EDAG: Evolutionary DAG Search for Neural Mechanism Discovery

Typed Directed Graphs, Behavioral Quality–Diversity, and a Self-Expanding Operator Library

Jovonni L. Pharr
Georgia Cyber Warfare Range
info@gacwr.org

January 3, 2026

v0.0.1

Abstract

EDAG is an evolutionary neural architecture search system whose unit of search is not a scalar accuracy over a fixed menu of blocks but a *learning mechanism*: a candidate is a typed directed graph whose edges carry one of three kinds—feedforward, state, and recurrent—so a single representation expresses feedforward stacks, convolutional and attention blocks, graph message passing, recurrence, and persistent memory without a change of formalism. A genome is declared valid when its feedforward edges form a directed acyclic graph with a path from an input node to an output node; state and recurrent edges may close cycles, and it is through those cycles that memory cells and recurrence are expressed, so a candidate compiles to a single-step forward pass while still carrying internal state that persists across steps. The compiler walks the feedforward subgraph in topological order, materialises one `torch` module per node across eighteen node types, routes the input tensor through a modality adapter for image, sequence, or graph data, and runs memory cells under a `standard`, `ema`, or `gru` update rule. Search is a steady-state, age-regularised evolutionary loop that selects on fitness plus a behavioral-novelty bonus, draws parents both from a tournament and from a MAP-Elites quality–diversity archive binned on depth and statefulness, and recombines, mutates, and weight-inherits: a child is initialised from its primary parent's weights for every layer that still matches, then trained under a three-tier staged budget that only promotes a candidate to longer training once it clears a competence gate. A behavioral descriptor read from the run itself—the mean and standard deviation of the output logits, the norm of a single parameter update, and the change in memory state—places each candidate in the archive, and every twenty-five evaluations the system mines the graphs it has seen for recurring two-, three-, and four-node chains and for common memory update rules, promoting the most frequent of them into a macro library that later children can splice in as first-class operators, so the search space expands as the run proceeds. The engine is implemented twice, in Python over PyTorch and in Rust over `tch-rs`, with a deterministic batched evaluation order so the two stay aligned, and it runs over MNIST, KMNIST, FashionMNIST, SVHN, STL-10, CIFAR-10, and two checked-in tiny text and graph datasets, emitting for each run a metrics report with a rendered genome, a JSON-Lines lineage of every birth with its parents and operator, the best genome and its weights, and a TorchScript (optionally ONNX) export of the discovered model.

Contents

I Preliminaries **2**

1 Introduction **2**

2 Related Work **3**

II The EDAG System **4**

3 The Genome: A Typed Directed Graph with Three Edge Kinds **4**

4 Compiling and Running a Genome **5**

5 Mutation, Crossover, and Macro Injection **5**

6 The Search Loop **6**

7 Behavioral Novelty and the Quality–Diversity Archive **7**

8 Macro-Evolution: A Self-Expanding Operator Library **7**

9 Weight Inheritance and Staged Evaluation **8**

10 Search Modes **8**

11 Implementation **8**

12 Deployment **9**

13 Reporting and Export **9**

14 A Run **9**

III Conclusion **9**

15 Conclusion **10**

Part I
Preliminaries

1 Introduction

Neural architecture search is usually posed as the optimisation of a scalar. A space of architectures is fixed in advance—a backbone with a handful of choice points, a cell to be repeated, a menu of operations—and a controller, an evolutionary population, or a differentiable relaxation drives a single number, validation accuracy, perhaps penalised by size or latency, toward its maximum. The product of such a search is a point in a space whose shape a human has already decided; the search rearranges known parts.

EDAG makes a different commitment. Its unit of search is a *mechanism*. A candidate is a typed directed graph, and the operators that create candidates are allowed to change not only which layers are present and how they are wired but whether the model carries state at all, how that state is updated, and whether computation feeds back on itself across steps. The representation is designed so that feedforward stacks, convolutional and pooling blocks, attention, graph message passing, recurrence, and explicit memory are all expressible as the same object—a graph over typed nodes—rather than as separate templates. A candidate is therefore free to become a convolutional network, a recurrent network, a memory-augmented sequence model, or a hybrid that corresponds to no standard family, and the search is free to move between those regions by local edits.

