# NEAT-Python Documentation

***Release 0.1***

**CodeReclaimers, LLC**

February 27, 2016

Contents

NEAT (NeuroEvolution of Augmenting Topologies) is a method developed by Kenneth O. Stanley for evolving arbitrary neural networks. NEAT-Python is a Python implementation of NEAT.

The core NEAT implementation is currently pure Python with no dependencies other than the Python standard library. The *visualize* module requires graphviz, NumPy, and matplotlib, but it is not necessary to install these packages unless you want to make use of these visualization utilities. Some of the examples also make use of other libraries.

If you need an easy performance boost, JIT-enabled PyPy does a fantastic job, and may give you a ~10x speedup over CPython. Built-in C and OpenCL network implementations will be added later to speed up applications for which network evaluation is the primary bottleneck.

Support for HyperNEAT and other extensions to NEAT will also be added once the fundamental NEAT implementation is more complete and stable.

**Please note:** the package and its usage may change significantly while it is still in alpha status. Updating to the most recent version is almost certainly going to break your code until the version number approaches 1.0.

For further information regarding general concepts and theory, please see Selected Publications on Stanley's website, or his recent AMA on Reddit.

If you encounter any confusing or incorrect information in this documentation, please open an issue in the GitHub project.

Contents:

CHAPTER 1

# NEAT Overview

NEAT (NeuroEvolution of Augmenting Topologies) is an evolutionary algorithm that creates artificial networks. For a detailed description of the algorithm, you should probably go read some of Stanley's papers on his website.

Even if you just want to get the gist of the algorithm, reading at least a couple of the early NEAT papers is a good idea. Most of them are pretty short (8 pages or fewer), and do a good job of explaining concepts (or at least pointing you to other references that will).

In the current implementation of NEAT-Python, a population of individual genomes is maintained. Each genome contains two sets of genes that describe how to build an artificial neural network:

1. Node genes, each of which specifies a single neuron.
2. Connection genes, each of which specifies a single connection between neurons.

To evolve a solution to a problem, the user must provide a fitness function which computes a single real number indicating the quality of an individual genome: better ability to solve the problem means a higher score. The algorithm progresses through a user-specified number of generations, with each generation being produced by reproduction (either sexual or asexual) and mutation of the most fit individuals of the previous generation.

The reproduction and mutation operations may add nodes and/or connections to genomes, so as the algorithm proceeds genomes (and the neural networks they produce) may become more and more complex. When the preset number of generations is reached, or when at least one individual exceeds the user-specified fitness threshold, the algorithm terminates.

CHAPTER 2

# Installation

Because the library is still changing fairly rapidly, attempting to run examples with a significantly newer or older version of the library will result in errors. It is best to install and get the examples using one of the two methods outlined below.

## 2.1 From PyPI using pip

To install the most recent release (version 0.6) from PyPI, you should run the command (as root or using *sudo* as necessary):

```
pip install neat-python
```

Note that the examples are not included with the package installed from PyPI, so you should download the source archive for release 0.6 and use the example code contained in it.

You may also just download the 0.6 release source, and install it directly using *setup.py* (as shown below) instead of *pip*.

## 2.2 From source using setup.py

Obtain the source code by either cloning the source repository:

```
git clone https://github.com/CodeReclaimers/neat-python.git
```

or downloading the source archive for release 0.6.

Note that the most current code in the repository may not always be in the most polished state, but I do make sure the tests pass and that most of the examples run. If you encounter any problems, please open an issue on GitHub.

To install from source, simply run:

```
python setup.py install
```

from the directory containing setup.py.

CHAPTER 3

# Configuration file format

The configuration file is in the format described in the Python ConfigParser documentation. Currently, all values must be explicitly enumerated in the configuration file. This makes it less likely that code changes will result in your project silently using different NEAT settings.

