An Initial Investigation on the Generation of Logic Gate Networks using Cartesian Genetic Programming

An Initial Investigation on the Generation of Logic Gate Networks using Cartesian Genetic Programming

Francesco Lubrano
Fondazione LINKS
Turin, Italy, IT
francesco.lubrano@linksfoundation.com

Emanuel Messetti
Independent Researcher
Turin, Italy, IT
emanuelmessetti@gmail.com

Alberto Scionti
Fondazione LINKS
Turin, Italy, IT
alberto.scionti@linksfoundation.com

Giovanni Squillero
Politecnico di Torino
Turin, Italy, IT
giovanni.squillero@polito.it

Alberto Tonda
Université Paris-Saclay,
AgroParisTech, INRAE, Applied Mathematics and Computer Science Lab (UMR 518 MIA-PS)
Palaiseau, Ile-de-France, FR
alberto.tonda@inrae.fr

Abstract

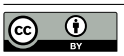
Logic Gate Networks (LGNs) are an appealing family of artificial intelligence models that efficiently perform inference, by replacing floating-point multiply–accumulate operations with Boolean ones. Despite their performance and efficiency, training still represents a problem: the selection and wiring of logic gates are discrete design choices, making the direct use of gradient-based optimization difficult, at least in its native form. This paper provides an initial investigation on whether Cartesian Genetic Programming (CGP) can be turned into a practical, gradient-free framework for evolving layered LGNs for image classification. To this purpose, we extend a CGP implementation with data handling functions specific for the modified National Institute of Standards and Technology (MNIST) dataset, redundant multi-bit output decoding, custom loss functions, batch-based evaluation, layered connectivity constraints, and adaptive mutation control. Experiments on binarized MNIST show that these design choices substantially improve the search process. In the final configuration, the proposed method reaches more than 80% test accuracy with a fully discrete evolutionary pipeline executed on a single processor core (CPU), providing a concrete indication that non-trivial logical classifiers can be obtained without back-propagation, Graphics Processing Units (GPUs), or continuous relaxations.

CCS Concepts

• **Computing methodologies** → **Computer vision**; **Genetic programming**; *Evolvable hardware*.

Keywords

Logic Gate Networks, Cartesian Genetic Programming, MNIST, Evolutionary Algorithms



GECCO Companion '26, San Jose, Costa Rica

ACM ISBN 979-8-4007-2488-6/2026/07
https://doi.org/10.1145/3795101.3814731

ACM Reference Format:
Francesco Lubrano, Emanuel Messetti, Alberto Scionti, Giovanni Squillero, and Alberto Tonda. 2026. An Initial Investigation on the Generation of Logic Gate Networks using Cartesian Genetic Programming. In *Genetic and Evolutionary Computation Conference (GECCO Companion '26), July 13–17, 2026, San Jose, Costa Rica.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3795101.3814731

1 Introduction

The increasing computational and energy cost associated with the training and inference of large artificial intelligence (AI) models has motivated a growing interest in designing more efficient architectures that offer a better trade-off between performance and cost over traditional Graphics Processing Units (GPUs). When focusing on resource-constrained scenarios, such as embedded systems [2, 3, 9], achieving high energy-efficient inference becomes imperative. Among the candidates, Logic Gate Networks (LGNs) stand out because they replace floating-point arithmetic with Boolean operators and bitwise computation, aligning the model directly with digital hardware primitives [7, 8]. In this regard, LGNs belong to a class of well studied approaches to make artificial neural networks achieving high throughput when executing on energy-efficient computing devices. Courbariaux et al. [1] introduced the idea of Binarized Neural Networks (BNNs), where almost all the operations can be reduced to binary ones, opening the door to potentially attain performance in the range of teraoperations per seconds on Field Programmable Gate Array (FPGA) devices. Umuroglu et al. [10] proposed a method that exploits the equivalence of artificial neurons with quantized inputs/outputs and truth tables to obtain neural networks that can be converted into very efficient lists of Boolean gates and their connectivity (i.e., netlists) on FPGAs.

In a LGN, each artificial neuron is represented by a two-input Boolean function, thus making the inference task extremely fast (low latency) and efficient. This high inference efficiency is counterbalanced by a major training challenge. The selection of the gate type at each node and the wiring between nodes (i.e., network topology) are discrete, combinatorial decisions. Recently, [7, 8] have shown that differentiable relaxations can make LGNs trainable with

gradient descent approaches, achieving strong results on image classification tasks. However, these methods still rely on continuous surrogate computations during training and benefit from highly optimized gradient-based implementations.