Three design choices follow from this commitment and organise the rest of the paper. First, the genome separates a model's feedforward skeleton, over which a single forward pass is compiled, from its state and recurrent edges, over which internal memory and feedback are carried (Section 3, Section 4). Second, the search does not optimise a single global best; it maintains a quality–diversity archive keyed on *behavior*, so that models which behave differently are preserved even when their structures are close, and it protects young and unusual candidates with a staged evaluation budget that spends compute only on candidates that have already shown competence (Section 6, Section 7, Section 9). Third, the operator set is not fixed: recurring substructures and memory update rules found among the elites are promoted into a macro library and become building blocks that later candidates can use directly, so the search space grows as the run proceeds (Section 8). The system is implemented twice, in Python and in Rust, with a deterministic evaluation order that keeps the two in step (Section 11), and every run records its metrics, the lineage of every candidate, and an exportable copy of the best model it found (Section 13).

2 Related Work

EDAG draws on four lines: evolutionary and reinforcement-learning neural architecture search, the direct encoding of networks as graphs, quality–diversity and novelty search, and the reuse of computation to make architecture evaluation affordable.

Evolutionary and RL architecture search. Large-scale evolution of image classifiers showed that a simple evolutionary algorithm, mutating a population of architectures from trivial seeds, can reach competitive networks given compute [1], and regularized evolution refined the selection rule by removing the *oldest* individual at each step rather than the worst, which keeps the population churning and favours architectures that remain good when retrained [2]. A parallel line drives search with a reinforcement-learning controller that emits architecture descriptions and is updated by policy gradient on the resulting accuracy [3]. EDAG implements the age-regularised steady-state loop as its default and keeps both a random-search and an RL-controller mode behind the same evaluation pipeline (Section 10).

Graph encodings of networks. Neuroevolution of augmenting topologies evolved both weights and structure from minimal seeds, using historical markings to align genomes and complexifying gradually [4]. EDAG inherits the direct-encoding stance—every node and edge is explicit and mutable—and the discipline of starting from a minimal seed, while widening the node alphabet to modern layer types and adding typed edges that carry state. The ambition to treat convolutional, recurrent, and graph networks as instances of one formalism is shared with recent algebraic accounts of architecture [5].

Quality–diversity and novelty. Novelty search selects for behavioral difference rather than for objective value [6], and MAP-Elites illuminates a space by keeping the best solution found in each cell of a behavioral grid [7]. EDAG combines the two: a novelty score computed as the sparseness of a candidate among its neighbours feeds the selection objective, and a MAP-Elites archive binned on behavioral coordinates both preserves diversity and supplies parents to the next generation (Section 7).

Weight inheritance and cheap evaluation. Reusing a parent's trained weights to warm-start a mutated child, and spending full training only on promising candidates, are standard devices for making evolutionary search affordable [8]. EDAG inherits weights for every layer that still matches after a mutation and gates training through three budget tiers (Section 9).

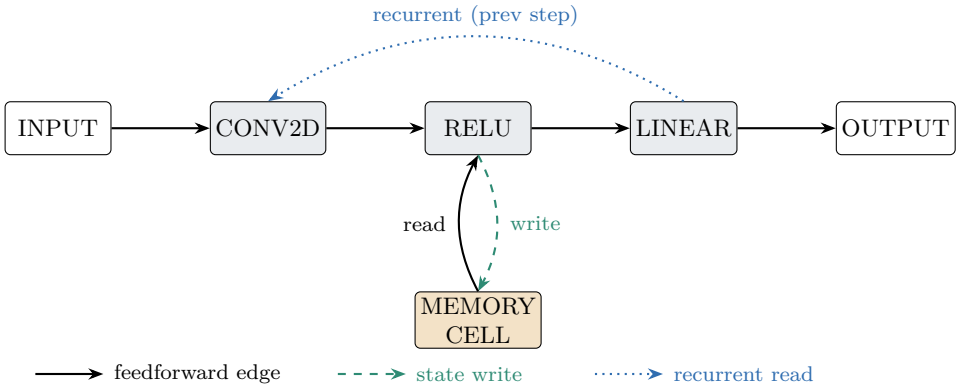


Figure 1: A genome is a typed directed graph. Feedforward edges (solid) carry the single-step forward pass and must form a directed acyclic graph with a path from an input to an output. A memory cell is written through a *state* edge (dashed) and read back through a feedforward edge, so it holds a persistent buffer across steps; a *recurrent* edge (dotted) feeds a later node's output into an earlier node on the following step. State and recurrent edges may close cycles that the feedforward validity condition ignores.

