Convolutional Differentiable Logic Gate Networks 1

Felix Petersen 2
Stanford University
InftyLabs Research
mail@felix-petersen.de

Hilde Kuehne 3
Tuebingen AI Center
MIT-IBM Watson AI Lab
h.kuehne@uni-tuebingen.de

Christian Borgelt 4
University of Salzburg 6
christian@borgelt.net

Julian Welzel 5
InftyLabs Research
welzel@inftylabs.com

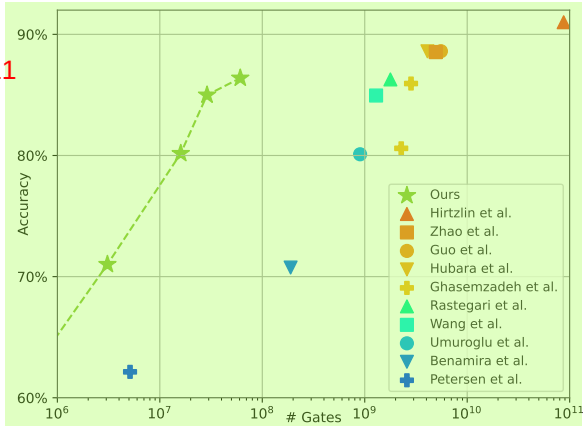
Stefano Ermon 7
Stanford University
ermon@cs.stanford.edu

Abstract 8

With the increasing inference cost of machine learning models, there is a growing interest in models with fast and efficient inference. Recently, an approach for learning logic gate networks directly via a differentiable relaxation was proposed. Logic gate networks are faster than conventional neural network approaches because their inference only requires logic gate operators such as NAND, OR, and XOR, which are the underlying building blocks of current hardware and can be efficiently executed. We build on this idea, extending it by deep logic gate tree convolutions, logical OR pooling, and residual initializations. This allows scaling logic gate networks up by over one order of magnitude and utilizing the paradigm of convolution. On CIFAR-10, we achieve an accuracy of 86.29% using only 61 million logic gates, which improves over the SOTA while being 29× smaller. 9

1 Introduction 10

Deep learning has led to a variety of new applications, opportunities, and use-cases in machine vision. However, this advancement has come with considerable computational and energy costs for inference [1]. Therefore, an array of methods has been developed for efficient deep learning inference [2]–[7]. These include binary weight neural networks (BNNs) [2], a set of methods for quantizing neural network weights down to binary representations (and sometimes also binary activations); quantized low-precision neural networks [3], a superset of BNNs and sparse neural networks [4]–[6], a set of approaches for pruning neural networks and increasing sparsity. These methods have been successfully utilized for efficient vision model inference. 11



12

Figure 1: Gate count vs. accuracy plot on the CIFAR-10 data set. Our models (★) are substantially above the pareto-front of the SOTA baselines. Gate counts are proportional to chip area. Our models are more efficient than the SOTA by factors of $\geq 29\times$. Note that the x-axis (gate count) is on a log-scale. 13

The state-of-the-art (SOTA) method for small architectures, deep differentiable logic gate networks (LGNs) [7], approaches efficient machine learning inference from a different direction: learning an LGN (i.e., a network of logic gates such as NAND and XOR) directly via a differentiable relaxation. 14

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

Differentiable LGNs directly learn the combination of logic gates that have to be executed by the hardware. This differs from other approaches (like BNNs) that require translating an abstraction (like matrix multiplication-based neural networks) into executable logic for inference, an inductive bias that comes with a considerable computational burden. By optimizing the logic directly on the lowest possible level instead of optimizing an abstraction, differentiable LGNs lead to very efficient inference on logic gate-based hardware (e.g., CPU, GPU, FPGA, ASIC). Recently, differentiable LGNs achieved SOTA inference speeds on MNIST [7], [8]. However, a crucial limitation was the random choice of connections, preventing LGNs from learning spatial relations, as they arise in images, which limited performance to an accuracy of only 62% on CIFAR-10 [7], [9]. To address this limitation, we propose to extend differentiable LGNs to convolutions. Specifically, we propose deep logic gate tree convolutions, i.e., kernels comprised of logic gate trees applied in a convolutional fashion. Using trees of logic gates, instead of individual gates, increases the expressivity of the architecture while minimizing memory accesses, improving accuracy and accelerating training as well as inference. Further, we adapt pooling operations by representing them with logical *or* gates (relaxed via the maximum t-conorm), improving the effectiveness of convolutions in LGNs. Additionally, we propose "residual initializations", a novel initialization scheme for differentiable LGNs that enables scaling them up to deeper networks by providing differentiable residual connections. These advances lead to an accuracy of 86.29% on CIFAR-10 using only 61 million logic gates, leading to cost reductions by $\geq 29\times$ compared to SOTAs as displayed in Figure 1.

2 Background

Our work builds on and extends differentiable logic gate networks [7]. To recap, logic gate networks (LGNs) are networks of nodes that are binary logic gates like AND, NAND, or XOR. LGNs are also known as binary circuits or logical circuits, and are the format in which any digital hardware is implemented on the lowest pre-transistor abstraction level. The function that an LGN computes depends on the choices of logic gates that form its nodes and how these nodes are connected. Optimizing an LGN requires choosing the connections and deciding on a gate for each node. A primary challenge when optimizing LGNs is that they are, by default, non-differentiable, preventing gradient descent-based training, making this problem conventionally a combinatorial problem. However, when applied to machine learning problems, solving the combinatorial problem conventionally becomes infeasible as we require millions of parameters or gates. Thus, a differentiable relaxation of randomly connected LGNs has been proposed, which allows training LGNs with gradient descent [7], overcoming the exponential difficulty of optimizing LGNs. In the remainder of this section, we cover the structure, relaxation, and training of differentiable LGNs, which we also illustrate in Figure 2.

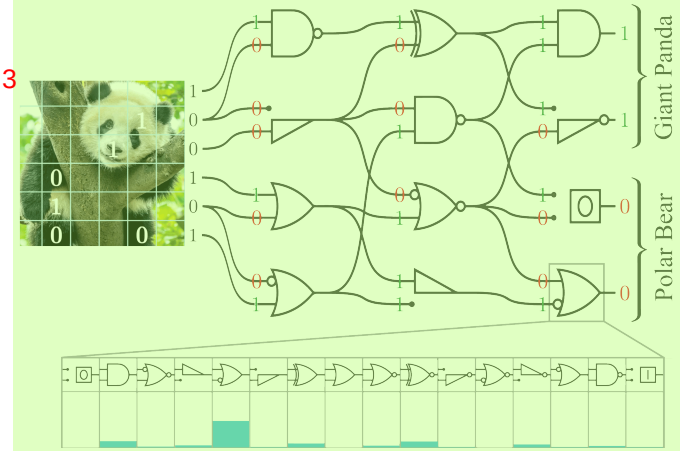


Figure 2: Architecture of a randomly connected LGN. Each node corresponds to one logic gate. During training, the distribution over choices of logic gates (bottom, 16 options) is learned for each node.

Structure LGNs follow a layered structure with each layer comprising a number of nodes, each comprising one logic gate (3 layers with 4 logic gates each in Fig. 2). As logic gates are inherently non-linear, LGNs do not require any activation functions. Further, LGNs do not have any weights nor any biases as they do not rely on matrix multiplications. Due to the binary (i.e., two-input) nature of the nodes, LGNs are necessarily sparse and cannot form fully-connected networks. The connectivity between nodes has so far been (fully) randomly selected, which works well for easier tasks but can become problematic if there is inherent structure in the data as, e.g., in images. During training, the connections remain fixed and the learning task comprises the choice of logic gate at each node.

