice, this suggests allocating capacity to width, especially because several node types (e.g., hybrid, DWN, probabilistic) have memory costs that scale as 2^n (or worse) with node fan-in.

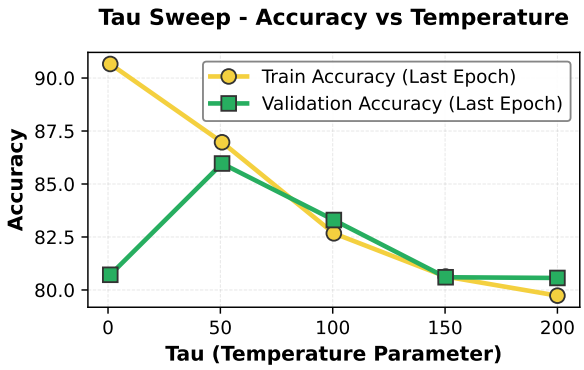
Table 2: Component-wise ablation on Fashion-MNIST. Base model: two-layer FFN with 4000 units per layer, trained for 20 epochs (batch size 128). Test accuracy is mean ± std over two runs. Each block (A–E) modifies one component. DiffLogic node memory grows as 2^{2^n}; we therefore only evaluate the case with two inputs per node.

Cfg	**Encoder**	**Head**	**Layer type**	**Fan-in**	**Node type**	**Acc. (%)**
Base	distributive	groupsum	topk_sparse	4	probabilistic	83.4±0.1
A1	binary					80.3±0.1
A2	distributive					83.5±0.1
A3	gaussian					81.8±0.1
A4	gray					81.0±0.0
A5	logarithmic					**83.8±0.2**
A6	onehot					83.2±0.2
A7	sign					79.7±0.1
A8	thermometer					82.4±0.0
B1			learnable			70.0±1.5
B2			random			77.3±0.0
B3			topk_sparse			**83.5±0.1**
C1				2	difflogic	78.8±0.1
C2				4	dwn	73.0±0.4
C3				4	fourier	77.8±0.7
C4				4	hybrid	**84.0±0.1**
C5				4	linear	70.8±1.5
C6				4	neurallut	81.7±1.0
C7				4	polylut	70.2±1.9
C8				4	probabilistic	83.5±0.1
C9				4	warp	69.2±2.5
D1				2		79.3±0.1
D2				3		82.0±0.1
D3				4		83.5±0.1
D4				5		84.2±0.2
D5				6		**84.9±0.0**
E1		grouped_dsp				80.9±0.1
E2		groupsum				**83.5±0.1**

Models with discretized nodes are prone to overfitting. We find that the temperature parameter τ in the GroupSum head provides a simple and effective way to prevent this. As shown in Figure 4b, the optimal τ depends on the dataset and architecture. With Fashion-MNIST (5 epochs), validation accuracy peaks at $\tau = 50$ (85.97%), while $\tau = 1.0$ causes severe overfitting (90.66% train vs 80.72% validation). This suggests treating τ as a tunable hyperparameter that controls the tradeoff between training performance and generalization in discrete networks.



(a) CIFAR-10 test accuracy as a function of network depth (rows) and width (columns). Accuracy peaks at two layers and 4000 nodes per layer; increasing depth yields diminishing or negative returns.



(b) FashionMNIST training and validation accuracy as a function of the temperature parameter τ in the Group-Sum head. Validation accuracy peaks at $\tau = 50$ (85.97%), while training accuracy monotonically decreases with increasing τ, suggesting that moderate temperature values prevent overfitting.

4.3 Multi-Platform Hardware Efficiency

We profile inference for the best Fashion-MNIST (see Section H): CPU (Intel Xeon Silver 4208, single-threaded), GPU (NVIDIA RTX 2080 Ti), and FPGA (Xilinx Zynq-7020). We report latency in microseconds for CPU/GPU and in nanoseconds for FPGA, together with a component-level breakdown that highlights bottlenecks.

Table 3: Hardware profiling on CPU, GPU, and FPGA. Latencies measured end-to-end in PyTorch (CPU/GPU) or post-synthesis timing (FPGA). FPGA energy per sample: $E = P \times t = 179.2\ \mathrm{W} \times 18.63\ \mathrm{ns} = 3.34\ \mathrm{nJ}$ (total on-chip power at full switching activity from post-synthesis estimation; includes 40.3 W slice logic, 31.0 W signals, and 106.8 W I/O power is overestimated due to unconstrained synthesis). The layer instances are optimized into encoder during synthesis, showing as a single instance in Vivado reports. FPGA model uses 4K nodes per layer rather than 128K due to Vivado synthesis limitations—the tool enforces a 1,000,000-element maximum for constant arrays, while 128K nodes require 8,192,000 elements for the LUT mapping table, exceeding this limit.