## 3.1 [phenotype] section

- ***input_nodes*** The number of nodes through which the network receives input.
- ***hidden_nodes*** The number of hidden nodes to add to each genome in the initial population.
- ***output_nodes*** The number of nodes to which the network delivers output.
- ***initial_connection*** Specifies the initial connectivity of newly-created genomes. There are three allowed values:
  - *unconnected* - No connection genes are initially present.
  - *fs_neat* - One connection gene from one input to all hidden and output genes. (This is the FS-NEAT scheme.)
  - *fully_connected* - Each input gene is connected to all hidden and output genes, and each hidden gene is connected to all output genes.
- ***max_weight, min_weight*** Connection weights (as well as node bias and response) will be limited to this range.
- ***feedforward*** If this evaluates to **True**, generated networks will not be allowed to have recurrent connections. Otherwise they may be (but are not forced to be) recurrent.
- ***activation_functions*** A space-separated list of the activation functions that may be used in constructing networks. Allowable values are: *abs*, *clamped*, *exp*, *gauss*, *hat*, *identity*, *inv*, *log*, *relu*, *sigmoid*, *sin*, and *tanh*. The implementation of these functions can be found in the nn module.
- ***weight_stdev*** The standard deviation of the zero-centered normal distribution used to generate initial and replacement weights.

## 3.2 [genetic] section

- ***pop_size*** The number of individuals in each generation.
- ***max_fitness_threshold*** When at least one individual's measured fitness exceeds this threshold, the evolution process will terminate.

- ***prob_add_conn*** The probability that mutation will add a connection between existing nodes. Valid values are on [0.0, 1.0].
- ***prob_add_node*** The probability that mutation will add a new hidden node into an existing connection. Valid values are on [0.0, 1.0].
- ***prob_delete_conn*** The probability that mutation will delete an existing connection. Valid values are on [0.0, 1.0].
- ***prob_delete_node*** The probability that mutation will delete an existing hidden node and any connections to it. Valid values are on [0.0, 1.0].
- ***prob_mutate_bias*** The probability that mutation will change the bias of a node by adding a random value.
- ***bias_mutation_power*** The standard deviation of the zero-centered normal distribution from which a bias change is drawn.
- ***prob_mutate_response*** The probability that mutation will change the response of a node by adding a random value.
- ***response_mutation_power*** The standard deviation of the zero-centered normal distribution from which a response change is drawn.
- ***prob_mutate_weight*** The probability that mutation will change the weight of a connection by adding a random value.
- ***prob_mutate_activation*** The probability that mutation will change the activation function of a hidden or output node.
- ***prob_replace_weight*** The probability that mutation will replace the weight of a connection with a new random value.
- ***weight_mutation_power*** The standard deviation of the zero-centered normal distribution from which a weight change is drawn.
- ***prob_toggle_link*** The probability that the enabled status of a connection will be toggled.
- ***elitism*** The number of most fit individuals in each species that will be preserved as-is from one generation to the next.
- ***reset_on_extinction*** If this evalutes to **True**, when all species simultaneously become extinct due to stagnation, a new random population will be created. If **False**, a *CompleteExtinctionException* will be thrown.

## 3.3 [genotype compatibility] section

- ***compatibility_threshold*** Individuals whose genomic distance is less than this threshold are considered to be in the same species.
- ***excess_coefficient*** The coefficient for the excess gene count's contribution to the genomic distance.
- ***disjoint_coefficient*** The coefficient for the disjoint gene count's contribution to the genomic distance.
- ***weight_coefficient*** The coefficient for the average weight difference's contribution to the genomic distance.

## 3.4 [species] section

- ***survival_threshold*** The fraction for each species allowed to reproduce on each generation.
- ***max_stagnation*** Species that have not shown improvement in more than this number of generations will be considered stagnant and removed.