Differentiable Relaxation To learn the choices of logic gate for each node with gradient descent requires the network to be differentiable; however, the LGN is by default not differentiable for two reasons: (i) Because a logic gate computes a discrete function of its (Boolean) inputs, it is not differentiable. (ii) Because the choice of logic gate is not a continuous parameter, but a discrete

decision, it is not differentiable. Petersen *et al.* [7] propose to differentiably relax each logic gate to real-valued logic via probabilistic logic [10], [11]. For example, a logical *and* ($a_1 \wedge a_2$) is relaxed to $a_1 \cdot a_2$ and a logical *exclusive or* ($a_1 \oplus a_2$) is relaxed to $a_1 + a_2 - 2 \cdot a_1 \cdot a_2$, which corresponds to the output probability when considering two independent Bernoulli variables with coefficients a_1, a_2. To make the choice of logic gate learnable, Petersen *et al.* [7] introduce a probability distribution over the 16 possible logic gates ($\mathcal{S}$), which is encoded as the softmax of 16 trainable parameters. For a trainable parameter vector $\mathbf{z} \in \mathbb{R}^{16}$ and all 16 possible logic gate operations as $g_0, ..., g_{15}$, the differentiable logic gate as the expectation over its outputs can be computed in closed-form as

$$f_{\mathbf{z}}(a_1, a_2) = \mathbb{E}_{i \sim \mathcal{S}(\mathbf{z}),\, A_1 \sim \mathcal{B}(a_1),\, A_2 \sim \mathcal{B}(a_2)} \Big[g_i(A_1, A_2) \Big] = \sum_{i=0}^{15} \frac{\exp(z_i)}{\sum_j \exp(z_j)} \cdot g_i(a_1, a_2)\,. \quad (1)$$

With these two ingredients, logic gate networks become end-to-end differentiable.

Initialization, Training, and Discretization Training differentiable logic gate networks corresponds to learning the parameters inducing the probability distributions over possible gates. The parameter vector $\mathbf{z}$ for each node has so far been initialized with a standard Gaussian distribution. The connections are randomly initialized and remain fixed during training. For classification tasks, each class is associated with a set of neurons in the output layer and active neurons in each set are counted composing a class score (group sum, right part of Fig. 2). After dividing them by a temperature τ, the class scores are used as logits in a softmax cross-entropy loss. Differentiable LGNs perform best when trained with the Adam optimizer [12]. Empirical evidence showed that the softmax distributions typically converge to concrete choices of logic gates. Thus, differentiable LGNs can be discretized to hard LGNs for deployment on hardware by selecting the logic gate with the largest probability. This discretization process incurs only a minimal loss in accuracy compared to the differentiable LGN [7].

Limitations Differentiable LGNs have shown significant limitations wrt. the available architectural components. Previously, they did not provide the option to capture local spatial patterns as they were randomly connected and only operated on flattened inputs [7]. Further, they previously performed well only up to a depth of 6 layers [7]. Thus, more complex relationships between inputs cannot be modeled. Finally, while they provide SOTA performance, differentiable LGNs are very computationally expensive to train, e.g., a vanilla 5 million gate network required 90 hours on an A6000 GPU [7]. In the following, we address these limitations by introducing convolutional logic tree layers, logical or pooling, residual initializations, as well as computational considerations for scaling.

3 Convolutional Logic Gate Networks

Convolutional neural networks (CNNs) have experienced tremendous success, being a core contributor to the current machine learning ascendancy starting with their progress on the ImageNet classification challenge in 2012 [13]. Underlying CNNs is the discrete convolution of an input tensor $\mathbf{A}$ (e.g., an input image or hidden activations) and a linear function / kernel $\mathbf{W}$, denoted as $\mathbf{A} * \mathbf{W}$. CNNs are especially effective in vision tasks due to the equivariance of the convolution, which allows the network to generalize edge, texture, and shapes in different locations by sharing the parameters at all placements. However, existing differentiable LGN methods do not support convolutions.

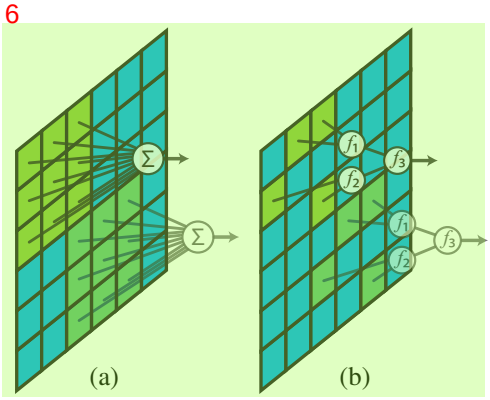


Figure 3: Conventional convolutional neural networks (a) compared to convolutional logic gate networks (b). The images illustrate the first and second to last kernel placements. The nodes correspond to weighted sums (a), and binary logic gates f_1, f_2, f_3 (b), respectively. The weights / choices of logic gates are shared between kernel placements. For visual simplicity, only a single input channel and kernel (output channel) is displayed.

In this work, we propose to convolve activations $\mathbf{A}$ with differentiable binary logic gate trees. While we could convolve $\mathbf{A}$ with an individual logic gate, we observe that actually convolving $\mathbf{A}$ with a (deep) logic gate network or tree leads to substantially better performance as it allows for greater expressivity of the model. Similar to how the inputs to each logic gate are randomly initialized and remain fixed in conventional differentiable LGNs, we randomly construct the connections in our

logic gate tree kernel function. However, we need to put additional restrictions on the connections for logic gate network kernels. Specifically, we construct each logic gate network kernel as a complete binary tree of depth d with logic gates as nodes and binary input activations as leaves. The output of the logic gate operation is then the input to the next higher node, etc. To capture spatial patterns, we select the inputs / leaves of the tree from the predefined receptive field of the kernel of size $s_h \times s_w$. Based on the depth of the tree, we randomly select as many inputs as necessary. For example, we could construct a binary tree of depth $d = 2$, which means that we need to randomly select $2^d = 4$ inputs from our receptive field, e.g., of size $64 \times 3 \times 3$, which corresponds to 64 input channels with a kernel size of 3×3. This tree structure allows to capture fixed spatial patterns and correlations beyond pair-wise binary inputs. Further, it extends the concept of spacial equivariance to LGNs as such trees can be used as kernel filters, capturing general patterns in different locations. Using trees of logic gates instead of individual logic gates also has the advantage of reducing memory accesses and improving training and inference efficiency. We remark that, as we apply convolution, the parameterization of each node is shared between all placements of the kernel (which contrasts convolution from mere local connectivity.) In Figure 3, we illustrate the difference between conventional CNN models and convolutional logic gate networks.

During training, the network learns which logic gate operation to choose at each node. Thus, each logic tree kernel is parameterized via the choices of each of the $2^d - 1$ logic gates, which are learnable. For a logic kernel of depth 2, we call these logic gates f_1, f_2, f_3 (or more formally $f_{\mathbf{z}_1}, f_{\mathbf{z}_2}, f_{\mathbf{z}_3}$ for parameter vectors $\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3$ corresponding to Equation 1). Given input activations a_1, a_2, a_3, a_4, the kernel is expressed as a binary tree of these logic gates:

$$f_3(\, f_1(a_1, a_2), f_2(a_3, a_4)\,). \tag{2}$$

For an input $\mathbf{A}$ of shape $m \times h \times w$ (m input channels; height; width) and connection index tensors $\mathbf{C}_M, \mathbf{C}_H, \mathbf{C}_W$[1], each of shape $n \times 4$ (n tree kernels / channels; 4 inputs per tree), the output is

$$\begin{aligned}\mathbf{A}'[k,i,j] = f_3^k\big(&f_1^k\big(\mathbf{A}\big[\mathbf{C}_M[k,1], \mathbf{C}_H[k,1]{+}i, \mathbf{C}_W[k,1]{+}j\big], \mathbf{A}\big[\mathbf{C}_M[k,2], \mathbf{C}_H[k,2]{+}i, \mathbf{C}_W[k,2]{+}j\big]\big),\\ &f_2^k\big(\mathbf{A}\big[\mathbf{C}_M[k,3], \mathbf{C}_H[k,3]{+}i, \mathbf{C}_W[k,3]{+}j\big], \mathbf{A}\big[\mathbf{C}_M[k,4], \mathbf{C}_H[k,4]{+}i, \mathbf{C}_W[k,4]{+}j\big]\big)\big)\end{aligned} \tag{3}$$

for $k \in \{1, ..., n\}$ where n is the number of tree kernels, $i \in \{1, ..., (h - s_h + 1)\}$, and $j \in \{1, ..., (w - s_w + 1)\}$ where $s_h \times s_w$ is the receptive field size. Note that, in Equation 3, for each output channel k the logic gates f_1^k, f_2^k, f_3^k (or their relaxed form) are chosen and parameterized independently. Per convolution, all placements (indexed via i, j) of one kernel share their parameters.

After introducing convolutional LGNs, in the remainder of the section, we introduce our additional components, training strategies, and our architecture.

