

---

# NEAT-Python Documentation

*Release 0.91*

**CodeReclaimers, LLC**

**Jul 17, 2017**



<b>1</b>	<b>NEAT Overview</b>	<b>3</b>
<b>2</b>	<b>Installation</b>	<b>5</b>
2.1	About The Examples . . . . .	5
2.2	Install neat-python from PyPI using pip . . . . .	5
2.3	Install neat-python from source using setup.py . . . . .	5
<b>3</b>	<b>Configuration file description</b>	<b>7</b>
3.1	[NEAT] section . . . . .	7
3.2	[DefaultStagnation] section . . . . .	8
3.3	[DefaultReproduction] section . . . . .	8
3.4	[DefaultGenome] section . . . . .	9
<b>4</b>	<b>Overview of the basic XOR example (xor2.py)</b>	<b>13</b>
4.1	Fitness function . . . . .	13
4.2	Running NEAT . . . . .	13
4.3	Getting the results . . . . .	14
4.4	Visualizations . . . . .	14
4.5	Example Source . . . . .	14
<b>5</b>	<b>Customizing Behavior</b>	<b>19</b>
5.1	New activation functions . . . . .	19
5.2	Reporting/logging . . . . .	19
5.3	New genome types . . . . .	20
5.4	Speciation scheme . . . . .	20
5.5	Species stagnation scheme . . . . .	20
5.6	Reproduction scheme . . . . .	20
<b>6</b>	<b>Overview of builtin activation functions</b>	<b>23</b>
6.1	abs . . . . .	23
6.2	clamped . . . . .	25
6.3	cube . . . . .	25
6.4	exp . . . . .	25
6.5	gauss . . . . .	25
6.6	hat . . . . .	25
6.7	identity . . . . .	27
6.8	inv . . . . .	27

6.9	log	27
6.10	relu	27
6.11	sigmoid	27
6.12	sin	27
6.13	softplus	27
6.14	square	27
6.15	tanh	27
<b>7</b>	<b>Continuous-time recurrent neural network implementation</b>	<b>29</b>
<b>8</b>	<b>Module summaries</b>	<b>31</b>
8.1	activations	31
8.2	attributes	32
8.3	checkpoint	34
8.4	config	35
8.5	ctrnn	36
8.6	genes	38
8.7	genome	40
8.8	graphs	44
8.9	indexer	45
8.10	iznn	45
8.11	math_util	47
8.12	nn.feed_forward	48
8.13	nn.recurrent	48
8.14	parallel	48
8.15	population	49
8.16	reporting	50
8.17	reproduction	52
8.18	six_util	54
8.19	species	54
8.20	stagnation	56
8.21	statistics	57
<b>9</b>	<b>Genome Interface</b>	<b>61</b>
9.1	Class Methods	61
9.2	Initialization/Reproduction	61
9.3	Crossover/Mutation	62
9.4	Speciation/Misc	62
<b>10</b>	<b>Reproduction Interface</b>	<b>63</b>
10.1	Class Methods	63
10.2	Initialization	63
10.3	Other methods	63
<b>11</b>	<b>Glossary</b>	<b>65</b>
<b>12</b>	<b>Indices and tables</b>	<b>69</b>
	<b>Python Module Index</b>	<b>71</b>

NEAT (NeuroEvolution of Augmenting Topologies) is a method developed by Kenneth O. Stanley for evolving arbitrary neural networks. NEAT-Python is a pure Python implementation of NEAT, with no dependencies other than the Python standard library.

---

**Note:** Some of the example code has other dependencies; please see each sample's README file for additional details and installation/setup instructions. In addition to dependencies varying depending on the example, visualization of the results (via `visualize.py` modules) frequently requires [graphviz](#) and/or [matplotlib](#).

---

Support for HyperNEAT and other extensions to NEAT is planned once the fundamental NEAT implementation is more complete and stable.

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:



*NEAT* (NeuroEvolution of Augmenting Topologies) is an evolutionary algorithm that creates artificial neural 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, and do a good job of explaining concepts (or at least pointing you to other references that will). [The initial NEAT paper](#) is only 6 pages long, and Section II should be enough if you just want a high-level overview.

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 (for a *fitness criterion function* of `max`; others are configurable) exceeds the user-specified *fitness threshold*, the algorithm terminates.

One difficulty in this setup is with the implementation of *crossover* - how does one do a crossover between two networks of differing structure? NEAT handles this by keeping track of the origins of the nodes, with an *identifying number* (new, higher numbers are generated for each additional node). Those derived from a common ancestor (that are *homologous*) are matched up for crossover, and connections are matched if the nodes they connect have common ancestry. (There are variations in exactly how this is done depending on the implementation of NEAT; this paragraph describes how it is done in this implementation.)

Another potential difficulty is that a structural mutation - as opposed to mutations in, for instance, the *weights* of the connections - such as the addition of a node or connection can, while being promising for the future, be disruptive in the short-term (until it has been fine-tuned by less-disruptive mutations). How NEAT deals with this is by dividing genomes into species, which have a close *genomic distance* due to similarity, then having competition most intense

within species, not between species (fitness sharing). How is genomic distance measured? It uses a combination of the number of non-homologous nodes and connections with measures of how much homologous nodes and connections have diverged since their common origin. (Non-homologous nodes and connections are termed *disjoint* or *excess*, depending on whether the *numbers* are from the same range or beyond that range; like most NEAT implementations, this one makes no distinction between the two.)



### About The Examples

Because *neat-python* 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 obtain matching example/library code by using one of the two methods outlined below:

### Install neat-python from PyPI using pip

To install the most recent release (version 0.91) 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 0.91](#) and use the example code contained in it.

You may also just get the 0.91 release source, and install it directly using *setup.py* (as shown below) instead of *pip*.

### Install neat-python 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 0.91](#).

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.

---

## Configuration file description

---

The configuration file is in the format described in the Python `configparser` documentation as “a basic configuration file parser language which provides a structure similar to what you would find on Microsoft Windows INI files.”

Most settings must be explicitly enumerated in the configuration file. (This makes it less likely that library code changes will result in your project silently using different NEAT settings. There are some defaults, as noted below, and insofar as possible new configuration parameters will default to the existing behavior.)

---

**Note:** TODO: Work on ways to make the existing defaults more integrated with the configuration code. This may involve a third, optional (defaulting to `None`) parameter for `ConfigParameter`, giving a default. Such a mechanism will be needed to convert the configuration in `DefaultReproduction` and `DefaultStagnation` to match that in other classes/modules. It would also be needed to add, for instance, an alternative uniform-distribution initialization for various parameters such as weight and bias, without requiring all configuration files to be changed to specify the gaussian distribution currently in use. Another example would be for a configuration file variable for whether more than one addition/connection *mutation* can take place at a time, as mentioned in `genome.py`.