CHAPTER 4

# Overview of the basic XOR example (xor2.py)

The xor2.py example, shown in its entirety at the bottom of this page, evolves a network that implements the two-input XOR function:

| Input 1 | Input 2 | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

## 4.1 Fitness function

The key thing you need to figure out for a given problem is how to measure the fitness of the genomes that are produced by NEAT. Fitness is expected to be a Python float value. If genome A solves your problem more successfully than genome B, then the fitness value of A should be greater than the value of B. The absolute magnitude and signs of these fitnesses are not important, only their relative values.

In this example, we create a feed-forward neural network based on the genome, and then for each case in the table above, we provide that network with the inputs, and compute the network's output. The error for each genome is 1 minus the root mean square difference between the expected and actual outputs, so that if the network produces exactly the expected output, its fitness is 1, otherwise it is a value less than 1, with the fitness value decreasing the more incorrect the network responses are.

This fitness computation is implemented in the `eval_fitness` function. The single argument to this function is a list of genomes in the current population. neat-python expects the fitness function to calculate a fitness for each genome and assign this value to the genome's `fitness` member.

## 4.2 Running NEAT

Once you have implemented a fitness function, you mostly just need some additional boilerplate code that carries out the following steps:

- Create a `neat.config.Config` object from the configuration file (described in Configuration file format).
- Create a `neat.population.Population` object using the `Config` object created above.
- Call the `epoch` method on the `Population` object, giving it your fitness function and the maximum number of generations you want NEAT to run.

After these three things are completed, NEAT will run until either you reach the specified number of generations, or at least one genome achieves the `max_fitness_threshold` value you specified in your config file.

## 4.3 Getting the results

Once the call to the population object's `epoch` method has returned, a list of the most fit genome for each generation is available as the `most_fit_genomes` member of the population. We take the 'winner' genome as the last genome in this list.

A list of the average fitness for each generation is also available as `avg_fitness_scores`.

## 4.4 Visualizations

Functions are available in the `neat.visualize` module to plot the best and average fitness vs. generation, plot the change in species vs. generation, and to show the structure of a network described by a genome.

## 4.5 Example Source

NOTE: This page shows the source and configuration file for the current version of neat-python available on GitHub. If you are using the version 0.6 installed from PyPI, make sure you get the script and config file from the archived source for that release.

Here's the entire example:

```
""" 2-input XOR example """
from __future__ import print_function

from neat import nn, population, statistics, visualize

# Network inputs and expected outputs.
xor_inputs = [[0, 0], [0, 1], [1, 0], [1, 1]]
xor_outputs = [0, 1, 1, 0]


def eval_fitness(genomes):
    for g in genomes:
        net = nn.create_feed_forward_phenotype(g)

        sum_square_error = 0.0
        for inputs, expected in zip(xor_inputs, xor_outputs):
            # Serial activation propagates the inputs through the entire network.
            output = net.serial_activate(inputs)
            sum_square_error += (output[0] - expected) ** 2

        # When the output matches expected for all inputs, fitness will reach
        # its maximum value of 1.0.
        g.fitness = 1 - sum_square_error


pop = population.Population('xor2_config')
pop.run(eval_fitness, 300)

print('Number of evaluations: {0}'.format(pop.total_evaluations))

# Display the most fit genome.
winner = pop.statistics.best_genome()
print('\nBest genome:\n{!s}'.format(winner))
```

```
# Verify network output against training data.
print('\nOutput:')
winner_net = nn.create_feed_forward_phenotype(winner)
for inputs, expected in zip(xor_inputs, xor_outputs):
    output = winner_net.serial_activate(inputs)
    print("expected {0:1.5f} got {1:1.5f}".format(expected, output[0]))

# Visualize the winner network and plot/log statistics.
visualize.plot_stats(pop.statistics)
visualize.plot_species(pop.statistics)
visualize.draw_net(winner, view=True, filename="xor2-all.gv")
visualize.draw_net(winner, view=True, filename="xor2-enabled.gv", show_disabled=False)
visualize.draw_net(winner, view=True, filename="xor2-enabled-pruned.gv", show_disabled=False, prune_u
statistics.save_stats(pop.statistics)
statistics.save_species_count(pop.statistics)
statistics.save_species_fitness(pop.statistics)
```