Platform	Component	Latency	Throughput	Energy/Sample	Resources
GPU	Encoder	2.55 μs			153.12 MB
	Layers	22.59 μs			1562.50 MB
	Head	1.69 μs			0.29 MB
	Total	26.80 μs	**37,307 FPS**	**130 μJ**	1715.92 **MB**
CPU	Encoder	131.33 μs			153.12 MB
	Layers	1,164.78 μs			1562.50 MB
	Head	87.08 μs			0.29 MB
	Total	1,382.54 μs	**723 FPS**	**5.63 mJ**	1715.92 **MB**
FPGA†	Total	**18.63 ns**	**53.7 M FPS**	**3.34 nJ**	**11,234 LUTs**

†Component-wise breakdown unavailable for FPGA due to aggressive logic optimization during synthesis—modules are fused together, making individual latency measurements impossible. Total latencies and resource utilization reported for end-to-end implementation.

5 Conclusion

This work evaluates BitLogic in two representative settings: a feedforward architecture and a convolutional architecture. We focused most of our tuning effort on the feedforward model, since it provides a clean baseline for analyzing node types, fan-in, and hardware cost. In this regime, BitLogic reaches state-of-the-art accuracy among logic-based neural approaches on the evaluated benchmarks.

In contrast, our convolutional variant underperforms the feedforward model in the current experiments. We believe this result is primarily due to limited tuning rather than a fundamental limitation of convolutional BitLogic. In particular, longer training, wider channel configurations, and more careful optimization of the layers could plausibly close part of the gap.

From a hardware perspective, convolution introduces a clear trade-off. Using local receptive fields can reduce the number of unique gates compared to a dense feedforward layer at similar representational capacity. However, when convolution is executed in an iterative (sliding-window) manner, latency scales with the number of patches. Fully parallelizing over patches can avoid this latency increase, but it substantially increases resource utilization. Which option is preferable depends on the target device and the deployment constraints (e.g., strict latency budgets versus strict LUT budgets).

5.1 Future Work

BitLogic provides an initial framework for FPGA-native neural networks, but several directions remain open.

Training stability at depth. While training is stable for shallow models, performance degrades as depth increases. Future work should study improved gradient flow and optimization in deep BitLogic networks, for example via normalization, better initialization, or residual-style wiring adapted to logic nodes.

Broader architectural coverage and tasks. The current implementation and evaluation focus mainly on image classification. Extending BitLogic to additional tasks such as segmentation, reconstruction, and sequence modeling will likely require stronger support for recurrent and encoder–decoder style components, as well as careful choices of output representations and heads.

More complete HDL export and sequential logic support. Automatic HDL export is robust for feedforward, fully combinational graphs, but support for more complex components is not yet complete. Improving the handling of sequential logic (e.g., explicit pipelining, stateful modules, and streaming interfaces) would enable cleaner hardware–accuracy trade-offs and make the framework easier to deploy in practical systems.

Hardware-aware training and co-design. A natural next step is to incorporate hardware constraints directly into training, e.g., by optimizing for LUT utilization, timing budgets, and target clock frequency. Combining BitLogic with hardware-aware Neural Architecture Search (NAS) could automatically discover architectures that best match a specific FPGA, narrowing the gap between model capacity and implementable efficiency.

References

Marta Andronic and George A. Constantinides. Neuralut-assemble: Hardware-aware assembling of sub-neural networks for efficient lut inference, 2025. URL https://arxiv.org/abs/2504.00592.

Marta Andronic, Jiawen Li, and George A. Constantinides. Polylut: Ultra-low latency polynomial inference with hardware-aware structured pruning, 2025. URL https://arxiv.org/abs/2501.08043.

Alan T. L. Bacellar, Zachary Susskind, Mauricio Breternitz Jr., Eugene John, Lizy K. John, Priscila M. V. Lima, and Felipe M. G. França. Differentiable weightless neural networks, 2025. URL https://arxiv.org/abs/2410.11112.

Simon Bührer, Andreas Plesner, Till Aczel, and Roger Wattenhofer. Recurrent deep differentiable logic gate networks, 2025. URL https://arxiv.org/abs/2508.06097.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations, 2016a. URL https://arxiv.org/abs/1511.00363.

Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1, 2016b. URL https://arxiv.org/abs/1602.02830.

Clement Farabet, Cyril Poulet, Jefferson Y. Han, and Yann LeCun. Cnp: An fpga-based processor for convolutional networks. In *2009 International Conference on Field Programmable Logic and Applications*, pp. 32–37, 2009. doi: 10.1109/FPL.2009.5272559.

Katarzyna Fojcik, Renaldas Zioma, and Jogundas Armaitis. Lilogic net: Compact logic gate networks with learnable connectivity for efficient hardware deployment, 2025. URL https://arxiv.org/abs/2511.12340.

Lino Gerlach, Liv Våge, Thore Gerlach, and Elliott Kauffman. Warp-luts - walsh-assisted relaxation for probabilistic look up tables, 2025. URL https://arxiv.org/abs/2510.15655.

Daniel Gerlinghoff, Benjamin Chen Ming Choong, Rick Siow Mong Goh, Weng-Fai Wong, and Tao Luo. Table-lookup mac: Scalable processing of quantised neural networks in fpga soft logic. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '24, pp. 235–245. ACM, April 2024. doi: 10.1145/3626202.3637576. URL http://dx.doi.org/10.1145/3626202.3637576.

