Recurrent Connections in Gene Expression Programming Neural Networks

Yichen Yang
Khoury College of Computer Sciences
Northeastern University
Portland, ME, USA
yang.yichen1@northeastern.edu

Jonathan Mwaura
Khoury College of Computer Sciences
Northeastern University
Boston, MA, USA
j.mwaura@northeastern.edu

***Abstract*—Gene Expression Programming Neural Networks (GEPNN) evolve neural network topologies through a fixed-length linear chromosome that decodes into expression trees. This encoding guarantees syntactic validity under genetic operations, but inherently produces acyclic structures that prevent GEPNN from solving temporal problems that require memory. This paper introduces the index terminal, a novel terminal type that references other nodes within the expression tree by their position on the chromosome. When a node references an earlier position, a recurrent connection is formed; when it references a later position, a skip connection is created. This mechanism enables arbitrary cycles in the computational graph while preserving fixed-length encoding and deterministic decoding properties found in Gene Expression Programming (GEP). The approach is validated on temporal XOR benchmarks that require one- and two-step memory. The results obtained confirm that evolution discovers recurrent connections through index terminals that maintain state across timesteps, enabling GEPNN to solve tasks previously inaccessible to the framework. Delayed regression experiments confirm that the mechanism generalizes to continuous-output tasks. This shows that index terminals support temporal sequence processing in both the classification and regression domains. These results establish that the proposed index terminals successfully extend GEPNN to temporal problems while maintaining the algorithmic simplicity that distinguishes GEP from variable-length neuroevolutionary encodings.**

***Index Terms*—Gene Expression Programming, Neural Networks, Neuroevolution, Recurrent Neural Networks, Genetic Algorithms, Temporal Sequence Processing**

I. Introduction

The design of automated neural architecture remains a central challenge in machine learning [1], [2]. Instead of relying on human expertise to specify network topologies, evolutionary strategies can autonomously discover solution architectures to a given problem. Among these, Topology and Weight Evolving Artificial Neural Networks (TWEANN) methods jointly optimize the structure and weights in the network. By evolving both topology and weights simultaneously, TWEANN methods have achieved significant success across a wide range of domains, as these approaches can discover novel network structures that would be difficult to design by hand. NeuroEvolution of Augmenting Topologies (NEAT) [3] is a prominent example that uses direct encoding of network graphs with historical markers for crossover alignment. An alternative representation within the TWEANN family utilizes Gene Expression Programming [4], which encodes candidate solutions as fixed-length linear chromosomes that are decoded into expression trees. When applied to neural architecture search, this framework is known as Gene Expression Programming Neural Networks (GEPNN) [5], [6].

GEPNN offers a different set of advantages that are intrinsic to the genotype-phenotype separation found in Gene Expression Programming (GEP). In GEP, candidate solutions are encoded as fixed-length linear chromosomes that are decoded into expression trees of varying size and shape. This separation ensures that all genetic operators produce syntactically valid offspring, eliminating the need for repair mechanisms that plague direct encoding schemes [4]. The fixed-length chromosome also simplifies the implementation of crossover and mutation, and the deterministic decoding process means that identical genotypes always produce identical phenotypes. Despite these strengths, GEPNN faces a fundamental limitation. Because GEPNN decodes the chromosomes into expression trees, a solution cannot contain recurrent connections which require that the decoded structure be a graph to allow cycles in it [6]. This prevents GEPNN from solving any sequential and temporal problems due to the lack of memory in the structure.

To combat this limitation, this paper introduces the index terminal, a novel encoding extension that enables GEPNN to evolve recurrent and skip connections within existing GEP constraints. An index terminal node takes the value of another node in the decoded structure, specified by the index. Throughout this paper, *terminal* is used to collectively refer to both input terminals and index terminals, while *input terminal* refers specifically to traditional GEP terminals that inject external data. As a proof of concept, this paper validates the index terminal mechanism on temporal XOR variants of increasing difficulty, demonstrating that GEPNN can now evolve solutions that not only require memory, but also memory with no immediate reward. These experiments establish that the proposed extension successfully enables recurrent computation within the GEP framework. The paper also presents results on delayed regression to demonstrate applicability beyond classification tasks.

The remainder of this paper is organized as follows. Section II reviews recurrent networks, GEPNN, and related work

in neuroevolution. Section III presents the index terminal mechanism and the experimental methodology. Section IV presents and discusses the results obtained, while Section V discusses future directions. Section VI concludes.

II. Recurrent Networks and GEPNN

A. Recurrent Neural Networks

Recurrent neural networks (RNNs) extend feedforward architectures by introducing connections that form cycles, allowing information to persist over time-steps [7]. In a feedforward network, activation flows in one direction from input to output, and the network has no mechanism to retain information from previous inputs [8]. Recurrent connections address this limitation by feeding a node's output back into the network, either to itself or to earlier nodes, on subsequent time-steps. This feedback enables the network to maintain a hidden state that accumulates information over a sequence, making RNNs suitable for tasks where the correct output depends on the context inserted earlier in the input stream. The structural requirement for recurrence is the presence of cycles in the computational graph: there must be at least one path that leads from a node back to itself.

