Adaptive Neuroevolution with Genetic Operator Control and Two-way Complexity Variation

Amir Behjat, *Member, IEEE*, Nathan Maurer, *Student Member, IEEE*, Sharat Chidambaran, *Student Member, IEEE*, and Souma Chowdhury, *Senior Member, IEEE*

***Abstract*—Topology and weight evolving artificial neural network (TWEANN) algorithms optimize the structure and weights of artificial neural networks (ANNs) simultaneously. The resulting networks are typically used as policy models for solving control and reinforcement learning (RL) type problems. This paper presents a neuroevolution algorithm that aims to address the typical stagnation and sluggish convergence issues present in other neuroevolution algorithms. These issues are often caused by inadequacies in population diversity preservation, exploration/exploitation balance, and search flexibility. This new algorithm, called the Adaptive Genomic Evolution of Neural-Network Topologies (AGENT), builds on the neuroevolution of augmenting topologies (NEAT) concept. Novel mechanisms for adapting the selection and mutation operations are proposed to favorably control population diversity and exploration/exploitation balance. The former is founded on a fundamentally new way of quantifying diversity by taking a graph-theoretic perspective of the population of genomes and inter-genomic differences. Further advancements to the NEAT paradigm occur through the incorporation of variable neuronal properties and new mutation operations that uniquely allow both the growth and pruning of ANN topologies during evolution. Numerical experiments with benchmark control problems adopted from the OpenAI Gym illustrate the competitive performance of AGENT against standard RL methods and adaptive HyperNEAT, and superiority over the original NEAT algorithm. Further parametric analysis provides key insights into the impact of the new features in AGENT. This is followed by evaluation on an unmanned aerial vehicle collision avoidance problem where maneuver planning models are learnt by AGENT with 33% reward improvement over 15 generations.**

***Impact Statement*—This paper presents AGENT, a new neuroevolution algorithm that simultaneously optimizes the topology and weights of neural networks. Through novel mutation and selection controllers that regulate diversity and convergence, along with flexible direction of mutation and flexible choice of nodes, AGENT advances the application and robustness of the neuroevolution paradigm. The potential impact of these advancements is apparent when comparing the performance of AGENT on reinforcement learning (RL) problems to the performance of state-of-the-art RL algorithms on these same problems. Various OpenAI Gym problems and a real-world collision avoidance problem for unmanned aerial vehicles (UAVs) are used to illustrate AGENT's performance relative to these other algorithms. AGENT is particularly deft at solving problems with small-to-moderately sized action/state spaces and complex environments with sparse, delayed and non-differentiable reward functions. Problems with these features are quite common in the engineering controls and robotics planning domains. In addition, the proposed mutation and selection control operators, and the novel graph-theoretic approach to quantifying diversity, could be adopted in other evolutionary machine learning algorithms to improve diversity preservation and address stalled convergence issues.**

***Index Terms*—Diversity preservation, Neuroevolution, Network topology, Reinforcement Learning, Open AI Gym, UAV**

I. Introduction

ARTIFICIAL neural networks (ANNs) are gaining a vital role as models for decision-making for a variety of intelligent autonomous systems [1]. ANNs are universal function approximators [2], which allows ANNs to act as policy models that provide state-to-action mappings for RL algorithms. Applications that require state-to-action mappings typically do not have a priori (i.e., labeled data) of the optimum state-to-action mappings. Reinforcment Learning (RL) methods [3] and its deep variants [4] constitute a dominant contemporary player in automated design of state-to-action models for such planning and control problems. However, drawbacks of using RL methods include relying on gradient information for backpropagation, which is not readily available for some problems, [5], requirement of a tedious (nested) hyper-parameter optimization or neural architecture search when ANNs are used as value or policy function approximators, and scalability limitations with the dimension of the problem [6]. As an alternative, Evolutionary algorithms [7], e.g., neuroevolution [8] and evolution strategies [9], have emerged to mitigate some of these limitations.

Neuroevolution allows for highly parallelized and distributed implementations of the learning process, and is suitable for problems with continuous and mixed state/action spaces. Key issues that neuroevolution methods suffer from are topological inflexibility, premature stagnation and poor convergence. To address these issues, this paper presents key advancements to the paradigm in co-evolving the topology and weights of neural networks, particularly for solving RL type problems related to planning and control. These advancements include the development of novel controllers within selection and reproduction operators to regulate population diversity and improvement rates, and the introduction of new neuronal properties and flexible initialization and mutation of neural topologies. The

Support from the DARPA award HR00111920030 and the NSF award CMMI-2048020 is gratefully acknowledged. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the DARPA or the NSF.

This paper was submitted on 03-21-2022

Amir Behjat was PhD student from University at Buffalo. He is a postdoctoral research in Purdue University, West Lafayette, IN 47906 USA (e-mail: abehjat@purdue.edu).

Nathan Maurer is a Master student from University at Buffalo, Buffalo, NY 14260 USA (e-mail: namaurer@buffalo.edu).

Sharat Chidambaran was Master student from University at Buffalo, Buffalo, NY 14260 USA (e-mail: sharatpa@buffalo.edu).

Corresponding author: Souma Chowdhury is Associate Professor in University at Buffalo, Buffalo, NY 14260 USA (e-mail: soumacho@buffalo.edu).

new neuroevolution algorithm is evaluated over benchmark RL and control problems adopted from the OpenAI Gym [10], and a practical robotics problem – namely collision avoidance between unmanned aerial vehicles (UAVs).

A. Neuroevolution: Overview

Neuroevolution is the process of designing ANNs through evolutionary optimization algorithms [11]. Some Neuroevolution approaches optimize only the ANN weights using Genetic Algorithms (GA) [12], [13]. The topology of the ANN must be prescribed by the user in these approaches, as is common in backpropagation-based training. With a predefined topology, there remains a risk of underfitting or overfitting of the state-to-action mapping model [14]. ***Topology and Weight Evolving ANNs*** (TWEANNs) directly addresses this issue. Some of the earliest work in TWEANNs include that by Miller et al. [15], where the adjacency matrix was evolved to alter the connection of nodes in the ANN, and by Schaffer et al. [16] where a GA was used to optimize hyper-parameters along with a hybrid architecture with specified restrictions.

B. Neuroevolution: Related Works

NeuroEvolution of Augmenting Topologies (NEAT) [17] is one of the most well-known implementations of the TWEANN concept. NEAT treats nodes, edges, and edge weights as phenotypes and these phenotypes are evolved using a GA. The initial population in NEAT includes minimalist topologies of feed-forward ANNs without any hidden nodes and with input and output layers designed according to the concerned problem. During the evolution process, NEAT performs classical genetic operations such as selection, crossover, and mutation on the population. Furthermore, a specialized operator called "speciation" is used in NEAT which helps to preserve genomes that are associated with complex newly created topologies, which otherwise may be destroyed due to their premature weights. Well known variations of NEAT, such as HyperNEAT [18], which provides an indirect genotype to phenotype encoding, and SUNA [19], have been used for controlling virtual agents in Atari games [20], evolving robot gaits [21], and geological prediction [22]. Evolving deep neural networks has also been explored with Neuroevolution [23]. Overall, there's been a rich variety of developments to neuroevolution in the past 10-15 years, particularly involving indirect encoding of topologies (e.g., HyperNeat [24], adaptive HyperNEAT [25], and ES-HyperNeat [26]), extension to deep learning and metalearning [23], [27], hybridization with gradient-based methods [28], open-ended evolution and embodiment of intelligence [29], [30]. A comprehensive survey of such developments can be found in the review paper by Stanley et al. [11]. Key issues with existing neuroevolution methods are discussed next.

***TWEANNs: Stagnation & Efficiency Concerns*:** Premature stagnation and sluggish rate of evolution are two crucial concerns in neuroevolution, especially when a highly non-linear state to action mapping is sought. There has been some notable works in investigating the issue of premature stagnation from different perspectives, such as in [31], [32], [33]. Two underlying phenomena have been found to be the major factors behind premature stagnation: i) *loss of diversity* and ii) *poor exploration/exploitation balance*.

Since nascent complex ANN topologies need more time to stabilize their weights compared to simpler counterparts, Neuroevolution methods are often unable to preserve genomic diversity which leads to local stagnation. Different approaches exist to solve the diversity preservation issue, such as Novelty search [34] and Surprise Search [35]. Some of these approaches create artificial multi-objective problems that trade off novelty with reward; however, the likely need for larger population sizes for such an approach and its impact on the generalizability [36] of the ensuing learnt behavior remains under-explored. More importantly, most of these existing methods provide limited capacity for *adaptive* diversity preservation during the learning process. There exists a rich body of literature on adaptive diversity preservation in the general domain of evolutionary computing (e.g., in GA [37] and particle swarm [38] algorithms). However, it is challenging to directly translate these approaches to neuroevolution due to the specialized encoding and reproduction operations used to respectively represent and evolve TWEANNs.

Thus we hypothesize that a quantification of diversity, one that accounts for the special combinatorial design space of TWEANN genomes, is needed to implement adaptive diversity preservation in neuroevolution. To investigate this hypothesis, we pursue a novel projection of candidate ANNs onto an undirected graph, based on their inter-genome distances [19], and then use the minimum spanning tree (MST) estimate to quantify the population diversity. This graph-based diversity quantification concept is motivated by how in general diversity between data sets, often occurring or posed as nodes in a network, is measured in various applications of network science [39]; our use of minimum spanning tree is specifically inspired by the literature on Genetic Linkage Mapping [40].

Ineffective balance between exploration and exploitation in neuroevolution, which leads to sluggish convergence or vulnerability to local optima, can be partly attributed to the inadequate recombination (e.g., due to permutation issues [41]) or unregulated mutation operations. *We posit that the current state of "exploration / exploitation balance" can be captured by analyzing the relative rates of improvement in the fitnesses of the population best and the population average during the neuroevolution process, with coherent rates being deemed desirable.* Based on this hypothesis, we propose an improvement metric that is then used to guide the rates of mutation operations, which regulates the degree of exploration allowed in the evolution process.

Initiating the population with minimalist ANN topologies as suggested in NEAT [42], [17] helps decrease the probability of overfitting and speeds up the convergence for problems with low-dimensional input spaces. However, a detrimental effect is encountered in problems with larger state spaces or when a highly non-linear policy function is required. Therefore, minimalist initial population can often lead to colossal computational effort, aka many generations of evolution, to reach topologies with adequate complexity for these problems. To address this issue, we have introduced two new capabilities: *problem size-adaptive initialization of the ANN topologies*, and *allowing topological complexity to both increase and decrease* during neuroevolution. Furthermore, by applying different activation function choices [19] and a novel memory property for

neurons, we seek to extend the applicability of neuroevolution to learn policies for a wider range of problems in autonomous systems without deviating from the feed-forward network structure. For example, the memory properties promote capture of derivatives of state inputs or intermediate states.