Part II
The EDAG System

3 The Genome: A Typed Directed Graph with Three Edge Kinds

A candidate architecture is a directed graph over typed nodes. A node carries a type drawn from an alphabet of eighteen kinds—an input and an output marker; the feedforward layers `LINEAR`, `RELU`, `CONV2D`, `MAXPOOL2D`, `BATCH_NORM`, `DROPOUT`, `FLATTEN`, and `LAYER_NORM`; the sequence and graph operators `TOKENIZE`, `ATTENTION`, `EMBEDDING`, `LSTM`, and `MESSAGE_PASS`; the structural join `RESIDUAL_ADD`; and the mechanism nodes `MEMORY_CELL` and `RECURRENT`— together with structural parameters (channel counts, kernel size, stride) and two mechanism fields: a boolean marking the node as stateful, a state size, and a string update rule. Every edge carries a *kind*,

$$\mathsf{kind} \in \{\, \texttt{ff},\ \texttt{state},\ \texttt{recurrent} \,\},$$

and it is this typing of edges, not the node alphabet alone, that lets one representation carry both a compilable forward pass and persistent internal state. Figure 1 shows a small genome with all three.

The feedforward subgraph is obtained by keeping only the `ff` edges. A genome is valid exactly when this subgraph is acyclic and connects an input to an output:

Definition 1 (Validity)**.** *Let G be a genome and G_{ff} the subgraph on its `ff` edges. G is* valid *iff G_{ff} is a directed acyclic graph and there exist an input node i and an output node o with a directed path from i to o in G_{ff}.*

Because validity is a condition on the feedforward subgraph only, the `state` and `recurrent` edges are free to form cycles. A memory cell is attached by a state edge that *writes* into it and a feedforward edge that *reads* from it on the same step; a recurrent edge connects two nodes so that the source's output on one step is available to the destination on the next. The genome serialises to a JSON object of nodes and typed edges and is reconstructed from it, so a candidate is a portable, inspectable artifact rather than a live object graph.

4 Compiling and Running a Genome

A valid genome is compiled into a `torch` module. The compiler walks the feedforward subgraph in topological order and materialises one module per node: a `RELU` node becomes a linear map followed by a rectifier, a `CONV2D` node a lazily sized convolution and rectifier, an `ATTENTION` node the four projections of a single-head attention, a `MESSAGE_PASS` node a self-and-neighbour pair that mixes along an adjacency supplied with the input, an `LSTM` node a recurrent layer, and so on across the alphabet; all internal activations share a fixed working width, and a final head projects to the number of classes. The input tensor is first routed through a modality adapter—images are kept as spatial tensors when convolutions are present, unfolded into patch tokens when sequence operators are present, split into features and an adjacency when message passing is present, and otherwise flattened—so that one representation serves image, sequence, and graph data.

The forward pass is a single sweep over the feedforward subgraph in topological order. A node's input is the sum of its feedforward predecessors' outputs on the current step plus the sum of its recurrent predecessors' outputs *from the previous step*. A memory cell is the one place where state edges act: it reads what its state-edge predecessors wrote, combines that write with its own previous buffer under its update rule, and emits the new buffer, which it also stores. The update rule is one of three forms,

$$\begin{aligned}
\textsf{standard}:&\quad h_t = \tanh\big(W[\,x_t \,\|\, h_{t-1}\,]\big),\\
\textsf{ema}:&\quad h_t = \alpha\, h_{t-1} + (1-\alpha)\, W x_t,\\
\textsf{gru}:&\quad z = \sigma\big(W_z[\,x_t \| h_{t-1}\,]\big),\quad \tilde{h} = \tanh\big(W_h[\,x_t \| z \odot h_{t-1}\,]\big),\quad h_t = (1-z)\, h_{t-1} + z\,\tilde{h},
\end{aligned}$$

where x_t is the aggregated write and h_{t-1} the retained buffer. Whether the buffer and the previous-step outputs actually persist is governed by a flag: during training, and during evaluation only when explicitly enabled, the cell commits its new state and the pass stashes node outputs for the next step's recurrent reads; otherwise the pass is a pure function and evaluation is deterministic. This is the precise sense in which a candidate is a *stateful computational program* rather than a static feedforward graph: the same genome can read and write internal memory as it runs.

