A Novel Library for Neuroevolution Algorithm Discovery

As a step towards creating smarter AI, how can a new library be built for designing experiments and extensions of the NEAT algorithm?

AP Research

Word Count: 5363

Abstract

NeuroEvolution of Augmenting Topologies (NEAT) is a revolutionary neuroevolution algorithm for training artificial intelligence, making it an ideal starting place in the quest to build more intelligent AI algorithms. Yet, the common method for customizing NEAT has been to rebuild it from scratch: hundreds of individual researchers have created their own implementations of NEAT, either to modify the original algorithm or bring it to a new programming language (Papavasileiou et al.). Additionally, NEAT has no official implementation for the Python language. Thus, this project explores the possibility of creating a novel Python library, dubbed BaseNEAT, that improves upon existing Python implementations of NEAT and provides a toolkit to facilitate the invention of new NEAT extensions. First, the field of neuroevolution was explored in the Python programming language, culminating in several experiments using an existing NEAT library. During this process, key limitations were realized regarding the library's lack of an interface for modifying the core NEAT algorithm. It was decided to reimplement NEAT from scratch, redesigning the algorithm as a base library with greater modularity than the existing library. Next, several experiments were performed to test the efficacy and ease of extension of the new BaseNEAT library against existing Python libraries and the original NEAT code. BaseNEAT was proven to be functional and produced comparable results to other implementations, showing that it is possible to consolidate the NEAT algorithm into its basic form such that writing new extensions of NEAT can be as simple as interfacing with a base API.

1 Introduction

For centuries, questions have been asked about the "causes and conditions" of life, as well as the ability of man to "make living creatures" (Aguilar et al.). Mary Shelley's 1818 novel *Frankenstein*, subtitled *The Modern Prometheus* after the eponymous Titan god who is credited with shaping humanity from clay, was written at a time when revolutionary scientific and technological advances prompted debate on the nature of life (Aguilar et al.). Modern science-fiction writers such as Ted Chiang, whose stories gave rise to the popular 2016 film *Arrival*, have debated the very same question in light of modern computer scientific advances such as the Internet, software, and machine learning/AI.

According to Aguilar et al., the term "artificial life" can take several different meanings, but broadly refers to the "the synthesis and simulation of living systems" using models and algorithms. Thus, the field of artificial life revolves around the idea: "build life in order to understand it better" (Aguilar et al.). Now, two centuries after *Frankenstein*, the collective dream remains the same: to strive to understand the natures of life and intelligence by building intelligent life.

Today, this dream is approaching its reality. AI researchers like Kenneth Stanley write algorithms, borrowing strategies from biology to create sapient artificial intelligence. One of Stanley's most famous algorithms is a neuroevolution algorithm called NEAT, or NeuroEvolution of Augmenting Topologies.

This paper describes BaseNEAT, a library building off of NEAT as yet another step towards creating man-made intelligent life, and reviews results from the first year of this ongoing project. The next section provides a brief review of NEAT and other NEAT libraries and the problem with them, situating BaseNEAT in a broader context. The process of enhancing NEAT to create BaseNEAT is then described. The last sections evaluate BaseNEAT's efficacy and suggest avenues for future work.

2 Literature Review

To understand why this project attempts to create a new NEAT library, it is important to review the existing body of literature and define the elements of a neuroevolution algorithm.

While at first this project sought to explore new ways of creating complex virtual life, i.e. digital organisms, it was determined that NEAT was the ideal place to start. The goal was to code an original algorithm or experiment and report on the findings. Over the course of the research process, avenues were explored and discarded until settling on writing a new modular toolkit for modifying the NEAT algorithm.

First, some context on neuroevolution algorithms is necessary to both understand the meaning of this mission and explicate the advantages of NEAT over other neuroevolution algorithms. Neuroevolution can be defined as the process of evolving "neural networks through evolutionary algorithms," making evolutionary algorithms and neural networks primary subjects for review before expounding neuroevolution and, specifically, NEAT ("Neuroevolution: A different kind of deep learning"). This section then reviews the existing literature surrounding NEAT and the problem that BaseNEAT seeks to address.

2.1 Evolutionary/Genetic Algorithms

Evolutionary, or genetic, algorithms are the first crucial element of a neuroevolution algorithm. Genetic algorithms are a category of search heuristics that solve problems by emulating the process of evolution in nature (De Jong; Goldberg). Given a problem to solve, genetic algorithms apply an evolutionary process to the "search space" of possible solutions. "Individuals" in an evolving population represent points in the search space–each one is a possible solution.

Genetic algorithms begin with an initial population of randomly generated possible solutions, then a "fitness function" evaluates each individual by measuring its success at solving the given problem. Fitter individuals are selected for reproduction and then "mated" with each other by recombining their genes to form offspring that inherit traits from both parents. Finally, offspring are "mutated," randomly perturbing their traits to expand to new areas of the search

space. The process repeats with the new population of offspring, continuing until a stop condition is reached.

Each individual consists of two parts: genotype and phenotype. An individual's genotype is analogous to its DNA. During the reproduction stage of a genetic algorithm, two parents' genotypes are broken up and recombined into a new offspring genome, with each gene randomly passed down from either parent. The genotype is like the blueprint that describes the phenotype. During the selection process, an individual's genotype is evaluated indirectly by testing its phenotype.

2.2 Neural Networks

Neural networks are the second vital piece of a neuroevolution algorithm. Neural networks are computerized models of interconnection between nodes inspired by the synapses and neurons of the human brain (Haykin). They are often organized into layers: an input layer, output layer, and hidden layers in between. While the input layer is fixed to receive a certain number of inputs, and the output layer is fixed to send a certain number of outputs, the hidden layer can be more flexible. NEAT, the algorithm explored in this paper, evolves its own network structure, so hidden neurons are not layered at all.

Like the human brain, a neural network computes its outputs by passing signals from neurons to other neurons through what are called connections. A neuron receives signals from its incoming connections, multiplies each one by the weight of connection, sums these weighted signals, applies a predefined activation function to the sum, then passes the resulting value through its outgoing connections to the next neurons.