C. Core Objectives of this Paper

This paper develops an *adaptive neuroevolution* approach to handle the premature stagnation issues and enhance the search efficiency in designing TWEANN-based policy models. Our primary contribution lies in incorporating novel mechanisms for insitu regulation of the genomic diversity and the rate of average fitness improvement. To these ends, new approaches are proposed respectively **i)** for measuring diversity in the combinatorial space of "*ANN topologies with variable node properties*", and **ii)** for comparing "*average vs. best fitness improvement rates*" to assess the balance between exploration and exploitation. The regulation of diversity and average improvement rate is respectively accomplished by dynamic adaptation of the selection and mutation operators, essentially acting as open-loop controllers in the neuroevolution process. Thus, the ***primary objective*** of this paper is to enable these new adaptation capabilities, along with novel node properties, problem-adaptive population initialization, and provision for both growth and pruning of networks in neuroevolution.

Our ***second objective*** is to test our neuroevolution algorithm on a suite of control problems adopted from the popular OpenAI Gym platform [10]. With these problems, we perform comparative analysis of our algorithm with existing RL benchmarks, state-of-the-art implementations of the original NEAT as well as a recent popular advancement of NEAT, namely adaptive HyperNEAT [25]. Selected problems are also used to perform parametric and ablation analysis to study the impact of the proposed new features. Our ***third objective*** is to demonstrate how the new neuroevolution algorithm can be used to solve complex robotics planning problems, by applying it to generate maneuver-planning models for reciprocal collision avoidance between quadcopter UAVs [43].

The next section describes the basic components of our new neuroevolution algorithm. Section III develops the controller features used for regulation of average improvement rate and population diversity. Sections IV and V respectively discuss the results of applying this new neuroevolution algorithm on the benchmark problems and the UAV collision-avoidance problem. Further discussion and concluding remarks are provided in Sections VI and VII.

II. AGENT (NEUROEVOLUTION) ALGORITHM

In light of its adaptation capability, the new neuroevolution algorithm is called ***Adaptive Genomic Evolution of Neural Network Topologies*** or **AGENT**. In this section, we describe the main components of AGENT, including the initialization steps, intra-generational stages, the encoding procedure, and the selection and reproduction operators. Figure 1 (a) shows the flowchart of the AGENT algorithm. [1] A pseudo code summarizing the main steps of AGENT is included in the supplementary material, as Algorithm-2 in Section S - VIII.

[1]To aid benchmarking and further adoption of AGENT, a MATLAB-based implementation of AGENT have been made available at the following repository: https://github.com/adamslab-ub/AGENT-Matlab.

As seen in Fig. 1, initialization of the algorithm involves generating the design of experiments of problem scenarios (similar to RL) over which the learning trials are conducted. This is followed by the genomic encoding of ANNs and the creation of the initial population.

At each generation, AGENT uses a ***two-stage*** *evolutionary approach*. The population of genomes (i.e., candidate ANNs) is grouped into a set of species or niches, based on topological similarities. All genomes participate in the first stage of evolution in each generation, while only the genomes with the highest fitness function from each species participate in the second stage of evolution. *Stage-1 processes*: In the first stage, the size of each niche is adjusted, followed by selection, crossover, and mutation. The resulting offspring genomes go through fitness evaluation (over the episodes of sample scenarios), followed by speciation. *Stage-2 processes*: In the second stage, similar to NEAT [17], the champions of each species are identified, which then undergo crossover and mutation. The resulting offspring are subsequently classified into the existing species groups. The new stagnation-mitigating controllers are then applied – this involves quantifying the average improvement rate and population diversity and using them as feedback to adapt the selection pressure and mutation rate, respectively. Fitness evaluation is then performed on the new genomes (resulting from second stage evaluation), and the algorithm moves on to the next generation.

A. Encoding of the Neural Network Genome

Information processing capacity in an ANN is encapsulated in its nodes (or neurons) and edges (or connections). Similar to the original NEAT algorithm [17], AGENT uses a direct bi-structural encoding, where each genome comprises of the encoding of the constituent nodes and edges of the candidate ANN. The genomic encoding is shown in the upper part of Fig. 1. In keeping with NEAT, AGENT encodes edges with two genomic properties, edge weight $w_{i,j}$ and innovation number $I_{i,j}$. The start and end nodes of an edge are denoted by i and j, respectively. Innovation number is used to distinguish between different edges based on their creation time, and sort edges thereof. This number is a useful identifier of common edges during the crossover operation.

AGENT differs from NEAT with regards to the encoding of nodes. AGENT introduces new nodal properties, namely the memory capacity M_i, and the activation function ϕ_i. In Fig. 1, the different symbols (hexagon for modified sigmoid [17], triangle for ramp, and pentagon for sigmoid) depict nodes with different activation functions. The memory size of each node is depicted by the color of the corresponding node symbol.

To allow greater flexibility of nonlinear transformations encapsulated by the network [44], three different activation functions are allowed to be selected, namely modified sigmoid (which was used in the original NEAT [17]), saturated linear or ramp, and sigmoid functions. Their mathematical expressions are given below:

$$y_i = \phi(v_i) = \begin{cases} \frac{1}{1+e^{-4.9v_i}}, & \text{if } \texttt{Mod. Sigmoidal} \\ \max(\min(v_i, 1), 0), & \text{if } \texttt{Ramp} \\ \frac{1}{1+e^{-v_i}}, & \text{if } \texttt{Sigmoidal} \end{cases} \quad (1)$$

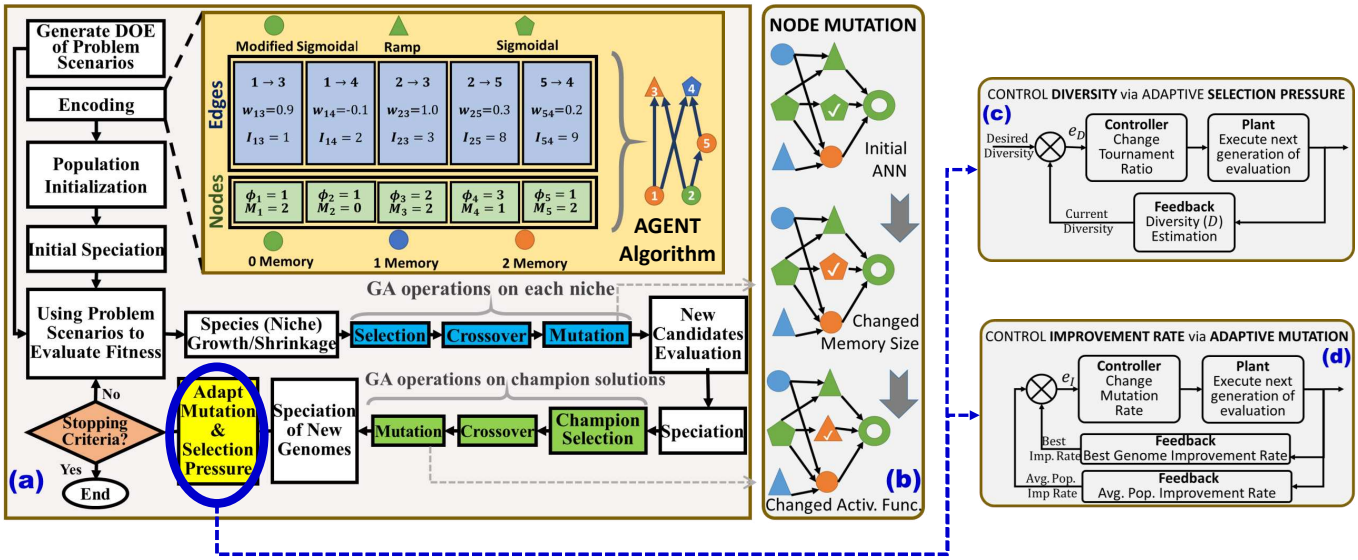


Fig. 1: AGENT Flowchart; (a) Overall AGENT algorithm, (b) New types of nodal mutation, (c) Diversity controller (via adaptive selection pressure), (d) Improvement rate controller (via adaptive mutation).

The memory property is introduced with the goal of capturing the temporal dynamic behavior that is typically demanded of neurocontrollers, without having to change the feed-forward structure (to say a recurrent structure). Here, the *memory* property of neurons is allowed to take one of three values: $M \in \{0, 1, 2\}$; a memory size of 0 designates using the current weighted aggregate input incoming into the node; memory sizes of 1 and 2 respectively designate using the first and second time derivatives of the weighted aggregate input incoming into the node. Equation 2 explains how the effective net synaptic input to any node-i is computed based on its memory property. In this equation, for a node connected to n_i upstream nodes, $V_i(\tau)$ gives the net synaptic input of node-i in time step τ, o_j gives the output of the upstream node-j, $\delta\tau$ is the time step used when implementing the neural network as a controller, and $U_i(\tau) = \sum_{j=1}^{n_i}(w_{j,i} \times o_j)$.

$$V_i(\tau) = \begin{cases} U_i(\tau), & \text{if } M = 0 \\ \dfrac{U_i(\tau) - U_i(\tau - \delta\tau)}{\delta\tau}, & \text{if } M = 1 \\ \dfrac{U_i(\tau) - 2U_i(\tau - \delta\tau) + U_i(\tau - 2\delta\tau)}{(\delta\tau)^2}, & \text{if } M = 2 \end{cases} \quad (2)$$

For control and planning problems, $\delta\tau$ can be defined to be the simulation time step at which state information is recorded during episodic trials. Here, we use $\delta\tau = 1$. Therefore, the above memory equations seek to estimate and propagate the first and second time derivatives of the information computed insitu w.r.t. the state inputs.

