Politecnico di Torino

Politecnico di Torino

Computer Engineering - Artificial Intelligence and Data Analytics

2025/2026
Graduation Session March 2026

Evolutionary Techniques for Generating Logic Gate Networks

Supervisors:
Giovanni Squillero
Alberto Scionti
Alberto Tonda
Francesco Lubrano

Candidate:
Emanuel Messetti

Abstract

Artificial intelligence (AI) has become a leading field in many scientific and technical research and industrial activities. This trend is supported by increasingly complex AI models (e.g., deep neural networks, transformers, LLMs, etc.) that require large-scale computing infrastructures. The ever-increasing energy consumption of these infrastructures is driving research towards more efficient AI systems. In this sense, research aims to develop AI models whose execution (inference) does not require complex mathematical operations (e.g., matrix-matrix or vector-matrix floating-point multiplications), allowing for the use of simpler and more energy-efficient computing systems. Recently, Logic Gate Networks (LGNs) have emerged as an energy-efficient alternative to traditional models (DNNs, CNNs, etc.). A LGN consists of a set of logic gates (AND, OR, etc.) organized in a layered structure (network), with the aim of performing complex tasks such as classifying input data. Unlike more traditional artificial neural networks, in a LGN each neuron consists of a Boolean function implemented by one or more logic gates. The aim of this thesis is to address this context. Specifically, we will study the state-of-the-art of evolutionary algorithms, focusing on genetic programming heuristics (such as Cartesian genetic programming – CGP, etc.) to identify the best approach for generating LGNs.

Acknowledgements

Ringrazio profondamente il Professor Giovanni Squillero per avermi dato la possibilità di intraprendere questo percorso e per avermi guidato nel compiere scelte ponderate durante la ricerca.

Ringrazio l'azienda Links che mi ha affiancato durante questo percorso. In particolare, ringrazio i relatori Alberto Scionti, Francesco Lubrano e Alberto Tonda per aver supervisionato la mia attività e per avermi suggerito idee brillanti durante lo svolgimento del lavoro.

Ringrazio la mia famiglia, in particolare i miei genitori, per il loro supporto emotivo ed economico durante questi cinque anni, scommettendo sul mio percorso e sapendo già in cuor loro che l'avrei portato a termine.

Ringrazio i miei coinquilini Andrea, Stefano e Gabriele per aver portato spensieratezza in questi cinque lunghi anni e per avermi motivato anche nelle situazioni più difficili.

Ringrazio i miei amici d'infanzia Nicola, Alessandro e Andrea per la leggerezza e per aver sempre creduto in me.

Infine, ringrazio la mia fidanzata Chiara per essere stata sempre al mio fianco, per avermi sostenuto sia emotivamente che nello studio e per essere stata un'ancora durante gli ultimi quattro anni di questo cammino.

Table of Contents

List of Tables

List of Figures

Glossary

LGN
: Logic Gate Networks

AI
: Artificial Intelligence

GP
: Genetic Programming

CGP
: Cartesian Genetic Programming

MAC
: Multiply And Accumulate

DNN
: Deep Neural Network

CNN
: Convolutional Neural Network

DDLGN
: Deep Differentiable Logic Gate Network

DAG
: Directed Acyclic Graph

LUT
: Look-up Table

BNN

Binary Neural Network

NEQ

Neuron EQuivalent

HRR

Holland Royal Road

Chapter 1

Introduction

In recent years, AI has made extraordinary progress thanks to increasingly large and accurate models, such as DNN, CNN and Transformers. However, this growth in complexity comes at an increasing cost in terms of computation, latency, and energy [1], especially in "edge" scenarios like embedded devices, IoT and real-time systems, where memory, power, and autonomy are limited. To meet these needs, alternative architectures are being explored that drastically reduce the cost of inference by replacing floating-point operations with simpler and more efficient operators ([2], [3]). Among other approaches, LGNs are networks composed of elementary logic gates (AND, OR, XOR, NAND, etc.) organized in a layered structure to perform learning tasks such as classification ([4], [5]). LGNs aim to align optimization to the underlying hardware structure, directly exploiting the Boolean operators that compose CPUs, GPUs, FPGAs, and ASICs, with the goal of extremely fast inference even with few resources.

Even if inference is extremely cheap, training a LGN is considerably more challenging: selecting the operator for each node is a discrete, non-differentiable decision, so standard gradient-descent methods do not apply directly. A highly promising direction is to use evolutionary algorithms, especially GP [6]. In particular, CGP is traditionally used to develop computer programs encoded as directed graphs [7]. This representation is highly versatile and can be applied to many domains, including neural networks. By encoding LGNs as graphs, CGP can evolve both its structure and the type of logical operations. This approach is gradient-free and potentially highly parallelizable.

Within this thesis, the goal is to study the state-of-the-art of evolutionary algorithms with a focus on CGP heuristics, and to develop and evaluate procedures for generating and optimizing LGNs of increasing complexity, comparing their accuracy and efficiency against quantized/binarized DNN alternatives on open-source datasets.

Starting with the theoretical foundations, Chapter 2 of this work aims at

providing the relevant background. First, it presents LGNs, discussing their key properties and motivating their usage within this thesis. Then, it provides an overview of GP, outlining many relevant variants with a particular focus on CGP.

Chapter 3 reviews the current state-of-the-art on LGNs. First, we introduce the results of DDLGNs and then cover the recent extension to the convolutional approach.

Chapter 4 gives an overview of the proposed approach, describing the development framework and the main steps of the evolutionary algorithm. The chapter also illustrates how the solution was first validated on the classic Holland Royal Road Problem, before being applied to the MNIST digit recognition task.

To validate this methodology, Chapter 5 reports the experimental evaluation of the proposed approach on the MNIST classification task, investigating the impact of several hyperparameters on overall performance.

Following the experimental phase, Chapter 6 analyzes the behavior of the proposed configuration, focusing not only on the results, but also investigating the computational cost and comparing the final model with the current state-of-the-art.

Finally, Chapter 7 concludes the thesis by summarizing the experimental results on the MNIST dataset, discussing the limitations of the proposed CGP approach, and suggesting possible future works.

Chapter 2

Background

This chapter provides the background necessary to frame the contributions of this thesis. First, it introduces LGNs, discussing their computational model, key properties, and the motivations that make them attractive for efficient inference, especially in resource-constrained scenarios. Then, it presents GP as a class of evolutionary methods for program synthesis and discrete optimization, highlighting why gradient-free approaches are suitable for training LGNs. Finally, the chapter outlines the main GP variants relevant to this work and motivates the choice of CGP as the reference evolutionary framework adopted in the experimental part of the thesis.

2.1 Logic Gate Networks

A LGN is a layered network in which each "neuron" is, in reality, a two-input Boolean function implemented by a logic gate. Consequently, there are no real synaptic weights or matrix-vector multiplications: the entire computation is a composition of bitwise logical operators. As a result, LGNs are intrinsically sparse (each node has only two inputs), do not require separate activation functions (the non-linearity is in the logic gates), and can be evaluated with simple bitwise operations. In the literature [4], "differentiable" variants temporarily relax the logical operators to real-valued counterparts (e.g., probabilistic T-norm/T-conorm), allowing the use of gradient descent to choose, for each node, which logic gate to use. At inference, the network is discretized by selecting the most probable operator and returns to being LGN executable only with logic gates. This pipeline has shown very fast inference with competitive accuracy on classic benchmarks.

Recently, LGN have been extended to the convolutional paradigm [5], introducing kernels made by logic gate trees, logic (OR) pooling and "residual" initializations to stabilize the depth. State-of-the-art results have been achieved on the CIFAR-10

benchmark, demonstrating an order-of-magnitude reduction in gate count compared to conventional binarized neural network implementations.

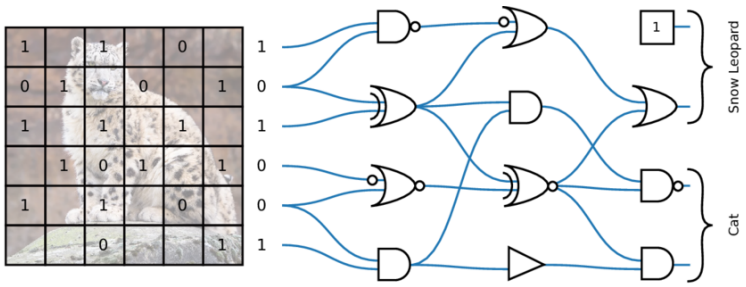

Figure 2.1: Image classification by a LGN.

Figure 2.1 illustrates the LGN architecture applied to a classification task. From it, we can appreciate the main differences with respect to a DNN [8]:

- No weights: there are no parameters to train, the network learns which gates and/or connections to use in each layer.
- Structural sparsity: each node receives only two inputs, not the entire previous vector. The complexity of a layer is $\Theta(n)$ with very small constants, instead of $\Theta(n \cdot m)$ in dense layers.
- A LGN uses Boolean operators instead of floating-point MACs, which allows for much faster execution. As an example, [4] has shown that LGNs are able to classify more than a million images of MNIST per second on a single CPU core.

Finally, we summarize the key advantages of leveraging LGNs, which serve as the primary motivation for this research:

- Inference efficiency: execution is entirely bitwise, on CPUs and especially on FPGAs/ASICs. Logic gates and bit-count operations are extremely economical, with throughput much higher than methods using binary weights.
- Minimal footprint: connections can be regenerated from a pseudo-random seed, and each node requires 4 bits to encode one of 16 possible Boolean functions; this drastically reduces the model's memory requirements.
- Suitable for the edge: the absence of floating-point operations (inference phase), low latency, and low power consumption make them ideal for embedded devices and accelerators, where energy, memory, and space budgets are constraints.

2.2 Genetic Programming

GP [6] is an evolutionary technique which evolves computer programs stochastically into optimized versions by using various operators such as selection, mutation, and crossover. In GP, each candidate solution is a program encoded using an internal representation, known as a genotype. Its quality is assessed by executing the program on a specific task and evaluating it via a fitness function. The search is stochastic and population-based: at each generation, a set of programs is iteratively modified through variation operators (e.g., mutation and crossover) and filtered through selection so that, on average, better programs become more frequent.

A standard GP workflow can be summarized as follows:

1. Initialization: generate an initial population by sampling random programs from a predefined *primitive set* (functions/operators, terminals/inputs, and possibly constants).

2. Evaluation: execute each program on the training set (or a subset) and compute its fitness.

3. Selection: choose parent programs based on fitness using different approaches (e.g., tournament selection, fitness-proportionate selection), optionally with elitism.

4. Variation: produce offspring by applying genetic operators to the parents' genomes, such as mutation (local edits) and/or crossover (recombination of substructures).

5. Replacement: build the next population from offspring and selected parents according to a generational or steady-state scheme.

6. Termination: stop when a fitness threshold is reached or a computational budget is exhausted.

The effectiveness of the GP approach lies in the separation between the genotype, i.e. the solution representation, and the phenotype, i.e. the actual program behavior. Selection operates on the phenotype by evaluating its fitness, while variation operators act exclusively on the genotype. This abstraction allows GP to explore complex search spaces by manipulating a simplified representation of the program structure. Consequently, one of the key design choices in GP is the program representation, since it determines (i) what kinds of structures can be expressed, (ii) how variation operators act, and (iii) the locality and redundancy of the genotype–phenotype mapping. Classical GP uses tree-structured programs where internal nodes are functions and leaves are terminals, as shown in the example in Figure 2.2.

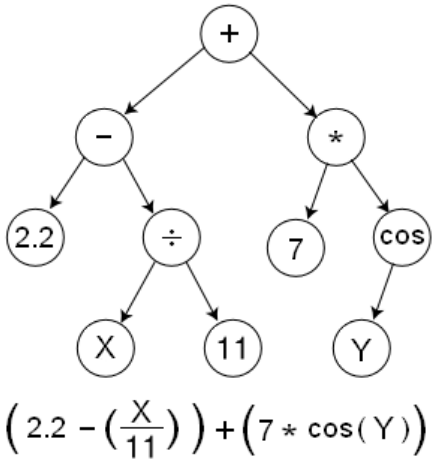

Figure 2.2: Classical GP tree-like individual.

Other variants adopt linear sequences of instructions, stacks, or graphs. In all cases, GP is particularly attractive when the objective is discrete and non-differentiable, because the optimization proceeds without requiring gradients. This property is central in this thesis, where an LGN can be seen as a compositional Boolean program: learning corresponds to selecting operators (gates) and wiring (connections), i.e., a combinatorial search problem.

2.2.1 Genetic Programming Variants

Over the years, many GP variants have been proposed, mainly differing in representation and, consequently, in variation operators and inductive biases [9]:

- **Tree-based or classical GP**: programs are syntax trees (Koza-style), well-suited to symbolic regression and expression discovery. It is often affected by *bloat*: the uncontrolled growth of program size over generations without a corresponding increase in fitness. Specific strategies must be applied to prevent it.