To activate a neural network, input values are provided, one for each of the network's input neurons. Then, each input neuron activates and transmits its outgoing signals to other neurons within the network. These neurons then activate and likewise send their outgoing signals to other neurons, eventually activating the network's output neurons, which return their outgoing signals as the output of the network as a whole.

In order to learn, neural networks must be trained: a network is fed input data and tasked with predicting the correct outputs from these inputs. These predictions are then evaluated against known correct outputs, and the network's internal parameters are adjusted to descend the error gradient towards more accurate predictions. This type of learning, in which input data are labeled with correct outputs, is called supervised learning. Barto and Dietterich describe these input-output pairs as "training examples," or examples of desired behavior (2).

However, neural network training can also be similar to dog training. This type of trial-and-error, reward-based learning, called reinforcement learning, is usually used when a database of correct input-output pairs is not available (Barto and Dietterich 4).

2.3 Neuroevolution

Neuroevolution is the application of genetic algorithms to train neural networks through a process of evolution ("Efficient Evolution of Neural Networks"). Neuroevolution algorithms

start with a population of randomly initialized genotypes, each genotype encoding a neural network phenotype. Then, the fitness function evaluates each one by testing its neural network phenotype. Next, the fittest genotypes are selected to reproduce and produce an offspring population, on which the process is repeated. Through iteration, this neuroevolutionary process produces more and more accurate neural networks. To continue with the Pavlovian dog analogy: rather than training a single dog to obey commands, neuroevolution takes a population of dogs, tests their ability to obey commands, and selectively breeds the most obedient ones.

A neuroevolution algorithm can evolve networks' connection weights, topology, or both. In neuroevolution algorithms that evolve only connection weights, networks' topologies are prespecified with unchanging numbers of connections and neurons. In algorithms that evolve network topology, neurons and connections are added and removed throughout the evolutionary process. Neuroevolution algorithms that evolve both weights and network topologies are called TWEANNs, or Topology and Weight Evolving Artificial Neural Network algorithms (Stanley and Miikkulainen).

Neuroevolution algorithms can also be classified by their use of direct encoding, in which a genotype directly specifies each neuron and connection of its phenotype, or indirect encoding, in which a genotype merely provides a set of rules for constructing its phenotype (Stanley and Miikkulainen). NEAT uses a direct encoding, while more advanced algorithms like HyperNEAT use an indirect encoding (Stanley and Miikkulainen; "A Hypercube-Based Encoding").

Neuroevolution is more widely applicable than strictly supervised learning, because neuroevolution algorithms utilize a fitness function to measure neural networks' performance at a task, rather than evaluating neural networks' accuracy against an entire database of inputs labeled with their correct outputs (Lehman and Miikkulainen). As such, neuroevolution is often classified as a type of reinforcement learning (Lowell et al.; Lehman and Miikkulainen).

2.4 NeuroEvolution of Augmenting Topologies (NEAT)

NeuroEvolution of Augmenting Topologies is a neuroevolution algorithm developed by Kenneth Stanley at the University of Texas at Austin in 2002 (Stanley and Miikkulainen). NEAT is considered "one of the most influential algorithms in the field" of neuroevolution (Papavasileiou et al.).

A NEAT genotype encodes a neural network as a list of *node genes* and another of *connection genes* (Stanley and Miikkulainen). Node genes, indexed by identification numbers, specify an input, output, or hidden neuron. Connection genes, indexed by "innovation numbers," specify a connection's in-node, out-node, weight, and whether or not it is disabled. New IDs and innovation numbers are assigned whenever mutations create new genes. In practice, innovation numbers can be represented by an ordered pair of the connection's in-node and out-node IDs, i.e. *(in-node, out-node)*.

NEAT allows mutations to network weights and topologies, making it a TWEANN (Stanley and Miikkulainen). Connection weights are mutated by perturbing every weight by a floating point number drawn randomly from a uniform distribution (Stanley and Miikkulainen).

NEAT also specifies a predefined probability that a connection weight will be not just perturbed, but completely replaced by a newly generated random value (Stanley and Miikkulainen).

Topological mutations change the size of the genotype by adding or removing genes (Stanley and Miikkulainen). “Add node” mutations add a new connection between two previously unconnected nodes (Stanley and Miikkulainen). “Add connection” mutations add a new node by splitting a connection, disabling it, and inserting the new node between the two previously-connected nodes (Stanley and Miikkulainen).

NEAT uses *historical markings* to enable the recombination of disparate topologies. During recombination, parents’ genes’ IDs are matched up, passing down only a single copy of each gene (Stanley and Miikkulainen). This technique enables NEAT to recombine genotypes in a way that preserves evolutionary progress, including neural “pathways” of connected neurons that have evolved to convey specific information through the network. Thus, NEAT elegantly circumvents the typical, inefficient approach of analyzing the parent topologies in order to calculate a way to recombine them (Stanley and Miikkulainen).

NEAT uses the biological principle of speciation to protect topological innovations (Stanley and Miikkulainen). Topological mutations can cause a new individual to perform poorly and die out before evolving potential usefulness. To protect these topological innovations, NEAT separates individuals so that they compete mainly within their species (Stanley and Miikkulainen).

The last major advantage of NEAT is its tendency to minimize topological complexity (Stanley and Miikkulainen). Because NEAT initializes its first-generation genotypes with the bare-minimum structure and only adds new neurons and connections when selectively favorable, NEAT usually finds a simpler solution to the problem it is applied to than other algorithms do (Stanley and Miikkulainen).

For these reasons, NEAT showed superior performance when it was evaluated on the standard reinforcement learning benchmark of “balancing two poles simultaneously” without providing velocity inputs to the network (Stanley and Miikkulainen).