5 Mutation, Crossover, and Macro Injection

New candidates are produced by mutation, by crossover, and by the injection of promoted macros. A mutation is chosen from six operators by the fixed distribution of Table 1. Two of the six act on mechanism directly: `add_memory` creates a `MEMORY_CELL` with a sixty-four wide state, wires a target node into it through a state edge (the write) and the cell back to the target through a feedforward edge (the read); `add_recurrence` adds a recurrent edge. The structural operators split an edge with a new node whose type is itself drawn from a distribution—most often a rectifier or linear map, less often a convolution, pooling, normalisation, attention, message-passing, or LSTM node—add or delete a feedforward edge subject to the validity condition, remove a non-terminal node and rewire its feedforward neighbours, or tweak a node's parameters. Every operator either preserves validity or is undone when it breaks it, so the population is always compilable.

Crossover takes two parents, keeps the first parent's nodes, adds a subset of the second parent's nodes under an identifier remapping that avoids collisions, merges the second parent's edges by a coin flip, and repairs the result to the first parent's clone if the merge breaks validity; it is real recombination of two graphs rather than innovation-aligned crossover. A child is produced by crossover with probability 0.25 and is otherwise a clone of its primary parent; in either case a promoted macro is spliced in with probability 0.10 before the mutation operator is applied.

Operator	Effect	Probability
`add_connection`	add a feedforward edge if it keeps the FF graph acyclic	0.35
`add_node`	split an edge with a new typed layer node	0.22
`tweak_node`	perturb a node's channels, kernel, dropout, or patch size	0.13
`remove_node`	delete a non-terminal node, rewire its FF neighbours	0.10
`add_memory`	attach a memory cell (state write, feedforward read)	0.10
`add_recurrence`	add a recurrent edge	0.10

Table 1: The mutation operators and their selection probabilities as implemented. The last two act on mechanism—internal state and feedback—rather than on feedforward topology alone.

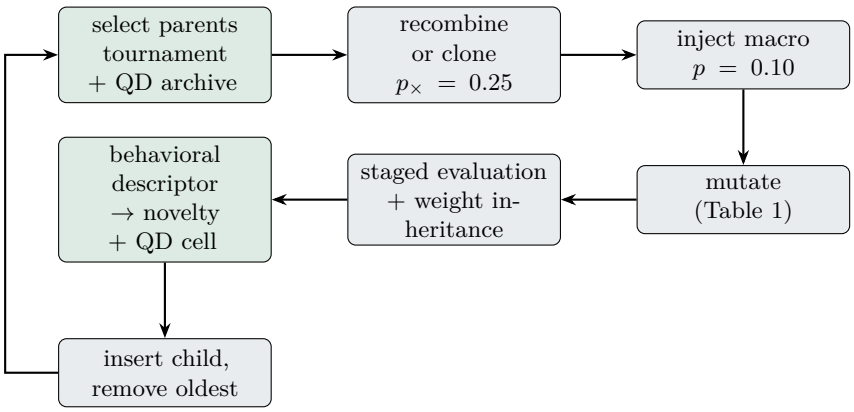


Figure 2: The steady-state loop. Each step selects parents from both a tournament on fitness-plus-novelty and the quality–diversity archive, recombines or clones them, optionally splices in a promoted macro, mutates, and evaluates the child under a staged budget with its parent's weights inherited. A behavioral descriptor read from the run sets the child's novelty and its archive cell; the child is inserted and the oldest individual retired.