B. Initialization

Unlike the original NEAT and the majority of its variations, the AGENT permits a small number of hidden neurons in genomes of its initial population, instead of a minimalist topology with no hidden neurons. This approach helps expedite the optimization process in most higher-dimensional problems or in problems with Multiple Inputs and Multiple Outputs (MIMO), which usually demand a more complex ANN than that provisioned by a minimalist topology. The number of hidden nodes in each genome is chosen from a distribution such that the expected value (over the population) of the number of hidden nodes is given by $\sqrt{n_I \times n_O}$ which helps diversifying in the initial population. Here n_I and n_O are respectively the numbers of input and output nodes. This average sizing of initial ANNs is motivated by the work of Stathakis [45]. This initialization is merely suggestive, since AGENT can both complexify and decomplexify the neural network by adding and removing neurons, and can thus work with other minimalist or prescribed initializations as well. Since AGENT evolves non-layered TWEANNs with neuronal properties that offer additional variability, numerical experiments expectedly showed that our initial size requirements were slightly smaller than that recommended in [45].

C. Speciation

Speciation is a crucial process in the AGENT algorithm. In this process, the population is divided into several groups or species (also referred to as niches). Speciation has two objectives. First, it protects newly generated genomes (which contain yet-to-be-stabilized weights) from elimination due to selection pressure. Second, it facilitates searches within each species. The speciation process adopted here is conceptually similar to that in SUNA [19]. The speciation process has three steps. In step 1, we order the networks based on their uniqueness. In step 2, the networks with the highest uniqueness values are selected as the *anchor genomes* to form *niches*. In step 3, the rest of the population is classified into these niches, based on their similarity to each *anchor genome*. While for the ease of implementation, the total number of groups are fixed (here, set at $\min(\lceil \frac{N_{pop}}{10} \rceil, 8)$), the size of groups could vary across generations. This is because the distribution of genomes w.r.t. their design space varies across generations.

The uniqueness of a genome is defined by how distinct a genome is from others in the population, based on a distance measurement in the design space of TWEANNs. This distance metric is described in Section III-A. Specifically, the unique-

ness (u_A) of a generic genome-A can be expressed as

$$u_A = \min_{\forall B \neq A} d_{A,B} \tag{3}$$

Here, B represents any genome in the population other than A, and the distance metric d_{AB} provides a measure of dissimilarity of the two genomes, A and B.

D. Selection

Here, tournament selection [46] is used in AGENT due to its property of being invariant to order-preserving transformations. Moreover, compared to the proportional selection [46], tournament selection provides the opportunity to adapt the selection pressure by varying the ratio between the number of genomes that participate in the tournament and the number of genomes that are allowed to win the tournament. The classical tournament selection uses one-to-one competitions, with each genome competing twice with randomly selected opponents. Here, we define a more generalized multi-player competition, comprising randomly selected N_T genomes, among which the fittest N_W genomes are allowed to win the tournament and get copied to the mating pool. A total of $2N/N_W$ such random tournaments are conducted. This allows the selection process to create a mating pool of size $2N$, from which N offspring are subsequently derived via the crossover operation. We will later see in Section III-B that, the probability of getting selected, i.e., the selection pressure, can be regulated by controlling the N_W/N_T ratio. Diversity increases when this ratio decreases. Note that since crossover here (described next) produces one offspring per pair of parent genomes, a mating pool of size $2N$ is created to preserve the population size over generations.

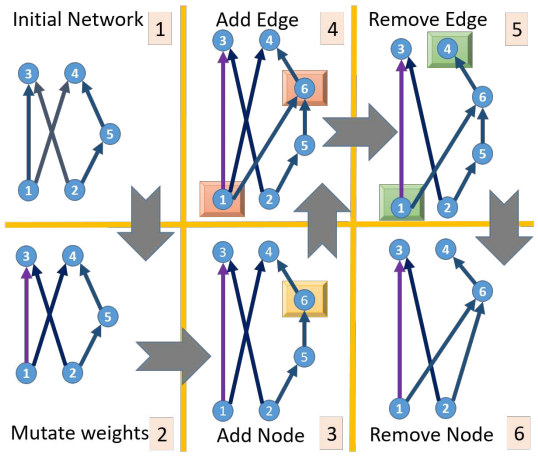
E. Crossover

AGENT expands on NEAT's crossover process by transferring the special nodal properties to the offspring. Figure 2 (c) illustrates this procedure. Weights of common edges are inherited randomly from one of the two parents, with an equal probability. The nodal properties, i.e., the activation function type, and memory type, as well as the weights associated with unique edges, are inherited from the parent genome with higher fitness.

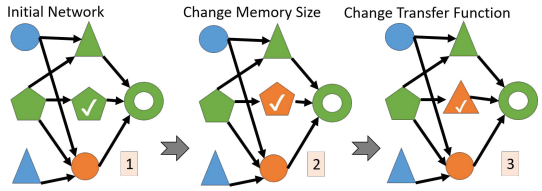
F. Mutation

The mutation operator in AGENT includes edge mutation that adds connections similar to that in NEAT, as well as node mutations as implemented in other later architectures [19], [47]. At the same time, we also implement newer types of edge mutations for decomplexifying the network (not present in NEAT or its popular variations), and newer node mutations that allow switching between nodes with different memory and functional characteristics. As illustrated in Fig. 2 (a, b), there are three major types of mutation in AGENT: 1) connective (continuous space): mutation of edge weights; 2) topological: addition and removal of edges and nodes of the network; and 3) neural (categorical space): mutation of nodal properties. Each type of mutation has its own mutation rate, whose initial value (μ_0) can be prescribed, and the rate (μ_t) is then controlled over generations (as described in Section III-D).

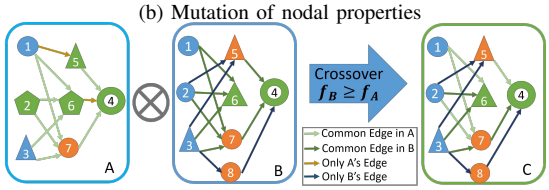
It should be noted that the addition of edges/nodes increases the complexity of the network, and their removal leads to the simplification of the network. With this capacity to both complexify and simplify network topologies, the mutation



(a) Mutation of edges and nodes (addition or removal) and mutation of real-valued edge weights



(b) Mutation of nodal properties



(c) Crossover: f_A and f_B are the fitnesses of genomes A and B; since $f_B \geq f_A$, unique edges (e.g., $3 \rightarrow 8$) and nodal properties (e.g., nodes 2 and 6) are inherited from B.

Fig. 2: Mutation and Crossover operations in AGENT

operation in AGENT is hypothesized to enhance the computational efficiency of the learning process, while still preserving the ability to generate parsimonious policy models (as in other TWEANNs). The different types/sub-types of mutation operations in AGENT are further described below.

Mutation of existing edge weights: For existing edges between any two nodes i and j, real-valued Gaussian mutation [48] of weights is undertaken, as given by:

$$\begin{aligned} w_{i,j} &= w_{i,j} + r \\ r &\in \mathcal{N}(0, \sigma_W^2) \end{aligned} \tag{4}$$

Here, $w_{i,j}$ is the weight of the edge connecting nodes i and j. The extent of mutation is controlled by the standard-deviation parameter σ_w, which can be prescribed by the user. A default value of 0.2, i.e., 10% of the initial range of edge-weight, is used in our case studies.

Addition of an edge: An edge can be added between any existing pair of nodes (i and j) as long as duplicate edges and cycles are not produced. The new edge is assigned a unique innovation number, $I_{i,j}$, and a weight $w_{i,j}$ chosen randomly from a uniform distribution in the range $[-1,\ +1]$.

Removal of an edge: An edge between any existing pair of nodes can be removed as long it does not produce a floating node. A floating node is one that either has no incoming edges or no outgoing edges. In our implementation,

the mutation rate for removing an edge is kept at 80% of the mutation rate for adding an edge, thereby allowing the slightly greater probability of network complexification (compared to simplification).

Addition of a node: A new node k can be added on an edge (previously connecting nodes i and j), resulting in the splitting of the edge into two edges. The two new edges thus created are assigned new innovation numbers ($I_{i,k}$ and $I_{k,j}$) and randomized new initial weights ($w_{i,k}, w_{k,j} \in [-1,\ +1]$).

Removal of a node: Any hidden nodes can be removed. After removing the hidden node, all incident downstream nodes (with respect to the removed node) are connected to all upstream nodes to which the removed node was connected using new connections. The weights of the new connections or edges are again randomly assigned from the range $[-1,\ +1]$. Node removal probability is kept at 80% of the probability of adding a node.

Mutation of nodal properties: Nodal properties are muted by probabilistic switching between the categorical values of these properties, i.e., different activation functions or memory values. In the case of memory, the probability of selecting $M = 0, 1, 2$ are respectively set at 0.5, 0.25, and 0.25 to give more weight to the memory-less type of node. In the case of activation function, all three types have an equal probability of getting selected during mutation.

The mutation operations of adding/removing edges or nodes are illustrated in Fig. 2(a) and those of changing nodal properties are illustrated in Fig. 2(b). Each of these mutations is carried out for a given node/edge of a candidate ANN genome in the population, only if the following criteria is satisfied: $r \leq \mu_t$; here r is a random number between 0 and 1, and μ_t is the rate at generation t associated with that particular type of mutation. Each of the above-described types of mutation has its own rate parameter. Supplementary materials Section S - III list the different mutation rate parameters, their subject of operation, their default initial value, and the recommended range. These rates are controlled over generations in order to adapt the rate of fitness improvement suitably.

Note that the list of crossover, mutation, elitism, selection, speciation and diversity operations in AGENT, and the associated parameters and their settings are provided as Table I in the Supplementary Material S - III.

III. Adaptation Mechanisms in AGENT

In this section, we explain the formulations proposed to adaptively regulate the diversity and fitness improvement rate in AGENT along with the requisite metrics and limits, ending with a description of how we quantify network complexity.

A. Diversity Preservation: Measure of Diversity

Population diversity is paramount to effective neuroevolution. This calls for adaptively regulating the population diversity – an abrupt decrease in diversity can lead to premature stagnation, but at the same time, a steady (low) rate of diversity reduction is needed for exploitation and eventual convergence. The first step in diversity preservation is measurement of diversity in the population. To develop a diversity measure, we need to first measure the net difference between the designs of any two neural networks, i.e., the difference in their structure and weights. Subsequently, an approach (which is currently lacking) is needed to use this measure of differences to quantify the overall diversity in the population. In this paper, we propose a new approach to quantifying the diversity in a population of ANN genomes.

