

TensorNEAT: A GPU-accelerated Library for NeuroEvolution of Augmenting Topologies

LISHUANG WANG, Southern University of Science and Technology, China

MENGFEI ZHAO, Southern University of Science and Technology, China

ENYU LIU, Southern University of Science and Technology, China

KEBIN SUN, Southern University of Science and Technology, China

RAN CHENG*, The Hong Kong Polytechnic University, China

The NeuroEvolution of Augmenting Topologies (NEAT) algorithm has received considerable recognition in the field of neuroevolution. Its effectiveness is derived from initiating with simple networks and incrementally evolving both their topologies and weights. Although its capability across various challenges is evident, the algorithm's computational efficiency remains an impediment, limiting its scalability potential. To address these limitations, this paper introduces TensorNEAT, a GPU-accelerated library that applies tensorization to the NEAT algorithm. Tensorization reformulates NEAT's diverse network topologies and operations into uniformly shaped tensors, enabling efficient parallel execution across entire populations. TensorNEAT is built upon JAX, leveraging automatic function vectorization and hardware acceleration to significantly enhance computational efficiency. In addition to NEAT, the library supports variants such as CPPN and HyperNEAT, and integrates with benchmark environments like Gym, Brax, and gymnas. Experimental evaluations across various robotic control environments in Brax demonstrate that TensorNEAT delivers up to 500x speedups compared to existing implementations, such as NEAT-Python. The source code for TensorNEAT is publicly available at: <https://github.com/EMI-Group/tensorneat>.

CCS Concepts: • Theory of computation → Evolutionary algorithms; Vector / streaming algorithms.

Additional Key Words and Phrases: Neuroevolution, GPU Acceleration, Algorithm Library

ACM Reference Format:

Lishuang Wang, Mengfei Zhao, Enyu Liu, Kebin Sun, and Ran Cheng. 2024. TensorNEAT: A GPU-accelerated Library for NeuroEvolution of Augmenting Topologies. 1, 1 (April 2024), 33 pages. <https://doi.org/XXXXXXX>. XXXXXXX

1 INTRODUCTION

Neuroevolution has emerged as a distinct branch within the field of artificial intelligence (AI). Unlike the common approach in machine learning that uses stochastic gradient descent, neuroevolution employs evolutionary algorithms for network optimization. This method not only optimizes parameters but also improves more complex aspects such as activation functions, hyperparameters,

*Corresponding Author

Authors' addresses: Lishuang Wang, wanglishuang22@gmail.com, Southern University of Science and Technology, Shenzhen, Guangdong, China, 518055; Mengfei Zhao, 12211819@mail.sustech.edu.cn, Southern University of Science and Technology, Shenzhen, Guangdong, China, 518055; Enyu Liu, 12210417@mail.sustech.edu.cn, Southern University of Science and Technology, Shenzhen, Guangdong, China, 518055; Kebin Sun, sunkebin.cn@gmail.com, Southern University of Science and Technology, Shenzhen, Guangdong, China, 518055; Ran Cheng, ranchengcn@gmail.com, The Hong Kong Polytechnic University, Hong Kong SAR, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

XXXX-XXXX/2024/4-ART \$15.00

<https://doi.org/XXXXXXX.XXXXXXX>

and the overall network architecture. Furthermore, neuroevolution offers a distinct advantage over traditional machine learning methods, which often converge to a single solution by relying on gradient-based techniques; instead, neuroevolution maintains a diverse population of solutions throughout its search process, allowing it to explore a broader range of possibilities [Stanley et al. 2019]. This characteristic of sustaining a variety of potential solutions makes neuroevolution particularly well-suited for problems with non-stationary environments or open-ended objectives, where the optimal solution may change over time or where there may be multiple viable solutions [Lehman and Stanley 2011; Mouret and Clune 2015]. These qualities not only emphasize neuroevolution's capacity for exploration but also highlight its potential for creative problem solving and its ability to tackle challenges that require adaptability and long-term innovation.

The NeuroEvolution of Augmenting Topologies (NEAT) [Stanley and Miikkulainen 2002] is well-recognized in the neuroevolution literature, particularly for its pioneering approach to evolving neural networks with both topological and weight diversity. Since its introduction in 2002, NEAT has demonstrated versatility across a wide range of domains, including game AI [Pham et al. 2018; Stanley et al. 2006], where it has been employed for evolving adaptive agents, robotics [Auerbach and Bongard 2011; Silva et al. 2012], enabling the design of efficient control strategies, and self-driving systems [Yuksel 2018], contributing to advancements in autonomous navigation. The original NEAT algorithm provided a solid foundation, and subsequent research has continued to expand its capabilities, leveraging its core principles to explore new areas. For example, extensions like HyperNEAT [Stanley et al. 2009] and ES-HyperNEAT [Risi et al. 2010] adopted indirect encodings to evolve large-scale networks with regular patterns, while DeepNEAT and CoDeepNEAT [Miikkulainen et al. 2019] integrated gradient-based methods to explore deeper neural architectures for more complex tasks. Recently, the RankNEAT algorithm [Pinitas et al. 2022] has applied NEAT principles to preference learning, where it optimizes networks based on subjective data, expanding NEAT's application into more nuanced, human-centered decision-making problems. The continued development of these variations not only underscores NEAT's enduring relevance but also showcases its adaptability in addressing diverse and evolving challenges within neuroevolution, making it a cornerstone in the field's ongoing innovation.

Over the past decade, the role of GPU acceleration has been a key factor in driving the rapid progress of artificial intelligence, particularly within deep learning. As models become increasingly complex, requiring the processing of hundreds of billions of parameters, such as in the case of modern large language models [Brown et al. 2020], hardware acceleration has proven crucial for handling the associated computational demands. GPUs, when employed for tasks like inference and back-propagation, have not only contributed to substantial reductions in training time but also enabled more scalable model architectures that can learn from vast datasets. Similarly, in the neuroevolution domain, there has been a growing effort to leverage the power of GPUs to achieve similar performance improvements. The use of frameworks like JAX [Frostig et al. 2018] has spurred developments in GPU-accelerated neuroevolution algorithms, with notable contributions from projects such as EvoJAX [Tang et al. 2022], evosax [Lange 2023], and EvoX [Huang et al. 2024]. These works aim to exploit the parallel processing capabilities of GPUs to significantly reduce the computational time required for evolutionary algorithms, which is particularly advantageous when dealing with large-scale problems or managing sizable populations. As a result, GPU acceleration is becoming increasingly integrated into neuroevolution workflows, paving the way for more efficient and scalable solutions in both research and practical applications.

Despite the rapid emergence of GPU-accelerated libraries across various machine learning and evolutionary computation frameworks, NEAT has remained largely underdeveloped in this regard. Our analysis revealed that while the NEAT algorithm has been implemented in several programming languages [b2developer 2022; McIntyre et al. 2023; peter ch 2019], very few of these implementations

leverage the computational power of GPUs to enhance performance. On the rare occasions where GPUs are utilized [Gajewsky 2023], the focus has primarily been on accelerating only the network inference aspect of NEAT, neglecting other vital components such as the evolutionary search and network mutation processes, which are critical to the algorithm’s success. Moreover, even in these GPU-accelerated implementations, the potential of parallel computation is not fully realized, as essential operations like fitness evaluation and mutation are still executed sequentially, preventing the framework from harnessing the full power of modern GPUs. This underutilization can be attributed to the inherent complexity of NEAT, which continuously evolves network topologies throughout its execution. This dynamic and unpredictable nature of topology evolution poses significant challenges for an efficient and scalable GPU implementation, as it complicates the parallelization of key operations that are essential for NEAT’s performance.

To address the performance limitations of traditional NEAT implementations, we introduce TensorNEAT, a GPU-accelerated library of the NEAT algorithm. TensorNEAT leverages a novel tensorization strategy that converts networks of diverse topologies into uniformly shaped tensors, enabling parallel execution of operations across the entire population. This approach ensures that the algorithm’s performance scales efficiently with the complexity of the task and the size of the population. By utilizing the JAX framework, TensorNEAT automatically benefits from GPU acceleration without requiring manual configuration or specialized hardware knowledge. When compared to widely-used open-source NEAT implementations, TensorNEAT demonstrates speedups of up to 500x, significantly reducing the time required for evolving neural networks. Overall, our contributions are summarized as follows.

- We propose a tensorization method, which enables the transformation of networks with various topologies and their associated operations in the NEAT algorithm into uniformly structured tensors for tensor computation. This method allows operations within the NEAT algorithm to be executed in parallel across the entire population, thereby enhancing the efficiency of the process.
- We develop TensorNEAT, a GPU-accelerated NEAT library based on JAX, characterized by high efficiency, flexible adaptability, and rich capabilities. TensorNEAT supports full GPU acceleration of representative NEAT algorithms, including the original NEAT algorithm, CPPN [Stanley 2007], and HyperNEAT [Stanley et al. 2009]. It also provides seamless interfaces with advanced control benchmarks, including Brax [Freeman et al. 2021] and Gymmax [Lange 2022], featuring GPU-accelerated environments with various classical control and robotics control tasks.
- We assessed TensorNEAT’s performance in a spectrum of complex robotics control tasks, benchmarked against the NEAT-Python library [McIntyre et al. 2023]. The results show that TensorNEAT significantly outperforms in terms of execution speed, especially under high computational demands and across various population sizes and network scales.

This paper is an extension of its prior conference version [Wang et al. 2024]. In particular, (a) we expanded the Implementation of TensorNEAT section, providing a more detailed description of the TensorNEAT library, including the introduction of its visualization capabilities; (b) we introduced multi-GPU support for TensorNEAT, and conducted experiments to evaluate the performance benefits of this feature; (c) we restructured the paper’s content and organization, including a title change, the enrichment of various sections, and improvements to the figures and visuals throughout the manuscript; and (d) we supplemented the experimental evaluation by testing TensorNEAT’s performance under different population size settings and comparing it with other GPU-accelerated EC algorithm libraries.

Algorithm 1 Main Process of the NEAT algorithm

Require: P (population size), I (number of input nodes), O (number of output nodes), f_{target} (target fitness value), G (maximum number of generations)

Ensure: best

```

 $pop \leftarrow \text{initialize } P \text{ networks with } I, O$ 
for  $g = 1$  to  $G$  do
     $fit \leftarrow \text{evaluate fitness values of } Pop$ 
    if  $\max(fit) \geq f_{\text{target}}$  then
        break
    end if
     $species \leftarrow \text{divide } pop \text{ by distances between networks}$ 
     $pop^* \leftarrow \{\}$ 
    for  $s$  in  $species$  do
         $c \leftarrow \text{determine the number of new individuals by } fit$ 
         $s \leftarrow \text{generate } c \text{ networks using crossover and mutation}$ 
         $pop^* \leftarrow pop^* \cup s$ 
    end for
     $pop \leftarrow pop^*$ 
end for
return  $pop[\arg \max(fit)]$ 

```

2 BACKGROUND

2.1 NeuroEvolution of Augmenting Topologies

Introduced by Kenneth O. Stanley and Risto Miikkulainen in 2002, the NeuroEvolution of Augmenting Topologies (NEAT) algorithm [Stanley and Miikkulainen 2002] represents a novel approach in neuroevolution. The NEAT algorithm manages a range of neural networks and simultaneously optimizes their topologies and weights to identify the most effective networks tailored for designated tasks. Algorithm 1 outlines NEAT's core procedure. Starting with a population of simple neural networks, it iterates through evolutionary cycles of species formation, fitness evaluation, and genetic operations. This process dynamically refines the networks until it achieves desired fitness levels or reaches a generational cap, ultimately yielding the optimal network structure.