B. GEPNN

Gene Expression Programming (GEP) [4] is an evolutionary algorithm that evolves solutions by manipulating fixed-length strings that are subsequently expressed as tree structures of varying complexity. Unlike genetic programming, where operators act directly on expression trees, GEP maintains a strict separation between the genotype (the linear chromosome) and the phenotype (the expression tree) [6]. This separation ensures that all genetic operations would produce syntactically valid offspring, as every chromosome is guaranteed to produce a valid expression tree.

A GEP chromosome consists of a series of genes, each divided into a head and a tail. Genes are made up of symbols drawn from a function set and a terminal set. Functions have associated arities (number of arguments), while terminals represent input variables or constants. The head may contain both functions and terminals, whereas the tail contains only terminals. The length of the tail, t, is determined by the head length h and the maximum function arity $n_{\max}$ according to:

$$t = h(n_{\max} - 1) + 1 \tag{1}$$

This formula guarantees that regardless of head composition, sufficient terminals exist to complete any possible expression tree. Expression trees are constructed by reading the chromosome in breadth-first order: beginning with the first symbol as the root, each function node receives subsequent symbols as children according to its arity, continuing until all branches terminate in terminals.

GEPNN [5], [6] extends the GEP framework to neural architecture search by interpreting expression trees as neural network topologies. In GEPNN, function nodes represent neurons that apply activation functions to their inputs, input terminals correspond to network inputs, and the tree structure defines connectivity. To encode connection parameters, GEPNN augments the standard head-tail chromosome structure by adding two additional domains: D_w for edge weights and D_t for activation thresholds [5], [6]. During the forward pass, activation flows from the input terminal leaf nodes upward through function nodes toward the root, with each connection scaled by its corresponding weight and each neuron applying its threshold before activation. All domains are evolved jointly through standard genetic operators, allowing GEPNN to search simultaneously over topology and parameters. The approach has been applied to the classification [5] and robotic control [9] domains.

C. Recurrent in TWEANNs

TWEANN methods can naturally represent recurrent connections because they typically encode networks as graphs rather than trees. Early work by Angeline et al. [10] introduced GeNeralized Acquisition of Recurrent Links (GNARL), an evolutionary algorithm specifically designed to construct recurrent neural networks [10], [11]. GNARL represented networks directly as graph structures and used mutation operators to add nodes, delete connections, and modify weights, allowing arbitrary recurrent topologies to emerge from minimal initial structures.

NeuroEvolution of Augmenting Topologies (NEAT) [3] addressed the problem of competing conventions through historical markings that assign a global innovation number to each new structural mutation, allowing meaningful crossover between networks with different topologies. Like GNARL, NEAT uses a direct graph-based encoding that allows arbitrary connection patterns, including recurrent connections. NEAT also introduced speciation to protect novel topologies from premature elimination, allowing structural innovations to be optimized before competing against established solutions. Implementations such as NEAT-python provide continuous-time recurrent neural network (CTRNN) support, modeling neuron dynamics as differential equations for temporally-dependent tasks [12].

Other TWEANN approaches have also demonstrated the ability to evolve recurrent structures. For example, Evolutionary Acquisition of Neural Topologies (EANT) encodes networks as a linear genome where genes represent neurons, inputs, or jumper connections [13]. Jumper genes specify forward or recurrent connections, enabling cyclic topologies within the linear representation. The genome can be evaluated directly without decoding to a separate phenotype structure, and a two-level evolutionary process separates structural exploration from weight optimization through the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [14]. Cellular Encoding takes an indirect approach, using a grammar tree to specify developmental instructions that construct the network, allowing complex recurrent architectures to emerge through graph-rewriting operations [15].

Despite their flexibility, these TWEANN methods employ fundamentally different representations than GEP. Direct encodings such as NEAT use variable-length genomes where genetic operators must account for topological alignment, and the phenotype is constructed by iterating over connection lists. Indirect encodings like Cellular Encoding require the execution of a developmental program to produce the network, adding computational overhead, and reducing transparency between genotype and phenotype. In contrast, GEP maintains a fixed-length linear chromosome with deterministic breadth-first decoding that guarantees syntactically valid offspring under any genetic operation. The challenge addressed in this paper is to enable recurrence within the GEP framework while preserving these properties.

D. Recurrence in Other Representations

Cartesian Genetic Programming (CGP) [16] offers an alternative graph-based encoding where programs are represented as nodes on a grid, with genes specifying node functions and connection indices. Originally restricted to directed acyclic graphs, CGP was extended to support recurrent programs by eliminating the acyclicity constraint and controlling the probability of recurrent connection through a parameter [17]. When applied to neural network evolution, this "Recurrent CGP of Artificial Neural Networks" (RCGPANN) approach executes chromosomes by updating each node in index order, with recurrent connections reading values from the previous timestep; nodes initialize to zero until computed, providing a simple mechanism for handling cycles during evaluation.