Zeyu Guo. A survey on lut-based deep neural networks implemented in fpgas, 2025. URL https://arxiv.org/abs/2506.07367.

Alireza Khataei and Kia Bazargan. Treelut: An efficient alternative to deep neural networks for inference acceleration using gradient boosted decision trees. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '25, pp. 14–24. ACM, February 2025. doi: 10.1145/3706628.3708877. URL http://dx.doi.org/10.1145/3706628.3708877.

Fabian Kresse, Emily Yu, and Christoph H. Lampert. Scalable interconnect learning in boolean networks, 2025. URL https://arxiv.org/abs/2507.02585.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

Shashank Nag, Alan T. L. Bacellar, Zachary Susskind, Anshul Jha, Logan Liberty, Aishwarya Sivakumar, Eugene B. John, Krishnan Kailas, Priscila M. V. Lima, Neeraja J. Yadwadkar, Felipe M. G. Franca, and Lizy K. John. Ll-vit: Edge deployable vision transformers with look up table neurons, 2025. URL https://arxiv.org/abs/2511.00812.

Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. Deep differentiable logic gate networks, 2022. URL https://arxiv.org/abs/2210.08277.

Felix Petersen, Hilde Kuehne, Christian Borgelt, Julian Welzel, and Stefano Ermon. Convolutional differentiable logic gate networks, 2024. URL https://arxiv.org/abs/2411.04732.

Lukas Rüttgers, Till Aczel, Andreas Plesner, and Roger Wattenhofer. Light differentiable logic gate networks, 2025. URL https://arxiv.org/abs/2510.03250.

Yaman Umuroglu, Yash Akhauri, Nicholas J. Fraser, and Michaela Blott. Logicnets: Co-designed neural networks and circuits for extreme-throughput applications, 2020. URL https://arxiv.org/abs/2004.03021.

Erwei Wang, James J. Davis, Peter Y. K. Cheung, and George A. Constantinides. Lutnet: Rethinking inference in fpga soft logic, 2019. URL https://arxiv.org/abs/1904.00938.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017. URL http://arxiv.org/abs/1708.07747.

Zeyu Yang, Karel Adamek, and Wesley Armour. Double-exponential increases in inference energy: The cost of the race for accuracy, 2024. URL https://arxiv.org/abs/2412.09731.

Shakir Yousefi, Andreas Plesner, Till Aczel, and Roger Wattenhofer. Mind the gap: Removing the discretization gap in differentiable logic gate networks, 2025. URL https://arxiv.org/abs/2506.07500.

List of Acronyms

DNN	Deep Neural Network	1
BNN	Binarized Neural Network	3
MAC	Multiply-Accumulate	2
DSP	Digital Signal Processing	5
BRAM	Block Random Access Memory	5
LUT	Lookup Table	1
MLP	Multilayer Perceptron	3
FPGA	Field-Programmable Gate Array	1
GPU	Graphics Processing Unit	1
CNN	Convolutional Neural Network	5
XNOR	exclusive NOR	3
TCL	Tool Command Language	7
HDL	Hardware Description Language	6
RTL	Register Transfer Level	1
DRC	Design Rule Check	6
DRC	Design Rule Check	6
NAS	Neural Architecture Search	11

A Implemented Differentiable LUT relaxations

A relaxation f is *boundary-consistent* if discretization maps $D_{\text{in}} : \mathcal{X}^n \to \{0,1\}^n$ and $D_{\text{out}} : \mathcal{Y} \to \{0,1\}$ exist with retrieval map $R : \Theta \to \{0,1\}^{2^n}$ such that for all $\mathbf{x} \in \{0,1\}^n$:

$$D_{\text{out}}\big(f(D_{\text{in}}(\mathbf{x}); \boldsymbol{\theta})\big) \;=\; g(\mathbf{x}; R(\boldsymbol{\theta})). \tag{9}$$

This ensures faithful transfer of learned relaxations to discrete hardware.

More compact parameterizations (e.g., $O(n)$ linear parameters or $O(n^d)$ degree-d polynomials) reduce VC dimension and induce implicit regularization, while full truth tables achieve expressiveness at cost of $O(2^n)$ parameters. Boundary-consistency holds across all parameterization choices.

A.1 Implemented Relaxations

Table 4 summarizes the differentiable LUT relaxations. All operate on continuous inputs $\mathbf{x} \in [0,1]^n$ during training, discretize to Boolean LUTs post-training, and maintain boundary-consistency. $H(\mathbf{x})$ denotes hard thresholding at 0.5; $\iota(\cdot)$ maps binary patterns to truth table indices; S_s is the subset corresponding to index s.