2.5 NEAT Extensions

NEAT’s popularity has inspired many to create their own “extensions” of the algorithm. According to Kenneth Stanley, independent researchers have devised countless of these extensions that either expand upon the standard NEAT algorithm or bring it to a new platform or programming language (“NEAT Users Page”). A search on Github, “the world's leading software development platform,” of “neuroevolution of augmenting topologies” confirms this, bringing up nearly 300 repositories that each independently implement the NEAT algorithm (“258 Repository Results”). Likewise, in a comprehensive review of NEAT’s “successors,” Papavasileiou et al. found “232 publications” each proposing a new NEAT extension. While it would be inconvenient to list all these extensions, some examples include upgrades to the original NEAT algorithm, such as HyperNEAT, ES-HyperNEAT, and NEAT with novelty search; extensions for evolving generative art, including DelphiNEAT-based Genetic Art,

SharpNEAT-based Genetic Art, and the *Living Image Project*; and those for solving fractured problems: RBF-NEAT, Cascade-NEAT, and SNAP-NEAT ("NEAT Users Page"; Secretan et al.; Kohl and Miikkulainen; Kohl). Furthermore, according to Stanley, official implementations of NEAT exist in several platforms and languages, including but not limited to C++, Java, Matlab, C#, Windows C++, and Delphi ("NEAT Users Page"). Although, the NEAT Users Page lacks an official extension for the Python programming language ("NEAT Users Page").

One important NEAT extension is real-time NEAT, or rtNEAT, invented by Kenneth Stanley. While standard NEAT replaces its entire population of individuals every generation, rtNEAT replaces one individual agent at a time ("Real-Time Neuroevolution"). As a result, agents evolve in *real time*. Enemy AIs can be trained while a user is playing against them in a video game.

Many researchers have written the NEAT algorithm de novo either to devise a new extension, bring support to a new programming language, or simply to challenge themselves. While these implementations and the official libraries allow researchers to use and experiment with the NEAT algorithm, no library exists to enable researchers to invent new NEAT extensions. Thus, in an effort to contribute to the research effort surrounding NEAT and neuroevolution, this paper introduces a new Python library, called BaseNEAT, providing researchers and developers an interface to not only use but also extend the NEAT algorithm.

3 Methodology

Python was used as the programming language of choice for this endeavor, due to its general flexibility, versatility, and readability as a language, as well as its large user base and wide range of packages (Srinath 2). Despite Python's reputation for slow run times, it was chosen over other languages such as JavaScript, TypeScript, and Java owing to the researchers' greater familiarity with the Python language (Srinath 2).

This section describes the explorative process, beginning with researchers' attempt to create a custom neuroevolution algorithm in Python, which exposed limitations with the standard approach to neuroevolution. After realizing these limitations, researchers discarded the custom algorithm in favor of a popular Python NEAT library. Several experiments were run using the existing library to reveal the limitations that motivated the development of a new NEAT library. Finally, the implementation of the new library is explicated, tests of its efficacy are devised, and the results are presented.

3.1 Custom Neuroevolution Algorithm

To begin, an attempt was made to design a custom neuroevolution algorithm utilizing a fully-connected feedforward neural network design. The algorithm could perform substitution mutations by applying random noise over all the weights of the network and insertion and deletion mutations by adding/removing layers and neurons. However, this design was very limited, in three ways.

First, unlike traditional genetic algorithms, it did not implement a genotype. Instead,

mutations were applied directly onto the phenotype–that is, the neural networks themselves. It would have been ideal to encode networks as a genetic representation, allowing not only for the direct encoding of connections and neurons, but also for the biologically-inspired indirect encoding of features such as symmetry and locality within the network structure.

Second, the algorithm was not able to recombine two topologically-disparate neural networks, making it impossible to implement both variable network topologies and sexual reproduction/recombination–two crucial elements of an effective genetic algorithm–in the same run of the algorithm.

Third, the custom algorithm was generally limited in its ability to generate variable network topologies, as it was anchored to the layered and fully-connected structure characteristic of feedforward neural networks. Ideally, the algorithm would have allowed for connections to be added between any two neurons in the network. As it was, each neuron connected exclusively to all neurons of the subsequent layer, in the standard feed-forward style.

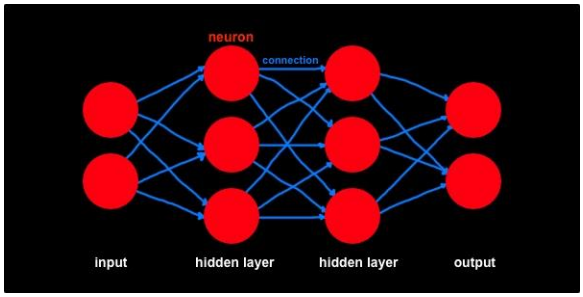

Figure 1. A fully-connected feed-forward neural network structure with two input neurons, two output neurons, and two hidden layers of three neurons each. As used in the custom algorithm, these networks vary only by their connection weight values, number of hidden layers, and number of neurons in each hidden layer.

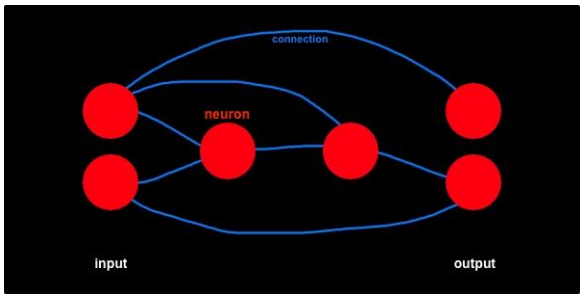

Figure 2. A NEAT-esque neural network structure with two inputs, two outputs, and a total of six neurons. This structure does not mandate full connection between neurons of adjacent layers, and also allows for connections between neurons of non-adjacent layers. In fact, "layers" are irrelevant in this structure, because individual neurons can freely connect to each other.

Although this algorithm served an important explorative role in helping to understand the specific methodology behind genetic algorithms, it was fatally limited in its practical use. As a

result, the algorithm was discarded in favor of NEAT, the tried-and-true, eminent neuroevolution algorithm.

3.2 NEAT-Python

In addition to a scarcity of alternatives, the NEAT-Python library was chosen for its "pure Python implementation of NEAT" in contrast to other libraries listed on the NEAT Software Catalog that were not designed for Python but provide added Python bindings ("Welcome to NEAT-Python's Documentation"; "Find the Right Version of NEAT for Your Needs"). Furthermore, NEAT-Python is listed as an official Python library on the Python Package Index, making it an ideal standard for comparison with any new Python NEAT library ("Neat-Python"). Several experiments were run to test NEAT-Python and evaluate the library's limitations. The two most significant experiments are expounded in this section.