Although both RCGPANN and the proposed approach use positional indices to create recurrence, the mechanism by which recurrence is introduced differs due to the underlying representations. In CGP, each node gene explicitly specifies its input connections as indices in the node array. Recurrence is therefore achieved by simply relaxing the existing acyclicity constraint on these indices, allowing a node to reference nodes that appear later in the array. GEP has no connection genes: connectivity is determined implicitly by symbol position during breadth-first decoding, and the tree structure fundamentally prevents cycles. Enabling recurrence, therefore, requires introducing a new symbol type rather than modifying an existing constraint. Despite this additional design challenge, the index terminal integrates into GEP without altering the chromosome structure, genetic operators, or decoding procedure, maintaining the properties that distinguish GEP from other neuroevolutionary encodings.

III. Methodology

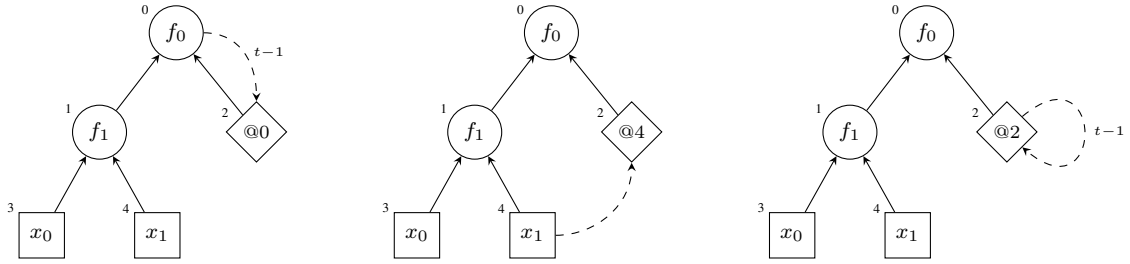
A. Index Terminal

This work introduces a novel terminal type for GEPNN chromosomes that enables recurrent and skip connections while preserving all existing GEP constraints. Conventionally, information within a GEP expression tree flows in only one direction, that is, from terminal leaf nodes toward the root. This unidirectional property is fundamental to GEP's design, ensuring that a linear chromosome deterministically decodes into a valid expression tree. The proposed work extends this framework by introducing the *index terminal*; a terminal that instead of sourcing data externally, retrieves its value from another node within the structure, specified by that node's position in the chromosome.

The index terminal respects all existing GEP constraints. Functionally, it operates identically to a standard terminal: it contributes to node counts, respects the arity requirements of its parent node, and follows the standard breadth-first unpacking order. However, an index terminal can only reference nodes within the head region, mirroring the hidden-to-hidden connectivity of traditional recurrent neural networks. Its behavior varies depending on the relationship between its own position and its target index (the node from which the index terminal draws data). The behavior is categorized into three cases:

1) *Target index smaller than self (Recurrent Connection).* When the target index is smaller than the index terminal's position (i.e., closer to the root node at index 0), the target node's current value has not yet been computed when the index terminal requires it. This configuration therefore represents a recurrent connection, and the index terminal retrieves the target node's value from the previous forward pass.
2) *Target index larger than self (Skip Connection).* When the target index is larger than the index terminal's position, the target node is farther from the root in the expression tree. During evaluation, leaf nodes and their ancestors are computed before nodes closer to the root, ensuring the target's value is already available when the index terminal is evaluated. This results in a skip connection within the current timestep.
3) *Target index equal to self (Self-Loop).* A self-referencing index terminal creates a non-functional recurrent connection. Since the node is initialized to have a value of zero and only ever reads its own previous value, it remains zero indefinitely. Such index terminals are effectively neutral and tend to be eliminated through evolutionary selection pressure.

To illustrate these cases, consider an expression tree with five nodes indexed 0 through 4. Nodes 0 and 1 are function nodes with arity 2, nodes 3 and 4 are input terminals, and node 2 is an index terminal. Following the standard GEP breadth-first unpacking, node 0 (root) takes nodes 1 and 2 as children, and node 1 takes nodes 3 and 4 as children. By varying the target of the index terminal at node 2, this work demonstrates the three connection types using Figure 1. In Figure 1(a), node 2 targets node 0; since the target index is smaller (i.e. at a position further upstream) than the index terminal's position, this creates a recurrent connection retrieving the root's value from the previous timestep. In Figure 1(b), node 2 targets node 4; since the target index is larger (i.e in a position further downstream from the index terminal), the value is already computed, resulting in a skip connection. In Figure 1(c), node 2 targets itself, creating a self-loop that perpetually outputs



(a) Recurrent (@0)
$[f_0, f_1, @0, x_0, x_1]$

(b) Skip (@4)
$[f_0, f_1, @4, x_0, x_1]$

(c) Self-loop (@2)
$[f_0, f_1, @2, x_0, x_1]$

Fig. 1: Index terminal connection types. Circles represent function nodes, rectangles represent input terminals, and rotated squares represent index terminals. Small numbers indicate node indices. (a) Recurrent connection: dashed arrow shows value retrieved from previous timestep. (b) Skip connection: dotted arrow shows value retrieved from current timestep. (c) Self-loop: node reads its own previous value, remaining zero. The active coding region of the corresponding chromosome for each variant is shown below.

zero.