Method	**Parameters $\boldsymbol{\theta}$**	**Forward $f(\mathbf{x}; \boldsymbol{\theta})$**	**Gradient Method**
Linear Umuroglu et al. (2020)	$\mathbb{R}^{n+1}$	$\sigma(\theta_0 + \sum_i \theta_i x_i)$	Autograd
DWN Bacellar et al. (2025)	$\mathbb{R}^{2^n}$	$\sigma(\boldsymbol{\theta})_{\iota(H(\mathbf{x}))}$	Extended Finite Difference
Probabilistic Rüttgers et al. (2025)	$\mathbb{R}^{2^n}$	$\sum_{\boldsymbol{a}\in\{0,1\}^n} \sigma(\theta_{\iota(\boldsymbol{a})}) \prod_j x_j^{a_j}(1-x_j)^{1-a_j}$	Autograd
Hybrid	$\mathbb{R}^{2^n}$	$\sigma(\boldsymbol{\theta})_{\iota(H(\mathbf{x}))}$	Probabilistic surrogate
DiffLogic Petersen et al. (2022)	$\mathbb{R}^{16m}$	$\sum_{k=1}^{16} \mathrm{softmax}(\theta)_k \cdot f_k(x_{i_1}, x_{i_2})$	Autograd
Polynomial Andronic et al. (2025)	$\mathbb{R}^{\sum_{d'\le d}\binom{n}{d'}}$	$\sigma(\sum_{\lvert\alpha\rvert\le d} \theta_\alpha \mathbf{x}^\alpha)$	Autograd
WARP Gerlach et al. (2025)	$\mathbb{R}^{2^n}$	$\sigma(\frac{1}{\tau}\sum_{s=0}^{2^n-1} \theta_s \prod_{j\in S_s}(2x_j-1))$	Autograd
Fourier	$\mathbb{R}^{3m}$	$\sigma(\mathrm{bias} + \sum_{k=1}^{m} \lvert a_k\rvert \cos(2\pi\langle \boldsymbol{k}_k, \mathbf{x}\rangle + \phi_k))$	Autograd
MLP Andronic & Constantinides (2025)	$\{\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)}\}_{\ell=1}^{L}$	$\sigma(\boldsymbol{W}^{(L)}\sigma(\cdots\sigma(\boldsymbol{W}^{(1)}\mathbf{x} + \boldsymbol{b}^{((1)})\cdots) + \boldsymbol{b}^{(L)})$	Autograd

Table 4: Differentiable LUT relaxations: parameterization, forward pass, and gradient computation. The cited works serve as conceptual inspiration; the implementations used in this work may deviate from the original implementation logic to ensure compatibility with the node input–output domain and to support residual initialization schemes.

Gradient computation follows standard automatic differentiation except where noted: DWN uses Extended Finite Difference as a surrogate gradient for the hard thresholding operation; Hybrid combines DWN's discrete forward pass with probabilistic surrogate gradients to enable smooth backpropagation despite non-differentiable binary thresholding.

B Implemented Layers

A layer $\mathcal{L}$ instantiates w parallel LUT nodes, each receiving n inputs according to a connection mapping $\mathcal{M}$. The mapping strategy is the central design choice, determining expressiveness, trainability, and hardware feasibility. We implement three principal layer types, each balancing different tradeoffs between flexibility, efficiency, and learnability.

Random Layers. Petersen et al. (2022) Random layers fix the connection mapping $\mathcal{M}$ at initialization and do not update it during training. Each LUT node j receives n inputs selected uniformly at random from the input vector. During training, only LUT configurations $\boldsymbol{\theta}^{(j)}$ are learned; the fixed random wiring provides implicit regularization and avoids learning overly specialized connections. This approach parallels random feature mappings in kernel methods, offering universal approximation guarantees in expectation when layer width is sufficiently large.

Learnable Layers. Bacellar et al. (2025) Learnable layers optimize the connection mapping jointly with LUT parameters, enabling task-specific adaptive routing. The mapping is parameterized as a weight matrix $\boldsymbol{W} \in \mathbb{R}^{w \cdot n \times w_{\text{in}}}$ where each input position learns which input dimension to select. During training, soft selection employs temperature-controlled softmax:

$$\mathcal{M}_j^{(\text{soft})}(x; \boldsymbol{W}, \tau) = \text{softmax}\left(\frac{\boldsymbol{W}_j}{\tau}\right)^{\top} x, \tag{10}$$

where $\boldsymbol{W}_j$ is the weight vector for position j. As training progresses, temperature τ is annealed toward 0, gradually hardening the selection to discrete indices. At inference, the mapping discretizes via arg max:

$$\mathcal{M}_j^{(\text{hard})}(x; \boldsymbol{W}) = x_{\arg\max_i W_{j,i}}, \tag{11}$$

yielding sparse, fixed wiring that enables the network to discover task-optimal connection patterns.