The above is done in the `config_work` branch.

---

Note that the `Config` constructor also requires you to explicitly specify the types that will be used for the NEAT simulation. This, again, is to help avoid silent changes in behavior. The configuration file is in several sections, of which at least one is required. However, there are no requirements for ordering within these sections, or for ordering of the sections themselves.

### [NEAT] section

The NEAT section specifies parameters particular to the generic NEAT algorithm or the experiment itself. This section is always required, and is handled by the `Config` class itself.

- ***fitness\_criterion*** The function used to compute the termination criterion from the set of genome fitnesses. Allowable values are: `min`, `max`, and `mean`

- *fitness\_threshold* When the fitness computed by `fitness_criterion` meets or exceeds this threshold, the evolution process will terminate, with a call to any registered reporting class' `found_solution` method.

---

**Note:** The `found_solution` method is **not** called if the maximum number of generations is reached without the above threshold being passed. TODO: Add a new configuration parameter to ignore the above, provided a maximum number of generations is passed to `population.Population.run()`, and if so call `found_solution` upon termination by a maximum number of generations. (This is mostly done in the `config_work` branch.)

---

- *pop\_size* The number of individuals in each generation.
- *reset\_on\_extinction* If this evaluates 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.

## [DefaultStagnation] section

The `DefaultStagnation` section specifies parameters for the builtin `DefaultStagnation` class. This section is only necessary if you specify this class as the stagnation implementation when creating the `Config` instance; otherwise you need to include whatever configuration (if any) is required for your particular implementation.

- *species\_fitness\_func* The function used to compute species fitness. **This defaults to “mean“.** Allowed values are: `max`, `min`, `mean`, and `median`

---

**Note:** This is **not** used for calculating species fitness for apportioning reproduction (which always uses `mean`).

---

- *max\_stagnation* Species that have not shown improvement in more than this number of generations will be considered stagnant and removed. **This defaults to 15.**
- *species\_elitism* The number of species that will be protected from stagnation; mainly intended to prevent total extinctions caused by all species becoming stagnant before new species arise. For example, a `species_elitism` setting of 3 will prevent the 3 species with the highest species fitness from being removed for stagnation regardless of the amount of time they have not shown improvement. **This defaults to 0.**

---

**Note:** TODO: `DefaultStagnation.write_config` uses a default of 15 for `species_elitism`, but the default by `DefaultStagnation.parse_config` is 0, which will override.

The above is handled in the `config_work` branch.

---

## [DefaultReproduction] section

The `DefaultReproduction` section specifies parameters for the builtin `DefaultReproduction` class. This section is only necessary if you specify this class as the reproduction implementation when creating the `Config` instance; otherwise you need to include whatever configuration (if any) is required for your particular implementation.

- *elitism* The number of most-fit individuals in each species that will be preserved as-is from one generation to the next. **This defaults to 0.**
- *survival\_threshold* The fraction for each species allowed to reproduce each generation. **This defaults to 0.2.**

---

**Note:** TODO: There is also a `min_species_size` configuration parameter, defaulting to 2, although it is not written out by `DefaultReproduction.write_config`.

The above is handled in the `config_work` branch, although documentation is needed.

---

## [DefaultGenome] section

The `DefaultGenome` section specifies parameters for the builtin `DefaultGenome` class. This section is only necessary if you specify this class as the genome implementation when creating the `Config` instance; otherwise you need to include whatever configuration (if any) is required for your particular implementation.

- **`activation_default`** The default *activation function attribute assigned* to new *nodes*. **If none is given, or “random“ is specified, one of the “activation\_options“ will be chosen at random.**
- **`activation_mutate_rate`** The probability that *mutation* will replace the node’s activation function with a *randomly-determined* member of the `activation_options`. Valid values are in [0.0, 1.0].
- **`activation_options`** A space-separated list of the activation functions that may be used by nodes. **This defaults to *sigmoid*.** The built-in available functions can be found in *Overview of builtin activation functions*; more can be added as described in *Customizing Behavior*.
- **`aggregation_default`** The default *aggregation function attribute assigned* to new *nodes*. **If none is given, or “random“ is specified, one of the “aggregation\_options“ will be chosen at random.**
- **`aggregation_mutate_rate`** The probability that *mutation* will replace the node’s aggregation function with a *randomly-determined* member of the `aggregation_options`. Valid values are in [0.0, 1.0].
- **`aggregation_options`** A space-separated list of the aggregation functions that may be used by nodes. **This defaults to “sum“.** The available functions (defined in `genome.DefaultGenomeConfig`) are: *sum*, *product*, *min*, and *max*
- **`bias_init_mean`** The mean of the normal/gaussian distribution used to *select bias attribute* values for new *nodes*.
- **`bias_init_stdev`** The standard deviation of the normal/gaussian distribution used to select bias values for new nodes.
- **`bias_max_value`** The maximum allowed bias value. Biases above this value will be *clamped* to this value.
- **`bias_min_value`** The minimum allowed bias value. Biases below this value will be *clamped* to this value.
- **`bias_mutate_power`** The standard deviation of the zero-centered normal/gaussian distribution from which a bias value *mutation* is drawn.
- **`bias_mutate_rate`** The probability that *mutation* will change the bias of a node by adding a random value.
- **`bias_replace_rate`** The probability that *mutation* will replace the bias of a node with a newly *chosen* random value (as if it were a new node).
- **`compatibility_threshold`** Individuals whose *genomic distance* is less than this threshold are considered to be in the same *species*.
- **`compatibility_disjoint_coefficient`** The coefficient for the *disjoint* and *excess gene* counts’ contribution to the *genomic distance*.
- **`compatibility_weight_coefficient`** The coefficient for each *weight*, *bias*, or *response* multiplier difference’s contribution to the *genomic distance* (for *homologous nodes* or *connections*). This is also used as the value to add for differences in *activation functions*, *aggregation functions*, or *enabled/disabled* status.

**Note:** It is currently possible for two *homologous* nodes or connections to have a higher contribution to the *genomic distance* than a disjoint or excess *node* or *connection*, depending on their *attributes* and the settings of the above parameters.

---

- ***conn\_add\_prob*** The probability that *mutation* will add a *connection* between existing *nodes*. Valid values are in [0.0, 1.0].
  - ***conn\_delete\_prob*** The probability that *mutation* will delete an existing connection. Valid values are in [0.0, 1.0].
  - ***enabled\_default*** The default *enabled attribute* of newly created connections. Valid values are True and False.
- 

**Note:** “Newly created connections” include ones in newly-created genomes, if those have initial connections (from the setting of the *initial\_connection* variable).