To implement this mechanism while avoiding infinite loops, the approach modifies the standard GEP forward pass to use a two-pass evaluation scheme. The first pass populates values for all recurrent index terminals by retrieving their targets' values from the previous timestep. The second pass performs the standard GEP tree evaluation, computing leaf nodes first and propagating values toward the root. For skip connections, the target node's value is guaranteed to be available since nodes further from the root are evaluated before those closer to it.

B. GEP Setup

This work follows standard GEPNN conventions [4], [6], maintaining the canonical chromosome structure of the head, tail, D_w and D_t domains. The index terminal extends the standard symbol pools: the head draws from the union of functions, input terminals, and index terminals, while the tail draws from input terminals and index terminals. During initialization, symbols are selected with uniform probability from their respective pools, weights in D_w and thresholds in D_t are sampled from a standard normal distribution $\mathcal{N}(0, 1)$, and head lengths and mutation rates are varied across trials to assess the effectiveness of the index terminal mechanism.

The function set, summarized in Table I, consists of unary and binary operations selected for general-purpose applicability. Binary variants of activation functions first sum their input before applying the activation. The complement function computes $(1+b_i)-x$ when combined with the post-activation bias, allowing evolution to select the center point of negation; with $b_i = 0$ it acts as a binary NOT for inputs in $[0, 1]$.

The evolutionary algorithm employs fitness-proportionate roulette wheel selection, whose stochastic nature preserves structural diversity by occasionally selecting suboptimal individuals that may carry useful configurations. Crossover uses a synchronized one-point scheme, where structural components and their associated weights and biases are exchanged at the crossover point, preserving learned parameter associations. Mutation regenerates each allele with probability equal to the mutation rate using the same mechanism as initialization, with numerical coefficients perturbed by adding Gaussian noise $\mathcal{N}(0, 1)$. The population size is set to 250, the top 10 individuals (4%) are preserved as elites, and the crossover rate is fixed at 0.8 following established guidelines [18], [19]. Mutation rates and head lengths are varied in the experiments. In this formulation, weights are associated with edges and biases with function nodes. For a function node i with function f_i, children $c_1, c_2, \ldots, c_n$, the corresponding edge weights $w_1, w_2, \ldots, w_n$, and bias b_i, the output is computed as:

$$v_i = f_i(w_1 \cdot v_{c_1}, w_2 \cdot v_{c_2}, \ldots, w_n \cdot v_{c_n}) + b_i \qquad (2)$$

This applies biases after the function computation, which differs from Ferreira's original formulation, which applies thresholds before activation [5], [6]. Post-activation biases provide equivalent expressiveness for monotonic functions [20] while allowing evolution to shift output ranges arbitrarily, for instance, shifting sigmoid's range from $(0, 1)$ to $(b_i, 1 + b_i)$.

TABLE I: Function set used in all experiments.

Function	Arity	Definition
sigmoid	1	$1/(1+e^{-x})$
tanh	1	$\tanh(x)$
ReLU	1	$\max(0, x)$
complement	1	$1 - x$
addition	2	$x + y$
subtraction	2	$x - y$
multiplication	2	$x \cdot y$
sigmoid_2	2	$\text{sigmoid}(x + y)$
$\tanh_2$	2	$\tanh(x + y)$
ReLU_2	2	$\max(0, x + y)$

C. XOR Experiment

Experimentation begins with the XOR problem as a baseline to validate the GEPNN framework before introducing temporal dependencies. Two configurations were tested: one excluding index terminals from the symbol pool (standard feedforward), and one including index terminals to examine whether evolution learns to ignore unnecessary recurrent structure. For

TABLE II: Experimental parameters across all tasks. Population size (250), elite count (10), and crossover rate (0.8) are constant across all experiments.

Experiment	Head Lengths	Mutation Rates	Gen. Limit	Runs
XOR	3–6	0.15–0.35	1000	100
$T_{0,-1}$	2–7	0.15–0.40	1000	100
$T_{-1,-2}$	2–7	0.15–0.40	2500	100
$T_{0,-2}$	2–7	0.15–0.40	4000	100
Delay=1	2–4	0.20–0.50	1000	100
Delay=2	3–6	0.20–0.50	2500	100

the configuration with index terminals, each individual was evaluated on all 24 permutations of the four XOR cases, processed as parallel sequences, preventing networks from exploiting positional patterns in the evaluation order.