Setting itself apart from alternative neuroevolution algorithms, NEAT employs three distinctive techniques:

- **Incremental Topological Expansion:** The NEAT algorithm begins its evolutionary process by initializing networks with a minimal configuration, typically consisting of just a single hidden node connecting inputs to outputs. This minimalist starting point is crucial for reducing complexity at the outset, allowing the algorithm to explore the problem space with simpler models before gradually introducing complexity. As evolution proceeds, the algorithm systematically adds new nodes and connections, incrementally expanding the network's topology. This controlled growth not only helps in maintaining a manageable search space but also allows the network to evolve in a more directed manner, progressively building toward more intricate and capable structures. By expanding the topology in stages, NEAT enables a balance between exploration and exploitation, ensuring that simpler network architectures are initially favored while providing the flexibility to evolve into more sophisticated configurations as required. This incremental approach proves especially effective for

solving complex problems, as it starts with streamlined architectures and organically grows into larger, more specialized networks as the evolutionary process advances.

- **Historical Markers for Nodes:** Each node in a NEAT network is tagged with a unique historical marker. During the crossover process in the NEAT algorithm, only nodes with identical markers are combined. This method adeptly navigates the challenges of combining networks that possess different topological configurations. By using historical markers, NEAT preserves important structural innovations, ensuring that new nodes and connections are not arbitrarily lost during evolution. This approach promotes stable and consistent network evolution, even as topologies grow more complex.
- **Species-based Population Segmentation:** NEAT categorizes its entire population into species based on genetic similarity, ensuring that networks with common traits evolve together. Conventional genetic procedures, such as selection, mutation, and crossover, are executed independently within each species. This approach serves multiple purposes: it protects promising, newly-formed network structures from being prematurely eliminated by more established competitors, allowing them time to develop and improve. Additionally, species-based segmentation fosters diversity by enabling different topological variations to evolve in parallel, increasing the variety of solutions explored during evolution. This helps prevent premature convergence and promotes a broader exploration of the search space, improving the chances of discovering optimal network configurations.

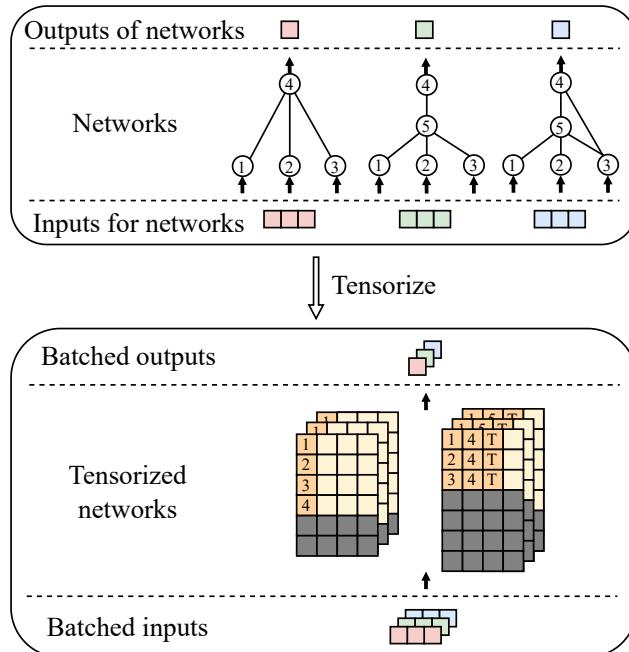


Fig. 1. Illustration of the fundamental acceleration principle underlying our tensorization method. In traditional network computations, each network individually processes its inputs. By contrast, with our tensorization method, a single computation suffices to derive the batched output for all networks.

2.2 NEAT Variants

Since its inception, the NEAT algorithm has undergone significant evolution, leading to the emergence of several innovative variants, each addressing unique challenges and applications. One of the most notable advancements was the introduction of HyperNEAT [Stanley et al. 2009], which employed an indirect encoding scheme to enable the efficient generation and handling of large-scale neural networks. By leveraging the inherent geometric regularities found in the problem space, HyperNEAT was able to create intricate network topologies suited for tasks requiring high representational capacity. Building upon this foundation, ES-HyperNEAT [Risi et al. 2010] extended the principles of HyperNEAT by incorporating more sophisticated techniques for evolving network structures. ES-HyperNEAT not only retained the scalability benefits of its predecessor but also introduced mechanisms that improved adaptability to complex, high-dimensional problem spaces, thereby broadening the range of tasks for which neuroevolution could be effectively applied. This progression reflects the continuous effort within the neuroevolution community to push the boundaries of what can be achieved with evolving neural architectures, as each variant strives to address the increasing demands of modern applications.

The development of DeepNEAT and CoDeepNEAT [Miikkulainen et al. 2019] marked a pivotal advance in integrating neuroevolution with deep learning methodologies, significantly broadening the scope and capabilities of these systems. DeepNEAT took the foundational principles of the NEAT algorithm and extended them into the domain of deep neural networks, allowing for the evolution of more complex, layered architectures that could be fine-tuned using gradient descent. This integration facilitated the exploration of both structure and parameter optimization, giving rise to more sophisticated models that could adapt to a wide range of tasks. CoDeepNEAT further evolved this idea by introducing a co-evolutionary framework, which enabled the simultaneous optimization of both network topologies and their components, such as activation functions or layer types. This dual optimization approach not only increased the flexibility of the models but also enhanced their performance in tasks requiring hierarchical learning and multi-level feature extraction. The combination of evolutionary strategies with gradient-based methods proved to be a powerful tool for discovering novel neural architectures, pushing the boundaries of what was achievable in terms of both accuracy and efficiency in deep learning, and opening new possibilities for more complex, adaptive systems in areas like automated feature learning and the construction of modular, scalable networks.

RankNEAT [Pinitas et al. 2022] is another variant of the NEAT algorithm that leverages neuroevolution to overcome the limitations typically associated with Stochastic Gradient Descent (SGD), particularly in terms of preventing overfitting during network optimization. Unlike SGD, which can struggle with local minima and the complexities of non-convex loss landscapes, RankNEAT explores a broader search space by evolving architectures rather than relying solely on gradient-based updates. This approach has proven highly effective in affective computing, outperforming traditional ranking algorithms such as RankNet, particularly in predicting annotated player arousal from game footage across three diverse games. RankNEAT's success is particularly evident in tasks involving the analysis of subjective data, such as assessing player arousal from game footage. Its ability to better capture and interpret the nuances of emotional responses makes it a powerful tool in this domain, demonstrating its potential for broader applications where subjective, human-centered data needs to be processed in an adaptive and efficient manner. Furthermore, its robustness in handling non-linear and noisy data suggests that RankNEAT could be adapted for other complex domains, further expanding its utility beyond just affective computing.

Together, these variants of NEAT demonstrate the algorithm's versatility and its ability to continuously adapt to the growing demands of neural network design, application, and optimization.

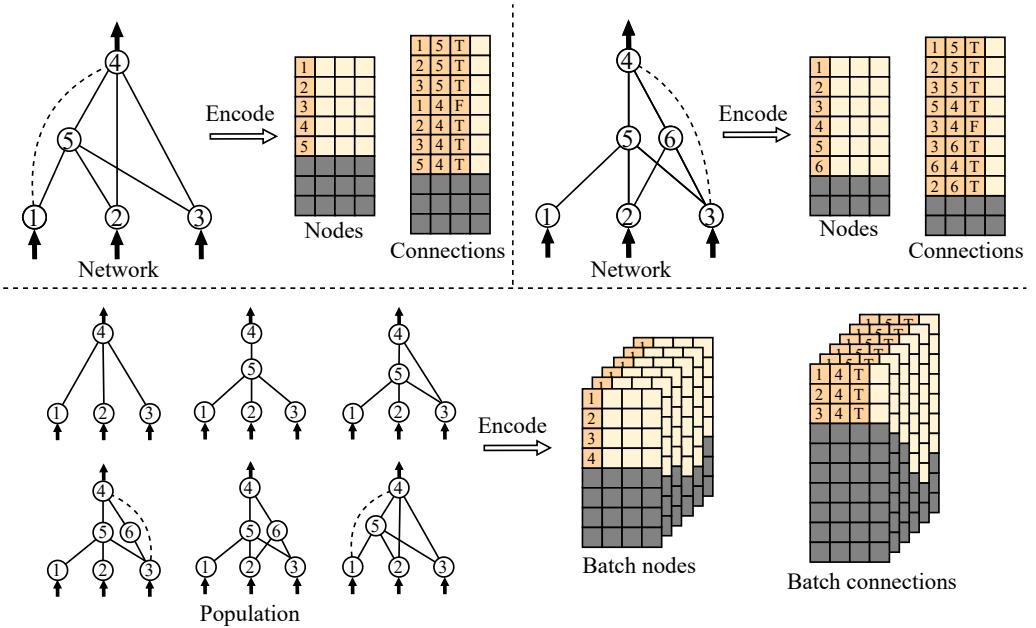


Fig. 2. Illustration of the network encoding process. In our method, networks with varying topological structures are transformed into uniformly shaped tensors, enabling the representation of the entire network population as batched tensors. The orange grids symbolize attributes specific to the NEAT algorithm, such as historical markers and enabled flags. The yellow grids denote the attributes of the network's nodes and connections, including biases, weights, and activation functions. The gray grids represent sections filled with NaN to ensure consistent tensor shapes.

By supporting diverse architectures and a range of problem domains, NEAT has proven effective in evolving both small-scale and large-scale neural networks with varying degrees of complexity.

2.3 NEAT Libraries

Over the past two decades, the research community has seen the development of various NEAT libraries, including NEAT-Python [McIntyre et al. 2023], MultiNEAT [peter ch 2019], as well as MonopolyNEAT [b2developer 2022], each contributing to the growing interest in evolving neural networks. Among these, NEAT-Python [McIntyre et al. 2023] has emerged as the most widely recognized open-source implementation of NEAT, boasting over 1,200 GitHub stars and serving as the foundation for numerous academic studies and practical applications of the NEAT algorithm [Gao and Lan 2021; Pinitas et al. 2022; Sarti and Ochoa 2021]. The popularity of NEAT-Python is largely due to its accessibility, ease of use, and robust documentation, which have allowed researchers and developers to build upon its codebase for a wide range of evolutionary computing experiments.

Like many NEAT implementations, NEAT-Python leverages object-oriented programming (OOP) paradigm, where key components such as populations, species, genomes, and genes are encapsulated as objects. This OOP approach fosters code modularity and clarity, making it easier for newcomers to understand the underlying mechanisms of NEAT, while also simplifying debugging and extension of the code. However, this method of representing NEAT components as objects introduces additional memory and computational overhead, particularly when scaling to large population sizes or more

complex problem domains. The need to maintain and manipulate large numbers of objects can significantly increase processing time, limiting the efficiency of these libraries in high-performance scenarios, such as those requiring rapid evolution over multiple generations or when running on resource-constrained hardware.

Most existing NEAT implementations do not utilize GPUs to accelerate computation, with a few exceptions such as PyTorchNEAT [Gajewsky 2023]. These libraries integrate tensor-based deep learning frameworks like PyTorch [Paszke et al. 2019] and TensorFlow [Abadi et al. 2015], allowing networks generated by NEAT to perform inference on GPUs. This integration enables batch inference, which improves network inference speed, particularly when handling a large volume of input data. However, these libraries primarily focus on optimizing the inference aspect of the networks and still rely heavily on the object-oriented programming paradigm. While they offer some degree of performance gain in terms of network inference, the neuroevolution process—where NEAT evolves networks through mutation and crossover—remains unoptimized for GPU execution. Specifically, the computationally intensive process of evolving a population of neural networks has not been fully parallelized. Furthermore, although batch processing allows individual networks to leverage GPU capabilities, the evaluation of the entire population in NEAT, which is central to the algorithm’s evolutionary process, continues to be executed sequentially. As a result, current NEAT implementations fail to fully exploit the high parallel processing potential of modern GPUs, leading to inefficiencies in both the search and evolutionary phases of the algorithm.