---

- ***enabled\_mutate\_rate*** The probability that *mutation* will *replace* (50/50 chance of True or False) the enabled status of a connection. Valid values are in [0.0, 1.0].
- ***feed\_forward*** If this evaluates to True, generated networks will not be allowed to have *recurrent connections* (they will be *feedforward*). Otherwise they may be (but are not forced to be) recurrent.
- ***initial\_connection*** Specifies the initial connectivity of newly-created genomes. (Note the effects on settings other than *unconnected* of the *enabled\_default* parameter.) There are seven allowed values:
  - *unconnected* - No *connections* are initially present. **This is the default.**
  - *fs\_neat\_nohidden* - One randomly-chosen *input node* has one connection to each *output node*. (This is one version of the FS-NEAT scheme.)
  - *fs\_neat\_hidden* - One randomly-chosen *input node* has one connection to each *hidden* and *output node*. (This is another version of the FS-NEAT scheme. If there are no hidden nodes, it is the same as *fs\_neat\_nohidden*.)
  - *full\_nodirect* - Each *input node* is connected to all *hidden* nodes, if there are any, and each hidden node is connected to all *output nodes*; otherwise, each input node is connected to all *output nodes*. Genomes with *feed\_forward* set to False will also have *recurrent* (loopback, in this case) connections from each hidden or output node to itself.
  - *full\_direct* - Each *input node* is connected to all *hidden* and *output nodes*, and each hidden node is connected to all output nodes. Genomes with *feed\_forward* set to False will also have *recurrent* (loopback, in this case) connections from each hidden or output node to itself.
  - *partial\_nodirect* # - As for *full\_nodirect*, but each connection has a probability of being present determined by the number (valid values are in [0.0, 1.0]).
  - *partial\_direct* # - as for *full\_direct*, but each connection has a probability of being present determined by the number (valid values are in [0.0, 1.0]).

Changed in version 0.91-github: *fs\_neat* split into *fs\_neat\_nohidden* and *fs\_neat\_hidden*; *full*, *partial* split into *full\_nodirect*, *full\_direct*, *partial\_nodirect*, *partial\_direct*

- ***node\_add\_prob*** The probability that *mutation* will add a new *node* (essentially replacing an existing connection, the *enabled* status of which will be set to False). Valid values are in [0.0, 1.0].
- ***node\_delete\_prob*** The probability that *mutation* will delete an existing node (and all connections to it). Valid values are in [0.0, 1.0].
- ***num\_hidden*** The number of *hidden nodes* to add to each genome in the initial population.

- ***num\_inputs*** The number of *input nodes*, through which the network receives inputs.
- ***num\_outputs*** The number of *output nodes*, to which the network delivers outputs.
- ***response\_init\_mean*** The mean of the normal/gaussian distribution used to *select response multiplier attribute* values for new *nodes*.
- ***response\_init\_stdev*** The standard deviation of the normal/gaussian distribution used to select response multipliers for new nodes.
- ***response\_max\_value*** The maximum allowed response multiplier. Response multipliers above this value will be *clamped* to this value.
- ***response\_min\_value*** The minimum allowed response multiplier. Response multipliers below this value will be *clamped* to this value.
- ***response\_mutate\_power*** The standard deviation of the zero-centered normal/gaussian distribution from which a response multiplier *mutation* is drawn.
- ***response\_mutate\_rate*** The probability that *mutation* will change the response multiplier of a node by adding a random value.
- ***response\_replace\_rate*** The probability that *mutation* will replace the response multiplier of a node with a newly *chosen* random value (as if it were a new node).
- ***weight\_init\_mean*** The mean of the normal/gaussian distribution used to *select weight attribute* values for new *connections*.
- ***weight\_init\_stdev*** The standard deviation of the normal/gaussian distribution used to select weight values for new connections.
- ***weight\_max\_value*** The maximum allowed weight value. Weights above this value will be *clamped* to this value.
- ***weight\_min\_value*** The minimum allowed weight value. Weights below this value will be *clamped* to this value.
- ***weight\_mutate\_power*** The standard deviation of the zero-centered normal/gaussian distribution from which a weight value *mutation* is drawn.
- ***weight\_mutate\_rate*** The probability that *mutation* will change the weight of a connection by adding a random value.
- ***weight\_replace\_rate*** The probability that *mutation* will replace the weight of a connection with a newly *chosen* random value (as if it were a new connection).

*Table of Contents*





---

## 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

### 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  $1 - \sum_i (e_i - a_i)^2$  between the expected ( $e_i$ ) and actual ( $a_i$ ) 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_genomes` function. This function takes two arguments: a list of genomes (the current population) and the active configuration. `neat-python` expects the fitness function to calculate a fitness for each genome and assign this value to the genome's `fitness` member.

### 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 the *Configuration file description*).
- Create a `neat.population.Population` object using the `Config` object created above.
- Call the `run` method on the `Population` object, giving it your fitness function and (optionally) 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 `fitness_threshold` value you specified in your config file.

## Getting the results

Once the call to the population object's `run` method has returned, you can query the `statistics` member of the population (a `neat.statistics.StatisticsReporter` object) to get the best genome(s) seen during the run. In this example, we take the 'winner' genome to be that returned by `pop.statistics.best_genome()`.

Other information available from the default statistics object includes per-generation mean fitness, per-generation standard deviation of fitness, and the best N genomes (with or without duplicates).

## Visualizations

Functions are available in the `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.

## 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 0.91 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 -- this is most likely the simplest possible example.
"""

from __future__ import print_function
import os
import neat
import visualize

# 2-input XOR inputs and expected outputs.
xor_inputs = [(0.0, 0.0), (0.0, 1.0), (1.0, 0.0), (1.0, 1.0)]
xor_outputs = [(0.0,), (1.0,), (1.0,), (0.0,)]

def eval_genomes(genomes, config):
    for genome_id, genome in genomes:
        genome.fitness = 4.0
        net = neat.nn.FeedForwardNetwork.create(genome, config)
        for xi, xo in zip(xor_inputs, xor_outputs):
            output = net.activate(xi)
```

```

        genome.fitness -= (output[0] - xo[0]) ** 2

def run(config_file):
    # Load configuration.
    config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                        neat.DefaultSpeciesSet, neat.DefaultStagnation,
                        config_file)

    # Create the population, which is the top-level object for a NEAT run.
    p = neat.Population(config)

    # Add a stdout reporter to show progress in the terminal.
    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)
    p.add_reporter(neat.Checkpointer(5))