In this work, we investigate a complementary direction: the generation of LGNs through CGP, i.e., a graph-based form of genetic programming particularly suitable for discrete structures and circuit-like phenotypes [5, 6]. Our goal is not merely to maximize final accuracy, but to understand whether a discrete, gradient-free evolutionary pipeline can produce meaningful logic-based classifiers under limited computational resources.

The main contribution of this paper is as follows: we designed a CGP-based pipeline tailored for layered multi-class LGNs. We introduced problem-specific mechanisms, such as redundant output coding, custom loss shaping, large-batch evolution, layered connectivity constraints, and adaptive mutation scheduling, to make our approach being able to generate LGNs targeting image classification tasks. Finally, we provided an empirical study on binarized MNIST handwritten digit dataset, showing how these mechanisms affect performance and where the main bottlenecks of the approach remain. Experimental results show that a trade-off exists between the obtained LGN accuracy and the amount of computing resources used to generated the networks. Albeit the proposed approach does not match the accuracy of State-of-the-Art differentiable logic-based models, it clearly demonstrates that valid, non-trivial LGNs can be evolved with limited hardware resources.

2 Background

2.1 Logic Gate Networks

LGNs are layered computational graphs; each node in the graph computes a Boolean function of two inputs. Since neither floating-point weights are used nor matrix multiplications are performed, LGNs are intrinsically sparse (i.e., each neuron gets only two inputs instead of the entire previous layer output vector) and closer to a very efficient hardware implementation.

Recently Petersen et al. [7] introduced the idea of differentiable logic gate networks, where each node learns a probability distribution over a set of two-input Boolean functions, and which results in the selection of the most probable logic gate. This approach yields extremely fast inference, with reported throughput beyond 10^6 images per second, when classifying the MNIST on a single CPU. Extension of the approach to convolutional settings has been also shown in [8], where logic-tree kernels, logical OR pooling, and residual initializations significantly improved performance on more structured datasets, such as CIFAR-10.

2.2 Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) represents programs as directed acyclic graphs (DAGs) encoded as fixed-length integer genomes [6]. Each node is described by one *function gene* and a set of *connection genes*, while separated *output genes* define the program output. Interestingly, in a GCP genome, many genes can remain inactive, allowing for a many-to-one genotype–phenotype mapping (neutrality) and neutral drift during search [5]. This characteristic has been repeatedly associated with an effective search on circuit-like problems.

Compared with other genetic programming variants, CGP is well suited for Boolean circuit synthesis because it supports signal reuse, multiple outputs, compact encodings, and mutation-only optimization schemes [11]. As such, CGP is a good candidate for directly evolving LGNs in the Boolean domain.

3 Proposed approach

Our implementation builds on *CGP++*, a modern C++ based Cartesian Genetic Programming framework proposed by Kalkreuth and Bäck [4]. In this work, we made this framework capable of tackling the MNIST classification problem and fitting with the requirements of layered LGNs. Thus, we implemented a problem-specific data handling, binarized input processing, redundant multi-bit output decoding, custom loss function (see Section 3.1), batch-based evolution mechanism, layered connectivity constraints (see Section 3.2), and adaptive mutation control.

As we address a 10-class classification task, we binarized the MNIST dataset by flatting each image into a Boolean vector with a length of 784, and by setting a threshold on the pixel value equals to 127. Therefore, the resulting input tensors are compatible with bitwise logic operations. Each individual in the population encodes a DAG with 80,000 nodes arranged into 5 layers, where each node is one of 16 possible two-inputs Boolean functions [7]. The corresponding LGNs express the output classification over 500 bits, which are interpreted as 10 groups (i.e., classes) of 50 bits. As such, the chance the input image belongs to a given class is represented by a range of values rather than a single bit, thus easing the evolution process. Figure 1 shows a classification example using a LGN: input image is binarized and passed to a network of logic gates, which generates output bits for each class.