3.2.1 Experiment 1: Evolving Virtual Organisms (EVO)

In Experiment 1, each individual agent in the NEAT algorithm represents a virtual organism. An organism interacts with its environment by sending information from its senses as inputs to its neural network, which then computes outputs that determine the organism's behavior. See Appendix B for a detailed listing of organisms' inputs and outputs

The simulation initializes with a population of inactive organisms which soon evolve movement in straight lines with inexplicable stops. More advanced movements develop, including spinning, purposeful movement towards other organisms, and stopping when too close to another organism. They keep their mouths open in order to eat other organisms by chance contact and tend to increase their radius when near other organisms, then decrease it when no others are nearby, likely because the simulation is designed such that the bigger organism will eat the other on collision. They also begin to close and open their mouths. Subsequent runs of the simulation exhibit variable effectiveness at generating successful organisms in early generations. Organisms evolve new behaviors, but seldom develop complex movement patterns like following or flocking.

Figure 3. An initial population in EVO.

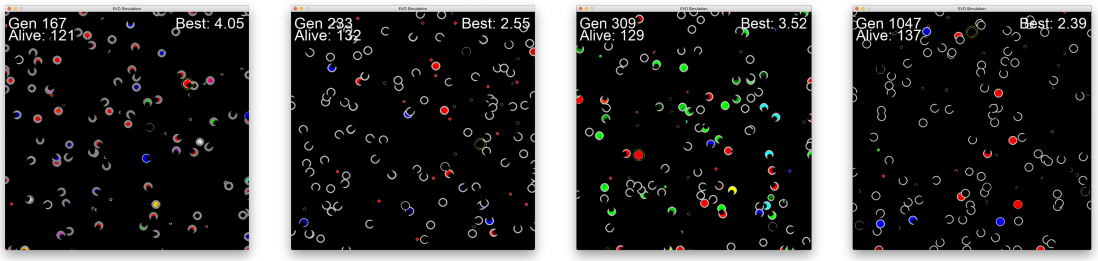

Figure 4. A population of creatures after evolving for 166, 232, 308, then 1,046 generations. Most creatures maintain maximum radius and open their mouths. At one point, a prominent species has decided not to change color, instead simply staying black. Then, a species of creatures who prefer green arises. Some differences in behavior are apparent, and gradual changes in color suggest that neural networks are evolving. The final generation exhibits a mouth-flashing effect, where they open and close their mouths rapidly, possibly to minimize the energy cost of keeping the mouth open.

This experiment clarified one important limitation of the NEAT-Python library: NEAT-Python does not allow the use of NEAT extensions at all, let alone enable researchers to create their own extensions. In fact, NEAT-Python's documentation states that support for extensions is "planned once the fundamental NEAT implementation is more complete and stable" ("Welcome to NEAT-Python's Documentation"). Thus, due to an incomplete design, NEAT-Python was unable to provide the tools necessary for extending the NEAT algorithm. While NEAT-Python's evolutionary algorithm demonstrated success in evolving new behaviors, it was only able to do so using standard NEAT; real-time NEAT and other extensions could not be implemented.

3.2.2 Experiment 2: Genetic Art

Experiment 2 took inspiration from the Picbreeder project, a website allowing online users to collaborate to evolve neural networks that produced art–a process called collaborative interactive evolution, or CIE (Secretan et al.).

Picbreeder's NEAT implementation used special neural networks called compositional pattern-producing networks, or CPPNs (Secretan et al.). CPPNs are applied over each coordinate of a 2-dimensional space, taking x and y coordinates as input and computing color values for each pixel (Secretan et al.). Internally, CPPNs are comprised of various functions "chosen from a canonical set"–in Picbreeder, they were sine, cosine, Gaussian, sigmoid, and identity–composed together in the structure of a connected-graph; i.e. a network (Secretan et al.).

While Picbreeder's collaborative interactive evolution was a type of neuroevolution, it did not use a fitness function to drive selection; instead, Picbreeder users interactively chose which CPPNs should survive and reproduce. Users could then publish their evolved CPPNs for other users to continue the process, constituting the the "collaborative" part of "collaborative interactive evolution."

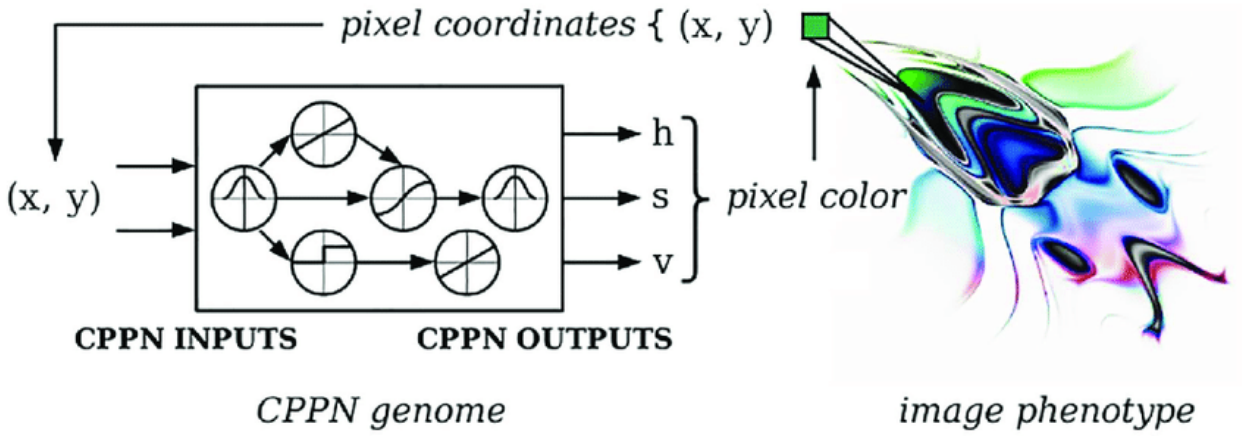

Figure 5: Image detailing the function and structure of compositional pattern-producing networks. For each pixel of a 2D space, a CPPN takes the x and y coordinates as input and feeds them through a network of functions defined by the CPPN's genome to produce an output representing the color of that pixel. Given the right genome, these can create complex patterns or even life-like images such as the one shown above.