    # Run for up to 300 generations.
    winner = p.run(eval_genomes, 300)

    # Display the winning genome.
    print('\nBest genome:\n{!s}'.format(winner))

    # Show output of the most fit genome against training data.
    print('\nOutput:')
    winner_net = neat.nn.FeedForwardNetwork.create(winner, config)
    for xi, xo in zip(xor_inputs, xor_outputs):
        output = winner_net.activate(xi)
        print("input {!r}, expected output {!r}, got {!r}".format(xi, xo, output))

    node_names = {-1:'A', -2:'B', 0:'A XOR B'}
    visualize.draw_net(config, winner, True, node_names=node_names)
    visualize.plot_stats(stats, ylog=False, view=True)
    visualize.plot_species(stats, view=True)

    p = neat.Checkpointer.restore_checkpoint('neat-checkpoint-4')
    p.run(eval_genomes, 10)

if __name__ == '__main__':
    # Determine path to configuration file. This path manipulation is
    # here so that the script will run successfully regardless of the
    # current working directory.
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, 'config-feedforward')
    run(config_path)

```

and here is the associated config file:

```

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

[NEAT]
fitness_criterion    = max
fitness_threshold    = 3.9
pop_size             = 150
reset_on_extinction  = False

```

```
[DefaultGenome]
# node activation options
activation_default      = sigmoid
activation_mutate_rate  = 0.0
activation_options     = sigmoid

# node aggregation options
aggregation_default    = sum
aggregation_mutate_rate = 0.0
aggregation_options    = sum

# node bias options
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

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient  = 0.5

# connection add/remove rates
conn_add_prob          = 0.5
conn_delete_prob       = 0.5

# connection enable options
enabled_default        = True
enabled_mutate_rate    = 0.01

feed_forward           = True
initial_connection     = full

# node add/remove rates
node_add_prob          = 0.2
node_delete_prob       = 0.2

# network parameters
num_hidden             = 0
num_inputs             = 2
num_outputs            = 1

# node response options
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

# connection weight options
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               = 2
survival_threshold   = 0.2
```



---

## Customizing Behavior

---

NEAT-Python allows the user to provide drop-in replacements for some parts of the NEAT algorithm, which hopefully makes it easier to implement common variations of the algorithm as 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.

### New activation functions

New *activation functions* are registered with your *Config* instance, prior to creation of the *Population* instance, as follows:

```
def sinc(x):  
    return 1.0 if x == 0 else sin(x) / x  
  
config.genome_config.add_activation('my_sinc_function', sinc)
```

The first argument to *add\_activation* is the name by which this activation function will be referred to in the configuration settings file.

This is demonstrated in the [memory-fixed](#) example.

---

**Note:** This method is only valid when using the *DefaultGenome* implementation, with the method being found in the *DefaultGenomeConfig* implementation; different genome implementations may require a different method of registration.

---

### Reporting/logging

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 *BaseReporter* (in *reporting.py*), probably partially by subclassing it.

*StdOutReporter*, *StatisticsReporter*, and *Checkpointner* may be useful as examples of the behavior you can add using a reporter.

## New genome types

To use a different genome type, you can create a custom class whose interface matches that of *DefaultGenome* and pass this as the *genome\_type* argument to the *Config* constructor. The minimum genome type interface is documented here: *Genome Interface*.

This is demonstrated in the [circuit evolution](#) example.

Alternatively, you can subclass *DefaultGenome* in cases where you need to just add some extra behavior. This is done in the [OpenAI lander](#) example to add an evolvable per-genome reward discount value. It is also done in the *iznn* setup, with *IZGenome*.

## Speciation scheme

To use a different speciation scheme, you can create a custom class whose interface matches that of *DefaultSpeciesSet* and pass this as the *species\_set\_type* argument to the *Config* constructor.

---

**Note:** TODO: Further document species set interface (some done in *module\_summaries*)

---

---

**Note:** TODO: Include example

---

## Species stagnation scheme

The default species stagnation scheme is a simple fixed stagnation limit—when a species exhibits no improvement for a fixed number of generations, all its members are removed from the simulation. This behavior is encapsulated in the *DefaultStagnation class*.

To use a different species stagnation scheme, you must create a custom class whose interface matches that of *DefaultStagnation*, and provide it as the *stagnation\_type* argument to the *Config* constructor.

This is demonstrated in the [interactive 2D image](#) example.

## Reproduction scheme

The default reproduction scheme uses explicit fitness sharing. This behavior is encapsulated in the *DefaultReproduction* class. The minimum reproduction type interface is documented here: *Reproduction Interface*

To use a different reproduction scheme, you must create a custom class whose interface matches that of *DefaultReproduction*, and provide it as the *reproduction\_type* argument to the *Config* constructor.



---

**Note:** TODO: Include example

---



---

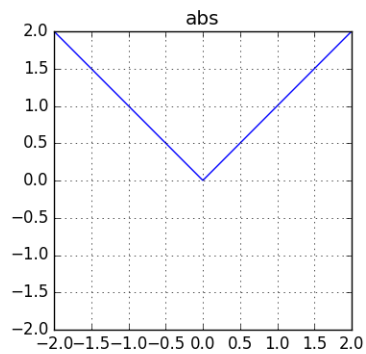
## Overview of builtin activation functions

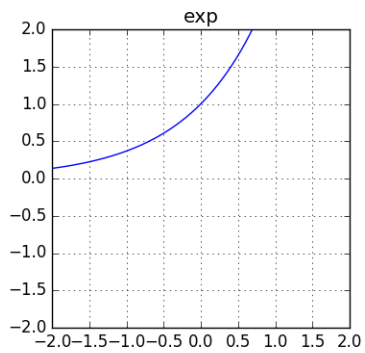
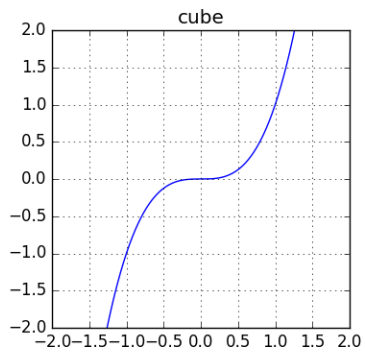
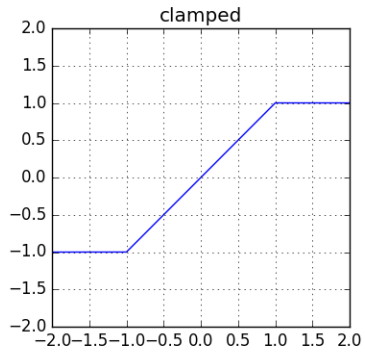
---

Note that some of these *functions* are scaled differently from the canonical versions you may be familiar with. The intention of the scaling is to place more of the functions' “interesting” behavior in the region  $[-1, 1] \times [-1, 1]$ .

The implementation of these functions can be found in the *activations* module.

### abs



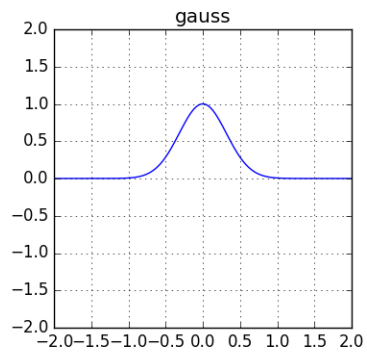


**clamped**

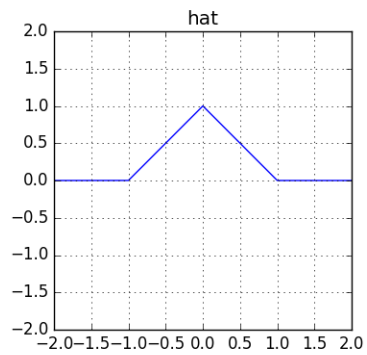
**cube**

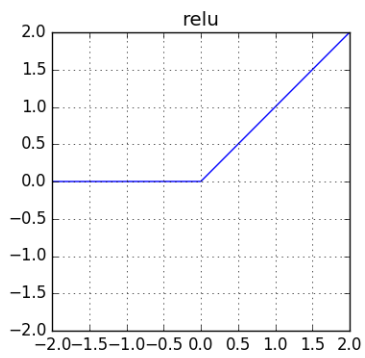
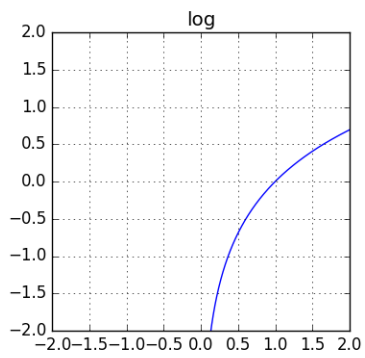
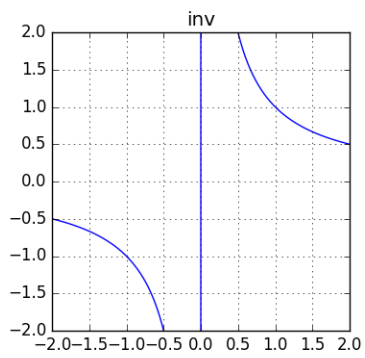
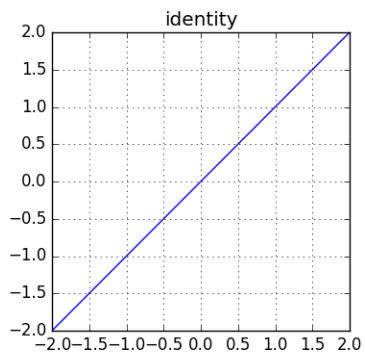
**exp**

**gauss**



**hat**





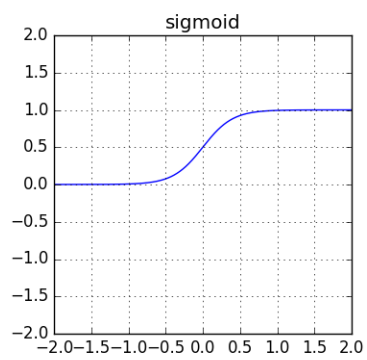
**identity**

**inv**

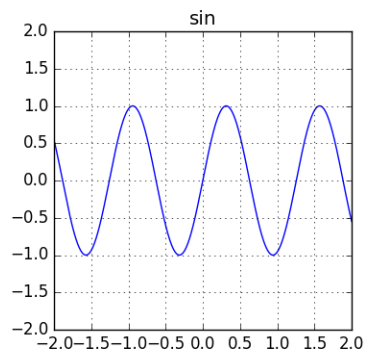
**log**

**relu**

**sigmoid**



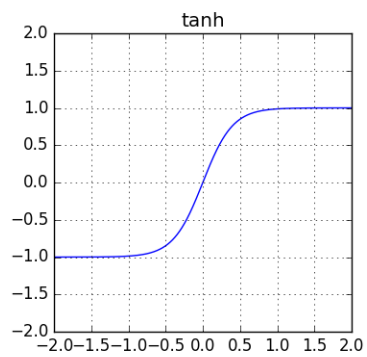
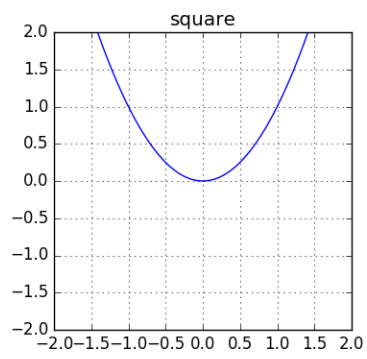
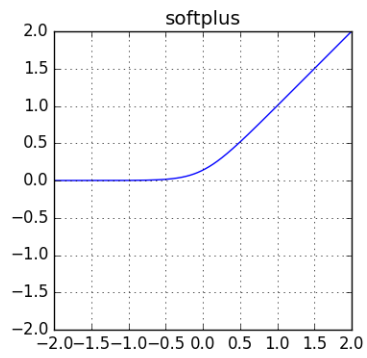
**sin**



**softplus**

**square**

**tanh**





---

## Continuous-time recurrent neural network implementation

---

The default *continuous-time recurrent* neural network (CTRNN) *implementation* in neat-python is modeled as a system of ordinary differential equations, with neuron potentials as the dependent variables.

$$\tau_i \frac{dy_i}{dt} = -y_i + f_i \left( \beta_i + \sum_{j \in A_i} w_{ij} y_j \right)$$

Where:

- $\tau_i$  is the time constant of neuron  $i$ .
- $y_i$  is the potential of neuron  $i$ .
- $f_i$  is the *activation function* of neuron  $i$ .
- $\beta_i$  is the *bias* of neuron  $i$ .
- $A_i$  is the set of indices of neurons that provide input to neuron  $i$ .
- $w_{ij}$  is the *weight* of the *connection* from neuron  $j$  to neuron  $i$ .

The time evolution of the network is computed using the forward Euler method:

$$y_i(t + \Delta t) = y_i(t) + \Delta t \frac{dy_i}{dt}$$



## activations

Has the built-in *activation functions*, code for using them, and code for adding new user-defined ones.

**exception** `activations.InvalidActivationFunction` (*TypeError*)

Exception called if an activation function being added is invalid according to the `validate_activation` function.

Changed in version 0.91-github: Base of exception changed to more-precise `TypeError`.

`activations.validate_activation` (*function*)

Checks to make sure its parameter is a function that takes a single argument.

**Parameters** `function` (*object*) – Object to be checked.

**Raises** `InvalidActivationFunction` – If the object does not pass the tests.

**class** `activations.ActivationFunctionSet`

Contains the list of current valid activation functions, including methods for adding and getting them.

**add** (*name, function*)

After validating the function (via `validate_activation`), adds it to the available activation functions under the given name. Used by `DefaultGenomeConfig.add_activation`.

**Parameters**

- **name** (*str*) – The name by which the function is to be known in the *configuration file*.
- **function** (*function*) – The function to be added.

**get** (*name*)

Returns the named function, or raises an exception if it is not a known activation function.