Top-K Sparse Layers. Fojcik et al. (2025) Top-K sparse layers balance expressivity with efficiency by restricting selection to a fixed set of k randomly sampled input candidates per node input position. For each position j, a set of k candidate input indices $\mathcal{C}_j$ is sampled uniformly at random and fixed at initialization. A learnable weight matrix $\boldsymbol{W}_k \in \mathbb{R}^{w \cdot n \times k}$ then selects among these k fixed candidates. During training, soft selection applies:

$$\mathcal{M}_j^{(\text{soft})}(x; \boldsymbol{W}_k, \mathcal{C}_j, \tau) = \sum_{i=1}^{k} \frac{\exp(W_{k,j,i}/\tau)}{\sum_{i'=1}^{k} \exp(W_{k,j,i'}/\tau)} x_{\mathcal{C}_{j,i}}, \tag{12}$$

where $W_{k,j,i}$ is the weight for the i-th candidate. Temperature annealing gradually sharpens the selection. At inference, hard selection picks the best candidate:

$$\mathcal{M}_j^{(\text{hard})}(x; \boldsymbol{W}_k, \mathcal{C}_j) = x_{\mathcal{C}_{j,\arg\max_{i=1}^{k} W_{k,j,i}}}, \tag{13}$$

This approach reduces parameters from $O(w \cdot n \cdot w_{\text{in}})$ to $O(w \cdot n \cdot k)$, enabling deployment in memory-constrained settings while providing implicit regularization through the fixed sparse candidate pool.

Training Enhancements. All layer types support optional mechanisms to improve learning dynamics and robustness. *Bit-flip augmentation* randomly inverts input bits during training with probability $p_{\text{flip}} \in [0, 1]$, encouraging nodes to learn robust functions invariant to input perturbations. *Gradient stabilization* normalizes activation statistics in the backward pass to prevent gradient explosion or vanishing, with three modes: `none` (default), `layerwise` (normalize across entire layer), or `batchwise` (normalize per batch element). Gradients are normalized as:

$$\tilde{\nabla} = \frac{\nabla - \mu(\nabla)}{\sigma(\nabla) + \epsilon} \cdot \sigma_{\text{target}}, \tag{14}$$

where σ_{target} is the desired output standard deviation and ϵ prevents numerical instability. Both mechanisms are applied only during training and improve convergence in deeper networks.

C Implemented Blocks

Blocks are composable modules that combine one or more layers to enable structured computation patterns for standard deep learning architectures. Unlike individual layers, blocks exploit domain-specific structure (e.g., spatial locality, token interactions) to provide parameter sharing and efficient information flow. They serve as building blocks for complete models while remaining flexible enough to support diverse topologies.

C.1 Convolutional Blocks

Convolutional blocks apply a LUT layer to spatial patches extracted from multi-dimensional input, enabling weight sharing across space. We implement 1D, 2D, and 3D variants for sequences, images, and volumetric data.

For 2D convolution with input $\boldsymbol{X} \in \mathbb{R}^{N \times C_{\text{in}} \times H \times W}$, patches are extracted at each spatial position (h, w):

$$\boldsymbol{P}_{h,w} = \text{unfold}(\boldsymbol{X})[h, w] \in \mathbb{R}^{k_H k_W C_{\text{in}}}, \tag{15}$$

processed through a LUT layer $\mathcal{L}$:

$$\boldsymbol{Y}_{h,w} = \mathcal{L}(\boldsymbol{P}_{h,w}) \in \mathbb{R}^{C_{\text{out}}}, \tag{16}$$

producing output shape $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ with output dimensions:

$$H_{\text{out}} = \left\lfloor \frac{H + 2p_H - d_H(k_H - 1) - 1}{s_H} \right\rfloor + 1. \tag{17}$$

Grouped convolution partitions channels into independent groups:

$$\text{input_dim per group} = \frac{C_{\text{in}}}{\text{groups}} \times k_H \times k_W, \quad \text{output_dim per group} = \frac{C_{\text{out}}}{\text{groups}}, \tag{18}$$

enabling parameter reduction and structured subnetwork learning. 1D and 3D variants follow analogous formulations applied to sequences and volumetric data respectively, supporting equivalent padding, stride, dilation, and grouping mechanisms.

C.2 Transposed Convolutional Blocks

Transposed convolutional blocks implement learnable upsampling via grid tiling. For each input position i with value $x_i \in \mathbb{R}^{C_{\text{in}}}$, a LUT layer produces a kernel-sized tile:

$$z_i = \mathcal{L}(x_i) \in \mathbb{R}^{C_{\text{out}} \times k_H \times k_W}, \tag{19}$$

which is placed directly in the output grid at position $(i_H \cdot k_H : (i_H + 1) \cdot k_H, i_W \cdot k_W : (i_W + 1) \cdot k_W)$ without overlapping. Output spatial dimensions are controlled by padding:

$$H_{\text{out}} = (H_{\text{in}} - 1) \cdot k_H + 2p_H + 1 + \text{output_padding}_H. \tag{20}$$

Stride and dilation are fixed to 1 for simplicity, distinguishing this from standard transposed convolution which permits overlapping summation.

C.3 Residual Blocks

Residual blocks chain multiple LUT layers sequentially with skip connections that concatenate selected input channels to the output, improving gradient flow and enabling feature reuse:

$$x_1 = \mathcal{L}_1(x), \quad x_2 = \mathcal{L}_2(x_1), \quad \ldots, \quad x_N = \mathcal{L}_N(x_{N-1}), \tag{21}$$

$$\text{output} = \text{concat}(x[:p], x_N), \tag{22}$$

where p is the number of residual channels (a configurable fraction of input dimension). Residual channel indices are sampled at initialization with a seeded random generator, ensuring reproducibility and providing implicit regularization. Unlike traditional residual addition, concatenation enables flexible dimension matching and richer feature combination.