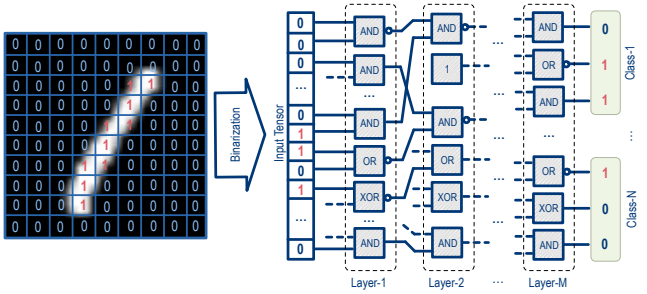


Figure 1: An example of a LGN inference: input image is binarized to form the input tensor; then, a network of logic gates organized in M layers provides output bits for N classes.

More specifically, a CGP individual is encoded as a structured sequence of integer values; Equation 1 provides the general form of the individual.

$$f_1\, S_{1,1}\, \ldots\, S_{1,a}\, \ldots\, f_n\, S_{n,1}\, \ldots\, S_{n,a}\, \ldots\, O_1\, O_2\, \ldots\, O_m \qquad (1)$$

Here, f_i selects the Boolean operator implemented by i-th node (with $i \in \{1, \ldots n\}$, being n the maximum number of nodes in the LGN); $S_{i,j}$ selects the j-th input source for the i-th node (with $j \in \{1, \ldots, a\}$, being a the arity of the node functions), while O_k specifies the source of the k-th output.

In this work, evolution follows a $(\mu + \lambda)$ strategy. At each generation, $\mu = 4$ parents produce $\lambda = 12$ offsprings by applying the

mutation operator; then all the 16 individuals are evaluated and the 4 best ones are retained as the next generation.

3.1 Multi-class decoding and Loss function definition

With the aim of facilitating the evolution process, classification output is expressed by a binary vector (i.e., output block) rather than a single bit. Let N_b be the number of output bits per class; then, the score associated to the class c is the number of bits set to 1 in its output block. Equation 2 shows how the prediction is achieved:

$$\hat{y} = \arg\max_{c} \sum_{j=1}^{N_b} b_{c,j}, \tag{2}$$

where $b_{c,j} \in \{0, 1\}$ is the j-th bit of the output block associated to class c, and $\hat{y}$ is the predicted class.

Given this scheme for representing the output classification for an input sample, let B_0 be the number of zeros in the output block of the correct class, and let B_1 be the number of ones in the output blocks of incorrect classes; then, we defined the *Loss* function l_i for the single input sample as follows:

$$l_i = \begin{cases} (B_0 \cdot p_{ma} + B_1) \cdot s_l, & \text{if } \hat{y}_i = y_i, \\ (B_0 \cdot p_{ma} + B_1), & \text{otherwise.} \end{cases} \tag{3}$$

and aggregate over a batch of size $\mathcal{B}_s$ to calculate the overall loss L:

$$L = \sum_{i=1}^{\mathcal{B}_s} l_i. \tag{4}$$

The parameter p_{ma} controls the penalty for missing activations in the output block of the correct class, while the parameter s_l scales the loss when the sample is already correctly classified (i.e., $\hat{y}_i = y_i$). Empirically, this design demonstrated to be more effective than using a purely bitwise loss(which reached only 41.80% accuracy without any further optimization), because it combines output-level correctness with internal bit-structure pressure.

3.2 CGP Framework Extensions

Evaluating each individual on the full training set at every generation is computationally expensive. To address this challenge, we introduced a batch-based evaluation as a first-class mechanism. A batch of training samples is loaded and kept active for a given number of generations before moving to a new batch. This simple approach substantially reduces per-generation cost and enables practical experimentation on MNIST.

Classical CGP uses a *levels-back rule* to constrain connections while preserving acyclicity. However, this does not guarantee a clean layered structure of the LGN. To better match with LGN architectures, we introduced three types of layered connectivity constraints:

- *Standard*: it follows the classical levels-back CGP rule.
- *Strict layered*: nodes connect only to the immediately previous layer.
- *Relaxed layered*: nodes connect to any previous layer, including the inputs.

Parameter	Value
Evolutionary strategy	(4 + 12)
Function nodes	80,000
Layers	5 (16,000 nodes per layer)
Function set	16 Boolean gates (2-input)
Output encoding	500 bits (50 bits/class)
Connectivity	Relaxed layered
Batch size ($\mathcal{B}_s$)	4,000
Mutation	Point mutation with 1/5 rule
Training schedule	46,000 generations + 55,000 fine-tuning

Table 1: Reference configuration used for the experimental campaign.