**Parameters** `name` (*str*) – The name of the function.

**Raises** `InvalidActivationFunction` – If the function is not known.

**is\_valid**(*name*)

Checks whether the named function is a known activation function.

**Parameters** *name* (*str*) – The name of the function.

**Returns** Whether or not the function is known.

**Return type** `bool`

---

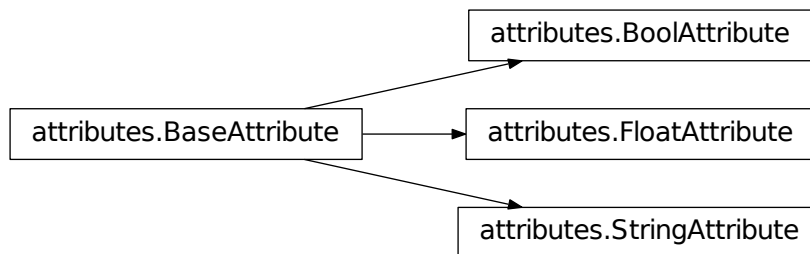
**Note:** TODO: Suggested simplification for the below: Make `__config_items__` a dict of lists, with name -> [value\_type, default] - no default if the last is None. This would also allow moving `get_config_params` into the `BaseAttribute` class, although `config_item_names` may require some modifications. (A default capability will be needed for future expansions of the attributes, such as different types of initializations, and for enabling better handling of the current activation/aggregation function defaults.)

The above is mostly/entirely done in the `config_work` branch.

---

## attributes

Deals with *attributes* used by *genes*.



**class** `attributes.BaseAttribute`(*name*)

Superclass for the type-specialized attribute subclasses, used by genes (such as via the `genes.BaseGene` implementation). Calls `config_item_names` to set up a listing of the names of configuration items using `setattr`.

**config\_item\_names**()

Formats a list of configuration item names by combining the attribute's name with the attribute class' list of `__config_items__`.

**Returns** A list of configuration item names.

**Return type** `list(str)`

**class** `attributes.FloatAttribute`(*BaseAttribute*)

Class for numeric *attributes* such as the *response* of a *node*; includes code for configuration, creation, and mutation.

**get\_config\_params**()

Uses `config_item_names` to get its list of configuration item names, then gets a `float`-type `config.ConfigParameter` instance for each and returns it.

**Returns** A list of `ConfigParameter` instances.

**Return type** `list(object)`

**clamp** (*value*, *config*)

Gets the minimum and maximum values desired from `config`, then ensures that the value is between them.

**Parameters**

- **value** (*float*) – The value to be clamped.
- **config** (*object*) – The configuration object from which the minimum and maximum desired values are to be retrieved.

**Returns** The value, if it is within the desired range, or the appropriate end of the range, if it is not.

**Return type** `float`

**init\_value** (*config*)

Initializes the attribute's value, (currently always) using a gaussian distribution with the configured mean and standard deviation followed by `clamp` to keep the result within the desired range.

**Parameters** **config** (*object*) – The configuration object from which the mean and standard deviation values are to be retrieved.

**Returns** The new value.

**Return type** `float`

**mutate\_value** (*value*, *config*)

May replace (as if reinitializing, using `init_value`), mutate (using a 0-mean gaussian distribution with a configured standard deviation from `mutate_power`), or leave alone the input value, depending on the configuration settings (of `replace_rate` and `mutate_rate`). TODO: Note that the `replace_rate` is likely to be lower, so should be checked second. (Done in the `config_work` branch.)

**Parameters**

- **value** (*float*) – The current value of the attribute.
- **config** (*object*) – The configuration object from which the parameters are to be extracted.

**Returns** Either the original value, if unchanged, or the new value.

**Return type** `float`

**class** `attributes.BoolAttribute` (*BaseAttribute*)

Class for boolean *attributes* such as whether a *connection* is *enabled* or not; includes code for configuration, creation, and mutation.

**get\_config\_params** ()

Uses `config_item_names` to get its list of configuration item names, then gets a `bool`-type or `float`-type `config.ConfigParameter` instance for each and returns it.

**Returns** A list of `ConfigParameter` instances.

**Return type** `list(object)`

**init\_value** (*config*)

Initializes the attribute's value, either using a configured default or (if the default is `None`) with a 50/50 chance of `True` or `False`.

**Parameters** **config** (*object*) – The configuration object from which the default parameter is to be retrieved.

**Returns** The new value.

**Return type** `bool`

**mutate\_value** (*value*, *config*)

With a frequency determined by the `mutate_rate` (which is more precisely a `replace_rate`) configuration parameter, replaces the value with a 50/50 chance of `True` or `False`; note that this has a 50% chance of leaving the value unchanged. TODO: Have different chances possible of *mutation* in each direction. Also, do not check vs `random` if the `mutate_rate` is 0 (done in the `config_work` branch).

**Parameters**

- **value** (*bool*) – The current value of the attribute.
- **config** (*object*) – The configuration object from which the `mutate_rate` parameter is to be extracted.

**Returns** Either the original value, if unchanged, or the new value.

**Return type** `bool`

**class** `attributes.StringAttribute` (*BaseAttribute*)

Class for string attributes such as the *aggregation function* of a *node*, which are selected from a list of options; includes code for configuration, creation, and mutation.

**get\_config\_params** ()

Uses `config_item_names` to get its list of configuration item names, then gets a `str`-type, `list`-type or `float`-type `config.ConfigParameter` instance for each and returns it.

**Returns** A list of `ConfigParameter` instances.

**Return type** `list(object)`

**init\_value** (*config*)

Initializes the attribute's value, either using a configured default or (if the default is either `None` or `random`) with a randomly-chosen member of the `options` (each having an equal chance).

Note: It is possible for the default value, if specifically configured, to **not** be one of the options.