To evaluate solution quality, this work defines a fitness function that combines mean squared error and classification accuracy in equal proportion:

$$f = 0.5 \cdot \max(0, 1 - 4 \cdot \text{MSE}) + 0.5 \cdot \text{accuracy} \tag{3}$$

where the MSE component is normalized by a factor of 4, corresponding to the maximum possible MSE for binary targets. This combination provides both smooth gradients for optimization through the MSE component and a direct classification signal through accuracy. Pure accuracy-based fitness creates a coarse landscape in which many structurally different solutions achieve identical scores. Pure MSE does not distinguish between errors on opposite sides of the classification boundary, allowing networks with different classification behavior to receive similar fitness. The equal weighting ensures that neither component dominates the selection pressure. A solution is considered successful when the fitness exceeds 0.9999, indicating both perfect classification and a near-zero prediction error.

The head length 2 is excluded because it cannot express a valid XOR solution with the given function set; all other experimental parameters are summarized in Table II. With head length 2, only one function can directly combine both inputs into an intermediate representation. The root function then receives this single intermediate value along with one raw input, which is insufficient to compute XOR since the problem requires comparing two independent intermediate computations.

This work also tested sets of more expressive functions, such as including the arithmetic OR function $f(a, b) = a + b - ab$. However, this proved too powerful –optimal solutions with head length 2 were trivially discovered. Such results, while validating the framework, do not demonstrate meaningful structure evolution and are discussed further in the future work section.

D. Temporal XOR Experiment

Having validated the GEPNN framework on static XOR, the next step introduces temporal dependencies to evaluate the index terminal's ability to enable recurrent computation. The temporal XOR (TXOR) task extends the temporal XOR problem introduced by Elman [7] as a reference for recurrent networks. Elman's formulation, requiring the XOR of current and previous inputs, serves as the simplest variant in this work. To evaluate whether the index terminal mechanism scales beyond minimal memory requirements, this work introduces additional variants with increasing difficulty. The task is generalized as follows.

$$T_{i,j} = x(t + i) \oplus x(t + j) \tag{4}$$

where $x(t)$ denotes the binary input in timestep t and $\oplus$ denotes the XOR operation. Unlike static XOR where all inputs are available simultaneously, temporal XOR requires the network to maintain memory of past inputs, making recurrent connections essential for solving the task.

This work evaluates three variants of increasing difficulty. The first variant, $T_{0,-1}$, requires the XOR of the current input and the input from one time-step ago, representing the simplest temporal dependency that can be solved with a single recurrent connection. The second variant, $T_{-1,-2}$, requires the XOR of inputs made one and two time-steps ago, demanding the memory of two consecutive past values. The third variant, $T_{0,-2}$, requires the XOR of the current input and the input from two time-steps ago; this is the most challenging because the network does not receive a fitness signal for storing the intermediate value at time-step $t - 1$. The value made at $t - 2$ must be preserved through a time-step where it does not provide immediate benefit, and evolution can only discover its utility once the complete two-step memory chain is in place.

The fitness function and the success criterion are identical to the XOR experiment. Generation limits were scaled to task difficulty through preliminary experiments, as summarized in Table II.

E. Delayed Regression Experiment

To evaluate the index terminal mechanism on continuous-output tasks, a delayed regression problem is introduced where the network must output the exact input value from a specified number of previous time-steps. Delayed regression tasks are well-established benchmarks for evaluating temporal memory in neural networks [21], [22], as they test both the ability to construct appropriate memory structures and the capacity to adjust parameters to preserve signal fidelity over time. At each time-step t, the network receives an input $x(t)$ sampled uniformly from $[-10, 10]$ and must produce an output $y(t) = x(t - d)$, where d is the delay. Two variants were evaluated: $d = 1$, which required memory of the immediately preceding input, and $d = 2$, which required memory over an intermediate time-step. Unlike temporal XOR where correct structure yields perfect classification, delayed regression requires precise weight tuning to preserve continuous values across timesteps.

Each individual was evaluated on 500 sequences of length 15, the remaining experimental parameters are summarized in Table II. Fitness was calculated as $f = 1/(1 + \text{MSE})$, where MSE is the mean squared error between the predicted and target values for all valid time-steps. A solution is considered

successful when the fitness exceeds 0.999999, indicating near-perfect reproduction of the delayed values.