C.4 Attention Blocks

Attention blocks compute pairwise token interactions and aggregate results, modeling long-range dependencies through two-stage processing on sequences of shape (B, N, D).

Stage 1: Pairwise Attention. For each pair of tokens (i, j), concatenate and process through a LUT layer:

$$\text{pair_input}_{ij} = \text{concat}(t_i, t_j) \in \mathbb{R}^{2D}, \tag{23}$$

$$\text{attn}_{ij} = \mathcal{L}_1(\text{pair_input}_{ij}) \in \mathbb{R}^{D_{\text{attn}}}, \tag{24}$$

producing an $N \times N \times D_{\text{attn}}$ attention tensor.

Stage 2: Aggregation. Flatten the attention tensor and process with the original input:

$$\text{agg_input} = \text{concat}(x, \text{flatten}(\text{attn})) \in \mathbb{R}^{D+N^2 D_{\text{attn}}}, \tag{25}$$

$$\text{output} = \mathcal{L}_2(\text{agg_input}) \in \mathbb{R}^{D_{\text{out}}}. \tag{26}$$

The block requires exactly two layer configurations for $\mathcal{L}_1$ and $\mathcal{L}_2$. Computational complexity is $O(N^2 D)$ due to the quadratic number of token pairs, inherent to any attention mechanism.

D Implemented Encoders

Encoders transform continuous or integer-valued inputs into binary representations suitable for LUT-based computation. They form a critical preprocessing layer, enabling LUT nodes to operate on diverse input modalities while maintaining the binary abstraction required for efficient FPGA synthesis. All encoders are fitted to training data to learn task-appropriate transformations, then applied deterministically at inference. This design decouples input preprocessing from core LUT-based computation, enabling flexible handling of diverse input modalities.

D.1 Encoding Principles

An encoder $\mathcal{E}$ maps each input dimension into a binary representation:

$$\mathcal{E} : \mathbb{R}^d \to \{0, 1\}^{d \cdot b}, \tag{27}$$

where b denotes the number of encoding bits per input dimension. The encoder is fitted to training data to learn thresholds, statistics, or other task-specific parameters, then applied deterministically at inference.

D.2 Implemented Encoders

Table 5 summarizes all implemented encoders. Each operates on continuous inputs and produces binary representations through different mechanisms: threshold-based comparisons, bit representations, or distribution-based mappings.

Table 5: Implemented encoders: parameterization, transformation, and fitting strategy.

Encoder	**Parameters**	**Encoding**	**Fitting**
Thermometer	$\boldsymbol{\theta} \in \mathbb{R}^b$	$(\mathbf{1}_{x>\theta_1}, \ldots, \mathbf{1}_{x>\theta_b})$	Quantiles
Gaussian Therm.	$\boldsymbol{\theta} \in \mathbb{R}^b$	$(\mathbf{1}_{x>\theta_1}, \ldots, \mathbf{1}_{x>\theta_b})$	Gaussian
Dist. Therm.	$\boldsymbol{\theta} \in \mathbb{R}^b$	$(\mathbf{1}_{x>\theta_1}, \ldots, \mathbf{1}_{x>\theta_b})$	Empirical
Binary	$(\min, \max)$	$\text{int2bits}(\lfloor (x - \min)/\Delta \cdot (2^b - 1) \rfloor)$	Min/max
Gray	$(\min, \max)$	$\text{gray}(\text{int2bits}(\cdot))$	Min/max
One-Hot	Edges	$\text{onehot}(\text{bucketize}(x))$	Quantiles
Sign-Magnitude	$\max \lvert \cdot \rvert$	$(\text{sign}(x), \text{bits}(\lvert \cdot \rvert))$	Max abs
Logarithmic	Offset, range	$\text{int2bits}(\lfloor (\log(x + \text{off}) - \min_{\log})/\Delta \rfloor)$	Log range

E Implemented Heads

Heads are output layers that transform the final layer's binary feature vectors into task-appropriate predictions. A head $\mathcal{H}$ maps from the final layer's output to the target space:

$$\mathcal{H} : \{0, 1\}^w \to \mathbb{R}^c, \tag{28}$$

where w is the final layer width and c is the number of output classes or targets. All heads partition the binary output into groups, aggregate within groups, and optionally apply learnable weights—enabling simple, hardware-friendly aggregation while maintaining expressiveness.

E.1 GroupSum Head

GroupSum partitions the final layer output into c groups and sums within each group:

$$\text{output}_i = \frac{1}{\tau} \sum_{j=1}^{g} x_{i,j}, \quad g = \frac{w}{c}, \tag{29}$$

where τ is a learnable temperature parameter controlling output magnitude. Each group receives g binary features; the sum ranges from 0 to g, providing a coarse but interpretable aggregation. If w is not divisible by c, input is zero-padded. GroupSum maps directly to addition trees on FPGA.