6 The Search Loop

The default search is a steady-state, age-regularised evolutionary algorithm (Figure 2). The population is seeded with a minimal, modality-appropriate genome—a single input wired straight to a single output for vector, sequence, and graph inputs, and a small convolutional stem (two convolutions each followed by pooling, then a flatten) for image inputs so the search starts shape-aware rather than flattening pixels. The seed is fully mutable: the operators grow, prune, and rewire it from there. Each step produces one child and removes the oldest individual, so the population size is held fixed and even a strong network is eventually retired, which keeps the search from settling onto one lineage. Selection is on an augmented objective,

$$\text{score}(c) \;=\; \text{fitness}(c) \;+\; w_{\text{nov}} \cdot \text{novelty}(c), \qquad w_{\text{nov}} = 0.25,$$

where fitness is the candidate's evaluation score and novelty is its behavioral sparseness (Section 7). Parents are drawn in two ways at once: with probability 0.40 the primary parent is sampled from the quality–diversity archive and otherwise from a size-three tournament on the augmented score, and the second parent is sampled from the archive with probability 0.35 and from a tournament otherwise. The child is then recombined or cloned, optionally given a macro, mutated, and evaluated with its primary parent's weights inherited; its behavioral descriptor places it in the archive and its score updates the archive cell it falls in.

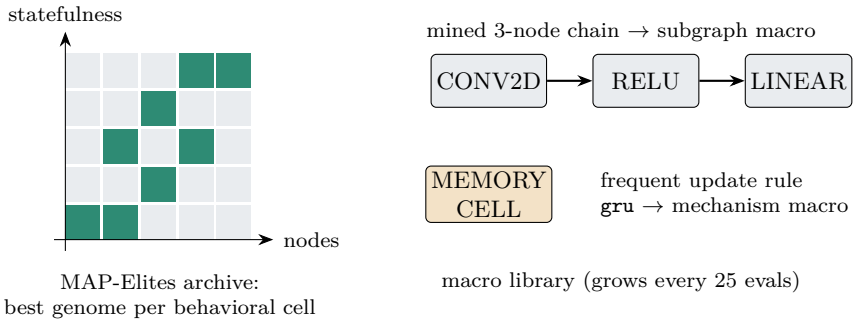


Figure 3: Left: the MAP-Elites archive keeps the best genome in each cell of a behavioral grid (occupied cells shaded) and supplies parents to the loop. Right: periodic mining promotes recurring feedforward chains into subgraph macros and common memory update rules into mechanism macros, which later children can splice in—so the operator library expands as the run proceeds.

7 Behavioral Novelty and the Quality–Diversity Archive

A candidate is described not by its structure but by how it *behaved* while being evaluated. The descriptor is read from the run: the mean and standard deviation of the output logits on a held-out batch, the norm of the parameter change produced by a single additional gradient step, and the norm of the change in memory state. Two candidates with nearly identical graphs but different training dynamics therefore receive different descriptors, and two structurally distant candidates that behave alike are drawn together.

Novelty is the sparseness of a candidate among the behaviors seen so far: the average distance from its descriptor to its nearest neighbours among the current population and an archive of past behaviors. A candidate is admitted to the novelty archive when it both clears a minimum competence (0.05 validation accuracy) and is sufficiently sparse (sparseness above 0.75), so that novelty is only credited to candidates that actually learn.

Diversity is also held structurally, by a MAP-Elites archive. Each genome is placed into a cell of a two-dimensional grid whose coordinates are a coarsened node count and a coarsened measure of statefulness—how many stateful nodes it has, together with its edge count—and each cell retains only the highest-scoring genome that has fallen in it. The archive is both a record of the best solution in each behavioral niche and an active source of parents: sampling a cell uniformly and breeding from its occupant keeps the search spread across niches rather than collapsed onto a single peak (Figure 3, left).

8 Macro-Evolution: A Self-Expanding Operator Library

The operator library is not fixed. Every twenty-five evaluations the system mines the genomes it has seen for recurring substructure: it counts the node-type sequences of every two-, three-, and four-node feedforward chain, and it counts the update rules carried by memory cells. The most frequent chains are promoted into a macro library as reusable subgraphs, and the most frequent memory update rules are promoted as first-class mechanism macros (Figure 3, right). Thereafter a child may, with probability 0.10, have a macro spliced in before mutation: a subgraph macro is inserted with its nodes and edges remapped onto fresh identifiers, and a rule macro rewrites the update rule of one of the child's memory cells. Because the promoted structures become building blocks that later candidates can use directly, the effective search space grows over the run: substructures discovered early become primitives for the architectures built later.

Tier	Train batches	Batch size	Val batches	Promotion gate
early	5	64	3	all candidates
mid	20	64	5	val. accuracy ≥ 0.12
full	50	64	10	val. accuracy ≥ 0.25

Table 2: The default staged evaluation budgets and the competence gates that promote a candidate from one tier to the next. A candidate that reaches the full tier is trained for five passes over its budget.