3.1 Logical *Or* Pooling

In CNNs, max-pooling is a crucial component selecting the largest possible activation over a predefined receptive field, e.g., for 2×2, $\max(a_{i,j}, a_{i,j+1}, a_{i+1,j}, a_{i+1,j+1})$ [13]. To adopt this for logic, we propose to use the disjunction of the binary activations $a_{i,j} \vee a_{i,j+1} \vee a_{i+1,j} \vee a_{i+1,j+1}$ via the logical *or*. Instead of using a probabilistic relaxation of the logical *or*, we can use the maximum t-conorm relaxation of the logical *or* ($\perp_{\max}(a, b) = \max(a, b)$). By setting the stride of the pooling operation to the size of its receptive field, this has a range of crucial computational advantages: (i) it is faster to compute than probabilistic relaxation; (ii) we only need to store the maximum activation and index; (iii) we only need to backpropagate through the maximum activations during training.

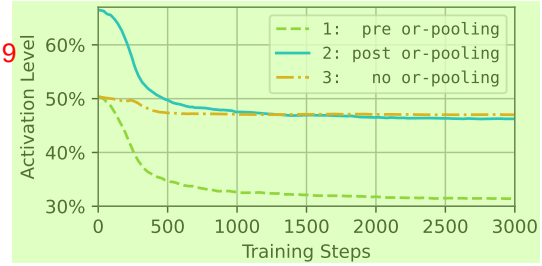


Figure 4: Plot of the density of activations for the second convolutional block of an *or*-pooling based convolutional LGN. It shows that training implicitly enforces that the outputs of the block have the activation level of a no-pooling network (i.e., with pure stride).

Intuitively, using many logical *or*s could lead to the outputs of the activations becoming predominantly 1. However, we find that, during training, this is not an issue as using *or* pooling causes an

[1] $\mathbf{C}_M \in \{1, ..., m\}^{n \times 4}$ indicates which out of m input channels is selected; $\mathbf{C}_H \in \{1, ..., s_h\}^{n \times 4}$ and $\mathbf{C}_W \in \{1, ..., s_w\}^{n \times 4}$ indicate the selected position inside of the receptive field of size $s_h \times s_w$.

automatic reduction of pre-pooling activations, resolving this potential concern. This phenomenon is shown in Figure 4. Here, the average activation of a convolutional block of a logic network with 2×2 strided *or* pooling is illustrated. For a random network without pooling, we expect and observe an average activation of 50% (dash-dotted). We observe that the post *or* pooling activations (solid line) for the initialized models is 66.5%, which follows expectation. The pre *or* pooling activations (dashed) are initialized at 50%, also following expectations. With training, the post *or* pooling activations (solid) rapidly converge to the average activations of a network without pooling, preventing any problematic saturation of activations. We do not introduce any explicit regularization enforcing this behavior, but instead found this to be an emerging behavior of training.

3.2 Residual Initialization

The parameters $\mathbf{z}$ of existing differentiable LGNs were initialized as random draws from a Gaussian distribution. Unfortunately, after applying softmax, this leads to rather "washed out" probability distributions over choices of logic gates. Accordingly, the expected activations, as computed via Equation 1, are also washed out, quickly converging towards 0.5 in deeper networks. This also leads to vanishing gradients in existing differentiable LGNs: With Gaussian initialization, during backpropagation, the gradient norm decays at each logic gate by a factor between 0.1 and 0.2 for an initialized network, exponentially slowing training for deeper networks.

In CNNs, a technique for preventing vanishing gradients and preventing loss of information in deep networks are residual connections. Residual connections conventionally add the input to a block to the output of this block [14]. However, when operating in logic, we cannot perform such additions.

To prevent the loss of information through washed out activations and reduce vanishing gradients with a joint strategy, we propose *residual initializations*. For this, we initialize each logic gate not randomly but instead to be primarily a feedforwarding logic gate. Here, we choose 'A' as a canonical choice and choosing 'B' would be equivalent. In our experiments, we found that initializing the probability for the logic gate choice 'A' to around 90% and setting all other gates to 0.67% works well. This corresponds to setting the parameter $z_3 = 5$ and all other $z_i = 0$ for $i \neq 3$ in accordance to Eq. 1. We illustrate an example of residual initializations compared to the existing Gaussian initializations in Figure 5.

Residual initializations prevent the loss of information as well as vanishing gradients in deeper networks. During training, whenever a residual connection is not required, the model learns to replace the feedforward logic gate choice by an actual operation. Thus, residual initializations are effectively a differentiable form of residual connections that does not require any hard-wiring. This also means that this form of residuals does not require additional logic gates for residuals. Residual initializations enable, for the first time, efficient and effective training of LGNs beyond 6 layers.

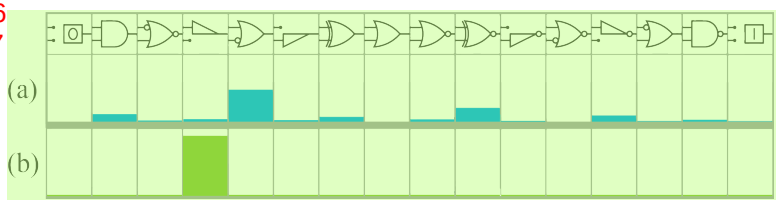


Figure 5: Gaussian initialization (a) [7] vs. our residual initialization (b).

3.3 Computational Training Considerations

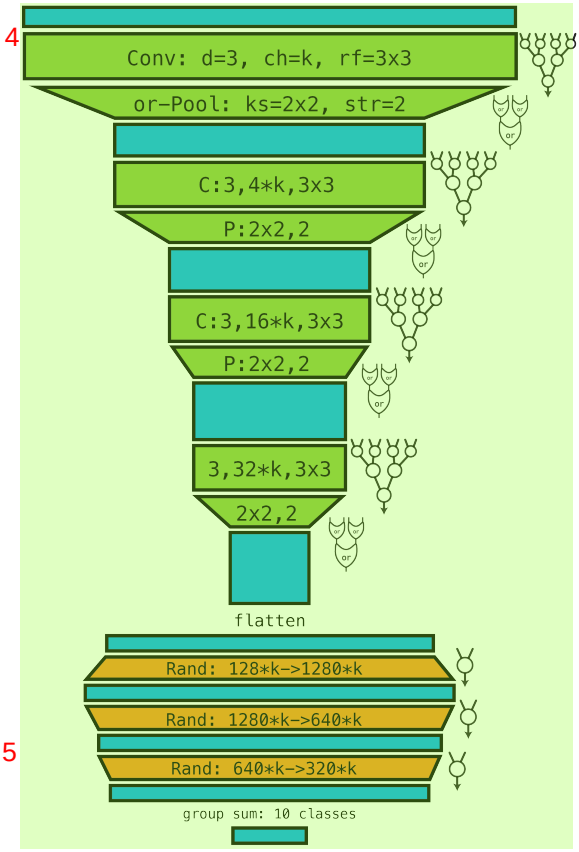
Using trees and pooling allows for substantially improved computational training efficiency and memory requirement reductions. This is because it allows intermediate activations to be used only by the current logic gate tree and because we only need to backpropagate through the maximum activations during *or* pooling. For example, using learnable trees with a depth of 2 and *or* pooling with a kernel size and stride of 2×2 corresponds to a logic gate tree of depth $2 + 2 = 4$ (2 levels are learnable + 2 from pooling) with 16 inputs and only a single output. For training, it is most efficient to discard all intermediate values and only store the output and information of which path through the pooling was selected, and during backward to recompute only this path, thereby reducing memory accesses. The reason for this is that training speed is limited by memory bandwidth and scalability is limited by GPU memory. On average, this strategy reduces memory accesses by 68% and reduces the memory footprint by 90% during training. For using LGNs in hardware designs, trees and pooling improve the locality of operations and routing, which also leads to more efficient chip layouts.