In neuroevolution, since genomes encode different topologies, their basic dimensionality varies across the population. Hence, a distance metric partly motivated by the novelty metric [19] is used here to quantify the difference between any two genomes. The distance, $d_{A,B}$, between two candidate ANNs, A and B, is thus given by the weighted sum of the difference between their node types, as well as the difference between edges connecting different types of nodes.

$$d_{A,B} = \alpha_P \sum_{i=1}^{P_T} |P_{i,A} - P_{i,B}| + \alpha_E \sum_{i=1}^{P_T} \sum_{j=1}^{P_T} |E_{i\to j,A} - E_{i\to j,B}| \tag{5}$$

In Eq. 5, $P_{i,A}$ and $P_{i,B}$ are the number of nodes of type i respectively in neural networks A and B; $E_{i\to j,A}$ is the number of edges from node type i to node type j in neural network A; and P_T is the number of types of activation functions allowed in the ANN. The weights α_P and α_E are prescribed to be 0.5 in this paper. Note that only the type and connectivity of neurons, and not the continuous parameters associated with them (namely weights and biases), are considered to quantify the difference between candidate neural networks. This approach is based on the premise that as long as diversity in the combinatorial characteristics (which includes topology) of the network can be preserved, the weights will anyways converge and follow suit, since the time constants of weight change is much higher than that of the topology. This approach also works well with the emerging hybrid concept of optimizing the continuous parameters insitu through a gradient descent approach [49], while the topology is being evolved.

To quantify the overall population diversity at any generation t, we construct a complete undirected graph K_N out of the population of N candidate genomes. The length of the edges connecting candidate genomes in this graph is given by the distance metric in Eq. 5. Then, by employing the concept of *minimum spanning tree* (MST) on the graph K_N, the overall population diversity is given by the total length of the MST. The concept of using MST to represent diversity has been explored in population-based optimization methods [50]. Figure 3 illustrates how the concept of MST is applied to the graph derived from the estimated distances between ANN genomes in the population. Thus, the population diversity D_t at the t^{th} generation can be expressed as:

$$D_t = \sum_{\forall\ A,B \in \text{M.S.T}} d_{A,B} \tag{6}$$

where A, B represents an edge connecting genomes A and B in the population graph. Kruskal's Algorithm [51] is used to determine the MST, which involves a frugal computational expense of the order of $O\left(|K_N| \log |K_N|\right)$, where $|K_N|$ is the number of edges in the graph. For reference, a pseudocode of Kruskal's algorithm is given in the Supplementary Material Section S - I.

In order to effectively calibrate diversity, we can now define a desired value for diversity and also delineate an approach

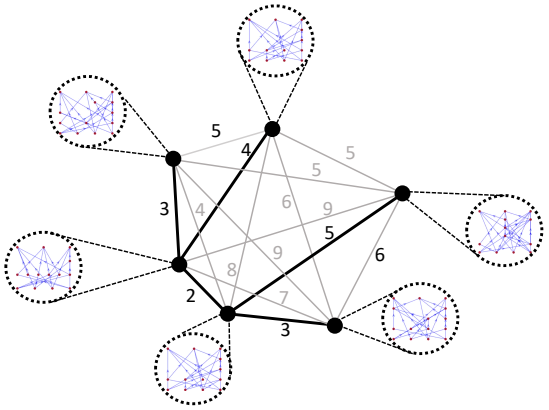


Fig. 3: Applying Kruskal's Algorithm [51] to compute the MST (dark edges get excluded) for the graph based on differences between ANNs in the population

to maintain this desired value. The desired diversity, $D_{d,t}$, can be defined as a linear (gradual) decay function of number of generations (t), as given by:

$$D_{d,t} = D_0 \times \frac{\beta_D \times t_{\max} - t}{\beta_D \times t_{\max}} \tag{7}$$

Here D_0 is the measured diversity in the initial population, the coefficient $\beta_D \geq 1$ facilitates gradual decay and seeks to facilitate a lower bound to the diversity retained in the final population, and $t_{\max}$ is the maximum allowed number of generations. The measurement of current diversity and the desired diversity decay formulation are together used to control the selection operation, in order to regulate the population diversity, as described next.

B. Diversity Preservation: Controlling Diversity

Since the selection process guides how greedy (biased towards the fittest genomes) or diverse will be the resulting mating pool, it can also serve as a medium for diversity preservation. Here, the tournament size in the selection operator is used as the control input to regulate the diversity.

In a population of N genomes, the probability of selecting a specific genome, to be copied into the mating pool, decreases with the number of genomes participating in the tournament (N_T) and increases with the number of the genomes chosen from the tournament (N_W). More specifically, the probability of the k^{th} ranked genome to be selected into the mating pool ($\mathbb{P}$) by resampling can be approximated by the following expression (where the higher order terms are ignored):

$$p(k \in \mathbb{P}) = 1 - \left[\frac{N - N_T}{N} + \frac{N_T \sum_{i=N_w}^{N_T} \binom{k-1}{i} \binom{N-k}{N_T-i-1}}{\binom{N}{N_T}} \right]^{\frac{2N}{N_W}} \tag{8}$$

where the rank k is determined by sorting the population simply based on fitness. The derivation of this probability of selection is provided in the supplementary document Section S - II. By decreasing N_T, the effect of the term $\left(1 - \frac{(k-1)! \times (N-N_W)!}{(N-1)! \times (k-N_W)!}\right)$ decreases and the probability of choosing all genomes become more uniform (less dependent on ranking). Similarly, increasing N_W tends to decrease the standard deviation in the probability of selection across the ranked population, considering that $\mathbb{P}$ is a function of k (ranking of genomes).

In summary, it can be seen that the probability of choosing lower ranked genomes increases by increasing the ratio $\frac{N_W}{N_T}$. Therefore this ratio can be used to decrease the selection pressure, thereby increasing the diversity, and thus serves as a suitable choice for a control input. For regulating diversity at any t^{th} generation, we define the control input as a negative exponential function:

$$\left.\frac{N_W}{N_T}\right|_t = \left.\frac{N_W}{N_T}\right|_{t-1} \times e^{-K_D(D_t - D_{d,t})} \tag{9}$$

K_D is the diversity gain coefficient, which regulates the degree of change that must be applied to the tournament ratio. We set $K_D = 0.1$. $(D_t - D_{d,t})$ represents the difference between the observed diversity (Eq. 6) and the desired diversity (Eq. 7). Due to undesirable diversity loss when this difference falls below zero, the above controller will increase the N_W/N_T ratio, therefore relaxing the selection pressure. This, along with speciation, is premised to provide lower ranked but potent genomes (e.g., more complex ANN topologies with premature weights) a greater chance of being selected into the mating pool.

C. Improvement Adaptation: Metric of Improvement

The premise behind tracking and controlling fitness is its ability to reflect whether adequate search dynamics are afforded by the population. Since diversity is simultaneously being preserved, steady improvement in average fitness over generations compared to the improvement in the fitness of the population's best fitness is reflective of a useful balance in exploration and exploitation, and thus a robust search process. With this premise, we first define an improvement metric that captures the history of improvement, as given by

$$I_t = \sum_{i=0}^{t-1} \left(\alpha_I \left(f_t - f_i\right)\right)^{\frac{i}{t}} \tag{10}$$

Here, f_t and f_i represent fitness function values at the t^{th} (current) generation and the i^{th} generations, respectively. The constant α_I is a scaling coefficient. This metric is designed such that more recent improvements have a greater influence. Both improvements in the average fitness ($f_{\text{av},t} = \sum_{i=1}^{N} f_i$) and the fitness of the population best ($f_{\text{be},t} = \max_{i=1}^{N} f_i$) are measured using Eq. 10.

D. Improvement Adaptation: Mutation Controller

If the rate of improvement in the average fitness of the population lags far behind the rate of improvement in the best fitness value, it demonstrates a weakening exploitation dynamic. In TWEANNs, mutation is the main driver of network innovation. Too high a rate of mutation leads to the generation of new niches of ANNs that do not get time to stabilize their weights, and the algorithm starts acting as random search – and thus the average fitness improvement starts lagging. Conversely, when the rate of fitness improvement in the population's best fitness lags behind that of the average fitness of the population, it is indicative of potential stagnation at local optima and weakening exploration. This situation calls for increasing the mutation rate to facilitate the discovery of new networks.

Thus, it is important to regulate the rate of mutation in order

to preserve a balance between the improvement in average population fitness and the population's best fitness. Thereof, we formulate the following mutation rate (μ_t) controller:

$$\mu_t = \mu_{t-1} \times e^{-K_I \times \frac{I_{\mathrm{be},t} - I_{\mathrm{av},t}}{I_{\mathrm{be},t}}} \tag{11}$$

where μ_t is the mutation rate in generation t, Here, $I_{\mathrm{av},t}$ and $I_{\mathrm{be},t}$ represent the fitness improvement metrics for the population's average fitness and the population's best fitness, respectively. These quantities are computed using Eq. 10. In Eq. 11, K_I is the mutation controller gain coefficient which can be prescribed to increase/decrease aggressive search dynamics. Here, K_I is set at 0.1. This controller is applied on the rates of different types of mutations, as listed in supplementary materials Section S - III. Essentially, this controller (Eq. 11) decreases the mutation rates when the average fitness improvement lags behind the best fitness improvement, and vice versa.

E. Measure of ANN Complexity

An understanding of how an evolutionary approach varies the network's topological complexity is critical to analyze its effectiveness in adapting expressibility to a given problem. When determining the complexity of a neural network, we use the popular measure of Von Neumann Entropy [52] with quadratic approximation, specifically the idea of Normalized Laplacian for directed graphs [53]. When expanded [54], this idea leads to the following expression for network complexity:

$$S = 1 - \frac{1}{n} - \frac{1}{n^2} \sum_{(u,v)\in E} \frac{1}{d_{\mathtt{out},u} \times d_{\mathtt{in},v}} \tag{12}$$

Here n is number of nodes and $(u, V) \in E$ is a directed edge connecting nodes, with $d_{\mathtt{out},u}$ and $d_{\mathtt{in},v}$ indicating the out and in degrees of nodes u and v respectively.

IV. Benchmark Testing AGENT: OpenAI Gym

In this section, we apply the new AGENT algorithm to solve a suite of benchmark problems in RL and control, with the next section describing an application of AGENT to solve a complex online planning problem in the UAV domain. The benchmark problems are adopted from OpenAI Gym [10], which is an open-source platform that has been growing in popularity for benchmarking and comparing RL algorithms [55], as well as other learning and optimization methods [23] to solve control problems and other RL type problems. In this paper, we showcase the performance of AGENT on four problems from the Classic Control and Box2D suites in OpenAI Gym. These problems include: **1)** *Mountain Car*, **2)** *Acrobot*, **3)** *Lunar Lander*, and **4)** *Bipedal Walker*. Brief descriptions of these problems are given in the Supplementary Material section S - IX. For ease of reference, visual snapshots of these problems and definitions of their state and action spaces are included as Table V in the Supplementary Material. Further information on these problems can be found at the OpenAI Gym website [2]. The problem settings such # inputs/outputs, high-level settings in AGENT such as population size and max generations, and size of neural networks resulting from AGENT are summarized as Table S - II in the Supplementary Material.