- **Linear GP** [10]: individuals are linear sequences of imperative instructions, often resembling assembly code. This can improve execution efficiency and makes it natural to target real processors, but it may require careful operator design to preserve syntactic validity and to maintain useful building blocks.

- **Graph-based GP**: programs are represented as directed graphs, enabling explicit reuse of intermediate values (fan-out) and compact representations for circuits and dataflow programs.

- **Grammar-based GP (e.g., Grammatical Evolution)** [11]: programs are generated from a grammar, allowing strict syntactic control and domain-specific languages.
- **Stack-based GP** [12]: programs operate on a stack (push/pop semantics), which can simplify variation but introduces its own execution model and biases.

For Boolean circuit synthesis and for LGNs in particular, graph-based representations are particularly suitable since they match the underlying computation model (composition of Boolean primitives), allow multi-output designs, and naturally capture the reuse of computed signals. Among graph-based GP approaches, CGP is widely adopted because it is simple, compact, and empirically competitive, while also exhibiting strong neutrality and limited bloat compared to classical GP [13].

2.2.2 Cartesian Genetic Programming

CGP is a form of GP in which programs are encoded as a fixed-length genome that maps to a feed-forward Directed Acyclic Graph (DAG). [13]. Historically motivated by evolvable hardware and digital circuit design, CGP represents a program as a grid of computational nodes arranged in columns (and optionally rows). Each node implements one function from a user-defined function set and receives its inputs from either primary inputs or from outputs of nodes in previous columns. The term "Cartesian" refers to this grid-based, coordinate-like organization.

Representation. In CGP, programs are represented by one- or two-dimensional DAG structures in which computational nodes embed the operations to perform. Consequently, the genotype is typically stored as a fixed-length sequence of integers that compactly specifies, for each node, both the operation it performs and its wiring. A common and convenient way to express this encoding is:

$$f_0 C_{01} \cdots C_{0a} \cdots\cdots f_j C_{j1} \cdots C_{ja} \cdots\cdots O_1 O_2 \cdots O_m \tag{2.1}$$

where f_j is the *function gene* of node j (i.e., it selects which operator the node implements; in this thesis f_j is an integer index into the allowed gate set), and C_{jk} are the *connection genes* of node j. Each C_{jk} specifies where the k-*th* input of node j comes from, with $k \in \{1, \ldots, a\}$ and a denoting the maximum arity among the available functions. In practice, each connection gene points either to a primary input or to the output of a previous node, thereby enforcing acyclicity in the decoded graph. Finally, O_i are the *output genes*: each O_i selects the source used as the i-*th* program output, with $i \in \{1, \ldots, m\}$, referencing either a primary input or a node output.

A common constraint is the *levels-back* parameter l, which limits how far back (in terms of previous columns) a node is allowed to connect, implicitly controlling the degree of locality in the search. In many practical implementations, the grid is flattened to a single row (i.e., a one-dimensional representation), and the number of columns becomes the maximum number of computational nodes. This simplifies both implementation and interpretation while preserving expressiveness for DAG encoding. Figure 2.3 shows a graphical example of a CGP program.

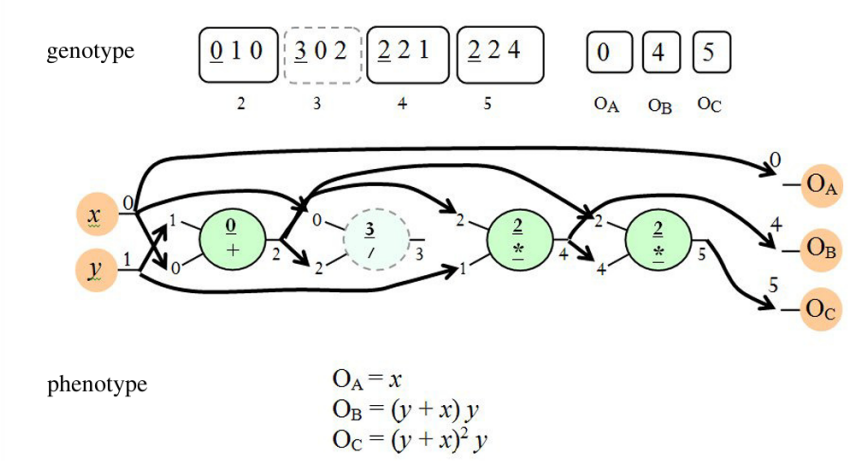

Figure 2.3: Example of a CGP program.

Genotype–phenotype mapping and neutrality. The phenotype (i.e., the executed program, here an LGN) is obtained by decoding the integer genome into a DAG and then identifying the *active* nodes, namely the subset of nodes that effectively contributes to the outputs. Operationally, active nodes are those reachable when tracing backward from the output genes O_i; conversely, nodes that are not reachable from any O_i are *inactive* (non-expressed). As a result, CGP exhibits a many-to-one genotype–phenotype mapping, where multiple distinct genomes can represent the same executed program. This redundancy induces **neutral drift**: mutations can modify inactive genes without changing the phenotype, enabling broad exploration of the search space while preserving current performance [13]. In practice, this mechanism is often considered one of the reasons CGP performs well on circuit-like problems and tends to avoid the bloat typical of tree-based GP.

Variation and evolutionary strategy. Most CGP systems rely primarily or often exclusively on point mutation: a small number of genes are randomly selected and replaced with valid alleles. This operation changes either a function gene, a connection gene, or an output gene. Unlike tree-based GP, where a single mutation

can lead to huge changes in program behavior, CGP mutation tends to have higher locality. Since the genome length is fixed and connections are indexed, a point mutation often results in a localized structural change. Selection is frequently implemented with a simple $(1 + \lambda)$ evolutionary strategy, commonly $(1 + 4)$: one parent produces λ offspring via mutation, and the best individual (ties included) becomes the parent of the next generation [13]. This "mutation-only" setup is computationally lightweight, and fits well when crossover provides limited benefit or is difficult to design.

Figure 2.4 reports pseudo-code representing the basic evolutionary strategy commonly adopted in CGP.

Algorithm 1 (1+4) Evolutionary Strategy

```
1  begin
2  | for all i such that 0<= i <μ do
3  | |     generate individual i randomly
4  | end for
5  |     fittest individual is selected as  parent
6  | while there is no solution or number of generations not exhausted do
7  | |     for all i such that 0<= i <λ do
8  | |     |   parent mutates and offspring i generated
9  | |     end for
10 | |     Generation of the fittest individual using the following
11 | |     if an offspring's fitness is equal or better than parents then
12 | |     |   choose the offspring
   | |     | else
13 | |     |   choose the parent
14 | |     end if
15 | end while
16 end
```

Figure 2.4: Basic algorithm used in CGP [7].

Why CGP for LGNs. CGP aligns naturally with the structure of LGNs: each node can be instantiated as a Boolean operator (AND, OR, XOR, etc.), and the evolved DAG directly corresponds to a gate-level circuit (or a gate-level computation graph) that can be executed with bitwise operations. Moreover, CGP supports:

- Compact encodings: a gate and its wiring are represented by a few integer genes, making genomes small and mutation efficient.
- Signal reuse: intermediate results can feed multiple downstream nodes, which is essential for circuit-like computations.
- Multiple outputs: CGP is naturally suited to MIMO settings, which is convenient for multi-class classification when outputs correspond to class logits or

class-specific bit patterns [13].

- Hardware-oriented mapping: the resulting phenotype is already a graph of Boolean operators, facilitating translation to low-level representations (e.g., netlists or instruction sequences) without additional quantization steps.

For these reasons, CGP provides a practical evolutionary framework for generating and optimizing LGNs: it matches the discrete nature of gate selection and wiring, leverages neutrality to explore large combinatorial spaces efficiently, and produces representations that are close to deployable digital logic.

Chapter 3

Related Works

This chapter reviews the current state-of-the-art on LGNs for machine learning, with a focus on differentiable training strategies that make large LGNs trainable with gradient-based optimization. We first summarize DDLGN, which introduced an end-to-end differentiable relaxation of binary logic gates and demonstrated extremely fast inference. We then cover the recent extension to the convolutional paradigm, which addresses the key limitations of the original approach on vision tasks by introducing logic-tree convolutions, logical pooling, and a depth-stabilizing initialization strategy.

3.1 Deep Differentiable Logic Gate Networks

LGNs promise exceptionally efficient inference because the forward pass is composed only of Boolean operators (e.g., AND, OR, XOR, NAND) rather than floating-point multiply-accumulate (MAC) operations. The central obstacle is training: the gate computation is discrete, and selecting which gate is used at each node is a categorical decision. In their "hard" form, these choices are non-differentiable, which prevents standard backpropagation.

To overcome this issue, in [4] authors introduced DDLGN by relaxing Boolean activations to probabilistic values in [0,1] and replacing each Boolean gate with a differentiable, real-valued counterpart derived from probabilistic logic (e.g., $a \wedge b \mapsto a \cdot b$, $a \oplus b \mapsto a + b - 2ab$). This step makes the gate output differentiable, but it does not yet solve the discreteness of the gate selection. To make the operator choice trainable, each node is parameterized as a categorical distribution over the 16 possible Boolean functions. In practice, each node stores a real-valued parameter vector (logits), mapped through a softmax. The node output is computed as the weighted average of all relaxed gates (i.e., the expectation under the learned distribution). After training, the network is discretized by selecting, at each node,

the most probable gate, producing a "hard" LGN executable purely with logic gates. Figure 3.1 shows a graphical representation of the probability distribution of gates for a single node.

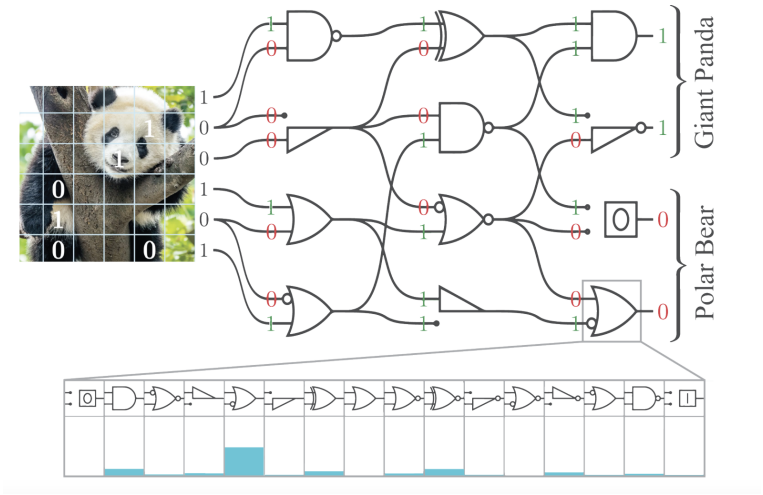

Figure 3.1: Representation of gate choice in a DDLGN.

DDLGN adopts a layered structure in which each node has exactly two inputs, implying intrinsic sparsity and linear-time layer cost $\Theta(n)$ (with small constants). For classification, multiple output bits are assigned to each class. Class scores are obtained by aggregating (bit-counting in the hard case, summation of probabilities in the relaxed case) and then applying a standard cross-entropy objective.

A key contribution is demonstrating that, once discretized, the model can be executed extremely fast on conventional hardware, exploiting bit-packing (e.g., batching bits into `int64`) and bitwise vectorization. The paper reports inference throughput beyond a million MNIST images per second on a single CPU core for a high-accuracy configuration. Moreover, because the connectivity can be generated from a pseudo-random seed, the deployed model can be stored compactly: the dominant information is the selected gate per node (conceptually encodable in 4 bits for 16 functions), leading to a small memory footprint.

In Table 3.1, we can see that this approach leads to state-of-the-art results and with an inference of 2 to 3 orders of magnitude faster than standard DNN.

Despite strong inference efficiency, DDLGN highlights two limitations relevant for the broader LGN framework:

- Training cost: training requires evaluating many relaxed operators per node (effectively a constant-factor overhead), and the paper notes training can be relatively expensive even if the model is sparse.

Model	Accuracy	Space	T. [CPU]	T. [GPU]
Linear Regression	91.6%	16KB	3μs	2.4ns
Neural Network (small)	97.92%	462KB	14μs	12.4ns
Neural Network	98.40%	86MB	2.2ms	819ns
Diff Logic Net (small)	97.69%	23KB	625ns	6.3ns
Diff Logic Net	98.47%	188KB	7μs	50ns

Table 3.1: Comparison of DDLGN to a classic DNN on MNIST. In the columns we report in order: the networks, the accuracy measured on the task and the memory occupied by the networks. The last two columns represent the inference time on the task respectively on a CPU and a GPU

- Architectural expressivity on images: the original differentiable LGNs rely on fixed random connectivity and flattened inputs, which makes it difficult to capture local spatial structure typical of vision tasks. This limitation is explicitly addressed by later convolutional extensions.