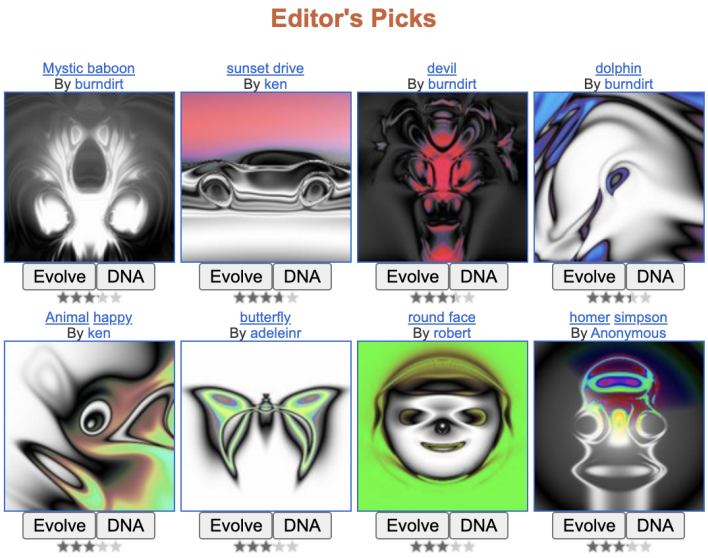

Figure 6: Screenshot of picbreeder.org, showing images created by online users' collaborative interactive evolution (CIE) during the Picbreeder project. These images have been evolved to illustrate complex, recognizable objects.

Picbreeder is an excellent example of a successful project applying NEAT, so it was chosen as a model for Experiment 2. Because Picbreeder's results can be clearly seen and evaluated by visual quality, they are easily compared with the results of this experiment, which can thereafter be easily compared to results of subsequent experiments using any new library. Within Experiment 2, two separate experiments were done: ART1 and ART2.

3.2.2.1 Genetic Art v1 (ART1)

ART1 attempted to emulate the Picbreeder project. Similarly to Picbreeder, in ART1, CPPNs generated images to be evaluated by a user; however, this experiment used single-user interactive evolutionary computation (IEC), rather than collaboration between multiple users. However, this project resulted in much less successful art pieces than Picbreeder, possibly because the canonical set of activation functions was too large. Additionally, since ART1 used matplotlib, a library for "creating static, animated, and interactive visualizations in Python," for image visualization, it was very slow ("Visualization with Python").

3.2.2.2 Genetic Art v2 (ART2)

ART2 improved upon ART1 by limiting the canonical set to sine, Gaussian, identity, sigmoid, cube, exp, hat, inv, log, softplus, square, and tanh, implementing new network inputs, and shifting to PyGame, a library for writing video games, for increased performance ("Pygame"). Rather than selecting a single winning image, ART2 users could select multiple images and click multiple times to control the fitness level of an image. With the faster visualization engine, larger image sizes were possible for easier viewing. In contrast to ART1, which used only two x and y inputs, ART2's CPPNs received five inputs as suggested by Hintze and Schossau:

1. X' = X normalized to [-pi, pi] range
2. Y' = Y normalized to [-pi, pi] range
3. Euclidian distance from (0, 0) to (X', Y')
4. Cosine X'
5. Sine Y'

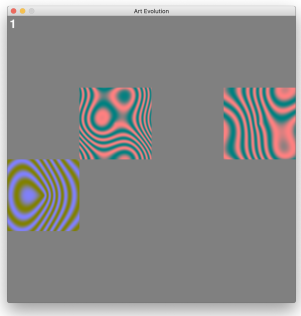

Figure 7. In this run, the first generation to produce viable images (shown above) was Generation 1 (the second generation, after Generation 0).

Figure 8. Three consecutive generations of ART2 evolution, showing images produced by CPPNs. The winning images, selected by the user, are outlined in yellow, with greater thickness indicating greater fitness. In this example, it is easy to see how the aesthetics of the images in one generation are greatly determined by the previous generation's winners. Slight as well as significant mutations can be observed.

ART2 showed much better aesthetic results than ART1, and possessed the capability to produce images that represented real objects such as scratch marks; however, it has not been observed to reach the advanced aesthetic capabilities seen in the Picbreeder project.

The failure of ART2 to produce comparable results to Picbreeder is likely due to a few key limitations. First, since Picbreeder takes advantage of collaborative rather than single-user interactive evolution, it is less limited by the effects of user fatigue, in which users lose interest before ever producing any viable results (Secretan et al.). Picbreeder images such as those displayed in Figure 6 are the result of many generations of evolution, while ART2's images resulted from only a few generations. If given enough generations, ART2 may produce similarly interesting results.

The importance of this experiment was in providing a replicable simulation for comparing a new NEAT library with the existing NEAT-Python library.

3.2.3 NEAT-Python Limitations

As shown by Experiment 1, NEAT-Python is limited by its inability to implement anything other than the standard NEAT algorithm. Extensions are not supported, and modifying the algorithm itself is impossible using NEAT-Python's API. Additionally, NEAT-Python does not allow for modifications, e.g. multiple independently-evolving populations in the same simulation environment, precluding the simulation of certain artificial ecosystems. Thus, a new NEAT library that allows for extensions like real-time NEAT and modifications like support for multiple populations would greatly widen the capabilities of the NEAT algorithm for researchers using Python.

3.3 Introducing BaseNEAT

In order to create this new library, NEAT was implemented from scratch in Python, then reworked into a more modular base library that allows for new NEAT extensions and modifications, including new gene types–e.g. body color genes–speciation algorithms, properties of individual agents–e.g. x and y coordinates–and selection processes.

First, the generational evolution algorithm was implemented and tested on its ability to generate genotypes, succeeding at a test of maximizing genotypes' ratio of connections to neurons. Then, the code was reorganized into a base module containing the core NEAT implementation and separate modules for NEAT extensions. The extension modules could interface with the base module by importing its API, thus enabling the developer to create new NEAT extensions. See Appendix A for the full source code.

After the library was completed, several tests were run to demonstrate its functionality and efficacy.

3.4 Testing & Results

This section describes the testing methodology and results that followed the construction of the new library. First, a qualitative comparison was made between BaseNEAT and NEAT-Python's results on the genetic art experiment described in section 3.2.2. Then, BaseNEAT was evaluated quantitatively using benchmarks from Stanley's original paper. Finally, mixed methods were utilized to evaluate BaseNEAT's usability for creating new NEAT extensions.