E.2 GroupedDSP Head

GroupedDSP extends GroupSum by adding learnable weights per group while maintaining DSP-safe integer quantization at inference. During training:

$$\text{output} = \boldsymbol{W} \cdot \text{group_sums}, \quad \boldsymbol{W} \in \mathbb{R}^{c \times c}, \tag{30}$$

where $\boldsymbol{W}$ contains continuous weights constrained to $[-1, 1]$ via tanh. During inference (eval mode), weights are quantized to w-bits integers:

$$\boldsymbol{W}_{\text{quantized}} = \text{clamp}(\text{round}(\tanh(\boldsymbol{W}) \cdot (2^{w\text{bits}-1} - 1)), -2^{w\text{bits}-1}, 2^{w\text{bits}-1}). \tag{31}$$

The quantized weights are normalized by their scale to maintain output magnitude consistency with training mode. With w-bits ≤ 15 (default: 12), weights fit safely in DSP48E1 blocks, enabling single-cycle inference on FPGA. GroupedDSP provides higher model capacity than GroupSum at the cost of using Latency due to use of DSPs

F Residual Initialization for Deep LUT Networks

Training deep differentiable lookup-table (LUT) networks is challenging due to vanishing gradients induced by repeated nonlinear compositions. Residual initialization mitigates this issue by initializing each node to approximate an identity mapping, such that early forward passes preserve the input signal while deeper layers learn residual corrections.

General principle. Consider a node with parameters θ and output

$$f(\mathbf{x}) = \sigma\left(\sum_i w_i \, \phi_i(\mathbf{x}) + b\right),$$

where ϕ_i are node-specific basis functions and σ is a sigmoid. Residual initialization enforces

$$f(\mathbf{x}) \approx x_1$$

by assigning a large positive logit to the basis function corresponding to the first input and a compensating negative bias, while all remaining parameters are set to zero or small Gaussian noise. Two hyperparameters control this behavior:

- *logit clarity* $c > 0$: determines how closely the initialized node approximates a binary identity ($\sigma(\pm c) \approx 1/0$),
- *noise level* σ_{noise}: standard deviation of optional Gaussian perturbations enabling residual diversity.

Example (Probabilistic LUT node). For a probabilistic node with k binary inputs,

$$f(\mathbf{x}) = \sum_{\boldsymbol{a} \in \{0,1\}^k} \sigma\big(\theta_{\iota(\boldsymbol{a})}\big) \prod_{j=1}^{k} x_j^{a_j} (1 - x_j)^{1 - a_j},$$

residual initialization sets

$$\theta_{\iota(\boldsymbol{a})} = \begin{cases} c + \varepsilon, & a_1 = 1, \\ -c + \varepsilon, & a_1 = 0, \end{cases} \qquad \varepsilon \sim \mathcal{N}(0, \sigma_{\text{noise}}^2).$$

For $\sigma_{\text{noise}} = 0$, the node implements an exact identity $f(\mathbf{x}) = x_1$ after the sigmoid; for $\sigma_{\text{noise}} > 0$, it realizes a noisy identity that supports residual learning. This initialization is boundary-consistent, fully differentiable, and preserves gradient flow.

Generality. The same construction applies across node families (linear, polynomial, neural, Fourier, DWN, hybrid, DiffLogic, WARP) by identifying the parameter associated with the first-input basis function, assigning it a logit of $+c$, introducing a compensating bias (typically $-c$), and optionally perturbing all other parameters with small Gaussian noise. This unified strategy enables stable training of very deep LUT networks by maintaining signal propagation at initialization.

G Node Regularization

Two complementary regularization techniques ensure learned LUT functions discretize cleanly to Boolean operations and remain robust under input perturbations:

Discretization gap. To avoid the ambiguous 0.5 region where discretization error is largest, discretization gap penalizes outputs not decisively 0 or 1:

$$\mathcal{L}_{\text{gap}} = \|\boldsymbol{y} - 0.5\|_p, \quad p \in \{1, 2, \infty\}, \tag{32}$$

where $\boldsymbol{y} \in [0,1]^m$ is the node output. This is computationally efficient ($O(m)$, no additional forward passes) and purely output-dependent. During training with weight $\lambda_{\text{gap}} > 0$, outputs are driven toward extremes, making post-training discretization via $D_{\text{out}}(y) = H(y - 0.5)$ faithful.

Bitflip consistency. For hardware robustness, penalize large output changes under single-bit input flips. For input $\mathbf{x} \in [0,1]^n$ and bitflipped variants $\mathbf{x}^{(i)} = \mathbf{x} \oplus \boldsymbol{e}_i$:

$$\mathcal{L}_{\text{bitflip}} = \frac{1}{n} \sum_{i=1}^{n} \left\| f(\mathbf{x}) - f(\mathbf{x}^{(i)}) \right\|_p, \tag{33}$$

where f is the node forward function. This measures Hamming sensitivity and costs $O(n \cdot \text{cost}(f))$ (requires n forward passes) but enables end-to-end gradient-based learning.