3.2 Convolutional Differentiable Logic Gate Networks

While randomly-connected LGNs can work well on simpler or tabular benchmarks, images contain strong local correlations and translation-equivariant patterns. The convolutional paradigm is the standard inductive bias to exploit this structure. The main issue is that classic DDLGN does not support parameter sharing across spatial locations and cannot easily encode localized receptive fields.

In [5], the authors present LogicTreeNet, which extends differentiable LGNs to convolutions by introducing deep logic gate tree convolutions. Each convolutional kernel is implemented as a small binary tree of logic gates whose leaves sample inputs from a local receptive field (e.g., 3×3) across channels. The kernel parameters, i.e. the gate-selection logits for the nodes of the tree, are shared across all spatial placements, exactly as in CNNs, providing translation equivariance while keeping computation gate-based. Figure 3.2 graphically compares a traditional CNN with a convolution logic gate network.

To emulate the downsampling role of max-pooling in CNNs, the authors propose *logical OR pooling*, implemented as a disjunction over a pooling window. Importantly, they use a max t-conorm relaxation (effectively $\max(\cdot)$ in the relaxed domain), which reduces backpropagation overhead and avoids saturating activations in practice.

A major empirical barrier in deeper differentiable LGNs is that softmax-initialized

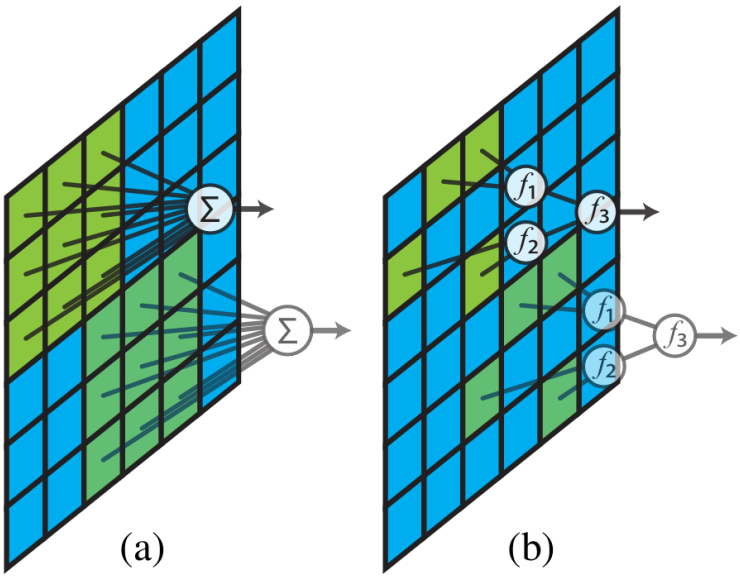

Figure 3.2: Conventional convolutional neural networks (a) compared to convolutional logic gate networks (b).

gate distributions can be "washed out," pushing activations toward 0.5 and causing vanishing gradients. The convolutional work introduces *residual initializations*: gates are initialized to strongly favor a feedforward operator (e.g., selecting input A with high probability), acting as a differentiable analogue of residual/skip behavior without explicit additions (which are unnatural in pure logic). This enables training substantially deeper logical architectures than previously feasible, and the paper reports large accuracy drops when residual initialization is removed, highlighting it as a key stabilizer.

Beyond architectural design, the convolutional work emphasizes efficient training implementations, including fused CUDA kernels for tree evaluation and pooling. This is positioned as essential to scaling training to larger gate counts and deeper models.

On CIFAR-10, the proposed LogicTreeNet reaches 86.29% accuracy with a reported gate count of 61 million, and the authors argue this improves the accuracy-vs-gate-count Pareto trade-off by large factors (e.g., tens of times fewer gates than several binarized baselines at similar accuracy).

The paper situates convolutional differentiable LGNs among similar families:

- Truth-table / LUT networks: networks that use n-input lookup tables as primitives can be effective and hardware-friendly, but are often derived from, or tightly coupled to, neural-network-style constructions (e.g., LUT-based

replacements inside BNN pipelines) rather than directly learning gate-level logic end-to-end.

- BNNs [2] and quantized networks: these methods reduce precision in weight-based models; although they can be compiled into logic for FPGAs/ASICs, they still reflect the computational structure of matrix multiplications and accumulations, whereas differentiable LGNs aim to learn the logic composition directly.

Overall, the current LGN state-of-the-art for supervised learning is dominated by differentiable training pipelines: DDLGN establishes the core relaxation-and-discretization framework with extreme inference efficiency, while convolutional differentiable LGNs (LogicTreeNet) introduce the missing convolutional inductive bias and the training stabilizers needed to scale depth and performance on vision benchmarks.

To conclude, works like [5] and [4] demonstrate that LGNs can achieve competitive accuracy while preserving an exceptionally efficient inference model, especially when training is enabled through differentiable relaxations and subsequent discretization. However, these approaches still rely on continuous surrogate operators and gradient-based optimization, which introduces non-negligible training overhead (e.g., evaluating mixtures of gates) and constrains the search to a specific parameterization of gate selection. In contrast, the focus of this thesis is to investigate evolutionary techniques and, in particular, CGP as a fundamentally discrete and gradient-free alternative for generating LGNs. From this perspective, differentiable LGNs serve as the main state-of-the-art reference points for accuracy and efficiency, while CGP provides a complementary approach to explore network topology and operator composition directly in the Boolean domain.

3.3 LogicNets: Co-Designed Neural Networks and Circuits for Extreme Throughput

An additional, closely related line of work to LGN is LogicNets [14], which targets extreme-throughput and ultra-low-latency inference by co-designing neural network topologies and their direct hardware realization on FPGAs. In contrast to DDLGN, which learns networks of logic gates through a differentiable relaxation, LogicNets starts from quantized and sparse neural networks and exploits the observation that a neuron with quantized inputs/outputs can be represented exactly as a truth table, and therefore mapped to FPGA LUTs without an explicit multiply-accumulate datapath.

LogicNets introduces the concept of converting a trained, quantized neuron into an equivalent truth table by enumerating all possible input combinations and recording the corresponding output. The resulting truth tables are instantiated in hardware as small ROM-like LUT structures, yielding circuits that are highly pipelinable and massively parallel. This approach removes the need for explicit MAC and activation datapaths, replacing them with pure combinational logic (plus optional pipeline registers).

A central challenge is that the hardware cost of implementing a truth table grows exponentially with the number of input bits (fan-in). For this reason, LogicNets constrains fan-in by combining:

- sparsity: each neuron connects to only a limited number of previous activations (small per-neuron fan-in), and
- low-bit activation quantization: activations are quantized to a small bitwidth (typically ≤ 4 bits),

so that each neuron can be mapped to a small truth table implementable with practical LUT resources. To guide topology selection, the paper derives an analytical LUT cost model and uses it to estimate resource usage before synthesis, enabling a direct accuracy–resource trade-off exploration.

The LogicNets workflow follows three steps:

1. define trainable NEQs that correspond to hardware building blocks (truth tables);
2. train sparse/quantized networks of NEQs in a standard deep learning framework;
3. convert the trained model into a Verilog netlist of truth tables for FPGA synthesis and deployment.

This results in a hardware-centric machine learning pipeline where the trained model is already structurally close to its final digital circuit. LogicNets is evaluated on two tasks with stringent throughput requirements (jet substructure classification and network intrusion detection), reporting FPGA implementations with sub-microsecond latency and throughput in the hundreds of millions of inferences per second, while maintaining competitive accuracy with respect to prior FPGA-oriented baselines. These results highlight that, when the model is explicitly designed around hardware primitives, extremely low-latency inference becomes achievable without relying on GPU-like execution paradigms.

LogicNets strengthens the broader thesis motivation: logic-level inference is a viable target for high-efficiency deployment, and the model representation can be

aligned with hardware building blocks (gates or LUT truth tables) rather than floating-point linear algebra. However, LogicNets still relies on gradient-based training and represents computation at the level of truth tables for constrained fan-in neurons, whereas LGNs operate directly as compositions of Boolean operators. From our perspective, LogicNets provides an important complementary reference point: it demonstrates the benefits of hardware-aware logic representations, while also emphasizing the practical limitations induced by fan-in constraints and the complexity of topology design. This thesis investigates whether evolutionary, discrete search via CGP can offer an alternative path by directly exploring gate-level structure and wiring in the Boolean domain, potentially producing compact logic networks without requiring differentiable relaxations or surrogate training.

Chapter 4

Proposed Solution

This chapter details the methodology and the experimental framework developed to evolve LGNs using CGP.

The first part describes the proposed approach for MNIST classification, focusing on data binarization and the multi-bit voting mechanism designed to translate graph outputs into class predictions. Subsequently, we introduce the computational backbone of this work: an extended version of the CGP++ framework. We detail the architectural modifications implemented to support layered topologies, batch-based training, and specialized mutation operators. Finally, we present a validation study on the HRR problem to demonstrate the framework's ability to evolve hierarchical structures before tackling the increased complexity of the MNIST digit recognition task.

4.1 Proposed Approach

This work proposes an alternative way to generate LGN based on CGP. The main goal is to develop a network that can correctly classify MNIST images [15]. The dataset contains images of handwritten digits, representing a benchmark for multiclass classification tasks with $n = 10$ classes. It is composed of 60,000 training examples and 10,000 test examples.

Data Preparation. To interface the MNIST dataset with the proposed CGP-based LGN, we convert each image into a fixed-length Boolean input vector. Starting from the original 28×28 grayscale images (pixel range [0,255]), we:

1. Flatten each image into a 1D vector of length 784;
2. Apply a simple threshold-based binarization: pixels with value greater than 127 are mapped to 1, and the remaining pixels are mapped to 0.

The resulting Boolean tensors are stored in two plain-text files:

- `mnist_boolean_train.txt`, containing binarized training samples
- `mnist_boolean_test.txt`, containing binarized test samples.

Each file follows the same format to simplify parsing in the CGP implementation: the first line is a header reporting the number of samples, the input size, and the number of classes (i.e., `NUM_SAMPLES 784 10`). Each subsequent line contains one labeled example as `LABEL` followed by the 784 binary pixel values separated by spaces. For readability, the following snippet shows the structure of the file (pixel sequences are truncated with `...`):

```
60000 784 10
8 0 0 0 0 ... 1 0 1 1 0 ... 0 0
7 0 0 0 0 ... 1 1 0 0 1 ... 0 0
```

This representation allows the LGN to be evaluated directly with bitwise operations, while maintaining a minimal and deterministic preprocessing pipeline.

Algorithm. In this work, CGP is optimized with an evolutionary strategy $(\mu + \lambda)$ and point mutation, following these steps:

1. **Initialization**: generate μ random genotypes with valid f, C, O values, using the encoding presented in Equation 2.1.
2. **Evaluation**: decode each genotype into a LGN and compute its Loss on the task.
3. **Offspring Generation**: create λ offspring by copying parents and applying n point mutations, i.e., randomly changing n genes at a time.
4. **Selection**: evaluate parents and offspring, then select the best μ individuals among the $\mu + \lambda$ candidates to form the next generation.
5. Repeat step 3 and 4 until a stopping criterion is met (max generations, target accuracy reached, time limit).

In our case, we use a 4+12 configuration; this means keeping 4 parents, generating 12 mutated offspring per generation, evaluating all 16 individuals, and retaining the best 4 according to the Loss. Over generations, selection biases the search toward genotypes whose decoded LGNs yield a lower Loss function, so the network progressively evolves toward more accurate solutions.

Classification. To perform 10-class classification, the 500 output nodes of the LGN are partitioned into 10 groups of $N_b = 50$ bits each. For each class c, we compute a classification score by counting the number of active bits (logical 1s) within its designated group. The final prediction $\hat{y}$ is retrieved by identifying the class with the highest score through the *argmax* operator:

$$\hat{y} = \arg\max_{c} \left(\sum_{j=1}^{N_b} b_{c,j} \right) \tag{4.1}$$

where $b_{c,j}$ represents the j-th output bit associated with class c. This redundant encoding, illustrated in Figure 4.1, provides a smoother landscape for the evolutionary search compared to a single-bit output. Figure 4.1 shows a simplified example with 2 bits per class.