IV. Results

A. XOR

The static XOR experiments validate the GEPNN framework on a well-understood benchmark. Both configurations, with and without index terminals, achieved 100% success rates across all tested head lengths (3–6) and mutation rates (0.15–0.40), with 100 independent runs per configuration. The median convergence ranged from 16 to 23 generations without index terminals and 19 to 24 with them, a modest increase reflecting the expanded search space. The head length 3 proved sufficient for both configurations, confirming the minimum structural requirement identified in the experimental design. Evolution consistently discovered feedforward solutions despite the availability of index terminals, demonstrating that the algorithm does not spuriously incorporate recurrent structure when it offers no fitness advantage. The uniformly high success rates indicate that XOR does not discriminate between configurations, motivating subsequent temporal experiments where structural requirements become more demanding.

B. TXOR

Figure 2 presents success rates across all three temporal variants of the XOR. The results confirm that the index terminal successfully enables recurrent computation within the GEPNN framework, with performance patterns that align with the structural demands of each task.

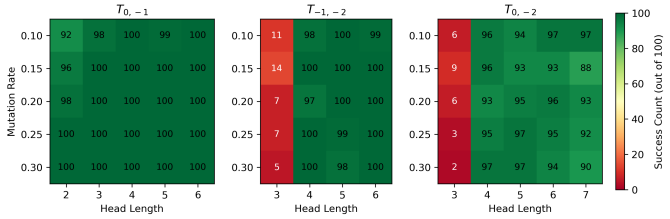


Fig. 2: Success rates (out of 100 runs) for Temporal XOR variants across head lengths and mutation rates.

The simplest variant, $T_{0,-1}$, achieves near-perfect success rates across all tested configurations, with head length 3 and above reaching 100% success at mutation rates of 0.25 and higher. This task requires only a single recurrent connection to store the previous input, representing the minimal memory requirement for temporal XOR.

The variant $T_{-1,-2}$ has a clear structural threshold at head length 4, where success rates jump from below 15% to above 97%. Solutions exist in the head length 3 space, as evidenced by non-zero success rates, but this configuration provides only the minimum structure necessary: one node to store the $t-1$ value and another to store the $t-2$ value, leaving little room for the computational machinery required to combine them. Head length 4 provides sufficient structural flexibility for evolution to reliably discover solutions.

The most challenging variant, $T_{0,-2}$, displays the expected difficulty pattern. This task requires the network to store the input from $t-2$ through an intermediate timestep where it does not provide a fitness signal, which means that evolution must discover the utility of $t-1$ storage only after the complete two-step memory chain is in place. Success rates at head length 3 remain below 10% across all mutation rates, while head length 4 and above achieve success rates exceeding 90%. The performance gap between $T_{-1,-2}$ and $T_{0,-2}$ at equivalent head lengths reflects the additional evolutionary difficulty of discovering memory structures without immediate reward. Based on these results, mutation rate 0.25 is used for subsequent analysis as it achieves the highest aggregate success across tasks and configurations.

Table III presents solution time statistics for successful runs on the two harder variants with a mutation rate of 0.25. The distributions are right-skewed, with most runs finding solutions within a few hundred generations, while a subset requires significantly longer. This pattern is consistent with evolutionary search, where the discovery of the correct recurrent structure depends on specific mutations occurring; once the structure emerges, weight optimization proceeds quickly.

TABLE III: Generations to solution for $T_{-1,-2}$ and $T_{0,-2}$ at mutation rate 0.25, computed over successful runs only.

Task	Head	Median	Mean	Max
$T_{-1,-2}$	4	138	252	2213
$T_{-1,-2}$	5	173	336	1983
$T_{0,-2}$	4	362	666	3795
$T_{0,-2}$	5	456	807	3220

Figure 3 shows the median generations to solution at a mutation rate of 0.25. The variant $T_{0,-1}$ solves rapidly across all head lengths with minimal variation as the structure increases, reflecting its low structural requirements. The variant $T_{-1,-2}$ exhibits a sharp drop in head lengths 3 to 4, after which the solution time stabilizes, indicating that head length 4 provides sufficient capacity for the two-step memory structure, and additional nodes offer diminishing returns. The variant $T_{0,-2}$ follows a similar pattern, but with notably higher generation counts, consistent with the delayed reward signal requiring evolution to maintain memory over a non-informative timestep. In particular, the minimum head length that achieves high success rates also yields the fastest median solution time for both $T_{-1,-2}$ and $T_{0,-2}$, suggesting that the excess structural capacity expands the search space without accelerating discovery.