3.4.1 Qualitative Comparison: BaseNEAT versus NEAT-Python

Code from section 3.2.2's genetic art experiment using NEAT-Python was adapted to interface with BaseNEAT's API in order to retest the same experiment using the new BaseNEAT library. See Appendix A for the full source code.

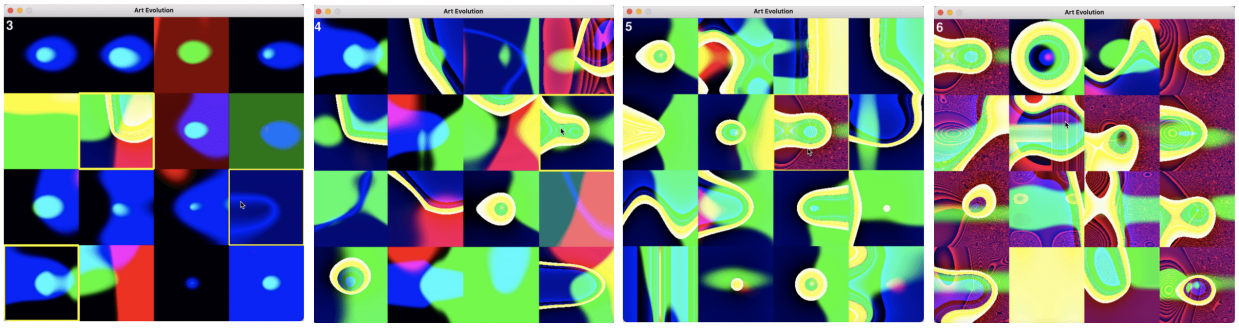

Figure 9. Four consecutive generations of genetic art in BaseNEAT. BaseNEAT shows comparable results to NEAT-Python (Figure 8) in the same genetic art experiment.

Results exhibited visual qualities of repeating patterns and symmetry similar to those seen in the NEAT-Python experiment. Additionally, BaseNEAT's evolutionary algorithm was

proven to be functional: images selected by the user passed down their visual characteristics to the next generation. This experiment demonstrated the functionality of BaseNEAT's standard NEAT implementation by comparing its genetic art experimental results with those of NEAT-Python.

3.4.2 Quantitative Comparison: BaseNEAT versus Original NEAT

To compare BaseNEAT to the original NEAT, two benchmark tests were performed: the XOR and pole-balancing tests. These tests were chosen for their use in the original NEAT paper as well as their popularity as benchmarks in the artificial intelligence community. All tests used a population size of 150 individuals, same as in the original NEAT paper (Stanley and Miikkulainen).

3.4.2.1 XOR Test

XOR, or Exclusive-OR, is a logical operation that returns true if and only if its inputs differ, i.e. one is true and the other is false. According to Stanley, XOR is not "linearly separable," meaning that in order to solve it, networks must evolve a hidden neuron (Stanley and Miikkulainen). This makes XOR ideal for testing the ability to evolve structure (Stanley and Miikkulainen). To test XOR, 50 trials were run measuring the number of generations needed for BaseNEAT to sufficiently pass the test.

Input		Output
A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Figure 10. A labeled dataset of input-output pairs for the XOR problem. Shown above, there are only four possible sets of inputs, and the values 1 and 0 represent true and false.

BaseNEAT was unable to pass the XOR test unless allowed hundreds or sometimes thousands of generations of evolution, suggesting that BaseNEAT's ability to evolve network structure is somewhat crippled compared to the original NEAT algorithm, which scored an average of 32 generations to pass the test over 100 runs (Stanley and Miikkulainen).

3.4.2.2 Pole-Balancing Test

The single pole-balancing test evaluates neural networks on their ability to balance a pole by moving the cart upon which the pole is attached (Stanley and Miikkulainen). However, in the original NEAT paper, single pole-balancing was discarded as "too easy for modern methods," and double pole-balancing was used instead (Stanley and Miikkulainen). The double pole-balancing test evaluates neural networks on their ability to simultaneously balance two

poles attached to the same cart (Stanley and Miikkulainen). Thus, both the single and double pole-balancing tests were used to evaluate BaseNEAT against the performance of the original NEAT algorithm.

The Python OpenAI Gym library, "a toolkit for developing and comparing reinforcement learning algorithms," was used to ensure the replicability and accuracy of the pole-balancing test environment (*OpenAI Gym*). Fifty trials were run measuring the number of generations needed to sufficiently pass each test.

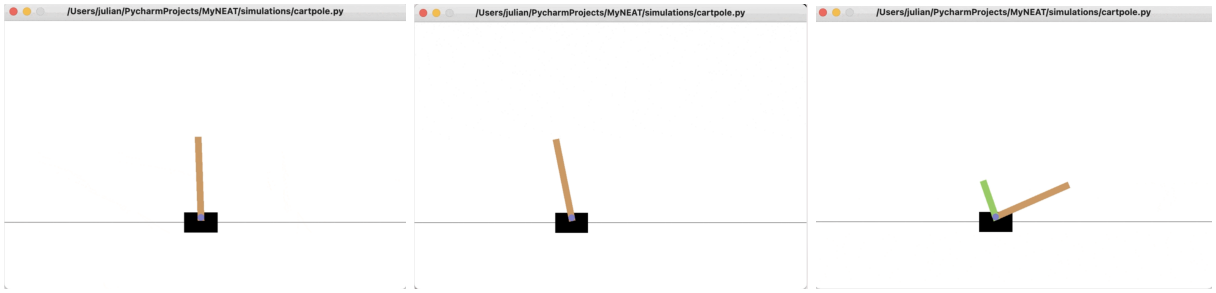

Figure 11. A relatively balanced pole (left), a pole falling to gravity as the neural network fails to keep it upright (middle), and a neural network failing the double pole-balancing test (right).

BaseNEAT was unable to pass the double pole-balancing test in a reasonable amount of time, likely due to the same deficiencies in structural evolution revealed by the XOR test.

However, it took an average of 1.0 generations with velocity inputs and 1.04 generations without velocity inputs for BaseNEAT to pass the single pole-balancing test, affirming that the single pole-balancing test is extremely easy, but also showing that BaseNEAT is still functional despite its specific deficiencies.