3 TENSORIZATION

To overcome the limitations of current NEAT implementations and fully harness modern hardware for improved efficiency, we introduce a novel tensorization approach. As shown in Fig. 1, this method accelerates the NEAT algorithm by converting various network topologies and their operations into structured tensors, which are well-suited for GPU-based computations. By vectorizing functions within network operations, our approach enables parallel processing across the entire population of networks, allowing GPUs to be used more effectively. The following sections detail the specific tensorization techniques applied to network encoding and operations. Table 1 presents the symbols used in this paper.

3.1 Tensorized Encodings

In the NEAT algorithm, a network \mathcal{N} can be represented as:

$$\mathcal{N} = \langle N, C \rangle,$$

where N, C are nodes and connections in \mathcal{N} , respectively. They can be represented as:

$$N = \{n_1, n_2, n_3, \dots\} \quad \text{and} \quad C = \{c_1, c_2, c_3, \dots\},$$

with n_i denoting the i -th node in N and c_i denoting the i -th connection in C .

In the NEAT algorithm, each node n can be represented as a tuple consisting of a historical marker and its attributes:

$$n = (k, \text{attr}_1, \text{attr}_2, \dots),$$

where $k \in \mathbb{N}$ is the historical marking, and attr_i is the i -th attribute of n . Similarly, a connection c can be represented as:

$$c = (k_i, k_o, e, \text{attr}_1, \text{attr}_2, \dots),$$

where $k_i \in \mathbb{N}$ and $k_o \in \mathbb{N}$ are the historical markings of the input and output nodes of c , respectively, $e \in \{\text{True}, \text{False}\}$ is the enabled flag. The attributes of a node n can be a bias b , an aggregation function f_{agg} (e.g., sum), and an activation function f_{act} (e.g., sigmoid), while the attributes of a connection c can be a weight w .

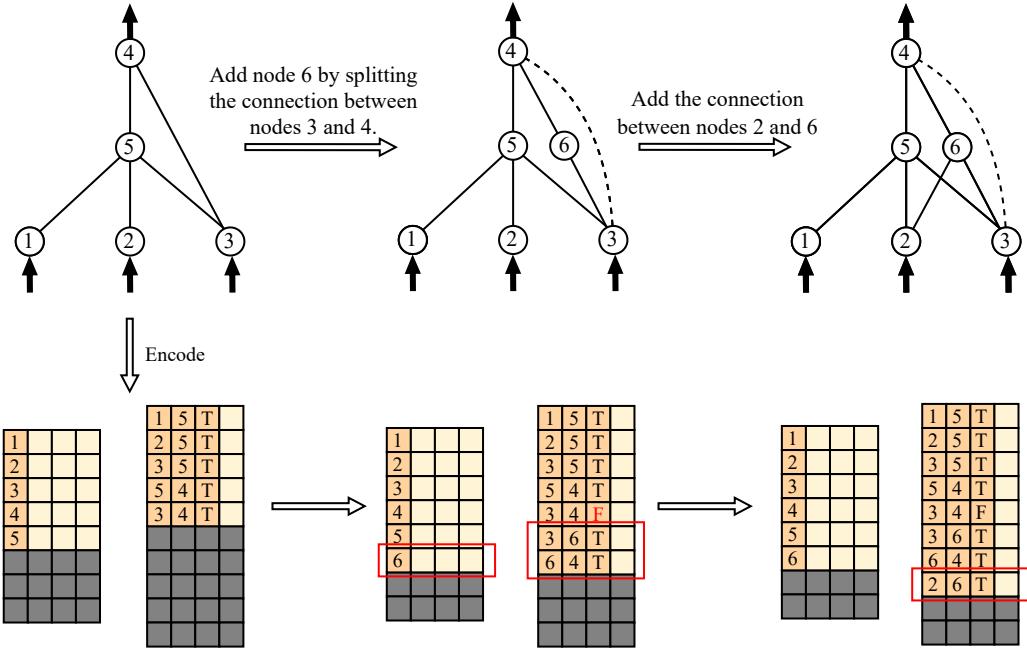


Fig. 3. An illustration of tensorized network operations. It demonstrates how traditional network operations are converted into equivalent tensor operations during tensorization. Changes from the original format are highlighted in red, underscoring the modifications made within the tensor.

We can use one-dimensional tensors to encode n and c :

$$\begin{aligned} \mathbf{n} &= [k, \text{attr}_1, \text{attr}_2, \dots] \in \mathbb{R}^{1+\text{noa}(n)}, \\ \mathbf{c} &= [k_i, k_o, e, \text{attr}_1, \text{attr}_2, \dots] \in \mathbb{R}^{3+\text{noa}(c)}, \end{aligned}$$

where $\text{noa}(i)$ denotes the number of attributes of i .

Then, the sets N and C can be represented as tensors \mathbf{N} and \mathbf{C} , respectively:

$$\begin{aligned} \mathbf{N} &= [\mathbf{n}_1, \mathbf{n}_2, \dots] \in \mathbb{R}^{|N| \times \text{len}(\mathbf{n})}, \\ \mathbf{C} &= [\mathbf{c}_1, \mathbf{c}_2, \dots] \in \mathbb{R}^{|C| \times \text{len}(\mathbf{c})}, \end{aligned}$$

where $|\cdot|$ is the cardinality of a set, $\text{len}(\cdot)$ is the length of a one-dimensional tensor, and \mathbf{n}_i and \mathbf{c}_j are the tensor representations of the i -th node and j -th connection, respectively.

To address the issue where each network in the population possesses a varying number of nodes and connections, we employ tensor padding using NaN values for alignment. We set predefined maximum limits for the number of nodes and connections, represented as $|N|_{\max}$ and $|C|_{\max}$, respectively. The resulting padded tensors, $\hat{\mathbf{N}}$ and $\hat{\mathbf{C}}$, are formulated as:

$$\begin{aligned} \hat{\mathbf{N}} &= [\mathbf{n}_1, \mathbf{n}_2, \dots, \text{NaN}, \dots] \in \mathbb{R}^{|N|_{\max} \times \text{len}(\mathbf{n})}, \\ \hat{\mathbf{C}} &= [\mathbf{c}_1, \mathbf{c}_2, \dots, \text{NaN}, \dots] \in \mathbb{R}^{|C|_{\max} \times \text{len}(\mathbf{c})}. \end{aligned}$$

Upon alignment, the tensors $\hat{\mathbf{N}}$ and $\hat{\mathbf{C}}$ of each network in the population can be concatenated, allowing us to express the entire population using two tensors: \mathbf{P}_N and \mathbf{P}_C . These tensors encompass

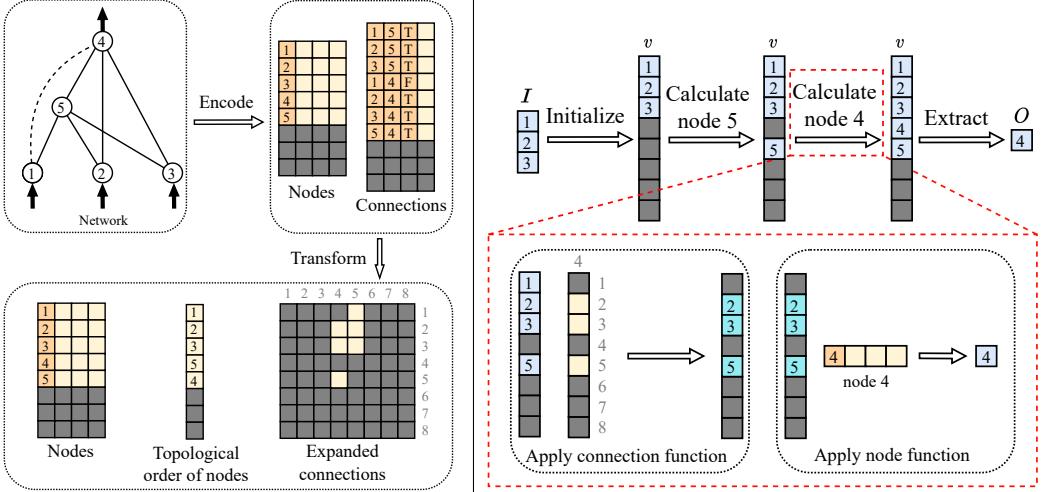


Fig. 4. Illustration of tensorized network inference process. It transforms a feedforward network’s topology into tensors and the subsequent calculation of node values for network inference. The network is first encoded into node and connection tensors, which are then ordered and expanded for processing. Finally, values are calculated through connection and node functions to produce the output.

all nodes and connections within the population, respectively. Formally, the concatenated tensors, P_N and P_C , are defined as:

$$P_N = [\hat{N}_1, \hat{N}_2, \dots] \in \mathbb{R}^{P \times |N|_{\max} \times \text{len}(n)},$$

$$P_C = [\hat{C}_1, \hat{C}_2, \dots] \in \mathbb{R}^{P \times |C|_{\max} \times \text{len}(c)},$$

where P denotes the population size, \hat{N}_i and \hat{C}_i represent the tensor representations of the node and connection sets of the i -th network, respectively. Fig. 2 provides a graphical representation of the tensorized encoding process.

3.2 Tensorized Operations

Upon encoding networks as tensors, we subsequently express operations on NEAT networks as corresponding tensor operations. In this subsection, we detail the tensorized representations of three fundamental operations: node modification, connection modification, and attribute modification.

3.2.1 Node Modification. Given a network $\mathcal{N} = \langle N, C \rangle$, node set modifications in N can involve either the addition of a new node n or the removal of an existing node n :

$$N' = N \cup \{n\} \quad \text{or} \quad N' = N \setminus \{n\}.$$

In the tensorized representation $\hat{N} \in \mathbb{R}^{|N|_{\max} \times \text{len}(n)}$ of the node set N , the tensorized operation for node addition can be represented as:

$$\hat{N}[r_i] \leftarrow n_{\text{new}},$$

where r_i denotes the index of the first NaN row in \hat{N} , n_{new} stands for the tensor representation of the node being added, $[\cdot]$ represents tensor slicing, and \leftarrow indicates the assignment operation.

Conversely, the tensorized operation for node removal can be depicted as:

$$\hat{N}[r_j] \leftarrow \text{NaN},$$

with r_j representing the index of the node for removal.

3.2.2 Connection Modification. Given a network $\mathcal{N} = \langle N, C \rangle$, modifications in the connection set C can entail either adding a new connection or eliminating an existing connection c:

$$C' = C \cup \{c\} \quad \text{or} \quad C' = C \setminus \{c\}.$$

In the tensorized representation, \hat{C} , of C , the operations of connection addition or removal can be depicted as:

$$\hat{C}[r_i] \leftarrow c_{\text{new}} \quad \text{or} \quad \hat{C}[r_j] \leftarrow \text{NaN},$$

respectively. Here, r_i indicates the index of the initial NaN row in \hat{C} , c_{new} represents the tensor form of the connection being introduced, and r_j signifies the index of the connection for removal.