[2]https://github.com/openai/gym/wiki/Leaderboard

This Section is structured as follows: First, we summarize each benchmark problem and present the results of the AGENT algorithm and its comparison with state-of-the-art RL results reported for that problem. We also compare AGENT's performance with that of a standard NEAT implementation [42] and Adaptive HyperNEAT [25]. While comparing the goal function or net reward value is straightforward for most problems, it is challenging to compare the computational costs across methods coherently. Typical measures of cost (e.g. number of episodes, number of time steps, CPU-time) are unreliable when comparing across different RL and evolutionary methods. This is because of the difference in the nature of episodic sampling in RL vs. Neuroevolution, greater cost accumulated by more successful policies in some problems, impact of the ODE solver's time-stepping in control problems, and different degrees of parallelizability of these methods (mostly in favor of evolutionary methods [23], [11]). We still report the # episode used by each method, mainly for reference. The results of AGENT and the comparative algorithms on the benchmark problems are all summarized in Table I, and discussed below. Note that, since the neuroevolution methods are stochastic, AGENT, NEAT and Adaptive HyperNEAT are each run 10 times on every OpenAI problem. Their results, i.e., net reward value and # episodes used, are then reported in terms of mean $\pm$ standard deviation over those 10 runs. Since the RL problems are from literature and NEAT function evaluation is controlled by its algorithm , it is difficult to make the number of function evaluations equal. Furthermore, neuroevolution follows a highly parallelizable algorithm which makes the computation cost of neuroevolution a function of number of generations instead of function evaluations.

Subsequently we discuss the convergence history and optimum network topology obtained by the best run of AGENT on each problem. We also provide an analysis of the change in the average/best improvement rates, population diversity and network complexity over generations in AGENT. Selected benchmark problems are also used to conduct ablation tests to analyze the impact of the unique features of AGENT, namely the selection and mutation controllers and the memory property of nodes. Note that an extended analysis of how the new diversity measure in AGENT compares to diversity in the outputs given by different networks in the population is provided in the Supplementary Materials Section S - V.

A. OpenAI Gym Problems: Comparative Analysis

Here we briefly describe the results obtained by AGENT for each of the four OpenAI benchmark problems, and how they compare to state-of-the-art RL methods reported in the literature. The choice of the RL methods used for comparison is based on the availability and reported superiority of their published results on these benchmark problems.

1) Reward Evaluation for OpenAI Gym Problems

The four OpenAI Gym problems used here involve stochastic elements (e.g., randomized initiations and action uncertainties) which makes it challenging to have a robust evaluation of the reward based on a single episode. Hence, multiple episodes are used to evaluate the quality of each genome, and for efficiency we allow each genome to progress to the next episode (during evaluation) only if it has gathered a threshold

amount of reward. Mathematically, this can be expressed as:

$$F_i = \sum_{j=1}^{N_{a,i}} R_{i,j}; \quad F = \frac{1}{N_s} \sum_{i=1}^{N_p} F_i \tag{13}$$

where $R_{i,j}$ is the reward the agent receives for each action taken; $N_{a,i}$ is the total number of actions taken in the i-th scenario; F_i represents the genome's accumulated reward in that scenario; F represents the net fitness function evaluated for an ANN genome; and N_s refers to the maximum number of scenarios available during training. In Eq. 13, N_p refers to the number of scenarios where the genome surpasses an adaptive threshold that is set based on the historic performance of genomes (e.g., a threshold of $0.8 \times F_i^*$ if $F_i^* \geq 0$ or $1.2 \times F_i^*$ if $F_i^* \leq 0$, where F_i^* is the reward accumulated by the previous best genome in the i-the scenario). Having multiple scenarios (here, five scenarios for AGENT) is posited to mitigate overfitting to training scenarios. The increased probability of mutations that complexify the network allows better exploration and thus addresses underfitting.

2) Mountain Car: Results

For this test problem, the performance of AGENT is compared to the performance reported for the Random Weight Guessing (RWG) method [56]. RWG is an exploration based method, and thus incurs a relatively high number of episodes to converge. From Table I, it can be seen that AGENT was able to find a 3% greater net reward value with negligible variance (and in fewer number of episodes) than that reported for the RWG method.

3) Acrobot: Results

For this test problem, the performance of AGENT is compared to that reported for the popular Q-learning method with Adaptive Memory Replay [56]. As can be seen from Table I, AGENT is able to achieve a clearly better net reward value compared to that of the RL method [57]). However, in this case AGENT requires a substantially greater number of episodes to converge, compared to RL; note that AGENT reaches a comparable net reward value much sooner, as evident from the convergence history plot in Fig. 4b.

4) Lunar Lander: Results

The optimum results obtained by AGENT for this problem is compared to that reported for a popular implementation of the SARSA method [58]. As seen from Table I, AGENT does not perform as well as the RL method. Considering the complexity of this problem, the results are expected to be better if a larger population size or greater # maximum generations are allowed. Moreover, the handcrafted design of the reward functions in such problems often aggregate multiple criteria which might be more favorable to particular solution approaches.

5) Bipedal Walker: Results

The optimum results obtained by AGENT for this problem is compared to that reported for a popular implementation of Policy-on Policy-off Policy Optimization (P3O) type RL method [61]. Since the number of steps is not reported, we used the maximum number of steps per episode to evaluate minimum number of episodes in learning (which is at best, an approximation). The NEAT implementation available to us [60] encountered programmatic errors for Bipedal Walker, and hence the results of NEAT is adopted from [59] for comparison, and appear to be marginally better than the mean performance of AGENT over ten runs for this problem (as seen in Table I). As observed from Table I, all the neuroevolution methods perform poorer to the RL (P3O) method, which is expected given the larger size of the state space for this problem. Larger population size and initiation with larger/deeper networks in the initial population might be able to address this issue in the future. Between AGENT and HyperNEAT, Table I shows that AGENT performs marginally better than HyperNEAT w.r.t. the optimum reward value obtained.

B. Comparison with NEAT and Adaptive HyperNEAT

We also compare AGENT with a state-of-the-art implementation of NEAT (with recurrent network) in Pytorch [60], based on [17] and an advanced variation of NEAT, aka HyperNEAT. We run NEAT with similar settings as that used in AGENT for the corresponding problem. Note that, when implementing NEAT on the Acrobot and Lunar Lander problems, each genome is evaluated over all five random episodes and the progressive reward computation concept (Eq. 13) is not used; without this allowance NEAT performance was very poor. Table I lists the results of this NEAT implementation. Unlike NEAT, HyperNEAT uses Compositional Pattern Producing Networks (CPPN) [24] to indirectly encode the weights, and facilitate evolution of larger networks. In our case, we compare with adaptive HyperNEAT, which uses CPPN to both encode weights and the Hebbian rule [62] for updating the network. For Adaptive HyperNEAT the progressive reward computation concept (Eq. 13) is used, as in AGENT.

As seen from Table I AGENT performed better than Pytorch NEAT in terms of net reward values in all three OpenAI problems. AGENT took roughly half the number of episodes to converge as NEAT did on the Acrobot and Lunar Lander problems. The performances of the two methods are comparable for the Mountain Car problem, with AGENT exhibiting lower variance in computing cost. While these results could change if PyTorch NEAT is implemented with further calibrated settings, the comparison provided here is intended to (and does) demonstrate the favorable effects of the fundamental modifications in AGENT over the standard NEAT algorithm.

From Table I, it can be seen that the performance of AGENT and Adaptive HyperNEAT are relatively similar in terms of reward values for the Mountain Car and Acrobat problems. However, Adaptive HyperNEAT is found to require a fewer number of episodes to converge on the corresponding optimum solutions. In the case of the Lunar Lander problem, AGENT is found to arrive at better reward values than Adaptive HyperNEAT. The higher episodic cost of AGENT is attributed to its explicit diversity preservation, which usually comes at some compromise in convergence rate, as well as likely due to the use of direct encoding of weights.

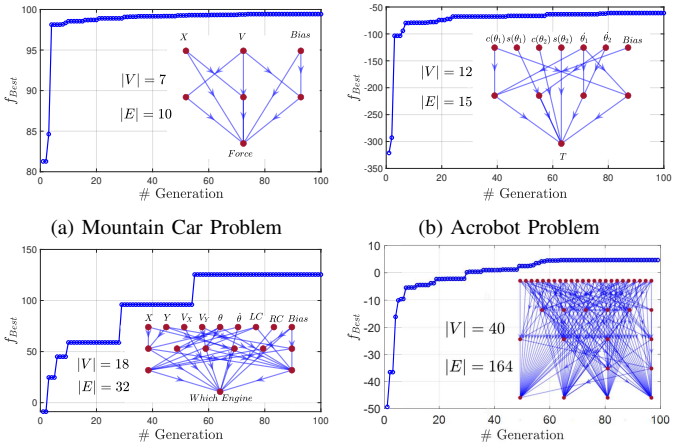
C. Computation Cost Analyses

As seen from Table I, in terms of the # episodes used, the training cost of AGENT appears comparable to (or slightly better than) that of NEAT and poorer than that of Adaptive HyperNEAT, across the OpenAI Gym problems. Here, all three neureovolution algorithms are subject to the same population size, the same maximum number of episodes (5 scenarios, here)

AGENT (our method)	**Mountain Car**	**Acrobat**	**Lunar Lander**	**Bipedal Walker**
Best Reward*	**99.25 ± 0.14**	**-65.20 ± 3.56**	86.69 ± 18.28	4.5 ± 0.25
# Episodes	$22,027 \pm 775$	$47,346 \pm 1,917$	$43,339 \pm 438$	$23,955 \pm 543$
RL Benchmark	Random Weight Guessing [56]	Adaptive Memory Replay [57]	SARSA [58]	P3O [59]
Best Reward*	96.1	~ -75.0	$\sim$ **200**	$\sim$ **250**
# Episodes	200,000	1,000	40,000	≥ 6250
NEAT (PyTorch) [60]	**Mountain Car**	**Acrobat**	**Lunar Lander**	**Bipedal Walker [59]**
Best Reward*	65.07 ± 42.64	-119.64 ± 1.57	10.63 ± 11.02	$\sim$ **6.75***
# Episodes	$17,887 \pm 8,932$	$100,143 \pm 87$	$99,980 \pm 52$	≥ 6250
Adaptive HyperNEAT (PyTorch) [25]	**Mountain Car**	**Acrobat**	**Lunar Lander**	**Bipedal Walker**
Best Reward*	99.24 ± 0.05	-67.00 ± 2.79	0.92 ± 47.98	1.62 ± 2.80
# Episodes	$6,790 \pm 794$	$11,180 \pm 518$	$10,899 \pm 435$	$10,646 \pm 252$

*The Bipedal Walker results for NEAT (from [59]) doesn't use settings similar to what AGENT and HyperNEAT uses in our simulations.