The residual initializations provide a bias towards the feedforward logic gate in trained LGNs. As feedforward gates only require a wire and no transistors, this further reduces the necessary transistor count for hardware implementations of the LGNs, reducing the required chip area. 1

We developed efficient fully-fused low-level CUDA kernels, which, for the first time, enable training of convolutional LGNs. The speed of our convolutional layer is up to 200× faster per logic gate than existing randomly connected LGN implementations [7]. We will make the code publicly available by including it into the `difflogic` library at `github.com/Felix-Petersen/difflogic`. 2

3.4 LogicTreeNet Architecture 3

In the following, we discuss the design of our convolutional logic gate tree network architectures (LogicTreeNet) for CIFAR-10, which we illustrate in Figure 6. We follow the pattern of conventional convolutional architectures and design the architecture by applying convolutional blocks with pooling at the end of each block. Each block reduces the size by a factor of 2×2 and we apply blocks until we reach a size of 2×2, increasing the number of channels in each stage. Following this, we apply two randomly connected layers and a group sum as our classification head. This architecture has an overall logical depth of 23 layers, including 4 convolutional blocks (`Conv`) with tree depths of $d = 3, 4$ *or* pooling layers (`or-Pool`), and 3 randomly connected layers (`Rand`). 15 of these layers are trainable (`Conv` blocks and `Rand`), and the pooling layers remain fixed. The architecture is defined in terms of a hyperparameter `k`, which controls the width of the overall network; we consider $\texttt{k} \in \{\text{S} \to 32, \text{M} \to 256, \text{B} \to 512, \text{L} \to 1\,024, \text{G} \to 2\,048\}$. In Appendix A.1, we describe LogicTreeNet layer-by-layer and include a LogicTreeNet for MNIST. 4



6

Figure 6: LogicTreeNet architecture. The logical architectures of the layers / blocks are illustrated on a per neuron basis. Circles indicate a logic gate that can be learned while the logical *or*s remain fixed. During training, for the trainable nodes, we use probabilistic relaxations of logic gates, which we parameterize via a softmax distribution over operators (Eq. 1/3). For the fixed logical *or*s, we use the continuous maximum t-conorm relaxation. 7

An additional architecture choice is the connectivity for the inputs to a convolutional tree. While we rely on random choices for the inputs, we restrict the choices of channels (C_M) such that each tree observes only 2 (rather than up to 8) input channels. This has the two advantages of enforcing spatial comparisons of values within one channel and is more efficient in hardware circuit designs. When creating hardware designs, for larger models, routing could become a problem due to congestion when connections between channels follow an arbitrary order. Thus, we restrict the connections between channels to ensure proper routing: we split the model into $\texttt{k}/8$ groups, ensuring no cross-connections between the channels of each group. This restriction as well as similar hardware specific routing restrictions can be implemented without affecting the accuracy due to the sparsity of the logic gate network model. 5 8

3.5 Input Processing 9

For our smaller CIFAR-10 models (S, M), we use 2 bit precision inputs, and encode them using 3 thresholds as in [7]. For our larger CIFAR-10 models (B, L, G), we use 5 bit precision inputs, and process them with low-level feature detectors, in particular, we use edge and curvature detector kernels with thresholds, converting them into binary encodings, which are converted into LGNs and not learned. We note that the gates for the input preprocessing are included in each of the gate counts. 10

4 Related Work

Beyond differentiable LGNs [7], [15] (covered in Section 2), the related work comprises truth table networks [16], [17], binary and quantized neural networks [2], [3], and sparse neural networks [4].

Lookup / Truth Table Networks Lookup table networks (aka. truth table networks) are networks comprised of lookup tables (LUTs) or equivalently (potentially complex) logic gates with n inputs.

There are different approaches for learning or constructing lookup table networks. Chatterjee [16] constructs truth table networks by "memorizing" training data in an explorative work to consider relations between memorization and generalization. Wang *et al.* [18], [19] replace the multiplication in BNNs by lookup tables (LUTNet). Benamira *et al.* [17] transform Heaviside step function activated CNNs into lookup tables by expressing the binary activation of each neuron via a lookup table that implicitly encodes the weight matrix (TTNet). This allows obtaining the binary activation of a neuron by "looking up" a value from the truth table at a location encoded via the binary inputs of the layer. Benamira *et al.* [17] use this as an intermediate representation to then convert the truth tables into LGNs via CNF/DNF (conjunctive / disjunctive normal form) conversion. The resulting LGNs allow for efficient and effective formal verification. These resulting LGNs differ from the LGNs considered in this work because they are derived from a conventional CNN and not directly learned, thereby having the inductive bias of the neural network architecture (matmul) and its computational overhead, which is similar to BNNs converted into LGNs. We remark that, while TTNets are LGNs, TTNets are not differentiable LGNs as there is no differentiable representation of LGNs involved. Recently, Bacellar *et al.* [20] extended differentiable LGNs to learning logic gates with more than two inputs.

Binary and Quantized Low-Precision Networks BNNs and quantized neural networks reduce the precision of the weight matrices of a neural network. For example, BNNs typically use the weights -1 and $+1$, but variations are possible. For quantized neural networks, a popular choice is 8-bit and other options (such as 4-bit [21]) are covered in the literature. This leads to substantially reduced storage requirements of neural networks at the cost of some accuracy. Instead of naïvely quantizing weights, these approaches typically involve, e.g., quantization-aware fine-tuning [3]. In addition, for some methods, BNNs and quantized neural networks also reduce the precision of the computations and activations, leading to speedups during inference [2], [3]. These approaches typically start with a conventional pre-trained neural network and then convert it into a low-precision representation. BNNs are among the fastest approaches for efficient inference [2].

While BNNs (with binary activations, e.g., XNOR-Net [22]) are converted into LGNs for inference on hardware (e.g., on FPGAs [23]), the resulting architectures are fundamentally different from directly trained logic gate networks. BNNs have weight matrices and require multiply-accumulate (MAC) operations to express matrix multiplications. Asymptotically, each MAC requires 8 logic gates while at the same time (with only 2 possible states of the weight) this leads to a smaller expressivity compared to a single learned logic gate (with 16 possible states). We include a technical discussion in the appendix. While it is disadvantageous for inference, for training, BNNs have the advantage of operating on a higher abstraction level, simplifying training and allowing for translation between conventional neural networks and BNNs. We remark that BNNs with binary input activations and binary weight quantization frequently do not use binary output activations [24], which means that only the multiplications within a matrix multiplication are binary, while the remainder of the respective architectures can require floating precision. In contrast to BNNs, differentiable LGNs are not parameterized via weight matrices but instead via the choices of logic gates at each node [7].

Sparse Neural Networks Sparse neural networks are networks that are not densely connected but instead have only selected connections between layers [4], [25], [26]. Conceptually, this means multiplying a weight matrix with a binary mask, setting a selection of weights to 0. Sparse nets can be utilized for efficient inference as the sparsity greatly reduces the number of floating-point operations that have to be executed. For an overview of sparse neural networks, we refer to Hoefler *et al.* [4].

Due to the binary (i.e., two-input) nature of logic gates, logic gate networks are intrinsically sparse. Thus, LGNs can be seen as sparse networks; however, sparse neural networks are typically not LGNs and typically operate on real values instead of Boolean values. As differentiable LGNs use randomly initialized and fixed connections, it is perhaps important to mention that choosing randomly initialized and fixed connections has been shown to also work well for conventional sparse neural networks [5].

5 Experiments

5.1 CIFAR-10

We train five sizes of LogicTreeNets on the CIFAR-10 data set [9] using the AdamW optimizer [12], [33] with a batch size of 128 at a learning rate of 0.02. Additional training details and hyperparameters are in Appendix A.2. We report our main results in Table 1 and Figure 1. Our primary evaluation is with respect to the number of logic gates (bin. ops), which corresponds to the cost in hardware implementations and is proportional to transistor count chip area for ASICs or occupancy on FPGAs.

Table 1: **Main results** for the CIFAR-10 experiments. Our LogicTreeNet models reduce the required numbers of logic gates by factors of $\geq 29\times$ compared to the state-of-the-art models. Our models are scaled to match accuracies.