**Parameters** **config** (*object*) – The configuration object from which the default and, if necessary, `options` parameters are to be retrieved.

**Returns** The new value.

**Return type** `str`

**mutate\_value** (*value, config*)

With a frequency determined by the `mutate_rate` (which is more precisely a `replace_rate`) configuration parameter, replaces the value with an one of the `options`, with each having an equal chance; note that this can be the same value as before. (It is possible to crudely alter the chances of what is chosen by listing a given option more than once, although this is inefficient given the use of the `random.choice` function.) TODO: Do not check vs `random` if the `mutate_rate` is 0 (done in the `config_work` branch). (Longer-term, add configurable probabilities of which option is used; eventually, as with the improved version of RBF-NEAT, separate genes for the likelihoods of each (but always doing some change, to prevent overly-conservative evolution due to its inherent short-sightedness), allowing the genomes to control the distribution of options, will be desirable.)

## checkpoint

Uses `pickle` to save and restore populations (and other aspects of the simulation state).

**class** `checkpoint.Checkpointer` (*generation\_interval=100, time\_interval\_seconds=300*)

A reporter class that performs checkpointing, saving and restoring the simulation state (including population, randomization, and other aspects). It saves the current state every `generation_interval` generations or `time_interval_seconds` seconds, whichever happens first. Subclasses `reporting.BaseReporter`. (The potential save point is at the end of a generation.)

**Parameters**

- **generation\_interval** (*int or None*) – If not `None`, maximum number of generations between checkpoints.
- **time\_interval\_seconds** (*float or None*) – If not `None`, maximum number of seconds between checkpoints.

**static save\_checkpoint** (*config, population, species, generation*)

Saves the current simulation (including randomization) state to `neat-checkpoint-generation`, with `generation` being the generation number.

**static restore\_checkpoint** (*filename*)

Resumes the simulation from a previous saved point. Loads the specified file, sets the randomization state, and returns a `population.Population` object set up with the rest of the previous state.

**Parameters** `filename` (*str*) – The file to be restored from.

**Returns** Object that can be used with `Population.run` to restart the simulation.

**Return type** `Population` object.

## config

Does general configuration parsing; used by other classes for their configuration.

**class** `config.ConfigParameter` (*name, value\_type*)

Does initial handling of a particular configuration parameter.

**Parameters**

- **name** (*str*) – The name of the configuration parameter.
- **value\_type** (*str*) – The type that the configuration parameter should be; must be one of `str`, `int`, `bool`, `float`, or `list`.

**\_\_repr\_\_** ()

Returns a representation of the class suitable for use in code for initialization.

**Returns** Representation as for `repr`.

**Return type** `str`

**parse** (*section, config\_parser*)

Uses the supplied configuration parser (either from the `configparser.ConfigParser` class, or - for 2.7 - the `ConfigParser.SafeConfigParser` class) to gather the configuration parameter from the appropriate configuration file *section*. Parsing varies depending on the type.

**Parameters**

- **section** (*str*) – The section name, taken from the `__name__` attribute of the class to be configured (or NEAT for those parameters).
- **config\_parser** (*object*) – The configuration parser to be used.

**Returns** The configuration parameter value, in stringified form unless a list.

**Return type** `str` or `list`

**interpret** (*config\_dict*)

Takes a `dictionary` of configuration parameters, as output by the configuration parser called in `parse()`, and interprets them into the proper type, with some error-checking.

**Parameters** `config_dict` (*dict*) – Configuration parameters as output by the configuration parser.

**Returns** The configuration parameter value

**Return type** `str` or `int` or `bool` or `float` or `list`

**format** (*value*)

Depending on the type of configuration parameter, returns either a space-separated list version, for `list` parameters, or the stringified version (using `str`), of `value`.

**Parameters** `value` (*str* or *int* or *bool* or *float* or *list*) – Configuration parameter value to be formatted.

`config.write_pretty_params(f, config, params)`

Prints configuration parameters, with justification based on the longest configuration parameter name.

#### Parameters

- **f** (*file*) – File object to be written to.
- **config** (*object*) – Configuration object from which parameter values are to be fetched (using `getattr`).
- **params** (*list*) – List of `ConfigParameter` instances giving the names of interest and the types of parameters.

**class** `config.Config`(*genome\_type, reproduction\_type, species\_set\_type, stagnation\_type, filename*)

A simple container for user-configurable parameters of NEAT. The four parameters ending in `_type` may be the built-in ones or user-provided objects, which must make available the methods `parse_config` and `write_config`, plus others depending on which object it is. (For more information on the objects, see below and *Customizing Behavior*.) `Config` itself takes care of the NEAT parameters. For a description of the configuration file, see *Configuration file description*.

#### Parameters

- **genome\_type** (*object*) – Specifies the genome class used, such as `genome.DefaultGenome` or `iznn.IZGenome`. See *Genome Interface* for the needed interface.
- **reproduction\_type** (*object*) – Specifies the reproduction class used, such as `reproduction.DefaultReproduction`. See *Reproduction Interface* for the needed interface.
- **species\_set\_type** (*object*) – Specifies the species set class used, such as `species.DefaultSpeciesSet`.
- **stagnation\_type** (*object*) – Specifies the stagnation class used, such as `stagnation.DefaultStagnation`.
- **filename** (*str*) – Pathname for configuration file to be opened, read, processed by a parser from the `configparser.ConfigParser` class (or, for 2.7, the `ConfigParser.SafeConfigParser` class), the NEAT section handled by `Config`, and then other sections passed to the `parse_config` methods of the appropriate classes.

**Raises** `AssertionError` – If any of the objects lack a `parse_config` method.

**save** (*filename*)

Opens the specified file for writing (not appending) and outputs a configuration file from the current configuration. Uses `write_pretty_params()` for the NEAT parameters and the appropriate class `write_config` methods for the other sections.