To illustrate structural constraints at head length 3, Figure 4 presents a rare successful solution to $T_{0,-2}$ found after 861 generations. The network contains two recurrent connections: node 4 (@2) retrieves the previous output of node 2, and node 5 (@1) retrieves the previous output of node 1, forming the two-step memory chain required to store the input from $t-2$. Every node in the structure serves a necessary role, leaving no room for redundancy. This minimal configuration explains why head length 3 achieves success rates below 10%: evolution

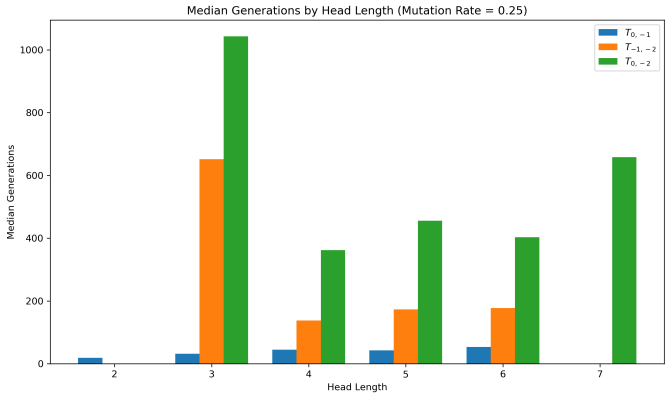


Fig. 3: Median generations to convergence by head length at mutation rate 0.25.

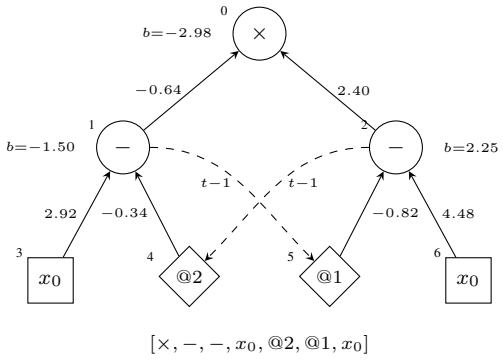


Fig. 4: A rare successful solution to $T_{0,-2}$ at head length 3. Dashed arrows indicate recurrent connections retrieving values from the previous timestep. Weights and biases are rounded to two decimal places, the sequence below is the acting coding region of the chromosome.

must simultaneously discover both the correct structure and precise weights within a tightly constrained search space.

C. Delayed Regression

The delayed regression task evaluates whether the index terminal mechanism generalizes from classification to continuous-output problems. Unlike temporal XOR where outputs are binary and fitness combines classification accuracy with MSE, delayed regression requires the network to reproduce an input value exactly after a specified delay, demanding precise weight tuning in addition to correct structure.

Table IV presents the delayed regression results.

The delay=1 variant achieves near-perfect success in all configurations, with head length 3 and above finding solutions in single-digit median generations as additional nodes provide more pathways for approximating the identity function. The delay=2 variant exhibits the same structural threshold observed in temporal XOR: head length 3 fails entirely, head length 4 achieves 90 to 95% success depending on the mutation rate, and head length 5 and above reaches near-perfect success. Within this range of mutation rates, performance remains

TABLE IV: Delayed regression results for successful runs. Delay=1 at head length 2 omitted as it represents a degenerate case requiring no index terminal.

Delay	Head	Mutation	Success	Median	Mean
1	3	0.25	100	12	74
1	3	0.30	100	13	60
1	3	0.35	100	8	59
1	4	0.25	99	5	28
1	4	0.30	96	4	31
1	4	0.35	100	4	18
2	4	0.25	90	255	418
2	4	0.30	93	242	415
2	4	0.35	95	236	427
2	5	0.25	96	45	173
2	5	0.30	100	59	136
2	5	0.35	99	44	81
2	6	0.25	100	21	37
2	6	0.30	100	16	24
2	6	0.35	100	24	35

stable, with modest improvements in success rate at higher mutation rates for head length 4.

The speed of solution for delay=2 reveals an interaction between structural capacity and parameter optimization. At head length 4, median generations exceed 200 even at the highest mutation rates tested, while head lengths 5 and 6 find solutions in under 60 generations despite solving the same task. The larger chromosome provides more degrees of freedom for weight optimization, allowing evolution to more easily tune continuous parameters that preserve signal fidelity over time steps. This contrasts with temporal XOR, where the minimum sufficient head length was also the fastest, reflecting the difference between tasks that primarily require structural discovery versus those that additionally require precise parameter tuning.

V. Future Work

The experiments presented in this work validate the index terminal mechanism on controlled benchmarks designed to isolate temporal memory requirements. A natural next step is to evaluate the approach on more challenging sequential tasks, including real-world time-series prediction and sequence classification problems, to assess both scalability and generalization to unseen data. Comparative evaluation against established recurrent TWEANN methods such as NEAT and RCGPANN on identical benchmarks would further contextualize the mechanism's performance, while topological analysis of evolved solutions would provide insight into the structural patterns that emerge through the index terminal. Computational cost analysis, particularly the overhead introduced by the two-pass evaluation scheme, is also warranted as the complexity of the problem increases.

The choice of primitive functions significantly impacts the structural complexity required to solve a given problem. During experimentation, this work tested an arithmetic OR function defined as $f(a, b) = a + b - ab$, which solved XOR with a head length of just two, one fewer than the minimum required by the standard function set. This demonstrates that composite functions encoding useful operations can reduce

the search space by eliminating the need for evolution to discover equivalent multi-node structures. Future work should systematically investigate which composite functions yield the greatest efficiency gains across different problem classes, as the optimal level of function expressiveness likely varies by domain.