9 Weight Inheritance and Staged Evaluation

Evaluation trains a freshly compiled candidate and measures its validation accuracy, and two devices keep this affordable. First, *weight inheritance*: a child is initialised from its primary parent's saved parameters, loaded non-strictly so that every layer whose name and shape still match is carried over and only genuinely new layers start from scratch. Because most mutations change the graph only locally, a child usually inherits the bulk of its parent and continues training rather than restarting it.

Second, *staged budgets*: training proceeds in three tiers, and a candidate is promoted to the next tier only when it has shown competence at the current one (Table 2). All candidates receive the early budget; a candidate advances to the mid budget only once its validation accuracy passes 0.12, and to the full budget only once it passes 0.25, where it is trained for several passes over the full-tier budget. Compute is therefore spent in proportion to demonstrated promise, and a young or unusual candidate is given a cheap first look before it is asked to justify a longer one. Optimisation is AdamW at learning rate 10^{-3} under a cross-entropy loss, and under the multi-objective setting the score subtracts a parameter and a wall-time penalty from accuracy; by default those penalties are zero.

10 Search Modes

The evolutionary loop is the default, and two further modes share the same compilation and evaluation pipeline behind a common interface. *Random search* proposes a candidate by applying a short sequence of mutations to a trivial seed and keeps the best evaluated so far. The *RL-controller* mode holds a small policy network that emits a short program of layer types forming a chain genome, evaluates it, and updates the policy by a REINFORCE step on the validation accuracy as reward. Because all three modes call the same evaluator, a run may be configured to any of them without changing the representation, the compiler, or the reporting.

11 Implementation

EDAG is implemented twice: in Python over PyTorch and in Rust over `tch-rs`, which binds the same `libtorch` backend, with the Rust engine building its genome graphs on `petgraph`. The two implementations mirror one another feature for feature—the typed genome and gene, the mutation operators, the novelty and MAP-Elites quality–diversity archives, the macro library, the evaluator, and the reporting each have a counterpart—and the Rust model builder instantiates the same node alphabet, from lazily sized convolutions and a single-head attention written as four projections to message passing as a self-and-neighbour pair and memory cells under the `standard`, `ema`, and `gru` update rules. The two agree on the representation: a genome serialises to a single JSON object, and a cross-language interoperability test loads a genome written by the Python side into the Rust engine and checks that it compiles and runs, so a

candidate produced by one engine is executable by the other. Evaluation can be batched across a thread pool; children in a batch are generated in a fixed order and inserted in that order regardless of the order in which their evaluations finish, so a run is deterministic and the two engines stay aligned, and the Rust engine seeds its generator explicitly for the same reason. Beyond the interoperability check, the test suite covers structural mutation, a forward pass over the tiny graph dataset, dataset loading, and TorchScript export. Data is loaded from the datasets checked into the repository: MNIST, KMNIST, FashionMNIST, SVHN, STL-10, and CIFAR-10 through the standard vision readers, and two small self-contained datasets, a tiny tokenised text set and a tiny graph set, that require no download and exercise the sequence and message-passing paths.

12 Deployment

A run is reproducible from a container rather than from a local environment. A multi-stage Dockerfile builds the Rust binary against a pinned `libtorch` and bundles the datasets into the image; a slim runtime image then carries only the binary and `libtorch` and runs the search as its entrypoint, and a Kubernetes deployment and service run that image on a cluster.

13 Reporting and Export

Every run is recorded. A per-dataset report records, at checkpoints through the run, the best candidate's accuracy, loss, node and edge counts, the size of the quality–diversity archive, and the number of promoted macros, together with a rendered diagram of the best genome—as both a Mermaid graph and an ASCII topology—at the start, middle, and end of the run, so the growth of the architecture is legible. A JSON-Lines history logs one event per birth—its step, identifier, both parents, the operator that produced it (seed, mutation, crossover, macro, random, or controller), and its fitness, novelty, and score—so the lineage of any final model can be traced back through its ancestors to a seed, and the run's configuration is written alongside it. The best genome and its weights are saved, and the model is exported to TorchScript by tracing, and optionally to ONNX, so the discovered architecture leaves the search as a deployable artifact rather than as a description.