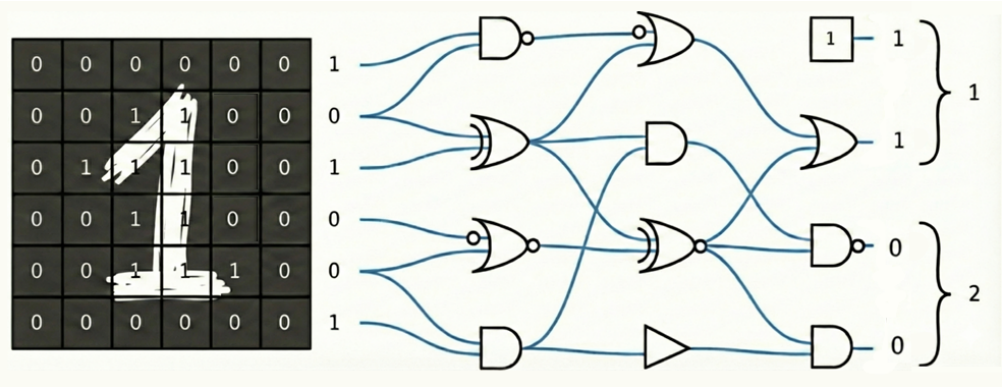

Figure 4.1: Classification of an MNIST tensor

Loss function. In a perfect classification, we should have 50 bits at 1 in the correct class and all the other bits at 0. The Loss function is specifically designed to reflect how many bits were wrongly set by the network. Given the resulting bitstring for one image: we define B_0 as the number of bits at 0 associated to the correct class and B_1 as the number of bits at 1 in incorrect classes. The Loss function used to measure the classification error is:

$$L = \sum_{i=0}^{S-1} (l_i) \tag{4.2}$$

where

$$l_i = \begin{cases} (B_0 X + B_1) Y & \text{if } \hat{y} = y \\ (B_0 X + B_1) & \text{otherwise} \end{cases} \tag{4.3}$$

Here, S is the batch size, and X and Y are constants chosen experimentally.

4.2 Framework

To implement and evaluate the proposed CGP-based approach for generating LGNs, we built upon CGP++, a modern high-performance C++ framework for Cartesian Genetic Programming developed by Roman Kalkreuth and collaborators [16]. CGP++ was designed as a reusable and extensible toolkit with the explicit goals of supporting new problem domains and improving the reproducibility of CGP experiments through a clear architecture, configuration interfaces, and checkpointing facilities.

CGP++ implements CGP using modern C++ (C++17) and follows an object-oriented and generic-programming approach. The framework is structured around modular components (e.g., initializer, evaluator, representation, variation, algorithm), enabling domain-specific logic to be added by specializing a small set of interfaces while reusing the common evolutionary loop and infrastructure. In particular, CGP++ supports template-based type configuration, allowing the user to select the evaluation type, genotype type, and fitness type according to the target domain.

CGP++ is operated through a command-line executable that takes as input:

- a data/benchmark file;
- a parameter file;
- a checkpoint file to resume an interrupted run, optionally.

The executable performs basic problem-domain dispatch based on the input file type and initializes the corresponding pipeline (e.g., logic-synthesis vs. symbolic-regression initializers). This initialization stage configures the evolutionary algorithm, allocates data structures for evaluation, and builds the function set and problem instance.

A key design choice of CGP++ is the separation between:

- static configuration (parameter file), used to define the overall experiment setup, and
- run-time overrides (command-line flags), used to quickly sweep hyperparameters.

The framework exposes the main CGP hyperparameters: number of nodes, levels-back, function set size, arity, mutation/crossover rates, and evolutionary strategy parameters μ and λ, as well as infrastructure parameters such as random seeds, number of jobs, and evaluation threads.

CGP++ provides native support for both the $(1 + \lambda)$ and $(\mu + \lambda)$ evolutionary strategies, which are selectable via configuration. This directly matches the

evolutionary protocol adopted in this thesis (Section 4.1), where we use a $(\mu + \lambda)$ strategy (e.g., a $4 + 12$ setup) and rely on mutation-driven exploration.

Beyond standard point mutation, CGP++ supports mutation pipelining, where multiple mutation operators can be activated and applied in sequence. The available operators include:

- probabilistic point mutation (gene-wise mutation with a given rate),
- single active gene mutation (focused mutations on expressed genes to bias changes toward functional parts of the graph),
- inversion mutation and duplication mutation (structural operators that modify genotype segments).

In addition, the framework includes crossover support (with selectable crossover types), although CGP commonly employs only mutation for integer encodings. In our experimental setup, the primary exploration mechanism is point mutation, otherwise explicitly stated.

CGP++ provides a checkpointing mechanism designed for reproducibility and fault tolerance in long experiments. When checkpointing is enabled, the framework can periodically store the evolution state (including generation/evaluation counters, global seed, and the population/genomes) according to a user-defined cadence (using the parameter `checkpoint_modulo`). A run can then be resumed by passing the checkpoint file to the executable, ensuring that experiments interrupted due to time limits or system failures can continue deterministically from the saved state. Together with explicit seed handling (fixed `global_seed` or generated seeds), this supports repeatable experimental conditions and controlled comparisons between configurations.

CGP++ includes dedicated components for logic synthesis, making it a natural starting point for evolving LGN-like structures. The logic-synthesis pipeline instantiates a dedicated problem class where the fitness metric is based on the Hamming distance between the candidate program outputs and the target truth table. The implementation also supports a compressed truth-table representation in which each fitness instance corresponds to a chunk of the truth table. This organization enables efficient evaluation and aligns with the typical CGP use case in Boolean domains.

For the Boolean function set, CGP++ provides a `FunctionsBoolean` implementation that:

- enforces the use of integer evaluation types (through compile-time checks) to avoid invalid floating-point usage, and
- computes gate outputs using native bitwise operators (e.g., `&`, `|`, `~`), enabling word-level parallelism when using multi-bit integer types.

The default Boolean function set in the reference implementation includes canonical two-input gates: AND, OR, NAND, NOR.

Overall, CGP++ provides a solid experimental backbone for this thesis: it offers a performant C++ implementation, configurable evolutionary strategies, mutation operator pipelines, and checkpointing for reproducible long runs. Its explicit support for Boolean program synthesis and bitwise evaluation makes it particularly suitable for evolving logic-centric architectures, which is precisely the setting of Logic Gate Networks.

4.3 Holland Royal Road Problem

Before working on MNIST, we validated the CGP++ pipeline using a well-known benchmark in genetic algorithms: Holland's Royal Road function [17]. The goal was to verify that our framework extensions worked correctly and that the evolutionary dynamics, such as selection, mutation and neutrality, were effective. This test can provide preliminary evidence that the system can handle problems where building blocks and hierarchical structures are essential before moving to the more complex MNIST task.

Problem definition. Holland's Royal Road function defines a maximization task over a binary string and is designed to reward the progressive construction and combination of "building blocks". With the default parameterization, the chromosome is a length-240 binary string partitioned into $2^k = 16$ *regions*, each of length $b + g = 15$, where the first $b = 8$ bits form a *block* and the remaining $g = 7$ bits are a *gap* ignored during fitness evaluation.

The fitness is computed as the sum of two components:

- PART: each block is scored according to the number of ones it contains (rewarding up to $m^\star$ ones and penalizing excess ones, while assigning 0 to complete blocks).
- BONUS: completed blocks (all b bits equal to 1) are rewarded hierarchically across levels, where at level ℓ the objective is to complete contiguous sets of 2^ℓ blocks. The first completed set at each level receives $u^\star$ and additional completed sets receive u.

For the default settings ($k = 4$, $b = 8$, $g = 7$, $m^\star = 4$, $v = 0.02$, $u^\star = 1.0$, $u = 0.3$), the maximum achievable fitness is 12.8.

Integration into CGP++. CGP++ natively supports logic synthesis through `.plu` truth tables, whereas the HRR benchmark is defined directly as a fitness

function over a produced bitstring. Therefore, we extended the framework by introducing a dedicated black-box problem and initializer:

- **`HollandRoyalRoadProblem`:** we implemented the PART and BONUS computations exactly as described in [17], using a phenotype that is a 240-bit output vector. The evaluator interprets the individual's outputs as the candidate chromosome and returns a fitness value based on PART+BONUS, with the theoretical maximum set to 12.8.

- **Loss L:** since CGP++ can operate in minimization mode, the problem returns $(12.8 - \text{fitness})$ so that the optimum corresponds to $L = 0$ (perfect solution).

- **`HollandRoyalRoadInitializer`:** HRR problem does not require a dataset. Thus, the initializer sets the experiment to a single instance and configures the expected I/O sizes (`num_variables`=2, `num_outputs`=240). Dummy inputs are provided only to satisfy the base `BlackBoxProblem` interface.

- **Domain dispatch and type validation:** the main entry point was extended to recognize a `.hrr` input file extension, instantiate the HRR initializer, and enforce an integer evaluation type (unsigned integer) for boolean logic.

- **Benchmark trigger file:** a minimal `dummy.hrr` file is used only to activate the HRR execution path in the framework.

Experimental configuration. The solution was obtained using the hyperparameters defined in the `parameters.txt` configuration file. The key settings are summarized in Table 4.1.

Parameter	**Value**
Evolutionary Strategy	$(\mu + \lambda)$ with $\mu = 4, \lambda = 8$
Genotype Size (N_{nodes})	1000
Levels-back	1000
I/O Size (Inputs : Outputs)	2 : 240
Function Set	{AND, OR, NAND, NOR} (Max Arity: 2)
Point Mutation Rate	0.1
Duplication Rate	0.05

Table 4.1: Experimental configuration for the HRR benchmark.

Results. With the above configuration, the framework reached a valid solution to the HRR problem in approximately 630,000 generations (about 44 minutes). This benchmark is specifically constructed to reward *hierarchical composition* and to test whether an evolutionary system can reliably assemble higher-order structures from lower-level building blocks. Therefore, successfully solving HRR provides evidence that the extended CGP++ pipeline (problem definition, evaluation loop, mutation operators, and parameterization) is operational and capable of non-trivial discrete optimization. In turn, this supports the working hypothesis of the thesis: if the framework can solve a hierarchical building-block benchmark such as HRR, it has good prospects to scale to the MNIST Boolean classification setting once the problem-specific components are in place.

4.4 MNIST Problem

To study LGNs in a realistic classification setting, we extended CGP++ to support the task of MNIST digit recognition from binarized images. The original CGP++ distribution mainly targets (i) logic synthesis from truth tables (`.plu`) and (ii) symbolic regression (`.dat`). In contrast, MNIST requires handling high-dimensional inputs (784 pixels per sample) and a multi-class objective (10 digits), which are not directly covered by the default problem classes. For this reason, we created a dedicated MNIST pipeline in our fork of the framework[1].

4.4.1 Problem Integration and I/O Handling

A new problem type (`MNIST_LOGIC`) was added to the main entry point (`cgp.cpp`). The framework now recognizes `.txt` datasets as MNIST logic benchmarks and instantiates the corresponding initializer (`MnistLogicInitializer`). Additionally, a compile-time validation routine (`validate_mnist_type`) enforces an integer evaluation type (`unsigned int`) to ensure that the execution remains in the Boolean domain and avoids unintended floating-point behavior.

The core extension is the introduction of `MnistLogicProblem`, a class derived from `BlackBoxProblem` specifically designed for classification. Unlike standard Boolean benchmarks with a few inputs, this class is configured to:

- accept 784 Boolean inputs per instance (flattened 28×28 image),
- store the full dataset in memory (`full_inputs`, `full_outputs`) and expose an *active* subset to the evaluation loop,

[1] `https://github.com/Ema725/Thesis-LGN/tree/main/cgp-plusplus`

- provide a dedicated `validate_individual` method that returns the exact number of correctly classified images, enabling accuracy reporting independently of the fitness definition.

To load MNIST-style data, we implemented `MnistLogicInitializer`, which parses the text format produced by the preprocessing step (header + labeled binary vectors). During `read_data()`, the initializer:

- reads the dataset header (`NUM_SAMPLES NUM_INPUTS NUM_CLASSES`) and automatically sets `num_variables` to 784,
- configures `num_outputs` for multi-class classification (see next paragraph),
- loads all samples into temporary buffers and then initializes the active input/output vectors,
- transfers ownership of the full dataset to `MnistLogicProblem` via move semantics (`set_full_dataset(std::move(...))`) to avoid expensive copies.

4.4.2 Multi-class Decoding and Fitness Design

Output encoding. Rather than using a single output per class, we adopt a redundant multi-bit output encoding to increase robustness and provide a richer discrete signal for evolution. The total number of outputs is constrained to be a multiple of 10:

$$\texttt{num\_outputs} = 10 \times \texttt{bits\_per\_class}.$$

Given an individual's output vector, the predicted class is computed via bit counting: outputs are partitioned into 10 contiguous blocks, one per class, and the class with the highest number of active bits is selected, as explained in Equation 4.1.

Loss. CGP++'s logic synthesis uses Hamming distance against truth tables. For MNIST we measure the classification quality; therefore,

`MnistLogicProblem::evaluate()` implements a family of Loss functions controlled by a new parameter `fitness_function`. The shared principle across variants is to reward solutions that:

- activate many bits in the correct class block, and
- suppress activations in the incorrect class blocks.

The details can be seen in Equation 4.2.