3.2.3 Attribute Modification. NEAT is designed not only to modify the network structures but also the internal attributes, either in a node or a connection. Specifically, given a node $n = (k, \text{attr}_1, \text{attr}_2, \dots)$, where $k \in \mathbb{N}$ is the historical marking, and attr_i is the i -th attribute of n , when modifying the j -th attribute in a node, the transformation can be represented as:

$$n' = (k, \text{attr}_1, \text{attr}_2, \dots, \text{attr}'_j, \dots),$$

Table 1. Symbols and Notations

Symbol	Explanation
\mathcal{N}	The network in the NEAT algorithm
N	The set of nodes in the network
C	The set of connections in the network
n	A node in the network
c	A connection in the network
$\text{noa}(\cdot)$	The number of attributes of a node or connection
\mathbf{n}	Tensor representing a single node in the network
\mathbf{c}	Tensor representing a single connection in the network
$\text{len}(\cdot)$	The length of a one-dimensional tensor
$ \cdot $	The cardinality of a set
\mathbf{N}	Tensor representing all nodes in the network
\mathbf{C}	Tensor representing all connections in the network
$ N _{\max}$	Maximum allowed number of nodes in the network
$ C _{\max}$	Maximum allowed number of connections in the network
$\hat{\mathbf{N}}$	Nodes tensor after padding with NaN
$\hat{\mathbf{C}}$	Connections tensor after padding with NaN
P_N	Tensor representing all nodes from all networks in the population
P_C	Tensor representing all connections from all networks in the population
$[]$	Tensor indexing operation
\leftarrow	Tensor assignment operation

and the corresponding tensorized operation is:

$$\mathbf{n}[1 + j] \leftarrow \text{attr}'_j.$$

Similarly, for a connection $c = (k_i, k_o, e, \text{attr}_1, \text{attr}_2, \dots)$, where $k_i \in \mathbb{N}$ and $k_o \in \mathbb{N}$ are the historical markings of the input and output nodes of c , respectively, $e \in \{\text{True}, \text{False}\}$ is the enabled flag, when modifying the j -th attribute in a connection, the transformation can be represented as:

$$c' = (k_i, k_o, e, \text{attr}_1, \text{attr}_2, \dots, \text{attr}'_j, \dots),$$

and the corresponding tensorized operation is:

$$\mathbf{c}[3 + j] \leftarrow \text{attr}'_j.$$

By combining the aforementioned three operations, operations for searching networks in the NEAT algorithm including Mutation and Crossover can be transformed into operations on tensors. Fig. 3 provides a graphical representation of the tensorized network operations.

3.3 Tensorized Network Inference

Another crucial component in NEAT is the inference process, where a network receives inputs and generates corresponding outputs based on its topologies and weights. For a network \mathcal{N} , given its node tensor $\hat{\mathbf{N}}$ and connection tensor $\hat{\mathbf{C}}$, the inference process can be represented as:

$$\mathbf{O} = \text{inference}_{\hat{\mathbf{N}}, \hat{\mathbf{C}}}(\mathbf{I}),$$

where \mathbf{O} and \mathbf{I} denote the outputs and inputs in the inference process, respectively.

In our tensorization method, the inference process is divided into two stages: transformation and calculation. ‘Transformation’ involves converting the node tensor $\hat{\mathbf{N}}$ and connection tensor $\hat{\mathbf{C}}$ into formats more conducive to network inference. ‘Calculation’ refers to computing the output using the tensors produced in the transformation stage. When a network undergoes multiple inference operations, it only needs to be transformed once. Networks in the NEAT algorithm can be categorized as either feedforward or concurrent, based on the presence or absence of cycles in their topological structure. Here, we primarily focus on the transformation and calculation processes in feedforward networks.

In feedforward networks, the transformation process creates two new tensors: the topological order of nodes $\mathbf{N}_{\text{order}} \in \mathbb{R}^{|N|_{\max}}$ and the expanded connections $\mathbf{C}_{\text{exp}} \in \mathbb{R}^{|N|_{\max} \times |N|_{\max} \times \text{noa}(c)}$, where $|N|_{\max}$ represents the predefined maximum limit for the number of nodes and $\text{noa}(c)$ denoting the number of attributes of connections in the network.

Given the absence of cycles in the network, topological sorting [Kahn 1962] is employed to obtain $\mathbf{N}_{\text{order}}$. For \mathbf{C}_{exp} , the generation rule can be represented as:

$$\mathbf{C}_{\text{exp}}[\hat{\mathbf{C}}[i][0, 1]] \leftarrow \begin{cases} \text{NaN}, & \hat{\mathbf{C}}[i][2] = 0 \\ \hat{\mathbf{C}}[i][2 :], & \hat{\mathbf{C}}[i][2] = 1 \end{cases},$$

where $i = 0, 1, 2, \dots, |C|_{\max}$, and $|C|_{\max}$ is the predefined maximum limit for the number of connections. Recall that in each line c of $\hat{\mathbf{C}}$, the values are $(k_i, k_o, e, \text{attr}_1, \text{attr}_2, \dots)$, with $c[0] = k_i$ and $c[1] = k_o$ indicating the indices of the input and output nodes of the connection, respectively, and $c[2] = e \in \{\text{True}, \text{False}\}$ representing the enabled flag. Locations not updated by this rule default to the value NaN. The tensors $\hat{\mathbf{N}}$, $\mathbf{N}_{\text{order}}$, and \mathbf{C}_{exp} are then used as inputs for the calculation process.

In the forward process, we utilize the transformed tensors $\hat{\mathbf{N}}$, $\mathbf{N}_{\text{order}}$, \mathbf{C}_{exp} , and the input \mathbf{I} to calculate the output \mathbf{O} . We maintain a tensor $\mathbf{v} \in \mathbb{R}^{|N|_{\max}}$ to store the values of nodes in the network. Initially, \mathbf{v} is set to the default value NaN and then updated with:

$$\mathbf{v}[k_{\text{input}}] \leftarrow \mathbf{I},$$

where k_{input} denotes the indices of input nodes in the network. The value of nodes is iteratively calculated in the order specified by N_{order} . The rule to obtain the value $v[k]$ of node n_k can be expressed as:

$$v[k] \leftarrow f_n(f_c(v | C_{\text{exp}}[:, k]) | \hat{N}[k][1 :]),$$

where $C_{\text{exp}}[:, k]$ indicates the attributes of all connections to n_k , and $\hat{N}[k][1 :]$ denotes the attributes of n_k . f_c and f_n are the calculation functions for connections and nodes in the network, respectively. For a network with connection attribute weight w , and node attributes bias b , aggregation function f_{agg} and activation function f_{act} , f_c and f_n can be defined as:

$$\begin{aligned} f_c(I_{\text{conn}} | w) &= wI_{\text{conn}}, \\ f_n(I_{\text{node}} | b, f_{\text{agg}}, f_{\text{act}}) &= f_{\text{act}}(f_{\text{agg}}(I_{\text{node}}) + b), \end{aligned}$$

where I_{conn} and I_{nodes} denote the inputs for connections and nodes, respectively.

After computing the values of all nodes, the tensor $v[k_{\text{output}}]$ represents the network's output, with k_{output} indicating the indices of output nodes in the network. Fig. 4 graphically depicts the tensorized network inference process.

Tensorization fundamentally embodies a space-time trade-off. By setting upper limits on network size, operations can be executed with fixed-shaped tensors on GPUs, maximizing parallel computation efficiency. Setting appropriate $|N|_{\text{max}}$ and $|C|_{\text{max}}$ is crucial as values that are too low restrict network complexity growth, while excessively high values waste GPU memory and reduce efficiency. In practice, these limits should be chosen based on problem requirements, and iterative tuning is often necessary to balance network scalability and computational speed.

4 IMPLEMENTATION

In this section, we outline the implementation of TensorNEAT, a library designed to enhance NEAT's scalability through tensorization and GPU/TPU acceleration via JAX. By converting network topologies into uniform tensors, TensorNEAT enables efficient parallelization of evolutionary processes, significantly boosting performance. We discuss its key components, including JAX-based hardware acceleration, user-friendly customization interfaces, and intuitive network visualization tools.

4.1 JAX-based Hardware Acceleration

JAX [Frostig et al. 2018] is an open-source numerical computing library, offering APIs similar to NumPy [Harris et al. 2020] and enabling efficient execution across various hardware platforms (CPU/GPU/TPU). Leveraging the capabilities of XLA, JAX facilitates the transformation of numerical code into optimized machine instructions. The optimization techniques provided by JAX have supported the development of numerous projects in evolutionary computation, as evidenced by [Tang et al. 2022], [Lange 2023], [Lim et al. 2022], and [Huang et al. 2024]. These advancements significantly contribute to JAX's growing popularity in the scientific community. Furthermore, JAX's flexibility and ease of integration with other machine learning frameworks, such as TensorFlow and PyTorch, have enabled researchers to experiment with cutting-edge models while benefiting from GPU acceleration and automatic differentiation. This has positioned JAX as a preferred tool for both traditional deep learning research and more specialized domains, like neuroevolution and reinforcement learning, where computational efficiency and scalability are crucial. As a result, JAX continues to be a key enabler for new methods and approaches in the field of evolutionary algorithms, pushing the boundaries of what can be achieved in terms of speed and performance.

TensorNEAT, integrating JAX, utilizes the functional programming paradigm to implement our proposed tensorization methods. This integration allows NEAT to effectively use hardware

accelerators such as GPUs and TPUs, enabling significant speedups in evolutionary processes. By maintaining uniform tensor shapes in network encoding, several key NEAT operations, such as mutation, crossover, and network inference, can be efficiently vectorized across the entire population dimension. This approach not only simplifies the computation but also ensures scalability, as operations can be parallelized and executed on large populations without compromising performance. Specifically, the use of JAX’s `jax.vmap` function enables automatic vectorization of operations over multiple individuals, while `jax.pmap` allows for further parallelization across multiple devices in a multi-GPU or multi-TPU setup. These combined capabilities make TensorNEAT adaptable to both small and large-scale experiments, facilitating faster training times and broader exploration of the search space, especially when dealing with complex neural architectures. Additionally, TensorNEAT’s design ensures that these enhancements are implemented without altering the core principles of NEAT, preserving its evolutionary behavior.

4.2 User-friendly Interfaces

Designed with user-friendly interfaces, TensorNEAT provides mechanisms for adjusting algorithms to specific requirements. It offers an extensive set of hyperparameters, allowing users to fine-tune various computational elements of the NEAT algorithm. Additionally, its modular problem templates facilitate the integration of specialized problems, ensuring adaptability to diverse application domains. By implementing a few functions, users can define the behavior of networks within the NEAT algorithm, making the customization process both flexible and straightforward. A notable feature is the interface supporting the evolution of advanced network architectures, including Spiking Neural Networks [Ghosh-Dastidar and Adeli 2009] and Binary Neural Networks [Hubara et al. 2016], which significantly broadens the scope of potential experiments. Furthermore, the system is designed to handle complex tasks with minimal setup, enabling users to quickly prototype new architectures or modify existing ones. This ease of use, coupled with its comprehensive support for cutting-edge network types, makes TensorNEAT a versatile tool for neuroevolution research. Details on the hyperparameters and the interfaces are elaborated in Appendix A and Appendix B, ensuring users have thorough guidance for fine-tuning and customization.

4.3 Intuitive Network Visualization

TensorNEAT provides two ways to visualize networks: topology diagrams and network formulations. The topology diagram visually represents the structure of the network by displaying the arrangement of nodes and connections, making it easier to understand the architecture and flow of information within the network. On the other hand, network formulations allow users to convert NEAT-generated networks into more formal representations, such as Latex formulas and Python code. This feature is particularly useful for researchers who want to mathematically analyze the network or integrate it into other software environments. These visualizations offer both an intuitive and a formal way of interacting with the evolved networks, enhancing both usability and interpretability. Fig. 5 demonstrate these two visualization methods. Together, they provide a comprehensive view of the networks, from their graphical structure to their formalized representations, enabling better understanding and facilitating further experimentation or integration into different workflows.