Method	Acc.	# Gates
DiffLogic Net (medium) [7]	57.39%	0.51 M
DiffLogic Net (largest) [7]	62.14%	5.12 M
Conv. TTNet (small) [17]	50.10%	0.57 M
Conv. TTNet (large) [17]	**70.75%**	189 M
FINN CNV [23]	**80.10%**	901 M
LUTNet [19]	**84.95%**	1 290 M
XNOR-Net [22] (NIN) [27]	**86.28%**	1 780 M
RebNet (1 residual) [28]	80.59%	2 270 M
RebNet (2 residuals) [28]	85.94%	2 830 M
BinaryNet [29]	88.60%	4 090 M
Zhao et al. [30]	88.54%	4 940 M
FBNA CNV [31]	88.61%	5 540 M
Hirtzlin et al. [32]	91. %	87 400 M
LogicTreeNet-S	**60.38%**	0.40 M
LogicTreeNet-M	**71.01%**	3.08 M
LogicTreeNet-B	**80.17%**	16.0 M
LogicTreeNet-L	**84.99%**	28.9 M
LogicTreeNet-G	**86.29%**	61.0 M

Comparing our model (M) with 3.08 M gates to the large TTNet model [17], we can observe that, while the accuracies are similar, our model requires only 1.6% of the number of logic gates. Increasing the model size, our model (B) matches the accuracy of FINN [23], while requiring only 16 M gates compared to 901 M gates, a $56\times$ reduction. Considering an even larger variant of our model (L) with 28.9 M gates, we achieve 84.99%. The smallest baseline model that achieves comparable accuracy (84.95%) is LUTNet [19], which requires $44.6\times$ as many logic gates. Finally, considering our largest model (G) with 61 M logic gates, we achieve 86.29% test accuracy. We match the accuracy of the Network-in-Network [27] XNOR-Net [22], while this baseline requires $29\times$ as many gates. Indeed, all networks in the literature below 4 billion gates perform worse than our 61 million gate network.

After covering the performance of the trained models, we demonstrate their applicability in hardware designs on a Xilinx FPGA as a proof-of-concept. On CIFAR-10 we limit the hardware development up to the base model (B) due to labor cost. In Table 2, we report the results. We can observe a very favorable FPGA timing trade-off compared to previous works. Indeed, using our model (B) we achieve 80.17% accuracy, matching the accuracy of the FINN accelerator, but decreasing inference time from 45.6 μs to 24 ns. In other words, our model achieves 41.6 million FPS, whereas the previously fastest FPGA model achieved 22 thousand FPS (even among all models with $\geq$70%). Herein, the limitation preventing us from reaching around 500 million FPS is the transfer speed onto the FPGA. Here, the difference between the smaller models (S & M) and the larger model (B) is that (S & M) receive the input at 2 bit precision whereas (B) receives the input at 5 bit precision. We want to remark that substantially accelerated speeds or reduced power consumption could be achieved by manufacturing custom hardware such as ASICs; however, this lies out of the scope of this work and is an interesting future research direction.

Table 2: Timing results for CIFAR-10. The time is per image on an FPGA. We use a Xilinx VU13P FPGA. Our times are bottleneck by the data transfer onto the FPGA. 'A' indicates the use of an ASIC.

Method	Acc.	FPGA t.
FINN CNV [23]	80.10%	45.6 μs
RebNet (1 residual) [28]	80.59%	167 μs
RebNet (2 residuals) [28]	85.94%	333 μs
Zhao et al. [30]	88.54%	5.94 ms
FBNA CNV [31]	88.61%	1.92 ms
FracBNN [34]	89.10%	356 μs
TrueNorth [35]	83.41%	A: 801μs
LogicTreeNet-S	60.38%	9 ns
LogicTreeNet-M	71.01%	9 ns
LogicTreeNet-B	80.17%	24 ns

We remark that all accuracies reported in the main paper are from discretized LGNs, and all gate counts maintain the full convolutional character (no location-based simplifications, e.g., at zero-padding). In Appendix A.4, we include a plot comparing the differentiable training mode accuracy to the discretized inference mode accuracy. Further, we refer to Figure 1 for a comparison of LogicTreeNet compared to the pareto-front of the state-of-the-art.

5.2 MNIST

We continue our evaluation on MNIST [8]. Here, we use a slightly smaller model architecture with only 3 (instead of 4) convolutional blocks due to the input size of 28×28. Each convolutional block has a depth of 3 and, to maintain valid shapes, we use no padding in the first convolutional block. Each block increases the number of channels by a factor of 3. This network architecture is described in greater detail in Appendix A.1.2.

Table 3: Results of the MNIST experiment. We use a Xilinx XC7Z045 FPGA, the same device as FINN CNV. All other baselines utilize equivalent or more powerful FPGAs.

Method	Acc.	# Gates	FPGA t.
DiffLogic Net (small) [7]	97.69%	48 K	—
DiffLogic Net (largest) [7]	98.47%	384 K	—
DWN [20]	98.77%	—	45 ns
TTNet (small) [17]	97.23%	46 K	—
TTNet [17]	98.02%	360 K	—
LUTNet [19]	98.01%	—	5 ns
FINN CNV [23]	98.40%	5.28 M	641 ns
FINN FCN [23]	98.86%	258 M	—
LowBitNN [36]	99.2 %	—	152 μs
FPGA-NHAP [37]	97.81%	—	4.9 ms
LogicTreeNet-S	98.46%	147 K	4 ns
LogicTreeNet-M	99.23%	566 K	5 ns
LogicTreeNet-L	99.35%	1.27 M	—

We display the results for MNIST in Table 3. Here, our models achieves a range of new SOTAs: compared to FINN [23], we can observe that our small model already improves the accuracy while simultaneously decreasing the model size by a factor of $36\times$, and reducing inference time by a factor of 160. Our medium model, with 99.23% test accuracy improves over all BNNs in the literature. When comparing to LowBitNN [36], a non-binary model, our medium model reduces the inference time by a factor of $30\,000\times$ while still improving accuracy, increasing throughput from 6 600 FPS to 200 000 FPS.

Within the, "one-classification-per-cycle" regime, comparing to LUTNet [19], we decrease the error from 1.99% to 0.77%, and we note that the larger FPGA that LUTNet uses should enable placing LogicTreeNet-L (0.65% error) multiple times, enabling multiple classifications per cycle.

Concluding, our MNIST models are both the most efficient models in the $\geq 98\%$ regime and at the same time also the highest accuracy models with an accuracy of up to 99.35%.

Variances For small models like the small (S) model for MNIST, which has only 16 kernels in the first layer, variance due to the fixed connectivity can become a significant concern. Thus, for the small models we train multiple models simultaneously, and use a validation set of 10 000 images that we hold-out from the training set (not the test set), and based on which we select the final models. We present the variations before this selection between individual model in Table 4. We can see that with increasing model size, the variance decreases.

Table 4: Variances between individual models on MNIST.

Model	Individual accs.
S	$98.21\% \pm 0.31\%$
M	$99.13\% \pm 0.11\%$
L	$99.29\% \pm 0.06\%$

5.3 Ablation Study

To demonstrate the importance of the provided architectural choices, we provide an ablation study in Table 5. Here, we observe that using trees, residual initializations, as well as *or* pooling are integral to the performance of convolutional LGNs. We also provide an ablation wrt. model depth.

Table 5: Ablation study on CIFAR-10 wrt. architectural choices.

Method	Accuracy	# train. layers	# total layers	*or*-pool	residual init.	weight decay	2 input channels
LogicTreeNet-L	84.99%	15	23	✔	✔	✔	✔
Conv. d: 1,1,1,1	80.98%	7	15	✔	✔	✔	✔
Conv. d: 1,1,2,2	82.68%	9	17	✔	✔	✔	✔
Conv. d: 2,2,2,2	83.32%	11	19	✔	✔	✔	✔
Conv. d: 2,2,3,3	84.13%	13	21	✔	✔	✔	✔
No *or* pooling	81.45%	15	15	✘	✔	✔	✔
Gaussian init.	76.18%	15	23	✔	✘	✔	✔
No weight decay	83.94%	15	23	✔	✔	✘	✔
8 input channels	83.53%	15	23	✔	✔	✔	✘

Starting with the model depth ablation, in Table 5, we can observe that the performance improves with increasing model depth. We observe that decreasing the model depth is detrimental to performance. We note that shallower

models do not directly correspond to reductions in gate counts because, for deeper models, the rates of trivial gate choices like 'A' that are removed during logic synthesis is significantly higher.

Next, we consider the omission of *or* pooling. We can observe that the accuracy drops by 3.5% when removing *or* pooling, demonstrating its importance. Setting weight decay to 0 causes a small reduction in accuracy by 1%. Allowing each tree to use 8 channels as the input, rather than just 2, reduces the accuracy (1.4%) because it is better to enforce the ability to perform comparisons within one channel at different x, y locations in the kernel. However, the more important effect of using only 2 input channels is the resulting improved routing in hardware design layouts.