4.4.3 Batch Training

Evaluating every individual on tens of thousands of images at every generation is computationally expensive. To make evolution feasible, we introduced batch training as a first-class mechanism:

- the `Parameters` class was extended with `batch_training`, `batch_size`, `batch_gen`, and `file_size`.
- `BlackBoxProblem` was extended with two virtual methods, `load_batch()` and `load_random_batch()`, which are overridden in `MnistLogicProblem`.

Two batch modes are supported:

- **Sequential batches** (`batch_training = 1`): every `batch_gen` generations, the active dataset window is shifted deterministically through the file using `load_batch(start, batch_size)`.
- **Random batches** (`batch_training = 2`): every `batch_gen` generations, a new random subset of size `batch_size` is sampled from the full dataset using `load_random_batch(batch_size, rng)`.

After a batch switch, the algorithm explicitly marks all individuals as *not evaluated* so that the Loss is recomputed on the new batch, ensuring consistency.

4.4.4 Search-space Shaping

Layered CGP connectivity. To better match the layered structure of LGNs, we extended `Species` so that the admissible ranges of connection genes can enforce layered wiring constraints. A new parameter `fixed_layers` controls the behavior:

- `fixed_layers = 0`: standard CGP levels-back connectivity.
- `fixed_layers = 1`: nodes in a layer can connect only to the immediately previous layer (first layer connects to primary inputs).
- `fixed_layers = 2`: nodes can connect to any previous layer (including inputs), which matches the architecture used in this thesis.

In this implementation, the parameter `levels_back` is repurposed as the *layer width* (number of nodes per layer), and the layer index is computed from the node position. This provides a direct mechanism to instantiate large, fixed-depth, layer-wise LGNs within the standard CGP genome format.

Extended Boolean gate set. To represent LGNs more faithfully, the Boolean function library was expanded from the reduced logic-synthesis set (AND/OR/-NAND/NOR) to a richer set of 16 Boolean operators, including XOR/XNOR, implication variants, constants (TRUE/FALSE), and unary buffers/negations. This allows evolution to explore a more expressive operator space at each node. Table 4.2 reports the complete list of the operators used in our problem.

ID	Operator
0	False
1	$A \wedge B$
2	$\neg(A \Rightarrow B)$
3	A
4	$\neg(A \Leftarrow B)$
5	B
6	$A \oplus B$
7	$A \vee B$
8	$\neg(A \vee B)$
9	$\neg(A \oplus B)$
10	$\neg B$
11	$A \Leftarrow B$
12	$\neg A$
13	$A \Rightarrow B$
14	$\neg(A \wedge B)$
15	True

Table 4.2: List of logic gates used.

Specialized mutation. In addition to the existing probabilistic point mutation and active-gene mutation, we introduced `FunctionMutation`, a unary operator that mutates only the function genes (gate types) while keeping wiring unchanged. This is useful in MNIST experiments because: topology search (connections) and operator search (gates) can be decoupled, once a promising connectivity pattern is found, targeted gate tuning can refine decision logic without destroying the structure. A dedicated parameter `function_mutation_rate` controls how many function genes are mutated per application.

4.4.5 Parameters Adaptation and Accuracy

The `MuPlusLambda` algorithm was extended to become batch-aware: at fixed intervals it triggers a batch switch, logs batch-level statistics (min/max accuracy

and Loss), and resets evaluation flags. Moreover, we added an optional 1/5th success rule [18] controller (parameters `one_fifth_rule`, `K`, `C`) to adapt mutation rates dynamically based on observed improvement rates, including coordinated adjustment of `function_mutation_rate` when active. The mutation rate m is updated every K generations according to the success rate p_s (the fraction of successful mutations) as follows:

$$m^{(g+K)} = \begin{cases} m^{(g)}/C & \text{if } p_s > 1/5 \\ m^{(g)} \cdot C & \text{if } p_s < 1/5 \\ m^{(g)} & \text{if } p_s = 1/5 \end{cases} \tag{4.4}$$

where C is a scaling factor that decreases the mutation rate to focus the search or increases it to escape local optima.

To make results interpretable during long runs, both the evolutionary reporting and the final job report were extended to print accuracy using `validate_individual()` whenever the active problem supports it. This yields outputs of the form:

$$\text{Accuracy} = \frac{\text{hits}}{\text{num\_instances}},$$

without conflating this diagnostic metric with the optimized fitness definition. With these extensions, CGP++ becomes a complete experimental platform for evolving LGN-like classifiers on MNIST: it supports high-dimensional Boolean inputs, multi-class decoding, scalable batch-based evaluation, layered topological constraints, and domain-specific mutation operators. As a result, the framework is now sufficiently tailored to run the experimental campaign presented in the next chapter, where we systematically evaluate different evolutionary configurations (population sizes, mutation strategies, batch regimes, and fitness variants) and measure their impact on both accuracy and computational efficiency.

Chapter 5

Experiments

This chapter reports the experimental evaluation of the proposed CGP-based approach for evolving LGNs on the binarized MNIST classification task. The goal is to quantify how different hyperparameters and design choices influence learning dynamics and final performance.

Unless otherwise stated, every experiment in this chapter evolves a LGN using a $(\mu + \lambda)$ evolutionary strategy with $\mu = 4$ parents and $\lambda = 12$ offspring per generation. Each individual encodes a network with 80,000 function nodes arranged into 5 layers (16,000 nodes per layer). The function set is the 16 Boolean operators described in Section 4.4.4, and the multi-class output is encoded using 500 output bits (50 bits per class). Variation is performed through point mutations and, if stated, function mutations. All these common parameters are summarized in Table 5.1. To improve readability, for each subsection we explicitly report the key run-time parameters of the corresponding sweep.

Parameter	**Value**
Evolutionary strategy	$(\mu + \lambda)$ with $(4 + 12)$
Function nodes	80,000
Layers	5 (16,000 nodes per layer)
Function set	16 Boolean gates (2-input)
Outputs	500 bits (50 bits/class)
Variation operator	Point mutation

Table 5.1: Common configuration.

5.1 Loss Function

The Loss function determines the optimization function used to guide evolution. In this work, we tested multiple Loss variants, each designed to balance two competing objectives: activating bits for the correct class and suppressing activations for incorrect classes. This section compares these variants.

As mentioned in Section 4.1, given the output bitstring, we define B_0 as the number of bits at 0 associated to the correct class and B_1 as the number of bits at 1 in incorrect classes.

Bitwise Loss. First, we focus on the Loss function presented in Equation 5.2, where we only did a bitwise confrontation on the output, penalizing more a bit at 0 in the correct class rather than a bit at 1 in a incorrect one, through the use of the X variable.

$$l_i = (B_0 X + B_1) \tag{5.1}$$

$$L0 = \sum_{i=0}^{S-1} (l_i) \tag{5.2}$$

To evaluate the impact of the Loss function variables on the network's performance, a sweep was conducted on the variable X. Table 5.2 summarizes the experimental setup and the resulting test accuracies.

Generations	Batch Size	Mut. Rate	X	Accuracy
30,000	200	0.005	1	23.20%
			2	33.00%
			5	40.00%
			10	**41.80**%
			100	11.20%

Table 5.2: Effect of the X parameter on MNIST accuracy.

For these experiments, X = 10 and X = 5 give the better results because those values give a better balance between the misclassified bits. X = 100 tends to produce outputs with all bits at 1, and X = 1 with all bits at 0.

Scaled Bitwise Loss. The next Loss treated is the one already presented in Equation 4.2:

$$l_i = \begin{cases} (B_0 X + B_1)Y & \text{if } \hat{y} = y \\ (B_0 X + B_1) & \text{otherwise} \end{cases}$$

$$L1 = \sum_{i=0}^{S-1} l_i$$

Here we introduce the variable Y, that works as a scale factor if the network predicts correctly the label for the tensor. By doing so, we create a new Loss function that takes into account bit-to-bit comparisons but also the number of correctly classified images. Table 5.3 reports the experimental results.

Generations	Batch size	Mut. Rate	X	Y	Accuracy
30,000	1,000	0.005	5	0.25	52.0%
			5	0.50	53.4%
			5	0.75	59.0%
			10	0.25	53.0%
			10	0.50	56.2%
			10	0.75	**62.90%**

Table 5.3: Impact of the X and Y parameters on MNIST accuracy

Compared to the previous experiment, with the variable Y added, we have an overall improvement in accuracy. The best performing models are those with Y = 0.75, probably because more aggressive values of Y, such as 0,5 or 0,25 tend to wash out the bit-to-bit logic of the Loss and randomize the evolution method.

Probability-normalized Loss. A limitation of the previous Loss functions is that they may converge to degenerate solutions (e.g., all zeros or all ones in the output bitstring), because the optimization is driven only by bitwise mismatches. To mitigate this behavior, we also tested a Loss inspired by a probabilistic interpretation of the output bits and by explicit constraints on the global activation level.

Let C be the number of classes ($C = 10$) and let L_C be the number of output bits assigned to each class (in our case $L_C = 50$). For each input sample j, let $b_{j,i,r} \in \{0,1\}$ be the r-th bit produced for class i ($r = 1, \ldots, L_C$). We interpret the fraction of active bits as an estimate of the probability that sample j belongs to class i:

$$p_{j,i} = \frac{1}{L_C} \sum_{r=1}^{L_C} b_{j,i,r}. \tag{5.3}$$

To avoid trivial solutions and enforce a consistent "probability mass" across classes, we introduce a normalization constraint requiring that the total number of ones across all classes equals L_C:

$$\sum_{i=1}^{C}\sum_{r=1}^{L_C} b_{j,i,r} = L_C \quad \Longleftrightarrow \quad \sum_{i=1}^{C} p_{j,i} = 1. \tag{5.4}$$

We then define the per-sample classification error as the distance from assigning probability 1 to the correct class y_j:

$$e_j = 1 - p_{j,y_j}. \tag{5.5}$$

In addition to minimizing the overall error $E = \sum_j e_j$, we include three penalty mechanisms:

- **Normalization penalty (P_0):** applied when Equation (5.4) is violated (i.e., the output contains too many or too few ones overall).
- **Misclassification penalty (P_1):** applied when the predicted class $\hat{y}_j = \arg\max_i p_{j,i}$ is not the true class (i.e., the correct class does not achieve the maximum probability).
- **Progressive difficulty penalty (P_2):** misclassified inputs are sorted (e.g., by error magnitude) and split into quartiles $s_{0.25}, s_{0.50}, s_{0.75}, s_{1.00}$; increasingly larger weights are assigned to worse quartiles to focus evolutionary pressure on the hardest cases.

Following this scheme, the final Loss can be written as:

$$\begin{aligned} L2 = P_2 \sum_{j\in s_{0.25}} \Big(P_1 \cdot (P_0 \cdot e_j)\Big) + 2P_2 \sum_{j\in s_{0.50}} \Big(P_1 \cdot (P_0 \cdot e_j)\Big) \\ + 4P_2 \sum_{j\in s_{0.75}} \Big(P_1 \cdot (P_0 \cdot e_j)\Big) + 8P_2 \sum_{j\in s_{1.00}} \Big(P_1 \cdot (P_0 \cdot e_j)\Big). \end{aligned} \tag{5.6}$$

Intuitively, this Loss (i) constrains the global activation level to prevent collapse to all-zeros/all-ones outputs, (ii) explicitly rewards correct *ranking* of class scores via P_1, and (iii) increases selection pressure on the hardest subset of samples through the progressive P_2 weighting. In the next experiments, we evaluate whether this more structured objective improves stability and generalization compared to the simpler bit-mismatch Loss functions. In practice, we observed that the progressive quartile term (P_2) brought limited additional benefit in our setting, while increasing the implementation complexity due to the need for a global batch-level ordering. For this reason, the best results were obtained with the simplified version of $L2$ that omits P_2 and uses only the normalization and misclassification penalties. Concretely, the strongest configuration was achieved using:

- a **normalization penalty** P_0 defined as

$$P_0 = \begin{cases} 1 + 0.1 \cdot \left| \sum_{i=1}^{C} \sum_{r=1}^{L_C} b_{j,i,r} - L_C \right| & \text{if } \sum_{i,r} b_{j,i,r} \neq L_C \\ 1 & \text{otherwise,} \end{cases}$$

  i.e., a 10% multiplicative increase for each unit of deviation from the target total number of ones ($L_C = 50$), and

- a **misclassification penalty** $P_1 = 5$ applied when $\hat{y}_j \neq y_j$ (and $P_1 = 1$ otherwise).

With these values, the per-sample Loss becomes $P_1 \cdot (P_0 \cdot e_j)$, where $e_j = 1 - p_{j,y_j}$.
given P_0 and P_1, we define $L3$ as:

$$L3 = \sum_{i=0}^{S-1} (P_1 \cdot P_0 \cdot e_j) \tag{5.7}$$

For comparison, an additional run was conducted using the $L3$ Loss function. Maintaining the baseline parameters of 30,000 generations, a batch size of 1,000, and a mutation rate of 0.005, this configuration results in a 60.8% accuracy on MNIST. The results are very similar to the one achieved with the Scaled Bitwise Loss (4.2). This shows how the limitations of the algorithm are not in the fitness function anymore but in other hyperparameters configuration.