4.4 Feature-rich Extensions

Extending beyond the conventional NEAT paradigm, TensorNEAT includes notable algorithmic extensions such as Compositional Pattern Producing Networks (CPPN) [Stanley 2007] and Hyper-NEAT [Stanley et al. 2009], tailored for parallel processing on hardware accelerators. In addition

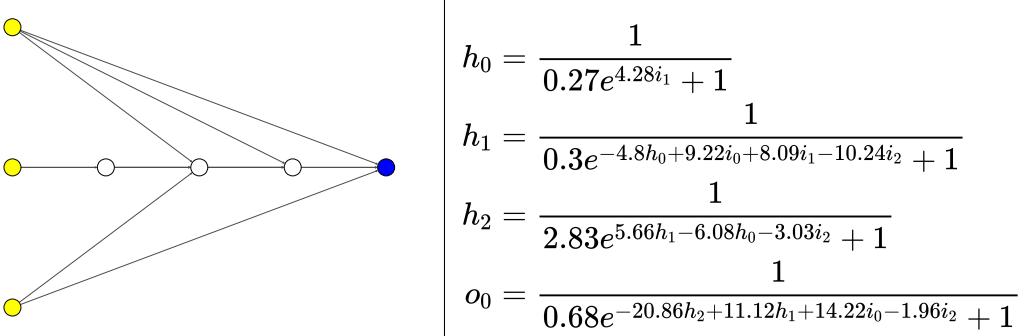


Fig. 5. Visualization methods for networks in TensorNEAT. The left side shows the topology diagram, and the right side represents the network formulations. This network has three input nodes, one output node, and three hidden nodes. In the topology diagram, yellow nodes represent input nodes, white nodes represent hidden nodes, and the blue node represents the output node. In the formula, the values of the input nodes are denoted by i_0 , i_1 , and i_2 ; the hidden nodes by h_0 , h_1 , and h_2 ; and the output node by o_0 .

to these algorithmic improvements, TensorNEAT is designed to leverage the computational capabilities of modern GPUs, enabling faster and more scalable evolution processes compared to traditional CPU-based approaches. For evaluation, TensorNEAT provides a comprehensive suite of standard test benchmarks, which span a wide range of domains from numerical optimization to complex function approximation tasks, ensuring broad applicability across various problem types. Moreover, TensorNEAT integrates seamlessly with leading reinforcement learning environments like Gym [Brockman et al. 2016], and hardware-optimized platforms such as gymnas [Lange 2022] and Brax [Freeman et al. 2021], allowing users to evaluate the performance of the NEAT algorithm in both simulated and hardware-accelerated settings. This flexibility makes TensorNEAT a robust and versatile tool for researchers and practitioners looking to apply NeuroEvolution techniques in high-performance computing environments.

5 EXPERIMENTS

This section presents a comprehensive evaluation of TensorNEAT, highlighting its performance across diverse robotic control tasks and hardware configurations. The experiments are designed to validate the effectiveness of the proposed tensorization method in addressing the computational challenges inherent in the NEAT algorithm. Specifically, we aim to answer the following key questions:

- (1) How does TensorNEAT compare with the existing NEAT implementation (NEAT-Python) in terms of execution time and solution quality?
- (2) How does TensorNEAT perform on multi-GPU setups?
- (3) What is the practical impact of the parallelism introduced by tensorization on the NEAT algorithm?
- (4) How does TensorNEAT compare with other GPU-accelerated algorithms in terms of execution time and solution quality?

By systematically addressing these questions, we demonstrate the scalability and practical benefits of the proposed approach.

5.1 Experimental Setup

This section outlines the experimental configuration used to evaluate TensorNEAT. The setup is designed to assess the library's performance across a variety of tasks, hardware configurations, and parameter settings.

Three tasks in Brax [Freeman et al. 2021] including Swimmer, Hopper and HalfCheetah are used to test TensorNEAT. These robotic control tasks require the robots to advance as far as possible in a forward direction. Fig. 6 displays representative screenshots of these environments, while Table 2 summarizes their key attributes.

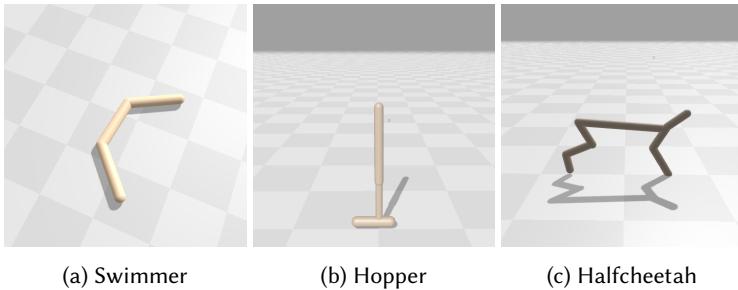


Fig. 6. Robotic control tasks in Brax.

Environment	Action Dimension	Observation Dimension	Termination Check
Swimmer	2	8	False
Hopper	3	11	True
Halfcheetah	6	18	False

Table 2. Attributes of the Brax environments

Details of the hardware specifications are provided in Table 3. In the experiments, unless otherwise specified, we used the Nvidia RTX 4090 GPU. To evaluate the performance of the algorithms, we varied parameters such as population size and the maximum number of generations while keeping other parameters fixed. Some of the key parameters used by TensorNEAT are listed in Table 4. For experiments involving other libraries including NEAT-Python [McIntyre et al. 2023] and evosax [Lange 2023], we employed their default parameter settings. A complete description of the experimental parameters is provided in Appendix C.

Table 3. Hardware Specifications

Component	Specification
CPU	AMD EPYC 7543
CPU Cores	8
GPU (Default)	NVIDIA RTX 4090
Host RAM	512 GB
GPU RAM	24 GB
OS	Ubuntu 22.04.4 LTS

Table 4. Core Parameters of TensorNEAT

Parameter	Value
max_nodes	50
max_conns	100
max_species	10
node_add_prob	0.2
conn_add_prob	0.4
survival_threshold	0.2

All experiments were repeated 10 times using different random seeds to ensure statistical robustness. Results are reported as mean values with 95% confidence intervals, providing a reliable basis for comparison.

5.2 Comparison with NEAT-Python

In this experiment, we compare the performance of TensorNEAT with the NEAT-Python library. First, we examined the evolution of average population fitness and the cumulative runtime of the algorithms across generations, with a constant population size of 10,000. As depicted in the first row in Fig. 7, TensorNEAT exhibits a more rapid improvement in population fitness over the course of the algorithm’s execution. We believe that the performance disparity between the two frameworks stems from modifications introduced in the NEAT algorithm after tensorization. To facilitate tensorization, TensorNEAT differs from NEAT-Python in several key aspects, such as population initialization, network distance computation, and species updating mechanisms. These differences have also resulted in variations in hyperparameter settings between TensorNEAT and NEAT-Python. We consider these discrepancies to be natural, as the purpose of this experiment is to validate that TensorNEAT provides a faithful implementation of the NEAT algorithm. Furthermore, the analysis of execution times, illustrated in the second row of Fig. 7, reveals a significant decrease in runtime with TensorNEAT compared to NEAT-Python.

Given the iterative nature of NEAT’s process, there is an expected increase in per-iteration time as network structures become more complex. This phenomenon was investigated by comparing the per-generation runtimes of both algorithms. In the Swimmer and Hopper tasks, as shown in the last row of Fig. 7, NEAT-Python exhibits a more marked increase in runtime, likely due to its less efficient object encoding mechanism, which becomes increasingly cumbersome with rising network complexity. By contrast, TensorNEAT, employing a tensorized encoding approach constrained by the pre-set maximum values of $|N|_{\max}$ and $|C|_{\max}$, achieves a consistent network encoding size, resulting in more stable iteration times.

Additionally, we investigated how runtime varies with changes in population size, ranging from 50 to 10,000. As illustrated in Fig. 8, NEAT-Python’s runtime significantly increases with larger populations, whereas TensorNEAT only experiences a marginal increase in runtime. It highlights TensorNEAT’s effective utilization of GPU parallel processing capabilities, further emphasizing its enhanced performance in large-scale computational tasks.

The adaptability of TensorNEAT was further validated by evaluating its performance across various hardware configurations, including the AMD EPYC 7543 CPU and a selection of mainstream GPU models. These evaluations were conducted with a consistent population size of 10,000. The total wall-clock time was recorded over 100 generations of the algorithm, and the results are summarized in Table 5. In every GPU configuration, TensorNEAT demonstrated a significant performance advantage over NEAT-Python. Notably, with the RTX 4090, TensorNEAT achieved a speedup surpassing 500 \times in the Halfcheetah environment. It is also noteworthy that TensorNEAT realized a speedup on the same CPU device in comparison to NEAT-Python, especially in the more complex Halfcheetah environment.

Table 5 shows that TensorNEAT achieves varying speedups across different Brax environments, with the highest acceleration in Halfcheetah and a lower effect in Hopper. This discrepancy arises from two factors: (1) network complexity and (2) the number of network inferences per episode. Halfcheetah has the largest observation and action spaces, leading to more complex networks that benefit significantly from TensorNEAT’s GPU acceleration. In contrast, Hopper includes an early termination mechanism where poorly performing policies result in shorter episodes, while better policies require more inferences per episode. As shown in Fig. 7, TensorNEAT evolves higher-fitness policies in Hopper, increasing the overall computational workload and reducing the

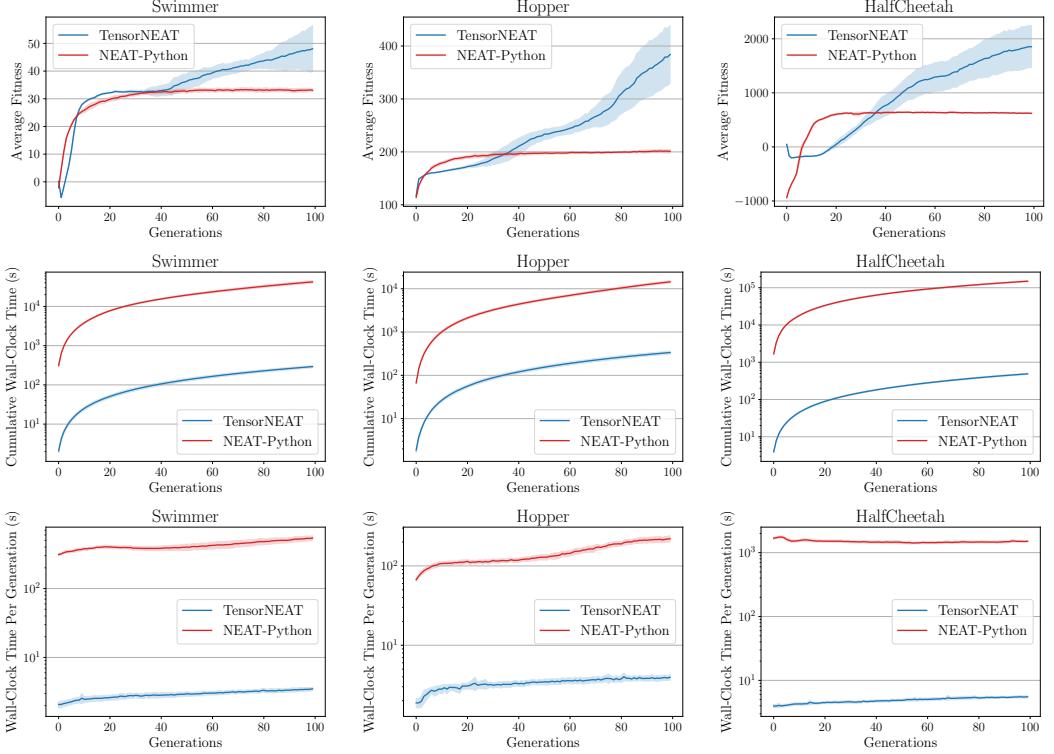


Fig. 7. Comparison of TensorNEAT and NEAT-Python with a fixed population size of 10,000, showing population fitness, cumulative runtime, and per-generation runtime.