TABLE I: Benchmark evaluation on OpenAI Gym problems: AGENT vs. reported results of state-of-the-art RL methods and of PyTorch implementations of NEAT, and Adaptive HyperNEAT. Due to their stochastic nature, AGENT NEAT and HyperNEAT results are reported in terms of mean ± standard deviation of the optimized fitness over 10 runs. [Note that reward values are higher the better]



(a) Mountain Car Problem (b) Acrobot Problem

(c) Lunar Lander Problem (d) Bipedal Walker Problem

Fig. 4: Convergence history of the best of 10 runs of AGENT on OpenAI Gym problems, and the topology of the corresponding optimized network

for evaluating each unique candidate network, and the same maximum number of generations for each problem, thereby promoting fair comparisons. Now note that, due to the progressive reward evaluation approach as explained in Section IV-A2, and that only uniquely new network genomes are evaluated in any generation, the executed number of episodes per candidate evaluation (thus ≤ 5 here) and per generation respectively both vary. Hence, even though the neureovolution algorithms are run till the same prescribed maximum number of generations (as seen from the convergence history plots in the Supplement, Fig. S - 3), the overall costs in terms of total # episodes could be different. Here, HyperNEAT cost in this respect is lower likely due to fewer candidates qualifying to be evaluated over all five episodes, and the presence of fewer uniquely new genomes in subsequent generations, compared to AGENT. On the other hand, while the reported RL results shows smaller computing cost compared to all three neuroevolution algorithms in terms of # episodes (as Table I shows), note that neuroevolution benefits from ready parallelization, and hence actual clock-time comparison could be different, as explained towards the start of Section IV.

For practical context, here we also report the computing time of a representative runs of AGENT – For a system with Windows 10, 16GB RAM and Ryzen 7, 2900 MHz 8-core CPU, AGENT took: **i)** 50 mins to run 50 generations of the OpenAI Mountain car problem with a population of 50 candidates; **ii)** 220 mins to run 50 generations of the OpenAI Acrobot problem with a population of 200 candidates; and **iii)** 65 mins to run 50 generations of the OpenAI Lunar lander problem with a population of 200 candidates.

D. Convergence History & Optimum ANN Topology

To provide insights into the nature of convergence of AGENT and the complexity of the resulting optimum ANN topologies, we use the best run of AGENT from each test problem, and show the corresponding convergence history outcomes in Fig. 4. It can be observed from Fig. 4 that significant improvement (between 15 to 150 % for different problems) in net reward values were accomplished during the neuroevolution process in each benchmark problem. The improvement is relatively gradual for the more complex Lunar Lander problem. We also observed that AGENT was able to generate small (parsimonious) policy models for each of these problems. To put that into comparative perspective, consider that for the Lunar Lander problem, SARSA (State–Action–Reward–State-Action) algorithm [58] trained a model with 2 hidden layers and a total of 32 nodes (16 RELU and 16 softmax nodes) and over 400 weight and bias parameters vs. a policy network with only 18 hidden nodes and 26 weight parameters evolved by AGENT (Fig. 4c). For the Mountain Car and Acrobot problems, the reported RL methods used for comparison do not clearly report their model sizes.

For comparison of convergence trends, the convergence history plot of the best of 10 runs of HyperNEAT on each of the four OpenAI Gym problems are included in the Supplement as Fig. S - 3. The AGENT convergence histories are re-plotted therein as well. We observe from that figure that HyperNEAT stalls early on for Mountain Car, Acrobot problems and Bipedal Walker problems. For Lunar Lander, while HyperNEAT shows gradual improvement, it converges to a premature ($\sim 45\%$) poorer reward value compared to AGENT.

A more in depth analysis of the change in ANN complexity over AGENT's generations, and the accompanying process of fitness improvement and variation in population diversity is provided in the Supplementary Materials section S - V.

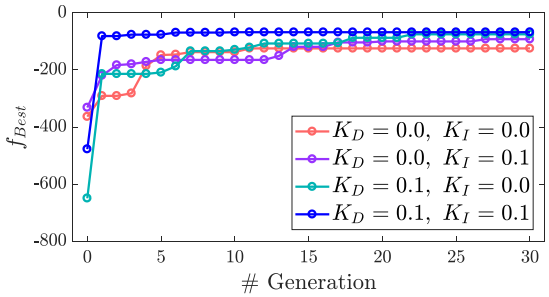


Fig. 5: Acrobot problem: AGENT's convergence history of the median case under each controller setting

E. Ablation Tests: Impact of the Adaptive Controllers

Here, we analyze the effect of diversity and improvement rate control in AGENT, achieved via regulation of selection pressure and mutation. Specifically, to study the individual and joint impact of the two controllers, we conduct an ablation test by using the following four different combinations of the controller gains: $(K_D, K_I) \in \{(0.0, 0.0), (0.0, 0.1), (0.1, 0.0), (0.1, 0.1)\}$; a zero gain represents a shut-off controller. With each $\{K_D, K_I\}$ setting, we run AGENT 5 times on the Acrobot problem. Given the high cost of testing multiple combinations, we limit the max generations to 30 in this test, as well as in the analysis of the effect of memory nodes presented in Section IV-F.

$K_D, K_I =$	0.0, 0.0	0.1, 0.0	0.0, 0.1	0.1, 0.1
Reward: Mean	-127.0	-105.8	-97.2	**-83.7**
Reward: Median	-124.50	-92.2	-75.0	**-67.5**
Reward: Std-Dev	49.5	44.3	47.6	**24.4**

TABLE II: Acrobot: AGENT's best genome's reward over 5 runs, under each *controller setting* – K_D: diversity controller gain (regulates selection), K_I: avg vs. best improvement rate controller gain (regulates mutation)

Table II summarizes the distribution of the reward value of the best genome obtained by AGENT over 5 runs, under each controller setting. The convergence history of the median case under each controller setting is shown in Fig. 5. The results in Table II show that using controllers has a clear positive effect on mean performance (higher) and robustness (smaller std-dev). Overall, the performance improvement, in terms of mean or median value (higher the better) and variance (lower the better) of the best reward obtained, is most significant when both controllers are used together, as seen from Table II. Now, when we consider the starting points (i.e., the quality of the best genome in the initial population) in the median cases, the diversity controller ($K_D \neq 0$) seems to provide the most significant convergence gains, as observed from Fig. 5.

F. Ablation Test: Analyze the Impact of Using Neurons with Memory in AGENT

To study the impact of using nodes with memory, i.e., nodes of types $M = 1$ & $M = 2$ (refer Eq. 2), we run AGENT on each of the benchmark problems with the memory choice deactivated (i.e., allowing only nodes of type $M = 0$). When memory nodes are available, neuroevolution is observed to yield a final network with at least one node with memory, e.g., the optimal model in the best "with memory" runs of Mountain Car, Acrobot and Lunar Lander problems contained (3/1/2), (8/1/1), and (9/2/2) nodes of types $M = 0/1/2$, respectively. The performance distribution of AGENT over 5 runs, obtained *with* and *without memory nodes* available to be selected, are summarized in Table III. The table shows that the realizable benefits of the availability of memory nodes is problem dependent. For example, while in the Acrobot problem significant mean performance gains are achieved when memory nodes are available, in the other two problems, slight decrease in mean performance is noted. These results are expected since, while the availability of nodes adds flexibility to the space of potential mappings (in continuous control), it increases the complexity of the neuroevolution search process (leading to a larger design space).

Problem	Lunar Lander	Acrobot	Mountain Car
W/O Memory	57.3 ± 23.8	-111.4 ± 15.1	**99.1 ± 0.410**
W/ Memory	**55.5 ± 23.1**	**-83.7 ± 24.4**	99.0 ± 0.276

TABLE III: Analysis of the impact of nodes with memory (Eq. 2) on AGENT performance (`Mean` ± `Std-Dev` in Net Reward value of optimum network)

V. AGENT Application: UAV-UAV Reciprocal Collision Avoidance

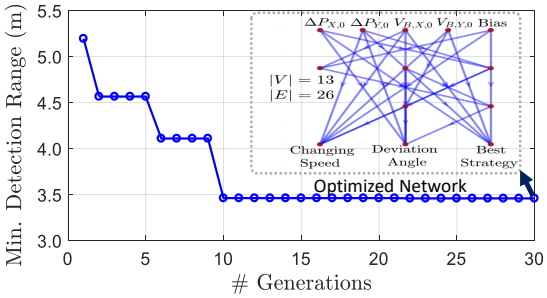
In this section, we briefly describe the reciprocal UAV-UAV collision avoidance problem, the simulation framework used for this problem, and the performance of AGENT in learning the collision avoidance maneuvers. As summarized in Supplementary Materials section S - VI-A, in the reciprocal collision avoidance strategy, two approaching identical quadcopter UAVs undertake mutually coherent maneuvers to avoid collision with each other, based on the same ANN-based maneuver planning model [43]. This model is developed via a framework called training reciprocal actions for collision evasion (TRACE) [43]. For successful collision avoidance, the distance of separation between the two UAVs must always remain greater than a safety threshold, $d_{\text{col}} = 2 \times$ `UAV_Diameter`. Two different local trajectory modification strategies are used to avoid a collision: **1)** *speed change* (SC): UAVs respectively accelerate/decelerate and decelerate/accelerate to avoid the collision; and **2)** *direction change* (DC): UAVs deviate respectively to the left of their original heading to avoid the collision. Both maneuvers are designed in a way such that the UAVs return to their original path and velocity at the end of the maneuver. Table IV lists the UAV specifications and problem settings used for training the maneuver model here.