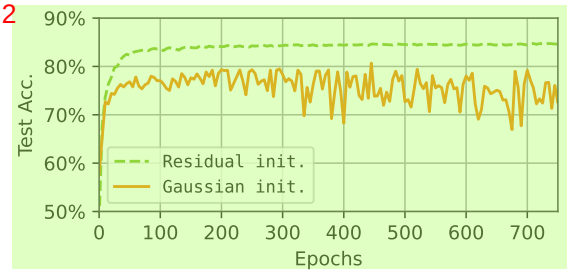


Figure 7: Residual initializations (green) drastically stabilize training of the LogicTreeNet compared to Gaussian initialization (orange).

Finally, we ablate the proposed residual initializations. We can observe in the table that the accuracy drops by almost 9% without residual initializations. This means the that the Gaussian initialization are almost unusable for such deep networks. In Figure 7, we display the test accuracy during training and observe that, without our residual initializations, training does not converge and is quite unstable.

We further ablate the effect of residual initialization on the distribution of gates in Figure 8. Here, we can observe that residual initializations not only stabilize training, but also lead to the favorable inductive bias of many gates being the 'A', which is automatically reduced during logic simplification.

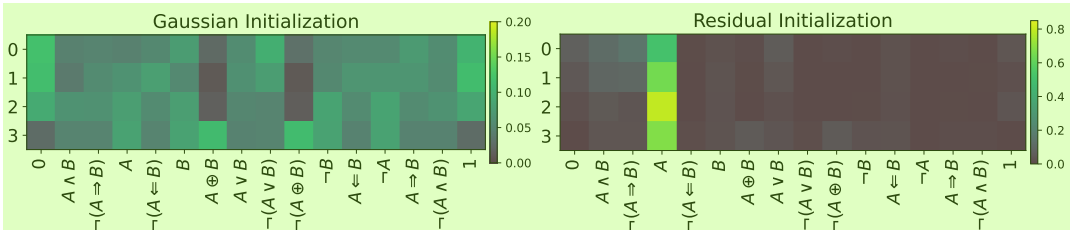


Figure 8: Distributions of choices of logic gates in a trained MNIST model, comparing Gaussian (left) and residual (right) initializations. The row number indicates the layer and the column indicates the logic gate.

6 Conclusion

In this paper, we introduced convolutional differentiable logic gate networks with logic gate tree kernels, integrating a range of concepts from machine vision into differentiable logic gate networks. In particular, we introduced residual initializations, which not only reduces loss of information in deeper networks, but also prevents vanishing gradients, enabling training of deeper LGNs than previously possible. Further, we introduced logical *or* pooling, which, combined with logic tree kernels, substantially improved training efficiency. Our proposed CIFAR-10 architecture, LogicTreeNet, decreases model sizes by factors of $\geq 29\times$ compared to the SOTA while improving accuracy. Further, our inference stack demonstrates that convolutional LGNs can be efficiently executed on hardware. For example, on MNIST, our model improves accuracy while achieving $160\times$ faster inference speed, and on CIFAR-10, our model improves inference speed by $1900\times$ over the state-of-the-art. An interesting direction for future research is applying convolutional differentiable logic gate networks to computer vision tasks with continuous decisions like object localization. We hope that our results motivate the community to adopt convolutional differentiable LGNs, especially for embedded and real-time applications where inference cost and speed matter most.

Acknowledgments and Disclosure of Funding

This work was supported in part by the Federal Agency for Disruptive Innovation SPRIN-D. CB is supported by the Land Salzburg within the WISS 2025 project IDA-Lab (20102-F1901166-KZP and 20204-WISS/225/197-2019). SE is supported by the ARO (W911NF-21-1-0125), the ONR (N00014-23-1-2159), and the CZ Biohub. We thank the reviewers for their supportive and helpful comments.

References

[1] R. Desislavov, F. Martinez-Plumed, and J. Hernandez-Orallo, "Compute and energy consumption trends in deep learning inference," *Computing Research Repository (CoRR) in arXiv*, 2021.

[2] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, "Binary neural networks: A survey," *Pattern Recognition*, vol. 105, p. 107 281, 2020.

[3] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," *Computing Research Repository (CoRR) in arXiv*, 2021.

[4] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *The journal of machine learning research (JMLR)*, 2021.

[5] S. Liu, T. Chen, X. Chen, L. Shen, D. C. Mocanu, Z. Wang, and M. Pechenizkiy, "The unreasonable effectiveness of random pruning: Return of the most naive baseline for sparse training," in *International Conference on Learning Representations (ICLR)*, 2022.

[6] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," in *International Conference on Learning Representations (ICLR)*, 2017.

[7] F. Petersen, C. Borgelt, H. Kuehne, and O. Deussen, "Deep Differentiable Logic Gate Networks," in *Proc. Neural Information Processing Systems (NeurIPS)*, 2022.

[8] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," 2010. [Online]. Available: `http://yann.lecun.com/exdb/mnist`.

[9] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," 2009.

[10] E. van Krieken, E. Acar, and F. van Harmelen, "Analyzing differentiable fuzzy logic operators," *Computing Research Repository (CoRR) in arXiv*, 2020.

[11] G. J. Klir and B. Yuan, *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice Hall, 1997.

[12] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference on Learning Representations (ICLR)*, 2015.

[13] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proc. Neural Information Processing Systems (NeurIPS)*, 2012.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[15] F. Petersen, "Training a neural network to perform a machine learning task," in *WIPO (PCT), WO2023143707A1, EP2022/051710, TW112101045A, CN202280089954.7A, KR1020247028676A, etc*, 2022.

[16] S. Chatterjee, "Learning and memorization," in *International Conference on Machine Learning (ICML)*, 2018.

[17] A. Benamira, T. Guérand, T. Peyrin, T. Yap, and B. Hooi, "A scalable, interpretable, verifiable & differentiable logic gate convolutional neural network architecture from truth tables," *Computing Research Repository (CoRR) in arXiv*, 2023.

[18] E. Wang, J. J. Davis, P. Y. K. Cheung, and G. A. Constantinides, "LUTNet: Rethinking inference in FPGA soft logic," in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2019.

[19] E. Wang, J. J. Davis, P. Y. K. Cheung, and G. A. Constantinides, "LUTNet: Learning FPGA configurations for highly efficient neural network inference," *IEEE Transactions on Computers*, 2020, to appear.

[20] A. T. Bacellar, Z. Susskind, M. Breternitz Jr, E. John, L. K. John, P. Lima, and F. M. França, "Differentiable weightless neural networks," in *International Conference on Machine Learning (ICML)*, 2024.

[21] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks," *Computing Research Repository (CoRR) in arXiv*, 2018.

[22] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *Proc. European Conference on Computer Vision (ECCV)*, 2016.

[23] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 65–74.

[24] A. Santos, J. D. Ferreira, O. Mutlu, and G. Falcao, "Redbit: An end-to-end flexible framework for evaluating the accuracy of quantized cnns," *Computing Research Repository (CoRR) in arXiv*, 2023.

[25] S. Liu, Y. Tian, T. Chen, and L. Shen, "Don't be so dense: Sparse-to-sparse gan training without sacrificing performance," *International Journal of Computer Vision*, 2023.

[26] S. Liu, T. Chen, X. Chen, X. Chen, Q. Xiao, B. Wu, T. Kärkkäinen, M. Pechenizkiy, D. Mocanu, and Z. Wang, "More convnets in the 2020s: Scaling up kernels beyond 51x51 using sparsity," in *International Conference on Learning Representations (ICLR)*, 2023.

[27] J. Yu, *PyTorch Implementation of XNOR-Net*, Archived in the Internet Archive on 2023/05/11 8:48, 2023. [Online]. Available: `https://github.com/jiecaoyu/XNOR-Net-PyTorch`.

[28] F. K. Mohammad Ghasemzadeh Mohammad Samragh, "Rebnet: Residual binarized neural network," in *Proceedings of the 26th IEEE International Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '18, 2018.

[29] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Proc. Neural Information Processing Systems (NeurIPS)*, 2016.

[30] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb. 2017.

[31] P. Guo, H. Ma, R. Chen, P. Li, S. Xie, and D. Wang, "Fbna: A fully binarized neural network accelerator," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

[32] T. Hirtzlin, B. Penkovsky, M. Bocquet, J.-O. Klein, J.-M. Portal, and D. Querlioz, "Stochastic computing for hardware implementation of binarized neural networks," *IEEE Access*, vol. 7, pp. 76 394–76 403, 2019.

[33] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," in *International Conference on Learning Representations (ICLR)*, 2019.

[34] Y. Zhang, J. Pan, X. Liu, H. Chen, D. Chen, and Z. Zhang, "Fracbnn: Accurate and fpga-efficient binary neural networks with fractional activations," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021.