3.4.3 Mixed Method Comparison: Ease of Extension Using BaseNEAT

While NEAT-Python does not support extensions, it was possible to implement multiple different extensions in BaseNEAT ("Welcome to NEAT-Python's Documentation"). Standard NEAT was implemented in less than 25 lines of additional code, real-time NEAT was added in less than 50 lines, and a custom extension was built in ~250 lines. See Appendix A for the full source code.

```
from neat.base.nn import RecurrentNetwork
from neat.base import BasePopulation, BaseSpeciesSet, BaseSpecies, BaseAgent, BaseGenome
class Population(BasePopulation):
    def evaluate(self, evaluate_fitness):
        self.species_set.speciate(self.agents)
        for agent in self.agents.values():
            agent.fitness = evaluate_fitness(agent)
            if self.fittest is None or agent.fitness > self.fittest.fitness:
                self.fittest = agent
        next_gen_genomes = []
        for species in self.species_set.index.values():
            fittest_in_species = max(species.members, key=lambda m: m.fitness)
            next_gen_genomes.append(fittest_in_species.genome)
        while len(next_gen_genomes) < self.config.pop_size:
            parent_species = self.species_set.random_species()
            parent1, parent2 = parent_species.random_members(2)
            if parent1.fitness < parent2.fitness:
                parent2, parent1 = parent1, parent2
            child_genome = self.genome_type(next(self.gid_indexer), self.config)
            child_genome.crossover(parent1.genome, parent2.genome)
            child_genome.mutate()
            next_gen_genomes.append(child_genome)
        return next_gen_genomes
```

```
from neat.base import BasePopulation, BaseSpeciesSet, BaseSpecies, BaseAgent, BaseGenome
from .config import Config
class Population(BasePopulation):
    def __init__(self, config: Config, species_set_type=BaseSpeciesSet, species_type=BaseSpecies,
                 agent_type=BaseAgent, genome_type=BaseGenome):
        super().__init__(config, species_set_type=species_set_type, species_type=species_type,
                         agent_type=agent_type, genome_type=genome_type)
        self.replacements = 0
    def do_replacement(self):
        eligible_agents = sorted(filter(lambda a: a.age >= self.config.minimum_age, self.agents.values()),
                                 key=lambda a: a.adjusted_fitness())
        if len(eligible_agents) == 0:
            return
        worst = eligible_agents[0]
        self.agents.remove(worst)
        parent_species = self.species_set.random_species()
        parent1, parent2 = parent_species.random_members(2)
        if parent1.fitness < parent2.fitness:
            parent2, parent1 = parent1, parent2
        child_genome = self.genome_type(next(self.gid_indexer), self.config)
        child_genome.crossover(parent1.genome, parent2.genome)
        child_genome.mutate()
        child = self.agent_type(child_genome)
        self.agents.add(child)
        parent_species.add(child)
    def do_reorganization(self):
        self.species_set.adjust_compat_threshold()
        self.species_set.speciate(self.agents, self.ticks)
        self.species_set.remove_empty()
    def update(self):
        self.fittest = max(self.agents.values(), key=lambda a: a.fitness)
        self.do_stagnation()
        if self.config.reset_on_extinction and len(self.agents) == 0:
            self.init()
            print("Reset on total extinction")
        if self.ticks % self.config.replacement_frequency == 0:
            self.do_replacement()
            print("Replacement")
            if self.replacements % self.config.reorganization_frequency == 0:
                self.do_reorganization()
                print("Reorganization")
            self.replacements += 1
        self.ticks += 1
```

Figure 12. Standard NEAT implemented in <25 lines of code (left) and real-time NEAT in <50 lines (right) using the BaseNEAT library.

4 Conclusion

As shown by the results of BaseNEAT, the NEAT algorithm can be consolidated into its basic form such that new NEAT extensions can be created simply by interfacing with the base code library. Discussion of implications, limitations, and further research for this project follows.

4.1 Implications

BaseNEAT acts as a proof-of-concept, showing that it is possible to create an even more expansive neuroevolution toolkit, and provides a much-needed alternative Python NEAT library. In the future, BaseNEAT can be published for researchers and novices alike to learn about NEAT and modify the core algorithm in order to create new neuroevolution algorithms. New neuroevolution algorithms enable researchers to evolve more intelligent agents, and smarter AI assists and empowers human intelligence and creativity. Thus, BaseNEAT is a small step towards building more useful algorithms that can aid productivity and enhance the human experience.

BaseNEAT also counters the common method among researchers of reimplementing NEAT from scratch, showing that researchers can create their own extensions of NEAT simply by extending off of a base library.

4.2 Limitations

BaseNEAT has several limitations that must be addressed in future research. According to Stanley and Miikkulainen, the XOR problem is limited in its usefulness as a test, because its methodologies vary between different researchers (16). Additionally, BaseNEAT's poor performance on the XOR and double pole-balancing benchmark tests reveals a possible deficiency in its ability to evolve neural network structure. Because NEAT's main advantage is in its ability to evolve network structure, this deficiency greatly limits the potential of BaseNEAT as a NEAT library. Finally, as BaseNEAT is written in Python, it is not compatible with other languages.

BaseNEAT is also limited due to the difficulty of gauging demand in the research community: do researchers really need a library like BaseNEAT? This paper assumes that BaseNEAT fills the need for a library that focuses on enabling researchers to create new NEAT extensions.

4.3 Future Research

More extensive testing can be done to ensure the completeness of BaseNEAT's NEAT implementation. For example, a shooter simulation can be devised to test BaseNEAT's real-time NEAT extension on a video game environment similar to the one which real-time NEAT was originally designed for. The evolving virtual organisms experiment described in section 3.2.1 can also be recreated in BaseNEAT.

Despite BaseNEAT's deficiency in the XOR and double pole-balancing tests, judging by its success in the single pole-balancing test, the deficiency only affects structural evolution, meaning that further research can be done to address and solve the specific problem without too much trouble. Resolving this issue should be the next main step for BaseNEAT, although future testing may opt for a different, more standardized test than XOR.