A. UAV-UAV Collision Avoidance: Framework

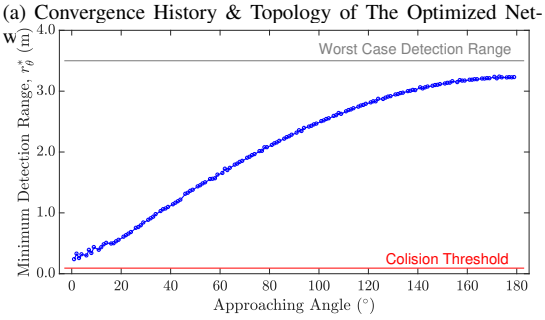
Parameters	Value
UAV Weight & Size (Dia)	28 g & 92 cm
UAV Nominal & Max Speed	8 m/s & 15 m/s
Safe Separation Distance (d_{col})	2× diameter
Reaction Time	0.1 seconds

TABLE IV: UAV collision avoidance: Simulation settings

Problem Formulation & Design of Experiments: Here the goal of designing the TRACE maneuver planning model is to decrease the dependency of the system on sensing capabilities. Dependency is represented in terms of the *minimum range of detection* demanded of the overall "sensing and peer-state estimation" system. As shown on the right part of Fig. 2, the input to the neural network based maneuver model includes the initial state parameters of the UAVs, and the output is the strategy to be applied (SC or DC strategy) and the action parameter associated with it, i.e., change in speed (δ_V) or change in heading angle (ϕ). Equation 14 summarizes the input



(a) Convergence History & Topology of The Optimized Network



(b) UAV collision avoidance: Performance of the maneuver model trained by AGENT over unseen test scenarios

Fig. 6: UAV-UAV Collision Problem: Convergence, optimum network and its performance on test cases

and output of the maneuver model.

$$[\delta_V, \phi, s] = f_\Psi(\Delta P_{X,0}, \Delta P_{Y,0}, V_{B,X,0}, V_{B,Y,0}) \quad (14)$$

where $\Delta P_{X,0} = P_{A,X,0} - P_{B,X,0}$ and $\Delta P_{Y,0} = P_{A,Y,0} - P_{B,Y,0}$ respectively represent the initial separation of the two UAVs (A: own UAV; and B: peer UAV) along the global X and Y coordinates; $V_{B,X,0}, V_{B,Y,0}$ are the initial velocities in X and Y directions for the peer UAV (UAV B). Here, f_Ψ denotes the neural network model, with Ψ encapsulating the TWEANN model description (topology, biases and weights). The output parameter s serves as a binary classifier that selects which strategy is to be used – by using the following discrimination: if $s \leq 0.5$, use SC strategy, otherwise use DC strategy. Thus, the maneuver planning neural network encompasses a multi-input-multi-output (MIMO) regression model.

In order to promote robust optimal maneuver planning, a worst case scenario perspective is taken here. The objective function to be minimized is thus defined as the worst case (largest) detection range required to just-about avoid collision across a set of uniformly distributed approach scenarios. Further details of how this objective function is computed can be found in [63]. Here neuroevolution is used to train the neural network based maneuver policy model with this objective being treated as the loss function to be minimized. Note that the possibility of the controller not being able to adequately follow the planned trajectory is also identified by using a Bagged Tree classifier (details of which can be found in the Supplementary Materials Section S - VI), thereby mitigating the need for large number flight (motion) simulations.

B. UAV-UAV Collision Avoidance: Results

The AGENT algorithm settings used to solve this problem is summarized earlier in supplementary materials and [63]. Figure 6a shows the convergence history of AGENT and the final network structure obtained for this problem. AGENT was able to discover a compact network with only 5 hidden nodes and a total of 26 edges. It can be seen from Fig. 6a that AGENT converged in $\approx$ 15 generations, and in that process achieved a 33% improvement in the fitness function over the random initial models – the minimum detection range decreased from 5.2 m to 3.5 m. This AGENT-optimized minimum detection range, that just about avoids collision, leads to a less than 1.7 seconds of detection lead-time across the different approaching angles. When HyperNEAT was applied to this UAV problem with settings comparable to that of AGENT, it was unable to find feasible solutions, with the final rewards value remaining at -200 even after 100 generations. As future work, we could look at specific reward shaping of the objective function to help HyperNEAT better solve this problem.

For further analysis, in Fig. 6b we plot the minimum detection range given by the AGENT-trained maneuver model for 144 different unseen approaching angles (i.e., test scenarios). The trained network is observed to generalize very well, with the minimum detection range showing a smooth trend, and always remaining smaller than the estimated worst case detection range of 3.5 m. In other words, the performance of the trained maneuver model persists on unseen test scenarios.

VI. Further Discussion

The analysis of the effect of memory nodes in AGENT showed that while significant performance gains can be achieved in some problems, in others the added (ANN) design space dimensionality might cause slight reduction in performance. The convergence histories, including that of rates of improvement and diversity over generations, showed that the controllers had desirable impacts, e.g., consistent changes in the fitness improvement rate of the population average and the best. Niche-averaged complexity was observed to follow a generally increasing but not necessarily monotonic trend – exhibiting an useful exploration/exploitation balance. Further statistical analysis was performed to elicit how our population diversity quantification was correlated with the observed diversity in the outputs of the networks in the population, with mostly noting a positive correlation except in the more complex lunar lander problem (where the weights of the more complex networks may not have stabilized within the allowed max generations). Statistical analysis of the impact of the controllers over the final outcomes showed that the most significant improvement across multiple runs of the algorithm was achieved when both controllers were used together.

It is important to note that mutation and selection operations impact both average improvement rates and diversity; and thus it remains challenging to discriminate and appropriately adapt their independent impacts. At present, AGENT separately adapts selection to control diversity, and mutation to control "average vs. best" improvement rate and the exploration/exploitation balance thereof. This leaves scope for further explorations regarding how to coherently adapt both of these operators to improve convergence. While more work remains to be done in the neuroevolution domain to formally describe and understand the interplay between diversity and exploration/exploitation balance, our work here offers insightful new mechanisms to quantify and potentially regulate these characteristics of neuroevolution.

VII. Conclusion

In this paper, we developed a new neuroevolution method for designing TWEANNs, by making important advancements to the NEAT formalism. The goal was to study how the new advanced features could help in mitigating premature stagnation and providing competitive performance on complex RL/control problems. The key contributions in this regard included: 1) quantifying diversity using a graph-theoretic concept and controlling diversity via adaptation of the tournament selection process; 2) controlling average fitness improvement via mutation rate adaptation; 3) include activation function choice and memory as nodal properties; and 4) allowing both growth and shrinkage of ANN topologies during evolution.

When applied to four benchmark control problems from the OpenAI gym platform, AGENT provided competitive results in terms of final reward values and network complexity, compared to results reported with state-of-the-art RL methods. AGENT was also compared to well-known PyTorch implementations of NEAT and Adaptive HyperNEAT, with AGENT providing superior results in terms of net reward value and usually smaller cost compared to NEAT. Results of AGENT was also comparable (slightly better) to Adaptive HyperNEAT on three out of four benchmark problems, and significantly better in the Lunar Lander problem. AGENT was also tested on a UAV-UAV collision avoidance problem, resulting in 33% improvement over random initial models in only 15 generations, with a population size of just 100 genomes. The resulting model was also found to generalize well over unseen scenarios.

We also studied the effect of the major new features in AGENT on its search process and performance. The insights gained therein points to the potential for translating AGENT's unique features, e.g., the controllers and the ability to both grow and shrink ANN topologies, to directly advance other neuroevolution algorithms, and is thus also an important direction of future work.

References

[1] A. I. Dounis and C. Caraiscos, "Advanced control systems engineering for energy and comfort management in a building environment—a review," *Renewable and Sustainable Energy Reviews*, vol. 13, no. 6-7, pp. 1246–1261, 2009.

[2] S. Sonoda and N. Murata, "Neural network with unbounded activation functions is universal approximator," *Applied and Computational Harmonic Analysis*, vol. 43, no. 2, pp. 233–268, 2017.

[3] J. Peters, S. Vijayakumar, and S. Schaal, "Reinforcement learning for humanoid robotics," in *Proceedings of the third IEEE-RAS international conference on humanoid robots*, 2003, pp. 1–20.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[5] W. Grathwohl, D. Choi, Y. Wu, G. Roeder, and D. Duvenaud, "Backpropagation through the void: Optimizing control variates for black-box gradient estimation," *arXiv preprint arXiv:1711.00123*, 2017.

[6] T. Poggio, H. Mhaskar, L. Rosasco, B. Miranda, and Q. Liao, "Why and when can deep-but not shallow-networks avoid the curse of dimensionality: a review," *International Journal of Automation and Computing*, vol. 14, no. 5, pp. 503–519, 2017.

[7] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: from architectures to learning," *Evolutionary Intelligence*, vol. 1, no. 1, pp. 47–62, 2008.

[8] S. Risi and J. Togelius, "Neuroevolution in games: State of the art and open challenges," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 9, no. 1, pp. 25–41, 2015.

[9] N. Hansen, S. D. Müller, and P. Koumoutsakos, "Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es)," *Evolutionary computation*, vol. 11, no. 1, pp. 1–18, 2003.

[10] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[11] K. O. Stanley, J. Clune, J. Lehman, and R. Miikkulainen, "Designing neural networks through neuroevolution," *Nature Machine Intelligence*, vol. 1, no. 1, pp. 24–35, 2019.

[12] E. Ronald and M. Schoenauer, "Genetic lander: An experiment in accurate neuro-genetic control," in *International Conference on Parallel Problem Solving from Nature*. Springer, 1994, pp. 452–461.

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

[14] A. J. Turner and J. F. Miller, "The importance of topology evolution in neuroevolution: a case study using cartesian genetic programming of artificial neural networks," in *International Conference on Innovative Techniques and Applications of Artificial Intelligence*. Springer, 2013, pp. 213–226.

[15] G. F. Miller, P. M. Todd, and S. U. Hegde, "Designing neural networks using genetic algorithms." in *ICGA*, vol. 89, 1989, pp. 379–384.

[16] J. D. Schaffer, R. A. Caruana, and L. J. Eshelman, "Using genetic search to exploit the emergent behavior of neural networks," *Physica D: Nonlinear Phenomena*, vol. 42, no. 1-3, pp. 244–248, 1990.

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

[18] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A hypercube-based encoding for evolving large-scale neural networks," *Artificial life*, vol. 15, no. 2, pp. 185–212, 2009.

[19] D. V. Vargas and J. Murata, "Spectrum-diverse neuroevolution with unified neural models," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 8, pp. 1759–1773, 2016.