[35] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, *et al.*, "Convolutional networks for fast, energy-efficient neuromorphic computing," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 113, no. 41, p. 11 441, 2016.

[36] J. Zhan, X. Zhou, and W. Jiang, "Field programmable gate array-based all-layer accelerator with quantization neural networks for sustainable cyber-physical systems," *Software: Practice and Experience*, 2020.

[37] Y. Liu, Y. Chen, W. Ye, and Y. Gui, "FPGA-NHAP: A General FPGA-Based Neuromorphic Hardware Acceleration Platform With High Speed and Low Power," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 69, no. 6, pp. 2553–2566, 2022.

[38] Z. Tu, X. Chen, P. Ren, and Y. Wang, "AdaBin: Improving Binary Neural Networks with Adaptive Binary Sets," in *Proc. European Conference on Computer Vision (ECCV)*, 2022.

[39] R. Gong, X. Liu, S. Jiang, T. Li, P. Hu, J. Lin, F. Yu, and J. Yan, "Differentiable soft quantization: Bridging full-precision and low-bit neural networks," in *Proc. International Conference on Computer Vision (ICCV)*, 2019.

[40] H. Qin, R. Gong, X. Liu, M. Shen, Z. Wei, F. Yu, and J. Song, "Forward and Backward Information Retention for Accurate Binary Neural Networks," in *Proc. International Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[41] H. Qiu, H. Ma, Z. Zhang, Y. Gao, Y. Zheng, A. Fu, P. Zhou, D. Abbott, and S. F. Al-Sarawi, "Rbnn: Memory-efficient reconfigurable deep binary neural network with ip protection for internet of things," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[42] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients," *Computing Research Repository (CoRR) in arXiv*, 2016.

[43] M. Shen, X. Liu, R. Gong, and K. Han, "Balanced binary neural networks with gated residual," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020.

[44] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Proc. Neural Information Processing Systems (NeurIPS)*, 2019.

A Implementation Details

A.1 Model Architecture Details

In this section, we discuss the convolutional LGN architectures in detail.

A.1.1 CIFAR-10 Architecture

In the following, we describe the model for CIFAR-10 from Figure 6 layer by layer:

- A convolutional block with k kernels with a receptive field of size 3×3 and tree depth $d = 3$, i.e., each kernel is a logic gate tree with seven logic gates, mapping 8 inputs to one output.
- An `or` pooling layer with kernel size 2×2 and stride 2. [shape after layer: $\texttt{k} \times 16 \times 16$]
- A convolutional block with `4*k` kernels with a receptive field of size 3×3 and depth $d = 3$.
- An `or` pooling layer with kernel size 2×2 and stride 2. [shape after layer: $\texttt{4*k} \times 8 \times 8$]
- A convolutional block with `16*k` kernels with a receptive field of size 3×3 and depth $d = 3$.
- An `or` pooling layer with kernel size 2×2 and stride 2. [shape after layer: $\texttt{16*k} \times 4 \times 4$]
- A convolutional block with `32*k` kernels with a receptive field of size 3×3 and depth $d = 3$.
- An `or` pooling layer with kernel size 2×2 and stride 2. [shape after layer: $\texttt{32*k} \times 2 \times 2$]
- Flattening the hidden state. [shape after flattening: `128*k`]
- Regular differentiable logic layer $\texttt{128*k} \to \texttt{1280*k}$ $^{(*)}$.
- Regular differentiable logic layer $\texttt{1280*k} \to \texttt{640*k}$ $^{(*)}$.
- Regular differentiable logic layer $\texttt{640*k} \to \texttt{320*k}$ $^{(*)}$.
- GroupSum with 10 classes $\texttt{320*k} \to 10$.

(All convolutional blocks are zero-padded with padding of size 1 to maintain respective shapes.)

$^{(*)}$: For the B & L size CIFAR models, we use $2\times$ as many gates in the final layers.

An additional implementation detail is that we can implement the last 3 layers in a fused fashion, using a single convolutional block/layer with depth $d = 3$, and only a single kernel application, which is functionally equivalent and faster due to our fused kernels that are available for convolution.

A.1.2 MNIST Architecture

The architecture for MNIST–like data sets has to account for the smaller input sizes (28×28). Thus, we use only 3 instead of 4 convolutional blocks for this architecture.

- A convolutional block with k kernels with a receptive field of size 5×5 and tree depth $d = 3$, without padding.
- An `or` pooling layer with kernel size 2×2 and stride 2. [shape after layer: $\texttt{k} \times 12 \times 12$]
- A convolutional block with `3*k` kernels with a receptive field of size 3×3 and depth $d = 3$.
- An `or` pooling layer with kernel size 2×2 and stride 2. [shape after layer: $\texttt{3*k} \times 6 \times 6$]
- A convolutional block with `9*k` kernels with a receptive field of size 3×3 and depth $d = 3$.
- An `or` pooling layer with kernel size 2×2 and stride 2. [shape after layer: $\texttt{9*k} \times 3 \times 3$]
- Flattening the hidden state. [shape after flattening: `81*k`]
- Regular differentiable logic layer $\texttt{81*k} \to \texttt{1280*k}$ $^{(*)}$.
- Regular differentiable logic layer $\texttt{1280*k} \to \texttt{640*k}$ $^{(*)}$.
- Regular differentiable logic layer $\texttt{640*k} \to \texttt{320*k}$ $^{(*)}$.
- GroupSum with 10 classes $\texttt{320*k} \to 10$.

$^{(*)}$: For the S & M size MNIST models, we use $2\times$ as many gates in the final layers.

A.2 Training Details

In Table 6, we summarize the hyperparameters for each model architecture configuration. We observe that the hyperparameter that depends the most on the data set is the learning rate η. The temperature τ and thus the range of attainable outputs $n_{\ell\ell/c}/\tau$ has a minor dependence on the data set. We use weight decay only for the CIFAR-10 models as it does not yield advantages for the smaller MNIST models. We note that convergence, when training with weight decay, is generally slightly slower but leads to slightly better models. Models trained wiht weight decay tend to have more gates.

Table 6: Hyperparameters for each model and data set: softmax temperatures τ, learning rates η, weight decays β, and batch sizes bs. For reference to show the relationship to τ, we include the number of output neurons in the last layer per class $n_{\ell\ell/c}$. The range of attainable class scores is $[0, n_{\ell\ell/c}/\tau]$.

Data set		CIFAR-10					MNIST		
Model identifier		S	M	B	L	G	S	M	L
Model scale	k	32	256	512	1 024	2 560	16	64	1 024
Temperature	τ	20	40	280	340	450	6.5	28	35
Learning rate	η	0.02	0.02	0.02	0.02	0.02	0.01	0.01	0.01
Weight decay	β	0.002	0.002	0.002	0.002	0.001	0	0	0
Batch size	bs	128	128	128	128	128	512	256	128
Output gate factor	ox	1	1	2	2	1	2	2	1
# input bits		2	2	5	5	5	1	1	1
Outputs/class	$n_{\ell\ell/c}$	1K	8K	32K	64K	80K	1K	4K	8K
Max score	$n_{\ell\ell/c}/\tau$	51	205	117	193	182	158	146	234

For the loss, we use different softmax temperatures τ depending on the model size. We observe two important relationships for choosing τ: (i) τ depends on the number of output neurons (larger number of output neurons $\Rightarrow$ larger τ) and (ii) τ depends on how certain the model will be after training on the respective dataset, i.e., for a hard task with low accuracy, we should choose a larger τ, while, for an easier task with a higher accuracy, we should choose a smaller τ. The reason for this is that cross-entropy requires smaller logit variances if the model is less certain and requires larger logit variances if a prediction is certain. A good rule of thumb during scaling is that the optimal temperature is proportional to the square-root of the number of output gates ($\tau^\star \propto \sqrt{n_{\ell\ell/c}}$).

For the CIFAR-10 B, L, and G models, we use a neural network teacher, supervising the class scores. When using a teacher on the class score level, a good rule of thumb is to increase the softmax temperature by a factor of $\sqrt{2}$.

For CIFAR-10, we split the training data into 45 000 training images and 5 000 validation images, and evaluate every 2 000 steps to select the best model. For MNIST, we split the training data into 50 000 training images and 10 000 validation images, and evaluate every 5 000 steps to select the best model.