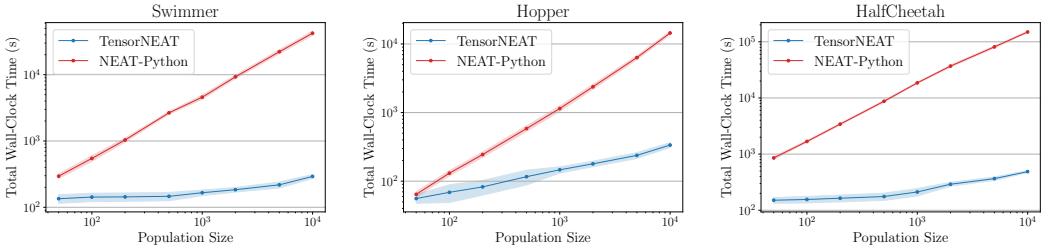


Fig. 8. Runtime comparison of TensorNEAT and NEAT-Python across different population sizes.

relative speedup. This suggests that acceleration gains depend on task characteristics, particularly network complexity and inference demands.

5.3 Performance on Multiple GPUs

We also tested the performance of TensorNEAT in multi-GPU computing using the HalfCheetah environment. The population size was fixed at 5,000, and each test involved running 50 generations of the NEAT algorithm. We conducted the experiments on Nvidia RTX 4090 hardware, varying the number of GPUs used (1, 2, 4, and 8). For each configuration, we averaged the total algorithm execution time over 10 independent runs to ensure reliable results. The results are presented in

Table 5. Runtimes over 100 generations on different hardware configurations, with a constant population size of 10,000.

Task	Library	Hardware	Time (s)	Speedup
Swimmer	NEAT-Python	EPYC 7543 (CPU)	42279.63 ± 3031.97	1.00
		RTX 4090	215.70 ± 6.35	196.01
	TensorNEAT	RTX 3090	292.22 ± 20.19	144.68
		RTX 2080Ti	434.94 ± 12.49	97.21
		EPYC 7543 (CPU)	14678.10 ± 616.85	2.88
Hopper	NEAT-Python	EPYC 7543 (CPU)	14438.13 ± 900.68	1.00
		RTX 4090	241.30 ± 9.41	59.83
	TensorNEAT	RTX 3090	336.08 ± 26.96	42.96
		RTX 2080Ti	518.40 ± 7.22	27.85
		EPYC 7543 (CPU)	13473.01 ± 544.07	1.07
Halfcheetah	NEAT-Python	EPYC 7543 (CPU)	149516.00 ± 4817.90	1.00
		RTX 4090	274.74 ± 14.21	544.21
	TensorNEAT	RTX 3090	487.82 ± 19.05	306.50
		RTX 2080Ti	705.13 ± 16.02	212.04
		EPYC 7543 (CPU)	15914.08 ± 4005.08	9.40

Fig. 9. The data is presented in two aspects: execution time and its inverse (execution performance). It can be observed that, as the number of GPUs increases, execution time decreases significantly. The improvement in execution performance is nearly linear, but the performance gains diminish when the number of GPUs increases from 4 to 8. This phenomenon is mainly attributed to two factors: 1. TensorNEAT employs multi-GPU acceleration only for the network evaluation phase, excluding the execution of the NEAT algorithm itself. As a result, only part of the overall execution time benefits from multi-GPU acceleration, leading to sub-linear performance improvements. 2. As the number of GPUs increases, TensorNEAT incurs additional overhead in managing communication between GPUs, which reduces the performance gain as more GPUs are utilized. The multi-GPU experiments validate the effectiveness of TensorNEAT’s acceleration while also demonstrating that its benefits tend to taper off as the number of GPUs increases.

5.4 Performance across Scalable Populations

In this part, we investigate the impact of varying population sizes on the performance of TensorNEAT across three Brax environments: Swimmer, Hopper, and HalfCheetah. The primary objective is to assess how increasing the population size influences both the quality of the solutions and the convergence rate, thereby highlighting the benefits of the tensorization-based parallelism.

Fig. 10 presents the experimental results, which are divided into two distinct parts. The top panel illustrates the best fitness achieved in the population after running a fixed number of generations for each population size. Across all three environments, larger population sizes consistently yield higher best fitness values. This result aligns with the intuition that a larger population enhances the search strength within each generation, leading to a more thorough exploration of the solution space.

The bottom panel of Fig. 10 compares the best fitness attained after running for an equivalent wall-clock time. In most cases, larger population sizes reach higher fitness scores more rapidly. This observation demonstrates the effectiveness of TensorNEAT’s parallel execution strategy. Unlike traditional serial implementations, where increasing the population size typically extends the

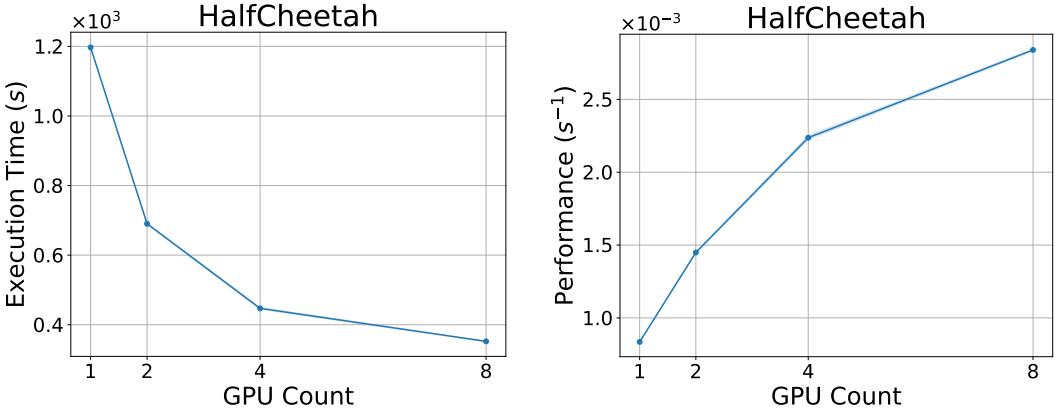


Fig. 9. Results of multi-GPU acceleration using 1 to 8 GPUs. Data is presented in two forms: execution time (left) and its inverse (right). The shaded regions in the figure represent the 95% confidence interval of the data, but due to the dense data distribution, the shading is not noticeable.

computation time per generation, the tensorized approach significantly reduces this overhead. As a result, larger populations can sustain a high optimization rate while remaining within the hardware limits.

However, the HalfCheetah environment exhibits a notable exception. In this scenario, a population size of 10,000 results in a lower optimization rate compared to a population size of 1,000. This outcome is attributed to the increased complexity of the HalfCheetah task, which demands computational resources beyond the hardware's parallel processing capacity. Consequently, when the population size exceeds the hardware-supported limits, the benefits of parallelism diminish, leading to longer execution times per generation and a subsequent reduction in the optimization rate.

Overall, the parallelism introduced by tensorization not only accelerates the algorithm's execution but also enables the use of larger population sizes to enhance optimization efficiency. However, the performance gains from larger populations are constrained by the complexity of the problem and the hardware's parallel processing capacity.

5.5 Comparison with evosax

In this set of experiments, we compare TensorNEAT with evosax [Lange 2023], a JAX-based GPU-accelerated evolutionary algorithm library. We selected several GA variants available in evosax, including SimpleGA [Such et al. 2018], SAMR-GA [Clune et al. 2008], LGA [Lange et al. 2023], and GESMR-GA [Kumar et al. 2022], to benchmark against the NEAT algorithm implemented in TensorNEAT. For evosax, the default hyperparameter settings were used, and the policy network was instantiated as a four-layer MLP with 16 neurons in each hidden layer ([inputs->hidden->hidden->outputs]). In all experiments, the population size was fixed at 5000 and the maximum number of generations was set to 100.

Fig. 11 and Table 6 summarize the experimental outcomes. On the relatively simple Swimmer task, the NEAT algorithm achieved performance comparable to that of the GA variants in evosax, indicating that both methods are effective in less complex environments. However, in the more challenging Hopper and Halfcheetah tasks, the NEAT algorithm consistently outperformed all

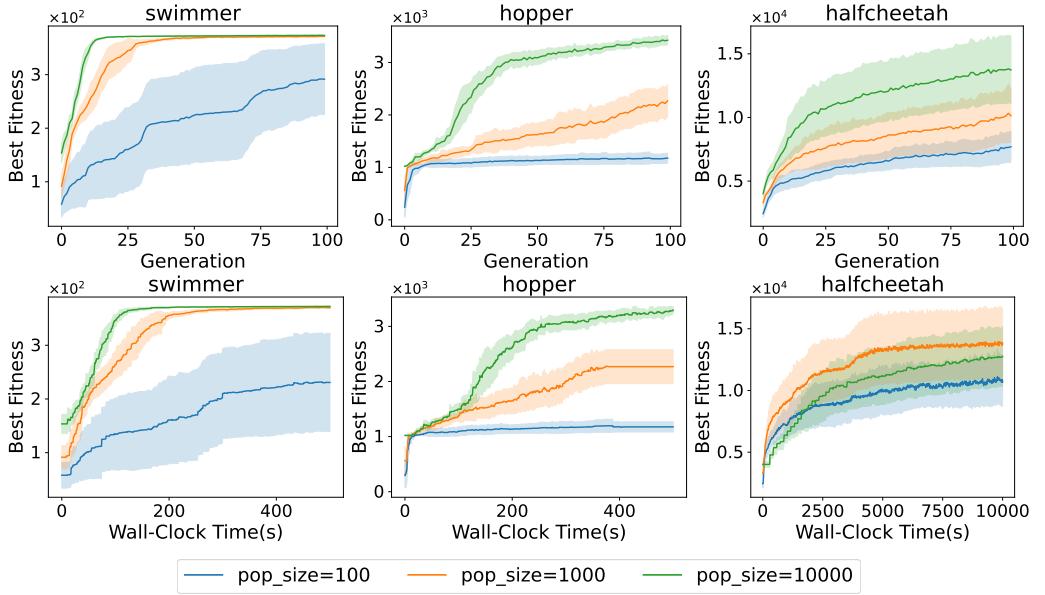


Fig. 10. Impact of population size on the best fitness achieved across Brax environments

GA variants. These results highlight the advantages of NEAT in robotic control tasks, particularly its ability to evolve network architectures that are better suited to the demands of complex environments.

The superior performance of the NEAT algorithm can be attributed to its incremental growth mechanism, which begins with minimal network structures and progressively expands the topology. This gradual expansion of the search space allows the algorithm to adaptively scale its complexity in accordance with the task requirements. Additionally, the speciation mechanism inherent in NEAT preserves promising network structures, further facilitating effective exploration of the solution space.

It is noteworthy that TensorNEAT incurred an approximate 10% increase in execution time relative to the GA variants in evoSAX across all tasks. We attribute this overhead to differences in network architectures. Specifically, the NEAT algorithm employs networks with irregular and randomly generated topologies, necessitating the individual computation of each node's value. This process involves numerous memory read and write operations on the GPU, which are less efficient compared to the streamlined matrix multiplication operations used in the MLPs of the GA variants. Despite this additional computational cost, the enhanced performance of NEAT in more complex tasks justifies the marginal increase in runtime.