In our experiments, the relaxed layered connectivity turned out to be the most effective, because it preserves a fixed-depth architecture while allowing skip-like shortcuts across layers.

Mutation provides the main exploration mechanism. Small mutation rates preserve useful substructures but slow down exploration; large mutation rates aggressively explore the search space, but often disrupt active computational paths. To find a good balance between these two configurations, we adopted an adaptive schedule based on the 1/5 success rule. Every $G = 500$ generations, the mutation rate is scaled by a multiplicative factor $C = 0.9$ as follows: the mutation rate is multiplied by $1/C$ when the recent success rate exceeds 1/5, and by C otherwise. In practice, this allows to explore more at the beginning of the evolution process, and gradually decaying as the search moves into refinement.

4 Experimental Results and Analysis

Our experimental study investigated on four main aspects: loss shaping, batch size, mutation rate, and connectivity constraints. The LGNs were obtained by first evolving a baseline network for $G = 46,000$ generations until reaching the accuracy of 75% (checkpoint); then, we fine-tuned this baseline model using different loss function (i.e., p_{ma} and s_l parameters – see equation 3) and mutation settings. All the experiments used the common configuration reported in Table 1, and have been performed using a single process running on a single X86_64 CPU. The machine was equipped with 16 GiB of main memory, and run a standard Ubuntu 24.04 Linux distribution.

Hyper-parameters influence. Larger batches produced a more stable estimate of classification quality and reduced overfitting to the currently active subset of the training set. At the same time, smaller fixed mutation rates improved the final accuracy by preserving useful active sub-graphs. The downside was that convergence became slower. These trends are summarized in Table 2.

The benefit of using a structured connectivity is particularly strong. Standard layered connectivity produced irregular graphs with deep and fragile dependency chains, whereas the other layered connectivity approaches stabilized genotype–phenotype mapping and made mutations more localized. Relaxed layered connectivity performed better than strict layered connectivity because it preserves the layered prior while providing additional expressiveness through shortcuts.

	Configurations	Test accuracy (%)
Batch size	100 samples	49.0
	200 samples	56.0
	1,000 samples	58.0
	2,000 samples	62.9
	4,000 samples	65.0
Mutation rate	0.005	45.0
	0.0015	52.0
	0.0005	62.0
	0.00025	72.0
	$1/n_{\text{genes}}$	75.0
Connectivity	Standard	60.0
	Strict layered	75.0
	Relaxed layered	78.0

Table 2: Most relevant ablations; experiments considered all their possible combinations.

p_{ma}	s_l	Mutation	Generations	Accuracy (%)
10	0.75	Function-only	20,000	78.3
10	0.75	Point mutation	55,000	78.8
10	0.25	Point mutation	55,000	80.1

Table 3: Best fine-tuning configurations starting from the 75% checkpoint.

Fine-tuning. The best overall results were obtained through fine-tuning from a good checkpoint rather than training from scratch with a single fixed configuration.

Table 3 reports the best fine-tuning experiments. Interestingly, setting $s_l = 0.75$ provided the best results during training from scratch, while setting $s_l = 0.25$ became superior in late-stage fine-tuning. A plausible interpretation is that the earlier phase mainly benefits from bit-level organization pressure, while the later phase profits more from a stronger push toward correct class-level decisions. Function-only mutation is also noteworthy: although it did not achieve the overall highest accuracy, it reached an accuracy of 78.3% in only $20,000$ generations. This suggests that once a reasonable topology becomes available, changing only the Boolean gate can be a useful short-term refinement mechanism.

Computational cost. Despite being promising, our approach suffers from using limited computing resources. Indeed, all experiments were performed on a single CPU, without exploring any specific execution optimization. For instance, running an experiment with $\mathcal{B}_s = 4,000$, a $(4 + 12)$ evolution strategy, and $G = 1,500$ generations requires about 24 hours, which corresponds to approximately 57.6 seconds per generation. As we are interested in making our approach more effective on computing environments with limited resources (e.g., embedded systems), further optimizations of the evaluation stage are needed to make long-run experiments affordable; this is the target for future works.