For Sections 5.2, 5.3 and 5.4 we will use Loss L1 4.2 with $X = 10$ and $Y = 0.75$, since it proved to be the best-performing one.

5.2 Batch Size

The evaluation of each individual on the complete MNIST training set is computationally expensive; therefore, evolution is performed using batches of training samples. The batch size affects both the computational cost per generation and the variance of the Loss function. Larger batches provide a more stable estimate of accuracy and smoother exploration, but increase evaluation time; smaller batches speed up evolution but may introduce noise and overfitting to the batch.

Table 5.4 reports the results of different experiments with varying batch size. As expected, smaller batches tend to overfit the batch rather than generalize the classification task. As we increase the batch size we notice an increase of the accuracy on the test set and the gap between training and test accuracy reduces. With batch size bigger than 4000 the accuracy did not increase significantly and the experiments were too slow to evaluate. Furthermore, we can notice that with a bigger batch size, the process requires fewer generations to improve its accuracy.

Mut. rate	Generations	Batch size	Test acc.	Train acc.
	200,000	100	49%	82%
	100,000	200	56%	79%
0.0005	20,000	1,000	58%	66%
	10,000	2,000	62.9%	65%
	5,000	4,000	**65%**	67%

Table 5.4: Effect of the batch size on MNIST accuracy

5.3 Mutation Rate

Mutation is the main exploration mechanism in CGP. The mutation rate is the percentage of genes that are randomly modified when generating an offspring, so it strongly affects the exploration–exploitation trade-off. In these experiments, we vary the point mutation rate while keeping the evolutionary strategy $(\mu + \lambda)$, batch regime, network size, and Loss function are fixed.

Batch size	Point mutation rate	Test accuracy (%)
	0.005	45%
	0.0015	52%
4,000	0.0005	62%
	0.00025	72%
	$1/ngenes$	75%

Table 5.5: Effect of point mutation rate on MNIST accuracy

As shown in Table 5.5, decreasing the mutation rate generally improves the final accuracy: smaller perturbations make it more likely to preserve useful substructures (i.e., gate compositions and wiring motifs) that have already emerged during evolution. This behavior is expected in CGP, where once a promising topology is discovered, large mutations tend to disrupt active paths and destroy partial "building blocks".

However, a lower mutation rate also introduces a clear trade-off. Since fewer genes are modified per offspring, exploration becomes less aggressive and the probability of generating a beneficial variation decreases. As a consequence, the algorithm may require a substantially larger number of generations to escape local optima or to discover new useful structures. In practice, very small mutation rates improve the best attainable accuracy but can slow down convergence, especially in the early stages of evolution when the population is far from a good solution.

To balance these competing effects, we introduce an adaptive mutation mechanism based on the 1/5 success rule, explained in Section 4.4.5. The underlying idea is to automatically adjust the mutation rate according to the observed effectiveness of recent mutations. In our implementation, this adaptation is controlled by a multiplicative factor C: the mutation rate is multiplied by C when $p_s < 1/5$ and by $1/C$ when $p_s > 1/5$ with $C = 0.9$ and $K = 500$.

Overall, the results in this section suggest that small fixed mutation rates can yield the highest accuracy, but at the cost of slower progress. The 1/5 rule provides a principled way to mitigate this issue by automatically annealing the mutation rate: high in the initial phases to promote exploration and progressively lower as evolution starts exploiting and refining promising LGN structures.

5.4 Layer Division

In classical CGP, node i can connect to a predecessor within the range $[i - l,\ i - 1]$, where l is the *levels-back* parameter. While this provides a simple acyclicity constraint, it does not explicitly enforce a layered structure: the resulting graphs may become very deep or irregular, which can be undesirable when the target architecture is a layered LGN.

To address this, we introduced a layered wiring scheme (Section 4.4.4) controlled by the `fixed_layers` option. In the experiments below, we compare different connectivity constraints:

- **Standard CGP connectivity**:: levels-back only.
- **Strict layered connectivity**: nodes connect only to the previous layer.
- **Relaxed layered connectivity**: nodes connect to any previous layer (including inputs).

For layered runs, `levels_back` is interpreted as the *layer width*, while the number of layers is fixed by the experiment design.

Mut. rate	Batch	Gen.	Connectivity setting	Test acc.
0.0005 with decay	4,000	90,000	Standard CGP (levels-back)	60%
			Strict Layered	75%
			Relaxed Layered	78%

Table 5.6: Effect of connectivity constraints on MNIST accuracy.

Table 5.6 shows that enforcing a fixed layered structure significantly improves

performance with respect to standard CGP connectivity. With the classical levels-back constraint, the evolved graphs tend to become structurally irregular: some functional paths may grow very deep while other regions remain inactive. In practice, this increases the difficulty of the search because mutations often affect long dependency chains, making improvements less local and more fragile. By contrast, the layered encoding introduces an architectural prior consistent with LGNs: information is progressively transformed across a small number of stages, and the genotype-to-phenotype mapping becomes more stable under mutation. This results in higher accuracy and more consistent evolutionary progress.

Furthermore, allowing relaxed layered connectivity, i.e., permitting nodes to connect to any previous layer (including the input layer), improves accuracy even more. This setting preserves the benefits of a fixed depth, while increasing expressiveness by enabling skip-like connections across layers. From an optimization perspective, these additional shortcuts reduce the need to propagate useful signals through many intermediate nodes and mitigate the risk of losing informative features due to destructive mutations in a single layer. In other words, the "any previous layer" option retains a clear layered structure but offers a richer wiring space, which appears beneficial for MNIST classification and leads to the best result among the tested connectivity constraints (78% test accuracy).

5.5 Summary of Best Configurations

This section summarizes the strongest configurations observed across the parameter sweeps and reports the best-performing settings under a consistent experimental protocol. Unless otherwise stated, all runs in this summary adopt the relaxed layered connectivity (nodes may connect to any previous layer, including inputs), use point mutation, and are trained with a fixed batch size of 4000 images.

Baseline training and fine-tuning protocol. As a first step, we evolved an LGN from scratch for 46,000 generations using an initial point mutation rate of 0.0005, adapted via the 1/5 success rule. Under this baseline configuration, the best checkpoint reached a test accuracy of 75%. Starting from this checkpoint, we then performed a fine-tuning phase, where we continued evolution while varying specific hyperparameters (primarily the Loss function and the mutation configuration) to assess how sensitive the final performance is to these choices. For both the initial training and the fine-tuning phase, the Loss used was L in Equation 4.2. The best fine-tuning runs obtained from this procedure are summarized in Table 5.7.

A few considerations can be drawn from Table 5.7. First, during the fine-tuning phase, the configuration with $Y = 0.25$ achieved better results than the one with $Y = 0.75$, even though in the earlier experiments the latter performed better during

X	Y	Mutation	Generations	Test accuracy (%)
10	0.75	Function mutation	20,000	78.3
10	0.75	point mutation	55,000	78.80
10	0.25	point mutation	55,000	80.1

Table 5.7: Best fine-tuning configurations starting from the 75% checkpoint.

training from scratch. A plausible explanation is that, once the network had already learned a reasonably good bit-level organization during the first 46,000 generations, fine-tuning benefited more from a Loss that pushed more strongly toward *correct classification* than toward further refining the bitwise arrangement. In other words, the bitwise work had already been mostly done in the pre-training stage, so a more aggressive scaling on correctly classified samples became advantageous in the refinement stage.

Another interesting observation concerns the **function mutation** experiment. Although its final accuracy did not exceed the best point-mutation configuration, it reached **78.3%** in only **20,000 generations**, which is a promising result in terms of convergence speed. This suggests that mutating only the function genes can be an effective way to quickly refine an already good topology, since it allows the algorithm to explore different gate assignments without disrupting the network connectivity. However, in our experiments this strategy did not continue improving beyond that point, indicating that function mutation alone may be useful as a short-term refinement operator, but insufficient to produce the best final solution without the support of topology-changing mutations.

Overall, the experiments presented in this chapter show that the proposed evolutionary pipeline is highly sensitive to a small set of key hyperparameters (Loss design, batch regime, mutation schedule, and connectivity constraints), and that consistent improvements can be obtained by combining relaxed layered wiring with adaptive mutation control and large batch evaluation.

Parameter	Value
Loss	Eq. 4.2 ($X = 10$; $Y = 0.75$ train, $Y = 0.25$ fine-tuning)
Mutation	Point mutation with 1/5 rule
Batch size	4000
Generations	46,000 + 55,000 (fine-tuning)
Best test accuracy	**80.1%**

Table 5.8: Reference configuration used for the final evaluation.

In the next chapter, we move from parameter sweeps to an analysis of the overall results: we select the reference configuration shown in Table 5.8 and examine its behavior in terms of accuracy, generalization gap, and computational cost. We also compare our best evolved LGNs against state-of-the-art logic-based approaches on MNIST.

Chapter 6

Results Analysis

This chapter analyzes the behavior of the proposed CGP-based approach beyond the final test accuracy alone. In particular, we focus on four aspects: (i) how the accuracy evolves across generations, (ii) how the logic gates are distributed across the layers of the evolved network, (iii) the computational cost of the training process, and (iv) how the final model compares with the current state-of-the-art. The goal is not only to report the best score achieved, but also to understand the strengths and limitations of the proposed methodology.

6.1 Accuracy Evolution Across Generations

Figure 6.1 reports the trend of the test accuracy during evolution. The overall behavior can be divided into two distinct phases. In the first phase, corresponding to the initial training from scratch, the accuracy increases rapidly during the first generations. This suggests that the evolutionary process is able to discover relatively quickly a set of useful logical building blocks and a coarse network structure capable of separating a non-trivial portion of the MNIST classes.

In the second phase, corresponding to fine-tuning from the best checkpoint, the improvement becomes much slower. The accuracy still increases, but only through small and irregular increments. A plausible explanation is the intrinsically *discrete* nature of the search space. Unlike gradient-based optimization, where the parameters can be updated through very small continuous steps, here every improvement requires a mutation that changes the genotype in a meaningful but non-destructive way. Even when the mutation rate is very small, the search space remains combinatorial, and therefore the optimization process cannot follow a smooth descent trajectory. As a consequence, once evolution reaches a reasonably good solution, further improvements become increasingly difficult and sparse.

This behavior is consistent with the results observed in Chapter 5: coarse

improvements are relatively easy to obtain, while pushing the model toward higher accuracy requires a much larger computational effort and a careful fine-tuning of the evolutionary hyperparameters.

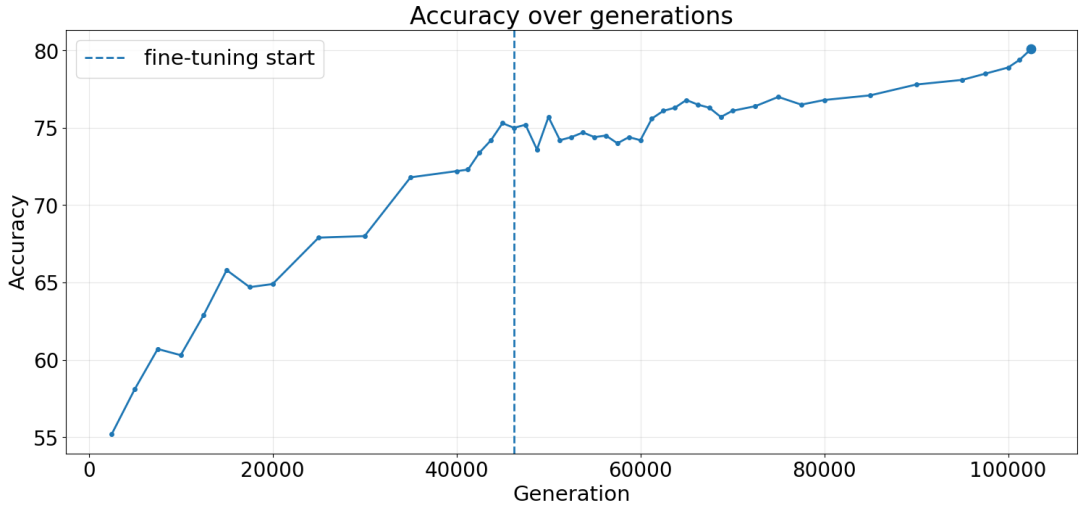

Figure 6.1: Evolution of test accuracy across generations. The curve shows a rapid increase during the early training stage and a much slower improvement during fine-tuning.