In summary, the experiments compared to evoSAX demonstrate that although the NEAT algorithm requires slightly more computation time, its adaptive network growth and speciation mechanisms deliver superior performance in challenging robotic control tasks. These findings validate the effectiveness of our GPU-accelerated implementation and underscore the practical merit of TensorNEAT.

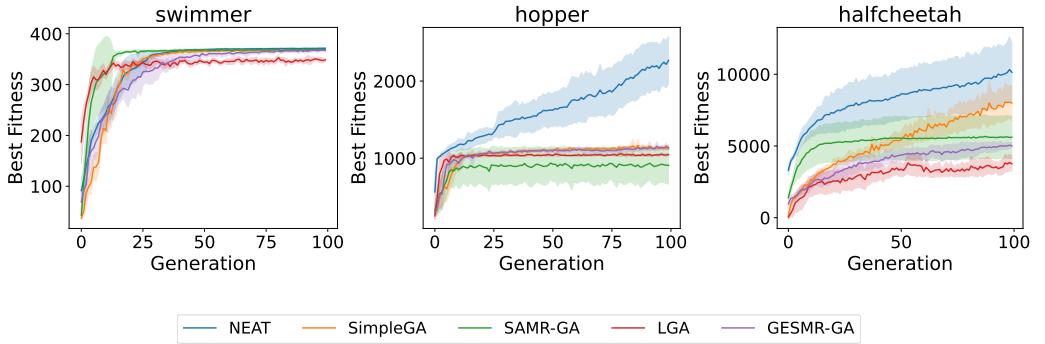


Fig. 11. Best fitness trends across Brax environments

Table 6. Performance and runtime comparison over 100 generations across tasks

Task	Library	Algorithm	Performance	Cost Time(s)
Swimmer	TensorNEAT	NEAT	372.06 ± 0.87	731.02 ± 60.25
		SimpleGA	370.97 ± 0.55	685.37 ± 45.15
		SAMR-GA	371.24 ± 0.82	721.35 ± 74.82
		LGA	361.68 ± 2.14	726.42 ± 82.89
		GESMR-GA	369.41 ± 2.21	641.73 ± 74.06
	evosax	NEAT	2353.97 ± 308.07	364.72 ± 7.47
Hopper	TensorNEAT	SimpleGA	1293.68 ± 89.78	304.16 ± 2.96
		SAMR-GA	983.02 ± 178.11	320.26 ± 7.81
		LGA	1140.61 ± 16.80	303.34 ± 1.80
		GESMR-GA	1194.43 ± 42.63	315.27 ± 11.49
		NEAT	10481.12 ± 2315.80	1431.43 ± 13.94
Halfcheetah	TensorNEAT	SimpleGA	8748.52 ± 1441.24	1407.41 ± 2.13
		SAMR-GA	5776.35 ± 1517.56	1406.16 ± 2.05
		LGA	4705.10 ± 554.95	1406.41 ± 2.16
		GESMR-GA	5275.45 ± 310.59	1408.14 ± 2.32
		NEAT		

6 CONCLUSION

In this paper, we tackled the scalability limitations of the NeuroEvolution of Augmenting Topologies (NEAT) algorithm by introducing a tensorization approach, which converts network topologies into uniformly shaped tensors for efficient parallel computation. Building on this foundation, we developed TensorNEAT, a library that utilizes JAX for automatic function vectorization and hardware acceleration, enabling seamless execution on modern GPUs and TPUs. TensorNEAT's flexible design allows for easy integration with widely-used reinforcement learning environments such as Gym, Brax, and gymnas, supporting extensive experimentation in robotics and control tasks. Our method not only simplifies the parallelization of the evolutionary process but also significantly reduces the computational cost of large-scale simulations. Consequently, TensorNEAT achieves up to 500x speedups compared to conventional NEAT implementations, establishing it as a powerful tool for high-performance neuroevolution, especially in scenarios requiring real-time feedback or complex multi-agent interactions. This substantial improvement in computational efficiency

paves the way for researchers to explore more advanced evolutionary algorithms and tackle more challenging real-world applications.

Our findings add to the current understanding and may pave the way for future research in this field. Looking ahead, our roadmap for TensorNEAT includes expanding its reach to distributed computing environments, transcending the limitations of single-machine setups. By enabling TensorNEAT to operate in distributed architectures, we aim to significantly improve scalability, allowing the system to handle larger datasets and more complex neural network topologies with greater efficiency. Furthermore, we plan to augment TensorNEAT's suite of functionalities by integrating advanced NEAT variants, such as DeepNEAT and CoDeepNEAT [Miikkulainen et al. 2019], to further enhance its potential in solving complex neuroevolution challenges. These variants introduce hierarchical and modular structures that can better exploit the representational power of deep learning models, potentially accelerating both the convergence speed and the quality of evolved solutions. Our long-term goal is to ensure TensorNEAT remains adaptable and robust, providing researchers and practitioners with a versatile tool for tackling a diverse range of evolutionary computation problems across various domains.

REFERENCES

- Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. *TensorFlow, Large-scale machine learning on heterogeneous systems*. <https://doi.org/10.5281/zenodo.4724125>
- Joshua E. Auerbach and Josh C. Bongard. 2011. Evolving complete robots with CPPN-NEAT: The utility of recurrent connections. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*. 1475–1482.
- b2developer. 2022. MonopolyNEAT: NEAT implemented into Monopoly with a knockout tournament scheme. <https://github.com/b2developer/MonopolyNEAT> Accessed: 2023-08-15.
- Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. arXiv:1606.01540
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.
- Jeff Clune, Dusan Misevic, Charles Ofria, Richard E. Lenski, Santiago F. Elena, and Rafael Sanjuán. 2008. Natural selection fails to optimize mutation rates for long-term adaptation on rugged fitness landscapes. *PLOS Computational Biology* 4, 9 (09 2008), 1–8. <https://doi.org/10.1371/journal.pcbi.1000187>
- C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. 2021. Brax—A differentiable physics engine for large scale rigid body simulation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*.
- Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* 4, 9 (2018).
- Alex Gajewsky. 2023. PyTorch NEAT. <https://github.com/uber-research/PyTorch-NEAT> Accessed: 2023-08-07.
- Zhenyu Gao and Gongjin Lan. 2021. A NEAT-based multiclass classification method with class binarization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery, New York, NY, USA, 277–278. <https://doi.org/10.1145/3449726.3459509>
- Samanwoy Ghosh-Dastidar and Hojjat Adeli. 2009. Spiking neural networks. *International Journal of Neural Systems* 19, 04 (2009), 295–308.
- Charles R. Harris, K. Jarrod Millman, Stéfan J. Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.
- Beichen Huang, Ran Cheng, Zhuozhao Li, Yaochu Jin, and Kay Chen Tan. 2024. EvoX: A Distributed GPU-Accelerated Framework for Scalable Evolutionary Computation. *IEEE Transactions on Evolutionary Computation* (2024), 1–1. <https://doi.org/10.1109/TEVC.2024.3388550>
- Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) (NIPS’16).

- Curran Associates Inc., Red Hook, NY, USA, 4114–4122.
- A. B. Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (Nov. 1962), 558–562. <https://doi.org/10.1145/368996.369025>
- Akarsh Kumar, Bo Liu, Risto Miikkulainen, and Peter Stone. 2022. Effective mutation rate adaptation through group elite selection. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Boston, Massachusetts) (GECCO '22). Association for Computing Machinery, New York, NY, USA, 721–729. <https://doi.org/10.1145/3512290.3528706>
- Robert Lange, Tom Schaul, Yutian Chen, Chris Lu, Tom Zahavy, Valentin Dalibard, and Sebastian Flennerhag. 2023. Discovering attention-based genetic algorithms via meta-black-box optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Lisbon, Portugal) (GECCO '23). Association for Computing Machinery, New York, NY, USA, 929–937. <https://doi.org/10.1145/3583131.3590496>
- Robert Tjarko Lange. 2022. *gymnax: A JAX-based reinforcement learning environment library*. <http://github.com/RobertTLange/gymnax>
- Robert Tjarko Lange. 2023. evosax: JAX-Based evolution strategies. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation* (Lisbon, Portugal) (GECCO '23 Companion). Association for Computing Machinery, New York, NY, USA, 659–662. <https://doi.org/10.1145/3583133.3590733>
- Joel Lehman and Kenneth O. Stanley. 2011. Evolving a diversity of creatures through novelty search and local competition. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*. 211–218.
- Bryan Lim, Maxime Allard, Luca Grillotti, and Antoine Cully. 2022. QDax: On the benefits of massive parallelization for quality-diversity. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery, New York, NY, USA, 128–131. <https://doi.org/10.1145/3520304.3528927>
- Alan McIntyre, Matt Kallada, Cesar G. Miguel, Carolina Feher de Silva, and Marcio Lobo Netto. 2023. Python implementation of the NEAT neuroevolution algorithm. <https://github.com/CodeReclaimers/neat-python> Accessed: 2023-08-07.
- Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrazad, Arshak Navruzyan, Nigel Duffy, et al. 2019. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier, 293–312.
- Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. arXiv:1504.04909 [cs.AI]
- Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
- peter ch. 2019. MultiNEAT: Portable NeuroEvolution library. <https://github.com/peter-ch/MultiNEAT> Accessed: 2023-08-15.
- Son Pham, Keyi Zhang, Tung Phan, Jasper Ding, and Christopher Dancy. 2018. Playing SNES games with neuroevolution of augmenting topologies. In *Proceedings of the AAAI Conference on Artificial Intelligence - Student Abstract Track*, Vol. 32.
- Kosmas Pinitas, Konstantinos Makantasis, Antonios Liapis, and Georgios N. Yannakakis. 2022. RankNEAT: Outperforming stochastic gradient search in preference learning tasks. In *Proceedings of the Genetic and Evolutionary Computation Conference*. Association for Computing Machinery, New York, NY, USA, 1084–1092. <https://doi.org/10.1145/3512290.3528744>
- Sebastian Risi, Joel Lehman, and Kenneth O. Stanley. 2010. Evolving the placement and density of neurons in the hyperneat substrate. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*. 563–570.
- Stefano Sarti and Gabriela Ochoa. 2021. A NEAT visualisation of neuroevolution trajectories. In *Applications of Evolutionary Computation*. Springer, 714–728.
- Fernando Silva, Paulo Urbano, Sancho Oliveira, and Anders Lyhne Christensen. 2012. odNEAT: An algorithm for distributed online, onboard evolution of robot behaviours. In *The International Conference on the Synthesis and Simulation of Living Systems*. 251–258. <https://doi.org/10.1162/978-0-262-31050-5-ch034>
- Kenneth O. Stanley. 2007. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines* 8 (2007), 131–162.
- Kenneth O. Stanley, Bobby D. Bryant, Igor Karpov, and Risto Miikkulainen. 2006. Real-time evolution of neural networks in the NERO video game. In *Proceedings of the 21st National Conference on Artificial Intelligence*, Vol. 2. AAAI Press, 1671–1674.
- Kenneth O. Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. 2019. Designing neural networks through neuroevolution. *Nature Machine Intelligence* 1, 1 (2019), 24–35.
- Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. 2009. A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life* 15, 2 (04 2009), 185–212. <https://doi.org/10.1162/artl.2009.15.2.15202>
- Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. *Evolutionary Computation* 10, 2 (2002), 99–127.

- Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. 2018. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. arXiv:1712.06567 [cs.NE] <https://arxiv.org/abs/1712.06567>
- Yujin Tang, Yingtao Tian, and David Ha. 2022. EvoJAX: Hardware-accelerated neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. Association for Computing Machinery, New York, NY, USA, 308–311. <https://doi.org/10.1145/3520304.3528770>
- Lishuang Wang, Mengfei Zhao, Enyu Liu, Kebin Sun, and Ran Cheng. 2024. Tensorized NeuroEvolution of Augmenting Topologies for GPU acceleration. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Melbourne, VIC, Australia) (GECCO '24). Association for Computing Machinery, New York, NY, USA, 1156–1164. <https://doi.org/10.1145/3638529.3654210>
- Mehmet Erkan Yuksel. 2018. Agent-based evacuation modeling with multiple exits using NeuroEvolution of Augmenting Topologies. *Advanced Engineering Informatics* 35 (2018), 30–55.

A HYPERPARAMETERS

The hyperparameters in TensorNEAT consists of those that influence the algorithm's dynamics and those who affect network behaviors:

- Algorithmic Controls:
 - `seed`: Random seed (integer).
 - `fitness_target`: Target fitness value for termintaion (float).
 - `generation_limit`: Maximum number of generations for termintaion (float).
 - `pop_size`: Population size (integer).
 - `network_type`: Network type, either feedforward (no cycles) or recurrent (with cycles).
 - `inputs`: Number of network inputs (integer).
 - `outputs`: Number of network outputs (integer).
 - `max_nodes`: Max nodes allowed in a network (integer).
 - `max_conns`: Max connections allowed in a network (integer).
 - `max_species`: Max species in the population (integer).
 - `compatibility_disjoint`: Weight for disjoint genes in distance calculation between genomes (float).
 - `compatibility_homologous`: Weight for homologous genes in distance calculation between genomes (float).
 - `node_add`: Probability of a node addition during mutation (float).
 - `node_delete`: Probability of a node deletion during mutation (float).
 - `conn_add`: Probability of a connection addition during mutation (float).
 - `conn_delete`: Probability of a connection deletion during mutation (float).
 - `compatibility_threshold`: Distance threshold for genome speciation (float).
 - `species_elitism`: Minimum species count to prevent all species are stagnated (integer).
 - `max_stagnation`: Stagnation threshold for species (integer). If a species does not show improvement for `max_stagnation` consecutive generations, then this species will be stagnated.
 - `genome_elitism`: Number of elite genomes preserved for next generation (integer).
 - `survival_threshold`: Percentage of species survival for crossover (float).
 - `spawn_number_change_rate`: Rate of change in species size over two consecutive generations (float).
- Network Behavior Controls:
 - `bias_init_mean`: Mean value for bias initialization (float).
 - `bias_init_std`: Standard deviation for bias initialization (float).
 - `bias_mutate_power`: Mutation strength for bias values (float).
 - `bias_mutate_rate`: Probability of bias value mutation (float).
 - `bias_replace_rate`: Probability to replace existing bias with a new value during mutation (float).
 - `response_init_mean`: Mean value for response initialization (float).
 - `response_init_std`: Standard deviation for response initialization (float).
 - `response_mutate_power`: Mutation strength for response values (float).
 - `response_mutate_rate`: Probability of response value mutation (float).
 - `response_replace_rate`: Probability to replace existing response with a new value during mutation (float).
 - `weight_init_mean`: Mean value for weight initialization (float).
 - `weight_init_std`: Standard deviation for weight initialization (float).
 - `weight_mutate_power`: Mutation strength for weight values (float).

- `weight_mutate_rate`: Probability of weight value mutation (float).
- `weight_replace_rate`: Probability to replace existing weight with a new value during mutation (float).
- `activation_default`: Default value for activation function.
- `activation_options`: Available activation functions.
- `activation_replace_rate`: Probability to change the activation function during mutation.
- `aggregation_default`: Default value for aggregation function.
- `aggregation_options`: Available aggregation functions.
- `aggregation_replace_rate`: Probability to change the aggregation function during mutation.

B INTERFACES

B.1 Network Interface

TensorNEAT offers users the flexibility to define a custom network that will be optimized by the NEAT algorithms by providing a network interface (as presented in Listing 1 and Listing 2). Users are required to define the specific behaviors associated with their network, including initialization, mutation, distance computation, and inference.

By implementing these interfaces, users can fit their specific requirements and leverage TensorNEAT's power for a wide range of neural network architectures.

```
class BaseGene:
    fixed_attrs = []
    custom_attrs = []

    def __init__(self):
        pass

    def new_custom_attrs(self):
        raise NotImplementedError

    def mutate(self, randkey, gene):
        raise NotImplementedError

    def distance(self, gene1, gene2):
        raise NotImplementedError

    def forward(self, attrs, inputs):
        raise NotImplementedError

    @property
    def length(self):
        return len(self.fixed_attrs) + len(self.custom_attrs)
```

Listing 1. Gene interface.

B.2 Problem Template

In TensorNEAT, users also have the flexibility to define custom problems they wish to optimize using the NEAT algorithms. This can be done by implementing the `Problem` interface as illustrated in Listing 3. Within this interface, users are required to provide the evaluation process for the problem, specify the input and output dimensions, and optionally implement a function to visualize the solution.

```

class BaseGenome:
    network_type = None

    def __init__(
        self,
        num_inputs: int,
        num_outputs: int,
        max_nodes: int,
        max_conns: int,
        node_gene: BaseNodeGene = DefaultNodeGene(),
        conn_gene: BaseConnGene = DefaultConnGene(),
    ):
        self.num_inputs = num_inputs
        self.num_outputs = num_outputs
        self.input_idx = jnp.arange(num_inputs)
        self.output_idx = jnp.arange(num_inputs, num_inputs + num_outputs)
        self.max_nodes = max_nodes
        self.max_conns = max_conns
        self.node_gene = node_gene
        self.conn_gene = conn_gene

    def transform(self, nodes, conns):
        raise NotImplementedError

    def forward(self, inputs, transformed):
        raise NotImplementedError

```

Listing 2. Genome interface.

```

from typing import Callable

from utils import State

class BaseProblem:
    jitable = None

    def setup(self, randkey, state: State = State()):
        """initialize the state of the problem"""
        raise NotImplementedError

    def evaluate(self, randkey, state: State, act_func: Callable, params):
        """evaluate one individual"""
        raise NotImplementedError

    @property
    def input_shape(self):
        raise NotImplementedError

    @property
    def output_shape(self):
        raise NotImplementedError

    def show(self, randkey, state: State, act_func: Callable, params, *args, **kwargs):
        raise NotImplementedError

```

Listing 3. Problem interface in TensorNEAT.

C EXPERIMENT DETAIL

In this section, we detail the hyperparameters in experiments.

C.1 Hyperparameters

Hyperparameters in TensorNEAT:

- Algorithmic Controls:
 - seed: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
 - fitness_target: Inf (to terminate at the fixed generation);
 - generation_limit: 100;
 - pop_size: [50, 100, 200, 500, 1000, 2000, 5000, 10000];
 - network_type: feedforward;
 - inputs: the same as the observation dimension of the environment.
 - outputs: the same as the action dimension of the environment.
 - max_nodes: 50;
 - max_conns: 100;
 - max_species: 10;
 - compatibility_disjoint: 1.0;
 - compatibility_homologous: 0.5;
 - node_add: 0.2;
 - node_delete: 0;
 - conn_add: 0.4;
 - conn_delete: 0;
 - compatibility_threshold: 3.5;
 - species_elitism: 2;
 - max_stagnation: 15;
 - genome_elitism: 2;
 - survival_threshold: 0.2;
 - spawn_number_change_rate: 0.5;
- Network Behavior Controls:
 - bias_init_mean: 0;
 - bias_init_std: 1.0;
 - bias_mutate_power: 0.5;
 - bias_mutate_rate: 0.7;
 - bias_replace_rate: 0.1;
 - response_init_mean: 1.0;
 - response_init_std: 0;
 - response_mutate_power: 0;
 - response_mutate_rate: 0;
 - response_replace_rate: 0;
 - weight_init_mean: 0;
 - weight_init_std: 1;
 - weight_mutate_power: 0.5;
 - weight_mutate_rate: 0.8;
 - weight_replace_rate: 0.1;
 - activation_default: tanh;
 - activation_options: [tanh];
 - activation_replace_rate: 0;

- aggregation_default: sum;
- aggregation_options: [sum];
- aggregation_replace_rate: 0;

Hyperparameters in NEAT-Python:

- fitness_criterion: max
- fitness_threshold: 999
- pop_size: 10000
- reset_on_extinction: False
- **[DefaultGenome]**
 - activation_default: tanh
 - activation_mutate_rate: 0
 - activation_options: tanh
 - aggregation_default: sum
 - aggregation_mutate_rate: 0.0
 - aggregation_options: sum
 - bias_init_mean: 0.0
 - bias_init_stdev: 1.0
 - bias_max_value: 30.0
 - bias_min_value: -30.0
 - bias_mutate_power: 0.5
 - bias_mutate_rate: 0.7
 - bias_replace_rate: 0.1
 - compatibility_disjoint_coefficient: 1.0
 - compatibility_weight_coefficient: 0.5
 - conn_add_prob: 0.5
 - conn_delete_prob: 0
 - enabled_default: True
 - enabled_mutate_rate: 0.01
 - feed_forward: True
 - initial_connection: full
 - node_add_prob: 0.2
 - node_delete_prob: 0
 - num_hidden: 0
 - num_inputs: the same as the observation dimension of the environment.
 - num_outputs: the same as the action dimension of the environment.
 - response_init_mean: 1.0
 - response_init_stdev: 0.0
 - response_max_value: 30.0
 - response_min_value: -30.0
 - response_mutate_power: 0.0
 - response_mutate_rate: 0.0
 - response_replace_rate: 0.0
 - weight_init_mean: 0.0
 - weight_init_stdev: 1.0
 - weight_max_value: 30
 - weight_min_value: -30
 - weight_mutate_power: 0.5

- weight_mutate_rate: 0.8
- weight_replace_rate: 0.1
- [DefaultSpeciesSet]
 - compatibility_threshold: 3.0
- [DefaultStagnation]
 - species_fitness_func: max
 - max_stagnation: 20
 - species_elitism: 2
- [DefaultReproduction]
 - elitism: -9999
 - survival_threshold: 0.2

Hyperparameters in evosax:

- [SimpleGA]
 - cross_over_rate: 0.0
 - sigma_init: 0.07
 - sigma_decay: 1.0
 - sigma_limit: 0.01
 - init_min: 0.0
 - init_max: 0.0
 - clip_min: -jnp.finfo(jnp.float32).max
 - clip_max: jnp.finfo(jnp.float32).max
 - elite_ratio: 0.5
- [SAMR-GA]
 - sigma_init: 0.07
 - sigma_meta: 2.0
 - sigma_best_limit: 0.0001
 - init_min: 0.0
 - init_max: 0.0
 - clip_min: -jnp.finfo(jnp.float32).max
 - clip_max: jnp.finfo(jnp.float32).max
 - elite_ratio: 0.0
- [LGA]
 - cross_over_rate: 0.0
 - sigma_init: 1.0
 - init_min: -5.0
 - init_max: 5.0
 - clip_min: -jnp.finfo(jnp.float32).max
 - clip_max: jnp.finfo(jnp.float32).max
 - elite_ratio: 1.0
- [GESMR-GA]
 - sigma_init: 0.07
 - sigma_meta: 2.0
 - init_min: 0.0
 - init_max: 0.0
 - clip_min: -jnp.finfo(jnp.float32).max
 - clip_max: jnp.finfo(jnp.float32).max
 - elite_ratio: 0.5

– **sigma_ratio**: 0.5