A.2.1 Memory Access Advantages through Fused Trees and Pooling

Using logic gate trees as convolutional kernels with pooling allows for a substantial speedup during training. For this, we fuse the entire tree as well as pooling into a single CUDA kernel operation. The reason behind this is two-fold, illustrated for the case of depth $d = 3$ and 2×2 pooling: (i) after reading the necessary 32 inputs, we perform $2 \times 2 = 4$ applications of the 7 learnable logic gates comprising the tree. Then, we apply the maximum t-conorm to pool the 4 outputs of the 4 tree applications to a single output value. Here, we do not need to read the intermediate results from memory but can instead keep them in registers. This prevents 28 additional memory read operations. (ii) as each set of 4 tree applications only has a single output after pooling, it is sufficient to write only this individual output (as well as the index of the pooling operation) to memory, saving 28 memory write operations, which are expensive. Further, this also reduces the memory footprint of training by a factor of around $10\times$. This procedure requires recomputing selected intermediate values within each block during the backward pass; however, the memory access savings offset this small additional computational cost.

A.2.2 Computational Requirements

The typical training time per epoch for the L model on a single NVIDIA RTX 4090 GPU is 30 seconds. Noteworthy is that $d = 3$ kernels are typically bottlenecked by CUDA compute cores, whereas $d = 2$ and $d = 1$ kernels are bottlenecked by memory bandwidth. While we explored $d = 4$ kernels, they (when fused) are very expensive ($> 10\times$) due to register pressure. Generally, but with very limited exploration, simply going from $d = 3$ to $d = 4$ did not improve performance / gate. $d = 4$ kernels can also be expressed, without fusing, using 2 logic gate tree layers; however, with this the memory consumption during training increases ($\approx 10\times$) which becomes a bottleneck.

A.3 Inference Details

For efficient hardware implementations, routing is an important consideration and the ability to place the LGN without congestion is paramount. Due to the sparsity of differentiable logic gate networks, limiting the choice of connections to a reasonable degree does not negatively affect accuracy. Accordingly, we select connections such that the model could be split into $\mathtt{k}/8$ separated models that are only recombined at the stage of output gates after accumulation, akin to using grouped convolutions with a constant number of groups throughout the network. This prevents congestions without reducing the accuracy. Additional routing restrictions can straightforwardly be implemented in differentiable LGNs without incurring performance penalties.

We illustrate the placement of our MNIST model (M) on a Xilinx XC7Z045 FPGA in Figure 9.

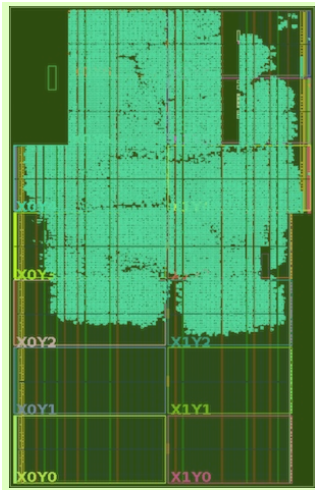


Figure 9: MNIST model (M).

We developed scalable logic synthesis tools that simplify the logic gate networks after training. For example, for our large MNIST model (L), during training, the main network has 3 216 128 gates. After training, in the discrete LGN, many of these gates are trivial (e.g., 'A' or constant) or not connected, and also further simplifications are possible. After logic synthesis, the number of logic gates was 697 758. Herein, the full convolutional nature of the network is maintained (and gates that have zero padded input, are still counted as full gates.) For the group sum operation, we use a tree adder that asymptotically uses 7 gates per output.

A.4 Train / Test Accuracy and Discretization Error

In Figure 10, we plot the training and test accuracies for training a convolutional LGN on CIFAR-10 [9]. Here, we can observe that the discretization error, i.e., the difference between the inference (hard) mode accuracy and the differentiable training mode accuracy is very small during late training. During early training, the discretization error is more substantial because the method first learns a "smooth" differentiable LGN (with high levels of uncertainty in the individual logic gate choices), which later converges to discrete choices for each logic gate. The discretization step chooses the logic gate with the highest probability for inference mode. Accordingly, during early training, the discretization causes larger changes, negatively affecting accuracy, while during late training, the discretization barely causes any changes and therefore does not considerably affect accuracy. We note that there is no noticeable overfitting behavior.

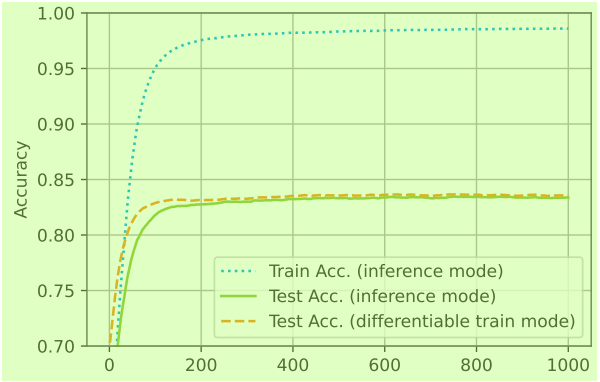


Figure 10: CIFAR-10 training and test accuracy plot. The discretization error, i.e., the difference between the inference (hard) mode accuracy and the differentiable training mode accuracy is very small during late training.

A.5 Ablation of the Residual Initialization Hyperparameter z_3

In Figure 11, we ablate the choice of z_3, which is the hyperparameter that indicates how strong the residual initialization is applied. We illustrate the ablation for an MNIST model. The model performs well, when $z_3 \geq 2$ ($z_3 = 1.5$ is included but reaches only 13%.) While $z_3 = 5$ is not the optimal choice for this particular model and training length, we have observed that, for larger models as well as for longer trainings, larger z_3 tend to be favorable. For example, on CIFAR-10, with the greater model depth, a z_3 of 2 is too small and prevents training, so we generally recommend using $z_3 = 5$.

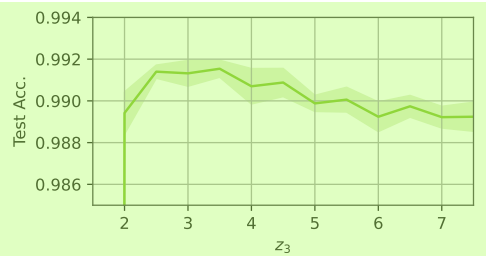


Figure 11: Test accuracy of an MNIST model with different choices of z_3 for the residual initialization, in steps of 0.5. Averaged over 5 seeds.

B Additional BMAC and BNN Discussions

B.1 BMACs in Logic

When translating a BNN into an LGN for inference on FPGAs, BNNs require BMACs (multiply-accumulate), which are often expressed via an XNOR ($\neg(a \oplus b)$) for the multiplication and a bitcount operation for the accumulation. In the case of n input bits, the necessary n MAC can be expressed using $\mathcal{O}(n) \approx 8 \times n$ logic gates: n logic gates are necessary for the XNORs and $\approx 7n$ logic gates for the accumulating bitcount. Further, this process adds a delay of $\mathcal{O}(\log n)$ to the logic circuit. This means that a BMAC is not one logical operation but instead typically requires around 8 binary logical operations (not accounting for additional thresholding or potentially batch-normalization operations).

B.2 Floats in BNNs

BNNs typically use 1-bit input activation and 1-bit weight quantization, but no output activation quantization [24]. This means that only the multiplications within a matrix multiplication are binary, while the remainder of the respective architectures is typically non-binary [24]. As in some residual BNN approaches [38]–[43], the residuals are not quantized, they require an additional FLOP (or integer operation) overhead between the layers; due to the cost of FLOPs in logic gates (>1000 binary OPs), effective deployment on a logic level has not been demonstrated for these unquantized residual approaches. Quantizing the residuals typically comes at a substantial accuracy penalty as demonstrated, e.g., by Ghasemzadeh *et al.* [28]. Thus, as these networks use full-precision residuals, a fair comparison is not applicable. Still, float-residual BNN approaches have important implications for speeding up GPU inference if (i) large matrix multiplications are necessary and (ii) the savings during the binary matrix multiplication itself outweigh the costs of converting between bit-wise representations and floats/integer representations; however, float-residual BNNs are not suitable for efficient logic gate based inference in hardware, e.g., on FPGAs or ASICs.

C List of Assets

- CIFAR-10 [9] [different open licenses]
- MNIST [8] [CC License]
- PyTorch [44] [BSD 3-Clause License]