H Experiment Details

To enable reproducibility and provide detailed insights into our baseline model configurations, we present a comprehensive description of the architectures, training procedures, and hyperparameters used in our comparative analysis.

H.1 Model Architecture Details

Our baseline study encompasses two primary architectures: feedforward networks (FFN) and convolutional neural networks (CNN). Both architectures utilize probabilistic logic gates as their fundamental computational units, with differences in their structural organization and parameter scaling.

H.1.1 Feedforward Network (FFN) Architecture

The feedforward baseline models follow a simple yet effective design pattern. The architecture consists of:

- **Input encoding layer:** A distributive thermometer encoder with 8 bits, flattening the input representation.
- **Hidden layer 1:** A top-k sparse layer with $k = 8$ and output dimension of $64,000$, using residual weight initialization with clarity parameter $\tau_{\text{logit}} = 5.0$ and noise factor $\sigma = 0.0$.
- **Hidden layer 2:** A second top-k sparse layer with identical dimensionality, sparsity ($k = 8$), and initialization parameters as layer 1.
- **Output head:** A GroupSum classifier head mapping to the appropriate number of classes (10 for CIFAR-10, MNIST, and FashionMNIST; 100 for CIFAR-100) with softmax temperature $\tau = 150$.

Input dimensions are dataset-dependent: CIFAR datasets ($32 \times 32 \times 3$ RGB) map to 24,576 dimensions after encoding, while MNIST/FashionMNIST (28×28 grayscale) map to 6,272 dimensions. Each hybrid probabilistic node in FFN layers uses input dimension 4 and output dimension 1 within the top-k sparse framework.

H.1.2 Convolutional Network (CNN) Architecture

The convolutional baselines employ stacked convolutional blocks followed by a fully connected layer. The architecture differs only in input encoding channels: CIFAR variants (32×32 RGB) use 24 input channels after encoding, while MNIST-like variants (28×28 grayscale) use 8 input channels. The core structure is:

- **Input encoding:** Distributive thermometer encoder with 8 bits, maintaining spatial structure (not flattened).
- **Convolutional block 1:** Input channels $\rightarrow$ 4,096 output channels, 3×3 kernel, stride 2, padding 1.
- **Convolutional block 2:** 4,096 $\rightarrow$ 16,384 channels, 3×3 kernel, stride 2, padding 1.
- **Fully connected layer (lut_0):** Flattened spatial representation mapped to 64,000 dimensions.
- **Output head:** GroupSum classifier with dataset-appropriate classes (10 or 100) and $\tau = 150$.

All convolutional blocks employ zero-padding to maintain specified spatial dimensions. Each convolutional layer is built from top-k sparse layers with $k = 8$, where each hybrid probabilistic node uses input dimension 6, output dimension 1, and residual weight initialization with $\tau_{\text{logit}} = 5.0$ and noise factor 0.0. The fully connected layer (lut_0) also uses a top-k sparse layer with $k = 8$ and hybrid nodes having input dimension 4 and output dimension 1, but with normal weight initialization (standard deviation 0.1) instead of residual initialization.

H.2 Training Details

H.2.1 Hyperparameters

Table 6 lists key hyperparameters. All models use Adam with $\eta = 0.01$, 50 epochs, and $\tau = 150$. Weight decay ($\beta = 10^{-4}$) applies only to CIFAR FFN models.

Table 6: Hyperparameter settings.

Dataset	**Arch.**	β	**Batch**
CIFAR (10/100)	FFN	10^{-4}	32
	CNN	0.0	8
MNIST/Fashion	FFN	0.0	32
	CNN	0.0	8

H.2.2 Node Complexity Analysis

Table 7 reports trainable and computational node counts. For CNN, computational nodes account for spatial patches: CIFAR variants (32×32) yield 256 and 64 patches for Conv_0/Conv_1; MNIST variants (28×28) yield 196 and 49 patches respectively.

Table 7: Node counts per architecture variant.

Model	**Trainable Nodes**	**Computational Nodes**
FFN	128K	128K
CNN (CIFAR)	84.48K	2.16M
CNN (MNIST)	84.48K	1.67M

H.3 Implementation Considerations

All experiments use random seed 42 for reproducibility and are trained on NVIDIA GPUs (TITAN RTX or RTX 3090) with CUDA 12.1 using PyTorch's Distributed Data Parallel framework with NCCL backend across 2 GPUs per node. The BitLogic framework provides efficient implementations of the differentiable logic gates and custom CUDA kernels. Training times per epoch and GPU utilization are shown in Table 8. This substantial difference reflects CNN's larger output dimensions and increased computational complexity compared to FFN. GPU utilization is consistently low (∼2%), indicating LUT-based operations are memory-bound rather than compute-bound—expected for logic gate architectures emphasizing bit-width reduction.

Table 8: Per-epoch training time on 2 Titan RTX using DDP. Times are averages across all datasets.

Architecture	**Time/Epoch (min)**	**Avg GPU Mem (GB)**
FFN	103.67	3.96
CNN	285.80	15.21