When BaseNEAT is ready to be released, it can be published on the Python Package Index for other Python developers and researchers to utilize in their own work. Furthermore, advanced NEAT extensions like HyperNEAT, ES-HyperNEAT, and novelty search can be implemented to demonstrate BaseNEAT's wider effectiveness and begin exploring the possibility of inventing even more advanced neuroevolution algorithms using BaseNEAT.

Works Cited

"258 Repository Results." *GitHub*, github.com/search?p=8&q=%22neuroevolution%2Bof%2Baugmenting%2Btopologies%22&type=Repositories.

Aguilar, Wendy, et al. "The Past, Present, and Future of Artificial Life." *Frontiers*, Frontiers, 19 Sept. 2014, www.frontiersin.org/articles/10.3389/frobt.2014.00008/full.

Barto, Andrew G, and Thomas G Dietterich. *Reinforcement Learning and Its Relationship to Supervised Learning*. web.engr.orst.edu/~tgd/publications/Barto-Dietterich-03.pdf.

De Jong, Kenneth Alan. *Analysis of the Behavior of a Class of Genetic Adaptive Systems*, 1 Jan. 1975, deepblue.lib.umich.edu/handle/2027.42/4507.

Goldberg, David E. "Genetic Algorithms in Search, Optimization, and Machine Learning." Choice Reviews Online, vol. 27, no. 02, 1989, doi:10.5860/choice.27-0936.

Haykin, Simon. *Neural Networks: A Comprehensive Foundation*.

Hintze, Arend, and Jory Schossau. "Sexual Selection Compared to Novelty Search." *The 2020 Conference on Artificial Life*, 2020, doi:10.1162/isal_a_00275.

Kohl, Nate F. "Learning in Fractured Problems with Constructive Neural Network Algorithms." *TexasScholarWorks*, 1 Aug. 2009, repositories.lib.utexas.edu/handle/2152/10658.

Kohl, Nate, and Risto Miikkulainen. "Evolving Neural Networks for Strategic Decision-Making Problems." *Neural Networks*, vol. 22, no. 3, 2009, pp. 326–337., doi:10.1016/j.neunet.2009.03.001.

Lehman, Joel, and Risto Miikkulainen. "Neuroevolution." *Scholarpedia*, vol. 8, no. 6, 2013, p. 30977., doi:10.4249/scholarpedia.30977.

Lowell, Jessica, et al. "Comparison of NEAT and HyperNEAT Performance on a Strategic Decision-Making Problem." *2011 Fifth International Conference on Genetic and Evolutionary Computing*, 2011, doi:10.1109/icgec.2011.33.

"Neat-Python." *PyPI*, Aug. 2017, pypi.org/project/neat-python/.

OpenAI Gym, gym.openai.com/.

Papavasileiou, Evgenia, et al. "A Systematic Literature Review of the Successors of 'NeuroEvolution of Augmenting Topologies.'" *Evolutionary Computation*, vol. 29, no. 1, 2021, pp. 1–73., doi:10.1162/evco_a_00282.

"Pygame." *PyPI*, pypi.org/project/pygame/.

Secretan, Jimmy, et al. "Picbreeder: A Case Study in Collaborative Evolutionary Exploration of Design Space." *Evolutionary Computation*, vol. 19, no. 3, 2011, pp. 373–403., doi:10.1162/evco_a_00030.

Srinath, K R. *Python – The Fastest Growing Programming Language*. www.irjet.net/archives/V4/i12/IRJET-V4I1266.pdf.

Stanley, Kenneth O, et al. "Real-Time Neuroevolution in the NERO Video Game." *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 6, 2005, pp. 653–668., doi:10.1109/tevc.2005.856210.

Stanley, Kenneth O. "Efficient Evolution of Neural Networks Through Complexification." *Neural Network Research Group*, nn.cs.utexas.edu/keyword?stanley%3Aphd04.

Stanley, Kenneth O. "Find the Right Version of NEAT for Your Needs." *NEAT Software Catalog*, eplex.cs.ucf.edu/neat_software/.

Stanley, Kenneth O. "Neuroevolution: A Different Kind of Deep Learning." *O'Reilly Media*, 13 July 2017, www.oreilly.com/radar/neuroevolution-a-different-kind-of-deep-learning/.

Stanley, Kenneth O. "The NeuroEvolution of Augmenting Topologies (NEAT) Users Page." *NeuroEvolution of Augmenting Topologies*, www.cs.ucf.edu/~kstanley/neat.html.

Stanley, Kenneth O., and Risto Miikkulainen. "Evolving Neural Networks through Augmenting Topologies." *Evolutionary Computation*, vol. 10, no. 2, 2002, pp. 99–127., doi:10.1162/106365602320169811.

Stanley, Kenneth O., et al. "A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks." *Artificial Life*, vol. 15, no. 2, 2009, pp. 185–212., doi:10.1162/artl.2009.15.2.15202.

"Visualization with Python." *Matplotlib*, matplotlib.org/.

"Welcome to NEAT-Python's Documentation!" *NEAT-Python*, neat-python.readthedocs.io/en/latest/.

Appendix A: Source Code

Source code hosted at: https://github.com/puffyboa/MyNEAT

Appendix B: Detailed Inputs & Outputs

Given a hypothetical Organism A, (x_A, y_A) represents the position of Organism A and (x_B, y_B) represents the position of the organism physically closest to Organism A, called Organism B.

Organism A's nine inputs would be as follows:

1. Constant 1.0
2. Random float [-1.0, 1.0]
3. Hunger of A [0.0, 1.0]
4. Radius of A divided by radius of B
5. Euclidean distance between (x_A, y_A) and (x_B, y_B) divided by A's radius
6. Difference between rotation of A and angle of the line that passes through (x_A, y_A) and (x_B, y_B)
7. Red value of B [0.0, 255.0]
8. Green value of B [0.0, 255.0]
9. Blue value of B [0.0, 255.0]

Organism A's network would then return seven outputs:

1. Forward movement
2. Rotational movement
3. Red value of A [0.0, 255.0]
4. Green value of A [0.0, 255.0]
5. Blue value of A [0.0, 255.0]
6. Radius of A [5.0, 15.0]
7. Mouth open, yes if >0.5, else no (allows eating, but costs energy)