[20] M. Hausknecht, P. Khandelwal, R. Miikkulainen, and P. Stone, "Hyperneat-ggp: A hyperneat-based atari general game player," in *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, 2012, pp. 217–224.

[21] J. Yosinski, J. Clune, D. Hidalgo, S. Nguyen, J. C. Zagal, and H. Lipson, "Evolving robot gaits in hardware: the hyperneat generative encoding vs. parameter optimization." in *ECAL*, 2011, pp. 890–897.

[22] G. Wang, G. Cheng, and T. R. Carr, "The application of improved neuroevolution of augmenting topologies neural network in marcellus shale lithofacies prediction," *Computers & geosciences*, vol. 54, pp. 50–65, 2013.

[23] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," *arXiv preprint arXiv:1712.06567*, 2017.

[24] K. O. Stanley, "Compositional pattern producing networks: A novel abstraction of development," *Genetic programming and evolvable machines*, vol. 8, no. 2, pp. 131–162, 2007.

[25] S. Risi and K. O. Stanley, "Indirectly encoding neural plasticity as a pattern of local rules," in *International Conference on Simulation of Adaptive Behavior*. Springer, 2010, pp. 533–543.

[26] S. Risi, J. Lehman, and K. O. Stanley, "Evolving the placement and density of neurons in the hyperneat substrate," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. ACM, 2010, pp. 563–570.

[27] S. Risi and K. O. Stanley, "Deep neuroevolution of recurrent and discrete world models," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019, pp. 456–462.

[28] J. Schmidhuber, D. Wierstra, and F. J. Gomez, "Evolino: Hybrid neuroevolution/optimal linear search for sequence prediction," in *Proceedings of the 19th International Joint Conferenceon Artificial Intelligence (IJCAI)*, 2005.

[29] D. Howard, A. E. Eiben, D. F. Kennedy, J.-B. Mouret, P. Valencia, and D. Winkler, "Evolving embodied intelligence from materials to machines," *Nature Machine Intelligence*, vol. 1, no. 1, pp. 12–19, 2019.

[30] A. Behjat, C. Zeng, K. K. Gabani, and S. Chowdhury, "Concurrent morphology-optimization and behavior-learning:co-designing intelligent quadcopters," in *AIAA Aviation 2020 Forum*, 2020.

[31] T. Weise, M. Zapf, R. Chiong, and A. J. Nebro, "Why is optimization difficult?" in *Nature-inspired algorithms for optimisation*. Springer, 2009, pp. 1–50.

[32] H. M. Pandey, A. Chaudhary, and D. Mehrotra, "A comparative review of approaches to prevent premature convergence in ga," *Applied Soft Computing*, vol. 24, pp. 1047–1077, 2014.

[33] M. de Wet, "Avoiding premature convergence in neuroevolution by broadening the evolutionary search," Department of Computer Science, The University of Texas at Austin, Undergraduate Honors Thesis HR-11-02, 2011. [Online]. Available: http://www.cs.utexas.edu/users/ai-lab?dewet:ugthesis11

[34] J. Lehman and K. O. Stanley, "Abandoning objectives: Evolution through the search for novelty alone," *Evolutionary computation*, vol. 19, no. 2, pp. 189–223, 2011.

[35] D. Gravina, A. Liapis, and G. Yannakakis, "Surprise search: Beyond objectives and novelty," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. ACM, 2016, pp. 677–684.

[36] K. Kawaguchi, L. P. Kaelbling, and Y. Bengio, "Generalization in deep learning," *arXiv preprint arXiv:1710.05468*, 2017.

[37] I. Vlašić, M. ðurasević, and D. Jakobović, "Improving genetic algorithm performance by population initialisation with dispatching rules," *Computers & Industrial Engineering*, vol. 137, p. 106030, 2019.

[38] S. Chowdhury, W. Tong, A. Messac, and J. Zhang, "A mixed-discrete particle swarm optimization algorithm with explicit diversity-preservation," *Structural and Multidisciplinary Optimization*, vol. 47, no. 3, pp. 367–388, 2013.

[39] T. G. Lewis, *Network science: Theory and applications*. John Wiley & Sons, 2011.

[40] Y. Wu, P. R. Bhat, T. J. Close, and S. Lonardi, "Efficient and accurate construction of genetic linkage maps from the minimum spanning tree of a graph," *PLoS Genet*, vol. 4, no. 10, p. e1000212, 2008.

[41] S. Haflidason and R. Neville, "On the significance of the permutation problem in neuroevolution," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, 2009, pp. 787–794.

[42] K. O. Stanley and R. Miikkulainen, "Efficient reinforcement learning through evolving neural network topologies," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc., 2002, pp. 569–577.

[43] A. Behjat, S. Paul, and S. Chowdhury, "Learning reciprocal actions for cooperative collision avoidance in quadrotor unmanned aerial vehicles," *Robotics and Autonomous Systems*, vol. 121, p. 103270, 2019.

[44] M. Dorofki, A. H. Elshafie, O. Jaafar, O. A. Karim, and S. Mastura, "Comparison of artificial neural network transfer functions abilities to simulate extreme runoff data," *International Proceedings of Chemical, Biological and Environmental Engineering*, vol. 33, pp. 39–44, 2012.

[45] D. Stathakis, "How many hidden layers and nodes?" *International Journal of Remote Sensing*, vol. 30, no. 8, pp. 2133–2147, 2009.

[46] P. J. Hancock, "Selection methods for evolutionary algorithms," in *Practical Handbook of Genetic Algorithms*. CRC Press, 2019, pp. 67–92.

[47] G. Howard, E. Gale, L. Bull, B. de Lacy Costello, and A. Adamatzky, "Evolution of plastic learning in spiking networks via memristive connections," *IEEE Transactions on Evolutionary Computation*, vol. 16, no. 5, pp. 711–729, 2012.

[48] K. Deb, *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, 2001, vol. 16.

[49] W. Huang, Y. Li, and Y. Huang, "Deep hybrid neural network and improved differential neuroevolution for chaotic time series prediction," *IEEE Access*, vol. 8, pp. 159 552–159 565, 2020.

[50] M. Li, J. Zheng, and J. Wu, "Improving nsga-ii algorithm based on minimum spanning tree," in *Asia-Pacific Conference on Simulated Evolution and Learning*. Springer, 2008, pp. 170–179.

[51] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[52] C. Ye, R. C. Wilson, C. H. Comin, L. d. F. Costa, and E. R. Hancock, "Approximate von neumann entropy for directed graphs," *Physical Review E*, vol. 89, no. 5, p. 052804, 2014.

[53] G. Minello, L. Rossi, and A. Torsello, "On the von neumann entropy of graphs," *Journal of Complex Networks*, vol. 7, no. 4, pp. 491–514, 2019.

[54] E. Hancock, "Lecture notes in network entropy," July 2016.

[55] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017.

[56] D. Oller, T. Glasmachers, and G. Cuccu, "Analyzing reinforcement learning benchmarks with random weight guessing," *arXiv preprint arXiv:2004.07707*, 2020.

[57] R. Liu and J. Zou, "The effects of memory replay in reinforcement learning," in *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2018, pp. 478–485.

[58] K. Asadi and M. L. Littman, "An alternative softmax operator for reinforcement learning," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 243–252.

[59] S. Zhang and O. R. Zaiane, "Comparing deep reinforcement learning and evolutionary methods in continuous control," *arXiv preprint arXiv:1712.00006*, 2017.

[60] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[61] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*. PMLR, 2015, pp. 1889–1897.

[62] G. Shaw, "Donald hebb: The organization of behavior," in *Brain Theory*. Springer, 1986, pp. 231–233.

[63] A. Behjat, K. K. Gabani, and S. Chowdhury, "Training detection-range-frugal cooperative collision avoidance models for quadcopters via neuroevolution," in *AIAA Aviation 2019 Forum*, 2019, p. 3312.

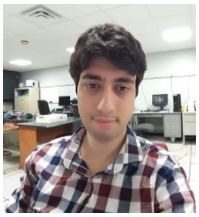
[64] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise reduction in speech processing*. Springer, 2009, pp. 1–4.

[65] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 2520–2525.

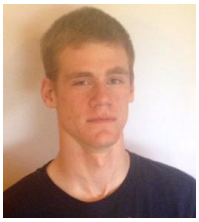
[66] T. Luukkonen, "Modelling and control of quadcopter," *Independent research project in applied mathematics, Espoo*, vol. 22, 2011.

[67] A. Moore, "Efficient memory-based learning for robot control," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, March 1991.

[68] A. Geramifard, C. Dann, R. H. Klein, W. Dabney, and J. P. How, "Rlpy: A value-function-based reinforcement learning framework for education and research," *Journal of Machine Learning Research*, vol. 16, no. 46, pp. 1573–1578, 2015. [Online]. Available: http://jmlr.org/papers/v16/geramifard15a.html



Amir Behjat is a Post-Doctoral researcher at Purdue university. He received his Ph.D. in Mechanical Engineering from University at Buffalo in 2021. His contributions to this paper occurred during his Ph.D. He graduated with his BS and MS degrees in Mechanical engineering from Sharif University of Technology. His research focuses on Neuroevolution, Physics-Aware machine learning, and UAV collision avoidance.



Nathan Maurer is an M.S. in Robotics student at University at Buffalo. He received his B.S. in Computer Engineering from University at Buffalo. His contributions to this paper is related to his ongoing M.S. Thesis research being performed under the supervision of Dr. Souma Chowdhury. His research interests include neuroevolution, neuromorphic computing and graph learning.

Sharat Chidambaran received his M.S. in Mechanical Engineering from University at Buffalo. His contributions to this paper occurred during his M.S. Thesis performed under the supervision of Dr. Souma Chowdhury at University at Buffalo. His research interests include autonomous systems, machine learning and data driven engineering.

Souma Chowdhury is an Associate Professor of Mechanical and Aerospace Engineering at University at Buffalo. Dr. Chowdhury received his B.S., M.S. and Ph.D. in Mechanical Engineering, respectively from IIT Kharagpur in India, Florida International University in Miami and Rensselaer Polytechnic Institute in Troy. His research interests lie at the intersection of multi-fidelity optimization and machine learning with applications to the design and control of autonomous systems, swarm robotics and energy systems. He has co-authored 150 articles in leading journals and full-length conference proceedings in related areas. His research has been sponsored by the NSF, DARPA, ONR, NASA and AFOSR.