6.2 Gate Distribution Across Layers

A second aspect worth analyzing is the internal composition of the evolved LGN. Since the genotype allows all 16 Boolean operators, it is useful to inspect whether evolution converges toward a strong preference for a small subset of gates, or whether different layers retain a more heterogeneous distribution.

Distribution over all nodes If all nodes in the network are considered, each layer contains 16,000 nodes by construction, and the gate distribution remains relatively uniform. This is expected, because a large fraction of the genotype is never expressed in the phenotype: many nodes are present in the genome only as latent genetic material and are never used in the active computational graph. Therefore, when all nodes are counted indiscriminately, the resulting distribution mostly reflects the random initialization and the broad exploration of the genotype space, rather than the actual logical structure used for classification.

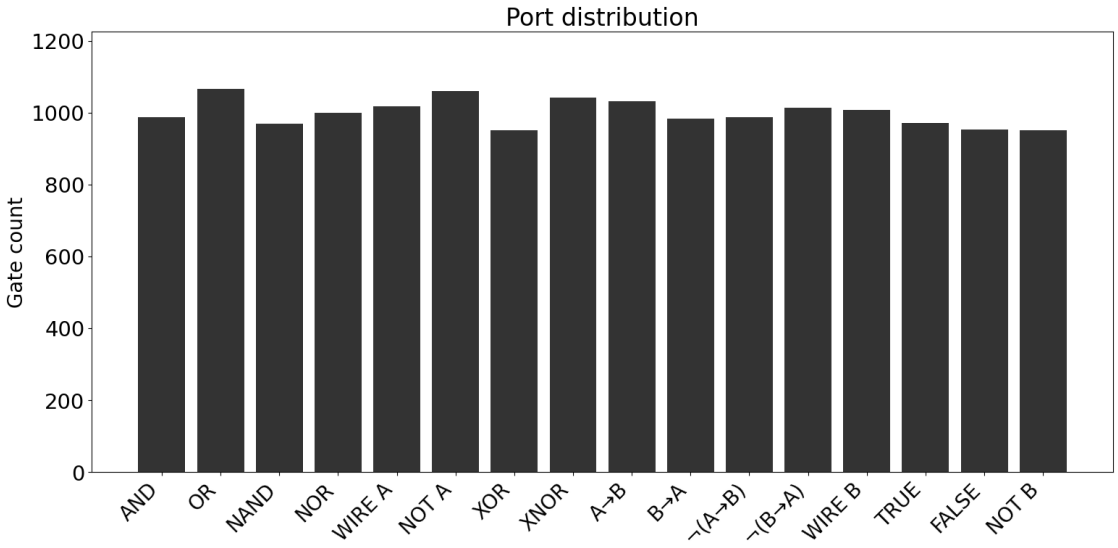

Figure 6.2: Gate distribution in a single layer

Distribution over active nodes A more informative analysis is obtained by considering only the *active* nodes, i.e., the subset of gates that effectively contributes to the final outputs. Figures 6.3 to 6.7 show the gate counts restricted to the active nodes of each layer.

The first relevant observation is that the number of active nodes decreases sharply from one layer to the next. In the analyzed checkpoint, the number of active nodes is:

$$2730 \to 1441 \to 711 \to 317 \to 109.$$

This trend shows that the effective computation graph becomes progressively sparser toward the output layers. In practice, although each layer contains 16,000 available nodes, only a small fraction is actually used, and this fraction is roughly halved at each successive layer. This is a direct consequence of the sparse nature of the network and of the CGP representation, where many nodes remain inactive and therefore act only as a reserve of unused genetic material.

A second interesting observation concerns the type of gates used in the deeper layers. In the final layers, the `TRUE` and `FALSE` constant gates appear much less frequently than in the earlier ones. This suggests that the last stages of the network rely less on trivial constant outputs and more on gates that combine or transform already extracted logical features. In other words, the deeper part of the active graph seems to be dedicated to refining class-discriminative combinations rather than injecting constant signals.

More generally, the active-gate histograms show that evolution does not collapse to a single dominant operator. Instead, several gates remain represented, although

with different frequencies. This indicates that the final classifier is not based on a single logical primitive, but rather on a heterogeneous combination of Boolean transformations distributed across layers.

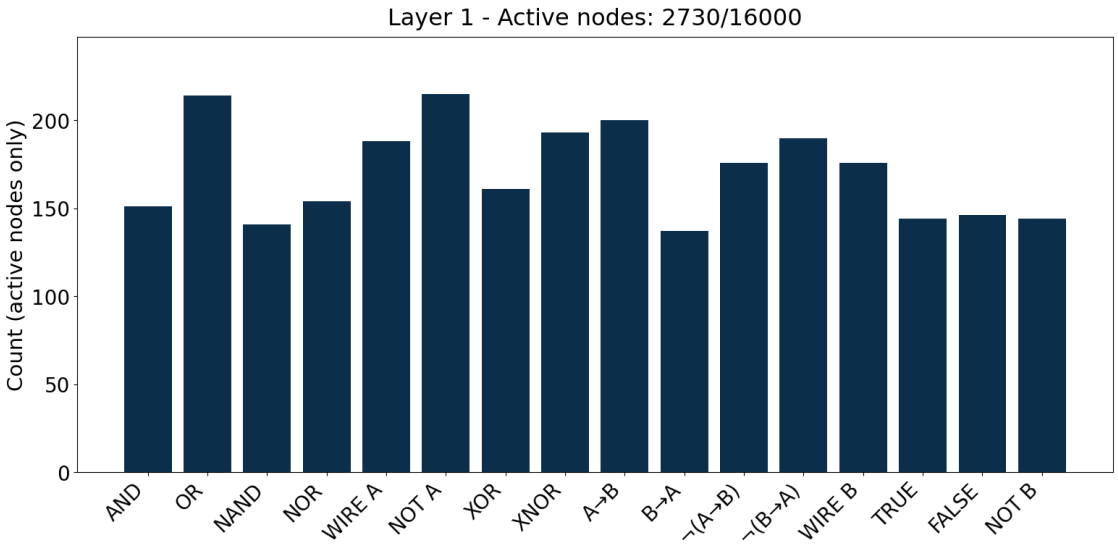

Figure 6.3: Distribution of active gates in Layer 1.

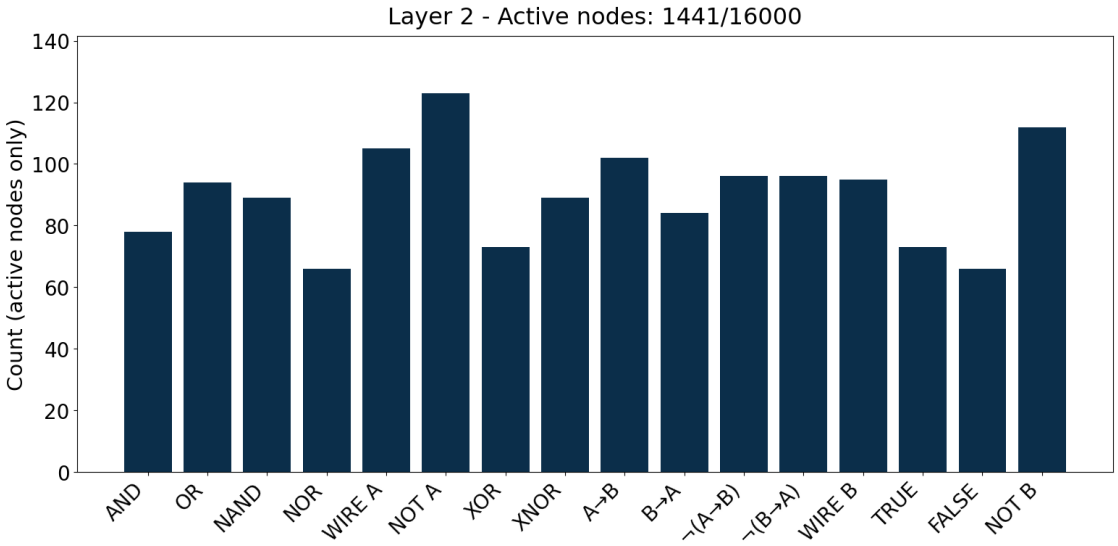

Figure 6.4: Distribution of active gates in Layer 2.

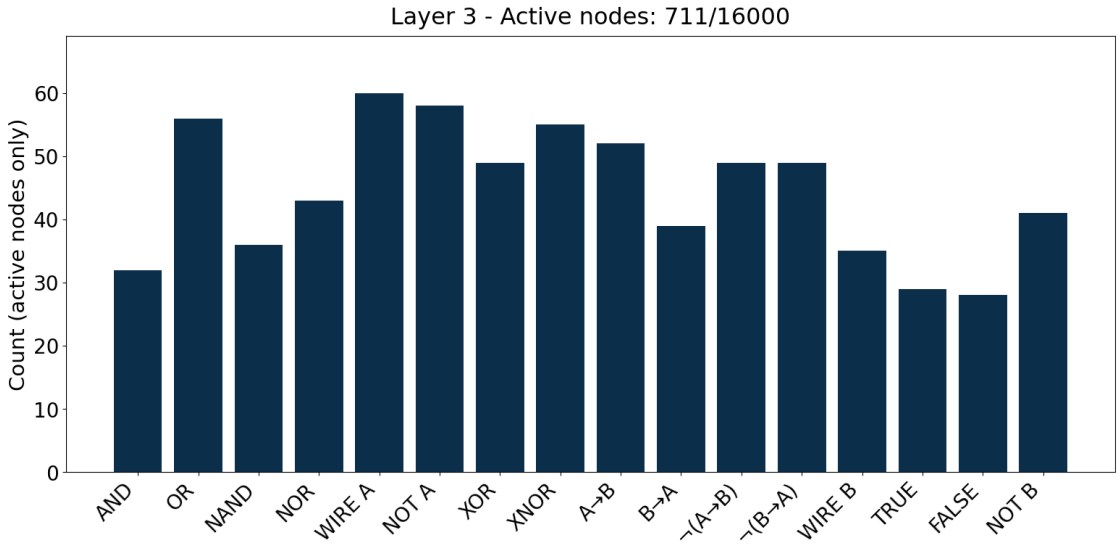

Figure 6.5: Distribution of active gates in Layer 3.

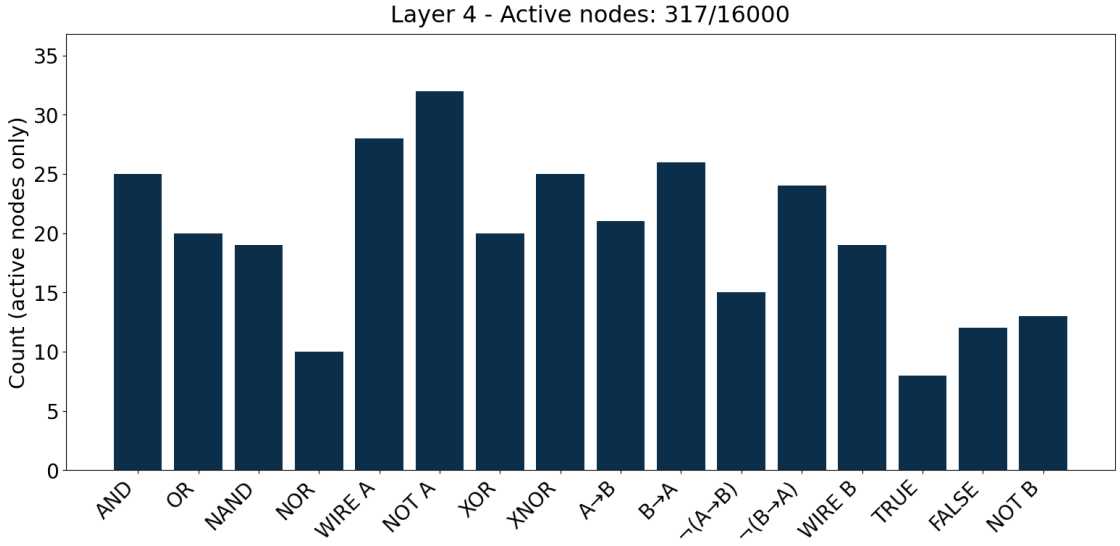

Figure 6.6: Distribution of active gates in Layer 4.