4.1 Comparing with Differentiable Methods

As previously stated, the main goal of this work is to assess how much performance (in terms of accuracy) is lost when moving to a fully discrete and resource-limited training setup, rather than using conventional gradient-based approaches and less constrained computing resources. To this end, we compared our best experimental result (i.e., 80.1%) accuracy with State-of-the-Art Differentiable Logic-based Networks on the same classification task (MNIST [7]). While this latter reached 98.47% accuracy (97.69% accuracy with small networks), the use of a more computing demanding training method makes it less viable on embedded systems. Further, we consider our results as a starting point for future optimizations that will close the gap.

4.2 Final Remarks

By analysing the performed experiments, some final remarks can be drawn. First, the search space must be carefully shaped. A naive application of the standard CGP connectivity rule is insufficient; indeed, an appropriate layering of the LGN is not just an architectural convenience, but a key ingredient for search stability.

Second, the optimization dynamics are strongly phase-dependent. Larger batches, smaller mutations, and checkpoint-based fine-tuning become increasingly important as the search approaches better solutions. This is consistent with the discrete nature of the problem: unlike gradient-based methods, there is no smooth path for making tiny late-stage improvements.

Third, in our implementation, only a small fraction of the nodes in each layer become active in the final phenotype. This is consistent with the neutrality of CGP and suggests that future works could exploit this characteristic (sparsity) more directly by designing more effective mutation operators and making the evaluation of individuals more efficient.

Finally, a clear trade-off using limited computing resources and the 'goodness' of generated solutions in a given time window exists. In this regards, this work represents a starting point for evolving LGNs directly in the Boolean domain.

5 Conclusions

This paper presented a CGP-based framework for evolving layered LGNs aimed to tackle MNIST image classification task. The proposed method extends standard CGP with redundant multi-bit output decoding, custom loss shaping, batch-based training, layered connectivity constraints, and adaptive mutation control. The best configuration reached 80.1% test accuracy by using a discrete search process executed on a single CPU. As such, experimental results set the path for further optimizations and improvements, including fast evaluation process, more informed mutation operators, hybridization with local or differentiable refinement approaches.

As such, the CGP demonstrated to be a viable framework for generating non-trivial LGNs without back-propagation, while experimental results support the idea that evolutionary computation can play a meaningful role in the design of efficient logic-based machine learning models.

References

[1] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. In Advances in neural information processing systems (NeurIPS), Vol. 29.

[2] Alex de Vries. 2023. The growing energy footprint of artificial intelligence. Joule 7, 10 (2023), 2191–2194. doi:10.1016/j.joule.2023.09.004

[3] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In Proceedings of the IEEE conference on computer vision and pattern recognition. 2704–2713.

[4] Roman Kalkreuth and Thomas Bäck. 2024. CGP++: A Modern C++ Implementation of Cartesian Genetic Programming. In Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '24). ACM, New York, NY, USA, 13–22. doi:10.1145/3638529.3654092

[5] Abdul Manazir and Khalid Raza. 2019. Recent Developments in Cartesian Genetic Programming and its Variants. Comput. Surveys 51, 6 (Jan. 2019), Article 122. doi:10.1145/3275518

[6] Julian F. Miller. 2011. Cartesian Genetic Programming. Springer Berlin Heidelberg, Berlin, Heidelberg, 17–34. doi:10.1007/978-3-642-17310-3_2

[7] Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. 2022. Deep Differentiable Logic Gate Networks. arXiv:2210.08277 [cs.LG] https://arxiv.org/abs/2210.08277

[8] Felix Petersen, Hilde Kuehne, Christian Borgelt, Julian Welzel, and Stefano Ermon. 2024. Convolutional Differentiable Logic Gate Networks. arXiv:2411.04732 [cs.LG] https://arxiv.org/abs/2411.04732

[9] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. 2020. Binary neural networks: A survey. Pattern Recognition 105 (Sept. 2020), 107281. doi:10.1016/j.patcog.2020.107281

[10] Yaman Umuroglu, Yash Akhauri, Nicholas J. Fraser, and Michaela Blott. 2020. LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications. In arXiv preprint arXiv:2004.03021. Xilinx Research Labs, Dublin, Ireland. https://arxiv.org/abs/2004.03021

[11] Garnett Wilson and Wolfgang Banzhaf. 2008. A Comparison of Cartesian Genetic Programming and Linear Genetic Programming. In Genetic Programming (EuroGP 2008) (Lecture Notes in Computer Science, Vol. 4971). Springer Berlin Heidelberg, 182–193. doi:10.1007/978-3-540-78671-9_16