14 A Run

We ran the default configuration—population ten, forty steps, the staged budgets of Table 2, AdamW at learning rate 10^{-3}—from the minimal modality-appropriate seed (a small convolutional stem for image inputs) on four image-classification benchmarks of rising difficulty: the grayscale digits of MNIST, the cursive Kuzushiji characters of KMNIST, the clothing images of FashionMNIST, and the natural colour images of CIFAR-10 (Table 3). The same search, unchanged across datasets, grows and refines the seed, fills the quality–diversity archive, and promotes macros into the operator library on every run.

On MNIST the best candidate's validation accuracy reaches 0.995 and on KMNIST 0.943; on FashionMNIST it reaches 0.865. CIFAR-10, three colour channels at 32×32, is the hardest of the four, and it is where the modality-aware seed and the full training tier matter most: the best candidate reaches 0.662, filling thirteen archive cells as the search explores convolutional structure. Across the four runs the occupied archive ranged from four to thirteen cells and the best genomes carried between six and thirteen nodes, several of them holding memory cells joined by a recurrent edge. Every figure is read from the run's own metrics report.

Dataset	Input	Val. accuracy	QD cells	Macros	Graph (nodes/edges)
MNIST	$1 \times 28 \times 28$	0.995	7	5	7/6
KMNIST	$1 \times 28 \times 28$	0.943	8	5	13/8
FashionMNIST	$1 \times 28 \times 28$	0.865	4	4	6/5
CIFAR-10	$3 \times 32 \times 32$	0.662	13	5	7/6

Table 3: One run of the default configuration from the minimal modality-appropriate seed on each of four image-classification benchmarks. Accuracy is the best candidate's validation accuracy under the staged evaluation budget; QD cells is the number of occupied MAP-Elites cells at the end of the run; macros is the number promoted into the operator library; the last column is the node and edge count of the best genome. Every value is read from the run's own metrics report.

Part III
Conclusion

15 Conclusion

EDAG searches over learning mechanisms by making the mechanism explicit in the representation. A candidate is a typed directed graph whose feedforward edges compile to a single-step forward pass and whose state and recurrent edges carry memory and feedback; the operators that build candidates edit structure, state, and update dynamics as separate dimensions; the search preserves behavioral diversity in a quality–diversity archive and protects young candidates with staged evaluation and weight inheritance; and the operator library expands itself by promoting the substructures and update rules that recur among the elites. The system is implemented in Python and in Rust against a common, deterministic evaluation pipeline, runs over eight datasets spanning image, sequence, and graph data, and emits for every run a full metrics report, a complete birth-by-birth lineage, and an exported copy of the best model it found. From a minimal modality-appropriate seed a run of the default configuration reaches 0.995 validation accuracy on MNIST, 0.943 on KMNIST, 0.865 on FashionMNIST, and 0.662 on colour CIFAR-10, evolving genomes that carry memory and recurrence as the quality–diversity archive and the macro library grow; the same search, unchanged, grows and refines the seed on every one. The representation, the operators, and the self-expanding library are offered as a platform for discovering architectures that do not correspond to known templates.

References

[1] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin. Large-Scale Evolution of Image Classifiers. *ICML*, 2017.

[2] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized Evolution for Image Classifier Architecture Search. *AAAI*, 2019.

[3] B. Zoph and Q. V. Le. Neural Architecture Search with Reinforcement Learning. *ICLR*, 2017.

[4] K. O. Stanley and R. Miikkulainen. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[5] B. Gavranović, P. Lessard, A. Dudzik, T. von Glehn, J. G. M. Araújo, and P. Veličković. Categorical Deep Learning: An Algebraic Theory of All Architectures. *ICML*, 2024.

[6] J. Lehman and K. O. Stanley. Abandoning Objectives: Evolution through the Search for Novelty Alone. *Evolutionary Computation*, 19(2):189–223, 2011.

[7] J.-B. Mouret and J. Clune. Illuminating Search Spaces by Mapping Elites. arXiv:1504.04909, 2015.

[8] H. Yuan et al. An Evolutionary Neural Architecture Search Method Based on Performance Prediction and Weight Inheritance. *Information Sciences*, 2024.