and here is the associated config file:

```
#--- parameters for the XOR-2 experiment ---#

# The `Types` section specifies which classes should be used for various
# tasks in the NEAT algorithm.  If you use a non-default class here, you
# must register it with your Config instance before loading the config file.
[Types]
stagnation_type      = DefaultStagnation
reproduction_type    = DefaultReproduction

[phenotype]
input_nodes          = 2
hidden_nodes         = 0
output_nodes         = 1
initial_connection   = unconnected
max_weight           = 30
min_weight           = -30
feedforward          = 1
activation_functions = sigmoid
weight_stdev         = 1.0


[genetic]
pop_size                = 150
max_fitness_threshold   = 0.95
prob_add_conn           = 0.988
prob_add_node           = 0.085
prob_delete_conn        = 0.146
prob_delete_node        = 0.0352
prob_mutate_bias        = 0.0509
bias_mutation_power     = 2.093
prob_mutate_response    = 0.1
response_mutation_power = 0.1
prob_mutate_weight      = 0.460
prob_replace_weight     = 0.0245
weight_mutation_power   = 0.825
prob_mutate_activation  = 0.0
prob_toggle_link        = 0.0138
reset_on_extinction     = 1
```

```
[genotype compatibility]
compatibility_threshold = 3.0
excess_coefficient      = 1.0
disjoint_coefficient    = 1.0
weight_coefficient      = 0.4

[DefaultStagnation]
species_fitness_func = mean
max_stagnation       = 15

[DefaultReproduction]
elitism              = 1
survival_threshold   = 0.2
```

CHAPTER 5

# Customizing Behavior

NEAT-Python allows the user to provide drop-in replacements for some parts of the NEAT algorithm, and attempts to allow easily implementing common variations of the algorithm mentioned in the literature. If you find that you'd like to be able to customize something not shown here, please submit an issue on GitHub.

## 5.1 Adding new activation functions

To register a new activation function, you simply need to call *neat.activation_functions.add* with your new function and the name by which you want to refer to it in the configuration file:

```
def sinc(x):
    return 1.0 if x == 0 else sin(x) / x

neat.activation_functions.add('my_sinc_function', sinc)
```

This is demonstrated in the *memory* example.

## 5.2 Reproduction scheme

The default reproduction scheme uses explicit fitness sharing and a fixed species stagnation limit. This behavior is encapsulated in the DefaultReproduction class.

TODO: document, include example

## 5.3 Speciation

If you need to change the speciation scheme, you should subclass *Population* and override the *_speciate* method.

## 5.4 Species stagnation

To use a different species stagnation scheme, you can create a custom class whose interface matches that of *FixedStagnation* and set the *stagnation_type* of your Config instance to this class.

TODO: document, include example

## 5.5 Diversity

To use a different diversity scheme, you can create a custom class whose interface matches that of *ExplicitFitnessSharing* and set the *diversity_type* of your Config instance to this class.

TODO: document, include example

## 5.6 Using different gene types

To use a different gene type, you can create a custom class whose interface matches that of *Genome*, and set the *node_gene_type* or *conn_gene_type* member, respectively, of your Config instance to this class.

TODO: document, include example

## 5.7 Using a different genome type

To use a different gene type, you can create a custom class whose interface matches that of *NodeGene* or *ConnectionGene*, and set the *genotype* member of your Config instance to this class.

TODO: document, include example

## 5.8 Reporting

The Population class makes calls to a collection of zero or more reporters at fixed points during the evolution process. The user can add a custom reporter to this collection by calling Population.add_reporter and providing it with an object which implements the same interface as StdOutReporter.

TODO: document, include example

## 5.9 Logging

If you need to change the logging scheme, you should subclass *Population* and override the *_log_stats* method.

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search