6.3 Computational Cost

An important limitation of the proposed approach is its computational cost. In the current implementation, using a batch of 4000 images, a $(4 + 12)$ evolutionary strategy, and a single CPU core, the algorithm requires approximately **24 hours**

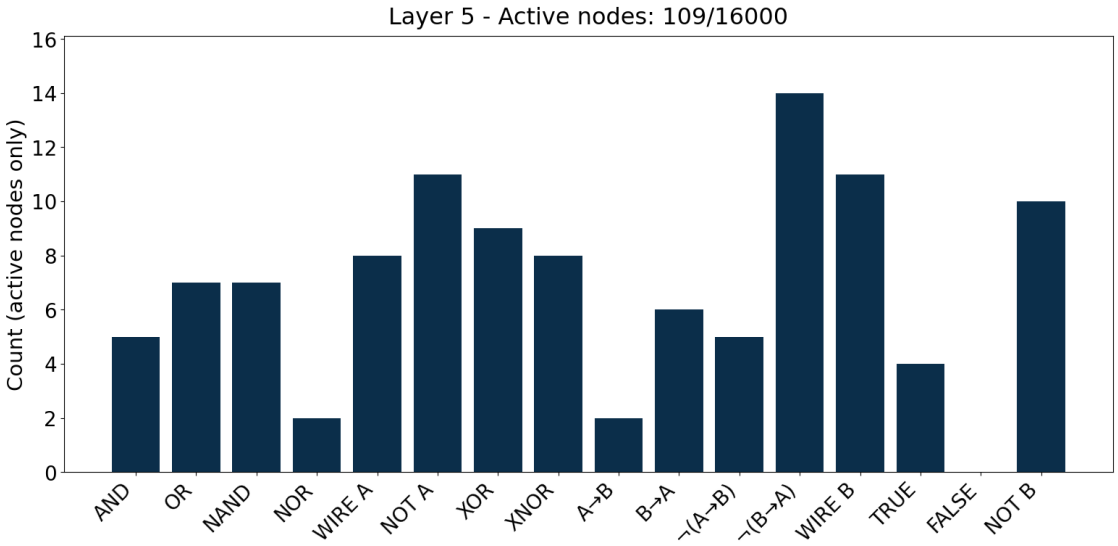

Figure 6.7: Distribution of active gates in Layer 5.

to execute **1500 generations**. This corresponds to about

$$\frac{24 \cdot 3600}{1500} \approx 57.6 \text{ seconds/generation.}$$

This runtime is significant, especially when compared to gradient-based methods, which can exploit highly optimized tensor operations and large-scale parallelization on GPUs. In our case, the cost is driven by the repeated evaluation of multiple individuals over large batches in a fully discrete setting. Since every offspring must be explicitly decoded and evaluated, the total training time grows quickly with the number of generations required for fine-tuning.

For this reason, one of the most promising directions for future work is the **parallelization of the Loss evaluation stage**. Since the evaluation of each individual and, to a large extent, the evaluation over samples are naturally parallelizable, a multi-core CPU implementation or a GPU-oriented reformulation could substantially reduce the wall-clock time of evolution. Such an improvement would make it feasible to explore larger populations, more generations, and richer hyperparameter sweeps.

6.4 Comparison With The State-of-the-art

Table 6.1 compares the best result obtained in this thesis with representative state-of-the-art logic-based approaches on MNIST. As expected, the differentiable approaches achieve higher accuracy than the model evolved in this work. In

particular, Deep Differentiable Logic Gate Networks (DiffLogic) reach substantially stronger classification performance, thanks to the use of gradient-based optimization and highly efficient implementations.

Model	**Training paradigm**	**Accuracy (%)**
Proposed CGP-LGN	Evolutionary / gradient-free	80.1
Diff Logic Net (small) [4]	Differentiable / gradient-based	97.69
Diff Logic Net [4]	Differentiable / gradient-based	98.47

Table 6.1: Comparison between the best evolved LGN obtained in this thesis and representative state-of-the-art logic-based models on MNIST.

Nevertheless, the comparison should be interpreted carefully. The model proposed in this thesis was trained **without gradient descent** and **without parallelization**, using a fully discrete evolutionary search over both topology and gate selection. From this perspective, a final accuracy of 80.1% should not be regarded as a competitive endpoint, but rather as a meaningful starting point. It demonstrates that even a pure CGP-based, gradient-free pipeline can discover non-trivial logical classifiers on MNIST, and that further improvements may be obtained through better mutation operators, more efficient evaluation, larger computational budgets, or hybrid search strategies.

Overall, the results suggest that the proposed approach is capable of evolving a valid LGN for image classification, but also highlight the main bottlenecks that currently limit its performance. On the positive side, the experiments show that: (i) the evolutionary process can significantly improve accuracy from scratch, (ii) structured layered connectivity is beneficial, and (iii) the final networks are highly sparse, using only a small subset of the available nodes. On the negative side, the discrete search space makes late-stage improvements slow, and the current implementation is computationally expensive.

Taken together, these observations support the main conclusion of this thesis: Cartesian Genetic Programming is a viable framework for generating Logic Gate Networks, but substantial room remains for improving both optimization efficiency and final predictive performance. In this sense, the results obtained provide a basis for future work rather than a final solution.

Chapter 7

Conclusions

This thesis investigated the use of evolutionary techniques for the generation of LGNs, with a particular focus on Cartesian Genetic Programming as a gradient-free alternative to differentiable training methods. The work started from the observation that Logic Gate Networks are attractive models for efficient inference, since they replace floating-point arithmetic with Boolean operations that are naturally aligned with digital hardware. At the same time, their training remains challenging because both gate selection and connectivity are discrete, making standard gradient-based optimization unsuitable in their native form.

Within this context, the main objective of the thesis was to study whether CGP could be adapted into a practical framework for evolving LGNs and applying them to a real classification task. To this end, the work proceeded along two parallel directions. First, it analyzed the theoretical background of LGNs, GP, and CGP, positioning the proposed approach with respect to the current state-of-the-art. Second, it extended the CGP++ framework to support the classification of binarized MNIST images, introducing problem-specific mechanisms such as layered connectivity constraints, custom Loss functions, batch-based training, and specialized mutation operators.

The experimental results show that the proposed pipeline is indeed capable of evolving valid logic-based classifiers. In particular, the thesis demonstrated that:

- CGP can be successfully adapted to the generation of layered LGNs for image classification;
- a fixed layered structure improves performance with respect to standard CGP connectivity;
- allowing connections from any previous layer further increases expressiveness and leads to better accuracy;

- adaptive mutation schedules and fine-tuning from good checkpoints are important to push the search beyond the first coarse solutions;
- even in a fully discrete and gradient-free setting, the evolved networks can reach meaningful predictive performance on MNIST.

The best configuration obtained in this work reached a test accuracy of **80.1%**. Although this result remains below the current state-of-the-art achieved by differentiable logic-based models such as DDLGN, it should be interpreted in light of the constraints of the present setup: the network was trained without gradient descent, without parallelized evaluation, and through a purely combinatorial search process over both topology and gate functions. From this perspective, the result provides evidence that CGP is not only theoretically compatible with LGNs, but also practically capable of producing non-trivial logical classifiers.

At the same time, the thesis also highlights the main limitations of the current approach. First, the optimization process is computationally expensive: the evaluation of many individuals over large batches on a single CPU core makes long runs costly in wall-clock time. Second, the discrete nature of the search space makes late-stage optimization difficult, since small improvements cannot be reached through smooth updates as in gradient-based methods. Third, while the proposed Loss functions and connectivity constraints improved the results, they do not completely solve the challenge of efficiently navigating a very large combinatorial design space.

These limitations naturally suggest several directions for future work. A first priority is the parallelization of evaluation function, either across multiple CPU cores or through GPU-oriented bitwise implementations. This would allow much larger evolutionary budgets and make the overall training process substantially more practical. A second direction concerns the design of more informed mutation operators, capable of modifying active subgraphs in a less destructive way and of exploiting the structure of the evolved network more effectively than uniform point mutation alone. A third avenue is the exploration of hybrid optimization strategies, where evolutionary search is combined with local improvement procedures or with differentiable relaxations in order to couple the flexibility of CGP with the efficiency of continuous optimization.

Further work may also investigate more expressive architectural priors, for example CGP representations that better mimic convolutional or residual structures, as well as the direct application of the evolved networks to hardware targets such as FPGAs. Since LGNs are intrinsically aligned with digital logic, one of the most promising long-term directions is to close the gap between model evolution and deployable hardware descriptions, turning the evolutionary process into a genuine logic-synthesis pipeline for machine learning tasks.

In conclusion, this thesis shows that Cartesian Genetic Programming is a viable

framework for generating Logic Gate Networks, even on a challenging benchmark such as MNIST. While the obtained results do not yet match the accuracy of the best differentiable approaches, they establish a concrete proof of concept for a fully discrete, gradient-free methodology. More importantly, they open a promising research direction at the intersection of evolutionary computation, logic synthesis, and efficient machine learning, where further methodological and computational improvements may significantly strengthen the potential of the approach.

Bibliography

[1] Alex de Vries. «The growing energy footprint of artificial intelligence». In: *Joule* 7.10 (2023), pp. 2191–2194. ISSN: 2542-4351. DOI: 10.1016/j.joule.2023.09.004. URL: https://www.sciencedirect.com/science/article/pii/S2542435123003653 (cit. on p. 1).

[2] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. «Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1». In: *Advances in neural information processing systems (NeurIPS)*. Vol. 29. 2016 (cit. on pp. 1, 15).

[3] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. «Quantization and training of neural networks for efficient integer-arithmetic-only inference». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713 (cit. on p. 1).

[4] Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. *Deep Differentiable Logic Gate Networks*. 2022. arXiv: 2210.08277 [cs.LG]. URL: https://arxiv.org/abs/2210.08277 (cit. on pp. 1, 3, 4, 11, 15, 46).

[5] Felix Petersen, Hilde Kuehne, Christian Borgelt, Julian Welzel, and Stefano Ermon. *Convolutional Differentiable Logic Gate Networks*. 2024. arXiv: 2411.04732 [cs.LG]. URL: https://arxiv.org/abs/2411.04732 (cit. on pp. 1, 3, 13, 15).

[6] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Vol. 1. Cambridge, MA, USA: MIT Press, 1992. ISBN: 9780262111706 (cit. on pp. 1, 5).

[7] Julian F. Miller. «Cartesian Genetic Programming». In: *Cartesian Genetic Programming*. Ed. by Julian F. Miller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 17–34. ISBN: 978-3-642-17310-3. DOI: 10.1007/978-3-642-17310-3_2. URL: https://doi.org/10.1007/978-3-642-17310-3_2 (cit. on pp. 1, 9).

[8] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. «Binary neural networks: A survey». In: *Pattern Recognition* 105 (Sept. 2020), p. 107281. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2020.107281. URL: http://dx.doi.org/10.1016/j.patcog.2020.107281 (cit. on p. 4).

[9] Abdul Manazir and Khalid Raza. «Recent Developments in Cartesian Genetic Programming and its Variants». In: *ACM Computing Surveys* 51.6 (Jan. 2019), Article 122. ISSN: 0360-0300. DOI: 10.1145/3275518. URL: https://doi.org/10.1145/3275518 (cit. on p. 6).

[10] Garnett Wilson and Wolfgang Banzhaf. «A Comparison of Cartesian Genetic Programming and Linear Genetic Programming». In: *Genetic Programming (EuroGP 2008)*. Vol. 4971. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 182–193. DOI: 10.1007/978-3-540-78671-9_16 (cit. on p. 6).

[11] Robert I. McKay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'Neill. «Grammar-based Genetic Programming: a survey». In: *Genetic Programming and Evolvable Machines* 11.3 (2010), pp. 365–396. DOI: 10.1007/s10710-010-9109-y. URL: https://doi.org/10.1007/s10710-010-9109-y (cit. on p. 7).

[12] Tim Perkis. «Stack-based genetic programming». In: *Proceedings of the First IEEE Conference on Evolutionary Computation*. Vol. 1. July 1994, pp. 148–153. ISBN: 0-7803-1899-4. DOI: 10.1109/ICEC.1994.350025 (cit. on p. 7).

[13] Julian Miller and Andrew Turner. «Cartesian Genetic Programming». In: *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*. GECCO Companion '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 179–198. ISBN: 9781450334884. DOI: 10.1145/2739482.2756571. URL: https://doi.org/10.1145/2739482.2756571 (cit. on pp. 7–10).

[14] Yaman Umuroglu, Yash Akhauri, Nicholas J. Fraser, and Michaela Blott. «LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications». In: *arXiv preprint arXiv:2004.03021*. Dublin, Ireland: Xilinx Research Labs, Apr. 2020. URL: https://arxiv.org/abs/2004.03021 (cit. on p. 15).

[15] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (cit. on p. 18).

[16] Roman Kalkreuth and Thomas Bäck. «CGP++: A Modern C++ Implementation of Cartesian Genetic Programming». In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '24. New York, NY, USA: ACM, 2024, pp. 13–22. DOI: 10.1145/3638529.3654092 (cit. on p. 21).

[17] Terry Jones. «A Description of Holland's Royal Road Function». In: *Evolutionary Computation* 2 (Dec. 1994), pp. 409–423. DOI: 10.1162/evco.1994.2.4.409 (cit. on pp. 23, 24).

[18] Ingo Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart: Frommann-Holzboog Verlag, 1973 (cit. on p. 29).