Similarly, the current function set was designed for general-purpose applicability, but task-specific function sets may offer substantial performance improvements. Binary classification, regression, and temporal tasks each impose different structural demands, for instance, temporal tasks may benefit from functions with natural state-preserving or delay properties, while regression tasks may require functions better suited to continuous output ranges. The design of domain-specific function sets remains largely unexplored in the GEPNN context and represents a promising direction to improve convergence speed and solution quality in specialized applications.

VI. Conclusion

This work introduced the index terminal, a mechanism that enables recurrent and skip connections in Gene Expression Programming Neural Networks by allowing nodes to reference other nodes by their position in the chromosome. This extension transforms GEP's expression trees into arbitrary directed graphs without requiring modifications to the genetic operators or chromosome structure.

The experimental results demonstrate that GEPNN with index terminals successfully evolves solutions to temporal problems requiring memory. On temporal XOR variants, structural capacity proved to be the primary bottleneck, with tasks requiring multi-step memory chains exhibiting sharp performance transitions at critical head lengths. The variant $T_{0,-2}$ proved to be the most challenging, as evolution must discover a complete two-step memory structure before receiving any fitness signal for the intermediate storage node. The delayed regression experiments confirmed that the mechanism is generalized beyond classification to continuous-output tasks. As a proof of concept, these results establish that index terminals provide GEPNN with the representational power necessary for temporal sequence processing while maintaining the algorithmic simplicity that distinguishes GEP from other neuroevolutionary approaches. Future work will evaluate the mechanism on more challenging benchmarks and against established recurrent neuroevolutionary methods.

References

[1] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *Journal of Machine Learning Research*, vol. 20, no. 55, pp. 1–21, 2019.

[2] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, B. B. Gupta, X. Chen, and X. Wang, "A comprehensive survey of neural architecture search: Challenges and solutions," *ACM Computing Surveys*, vol. 54, no. 4, pp. 1–34, 2021.

[3] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.

[4] C. Ferreira, "Gene expression programming: A new adaptive algorithm for solving problems," *Complex Systems*, vol. 13, no. 2, pp. 87–129, 2001.

[5] ——, *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence*, 2nd ed. Berlin, Germany: Springer, 2006.

[6] J. Mwaura and R. Heminway, "Connectivity schemas in neuroevolution: What neural architectures does GEPNN evolve?" in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*. IEEE, December 2023.

[7] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990, original source of temporal XOR benchmark for RNNs.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[9] J. Mwaura and E. Keedwell, "Evolving robotic neuro-controllers using gene expression programming," in *2015 IEEE Symposium Series on Computational Intelligence*, 2015, pp. 1063–1072.

[10] P. J. Angeline, G. M. Saunders, and J. B. Pollack, "An evolutionary algorithm that constructs recurrent neural networks," *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pp. 54–65, 1994.

[11] G. M. Saunders, P. J. Angeline, and J. B. Pollack, "Structural and behavioral evolution of recurrent networks," in *Advances in Neural Information Processing Systems 6 (NIPS 1993)*. Morgan Kaufmann, 1993, pp. 88–95.

[12] A. McIntyre, M. Kallada, C. G. Miguel, C. Feher da Silva, and M. L. Netto, "NEAT-Python," https://github.com/CodeReclaimers/neat-python, 2019.

[13] Y. Kassahun and G. Sommer, "Efficient reinforcement learning through evolutionary acquisition of neural topologies," in *Proceedings of the 13th European Symposium on Artificial Neural Networks (ESANN)*, 2005, pp. 259–266.

[14] N. T. Siebel and G. Sommer, "Evolutionary reinforcement learning of artificial neural networks," *International Journal of Hybrid Intelligent Systems*, vol. 4, no. 3, pp. 171–183, 2007.

[15] F. Gruau, "Neural network synthesis using cellular encoding and the genetic algorithm," Ph.D. dissertation, Ecole Normale Supérieure de Lyon, 1994.

[16] J. F. Miller and P. Thomson, "Cartesian genetic programming," pp. 121–132, 2000.

[17] A. J. Turner and J. F. Miller, "Recurrent cartesian genetic programming of artificial neural networks," *Genetic Programming and Evolvable Machines*, vol. 17, no. 3, pp. 205–251, 2016.

[18] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[19] K. A. De Jong, "An analysis of the behavior of a class of genetic adaptive systems," Ph.D. dissertation, University of Michigan, USA, 1975.

[20] X. Yao, "Evolving artificial neural networks," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1423–1447, 1999.

[21] H. Jaeger, "Short term memory in echo state networks," *GMD Report*, vol. 152, 2002.

[22] R. J. Hyndman and G. Athanasopoulos, *Forecasting: principles and practice*, 2nd ed. OTexts, 2018. [Online]. Available: https://otexts.com/fpp2/