**Parameters** **filename** (*str*) – The configuration file to be written.

## ctrnn

**class** `ctrnn.CTRNNNodeEval`(*time\_constant, activation, aggregation, bias, response, links*)

Sets up the basic *Continuous-time recurrent neural network implementation (continuous-time recurrent neural network) nodes*.

#### Parameters



- **time\_constant** (*float*) – Controls how fast the node responds;  $\tau_i$  from *Continuous-time recurrent neural network implementation*.
- **activation** (*function*) – *Activation function* for the node.
- **aggregation** (*function*) – *Aggregation function* for the node.
- **bias** (*float*) – *Bias* for the node.
- **response** (*float*) – *Response* multiplier for the node.
- **links** (*list (tuple (int, float))*) – List of other nodes providing input, as tuples of (*input key, weight*)

**class** ctrnn.**CTRNN** (*inputs, outputs, node\_evals*)

Sets up the *Continuous-time recurrent neural network implementation* network itself.

**reset** ()

Resets the time and all node activations to 0 (necessary due to otherwise retaining state via *recurrent* connections).

**advance** (*inputs, advance\_time, time\_step=None*)

Advance the simulation by the given amount of time, assuming that inputs are constant at the given values during the simulated time.

**Parameters**

- **inputs** (*list*) – The values for the *input nodes*.
- **advance\_time** (*float*) – How much time to advance the network before returning the resulting outputs.
- **time\_step** (*float*) – How much time per step to advance the network; the default of `None` will currently result in an error, but it is planned to determine it automatically.

**Returns** The values for the *output nodes*.

**Return type** *list*

**Raises** `NotImplementedError` – If a `time_step` is not given.

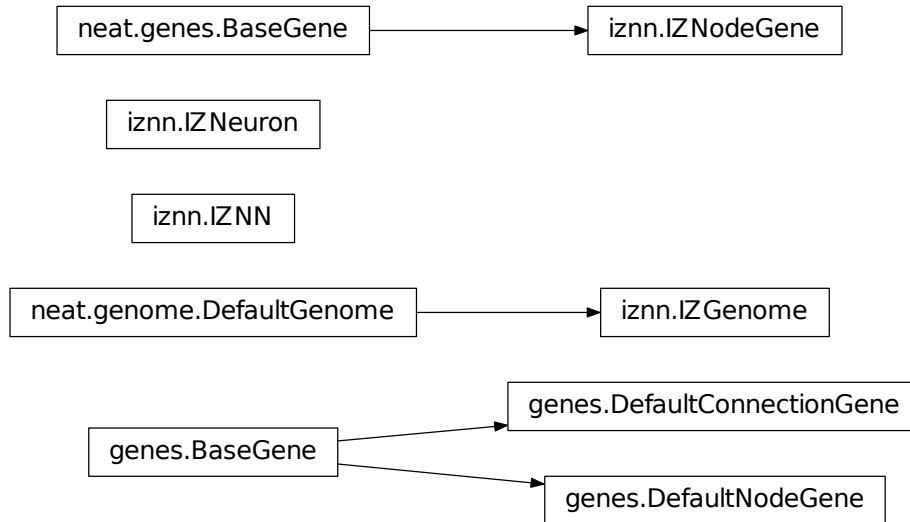
**static create** (*genome, config, time\_constant*)

Receives a genome and returns its phenotype (a *CTRNN* with *CTRNNNodeEval nodes*).

**Parameters**

- **genome** (*object*) – A *genome.DefaultGenome* instance.
- **config** (*object*) – A *config.Config* instance.
- **time\_constant** (*float*) – Used for the *CTRNNNodeEval* initializations.

## genes



**class** `genes.BaseGene` (*key*)

Handles functions shared by multiple types of genes (both *node* and *connection*), including *crossover* and calling *mutation* methods.

**Parameters** *key* (*int*) – The gene *identifier*. Note: For connection genes, determining whether they are *homologous* (for *genomic distance* and *crossover* determination) uses the identifiers of the connected nodes, not the connection gene’s identifier.

**\_\_str\_\_** ()

Converts gene attributes into a printable format.

**Returns** Stringified gene instance.

**Return type** `str`

**\_\_lt\_\_** (*other*)

Allows sorting genes by *keys*.

**Parameters** *other* (*object*) – The other *BaseGene* object.

**Returns** Whether the calling instance’s *key* is less than that of the *other* instance.

**Return type** `bool`

**classmethod** `parse_config` (*config*, *param\_dict*)

Placeholder; parameters are entirely in gene *attributes*.

**classmethod** `get_config_params` ()

Fetches configuration parameters from each gene class’ `__gene_attributes__` list (using `FloatAttribute.get_config_params`, `BoolAttribute.get_config_params`, or `StringAttribute.get_config_params` as appropriate for each listed attribute). Used by `genome.DefaultGenomeConfig` to include gene parameters in its configuration parameters.

**Returns** List of configuration parameters (as `config.ConfigParameter` instances) for the gene attributes.

**Return type** `list(object)`

**init\_attributes** (*config*)

Initializes its gene attributes using the supplied configuration object and `FloatAttribute.init_value`, `BoolAttribute.init_value`, or `StringAttribute.init_value` as appropriate.

**Parameters** **config** (*object*) – Configuration object to be used by the appropriate *attributes* class.

**mutate** (*config*)

*Mutates* (possibly) its gene attributes using the supplied configuration object and `FloatAttribute.init_value`, `BoolAttribute.init_value`, or `StringAttribute.init_value` as appropriate.

**Parameters** **config** (*object*) – Configuration object to be used by the appropriate *attributes* class.

**copy** ()

Makes a copy of itself, including its subclass, *key*, and all gene attributes.

**Returns** A copied gene

**Return type** `object`

**crossover** (*gene2*)

Creates a new gene via *crossover* - randomly inheriting attributes from its parents. The two genes must be *homologous*, having the same *key/id*.

**Parameters** **gene2** (*object*) – The other gene.

**Returns** A new gene, with the same *key/id*, with other attributes being copied randomly (50/50 chance) from each parent gene.

**Return type** `object`

**class** `genes.DefaultNodeGene` (*BaseGene*)

Groups *attributes* specific to *node* genes - such as *bias* - and calculates genetic distances between two *homologous* (not *disjoint* or excess) node genes.

**distance** (*other, config*)

Determines the degree of differences between node genes using their 4 *attributes*; the final result is multiplied by the configured *compatibility\_weight\_coefficient*.

**Parameters**

- **other** (*object*) – The other `DefaultNodeGene`.
- **config** (*object*) – The genome configuration object.

**Returns** The contribution of this pair to the *genomic distance* between the source genomes.

**Return type** `float`

**class** `genes.DefaultConnectionGene` (*BaseGene*)

Groups *attributes* specific to *connection* genes - such as *weight* - and calculates genetic distances between two *homologous* (not *disjoint* or excess) connection genes.

**distance** (*other, config*)

Determines the degree of differences between connection genes using their 2 *attributes*; the final result is multiplied by the configured *compatibility\_weight\_coefficient*.

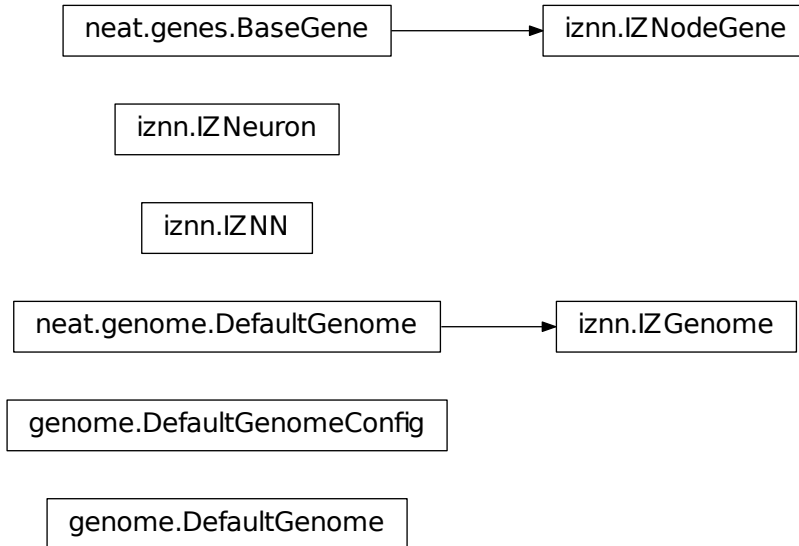
**Parameters**

- **other** (*object*) – The other `DefaultConnectionGene`.
- **config** (*object*) – The genome configuration object.

**Returns** The contribution of this pair to the *genomic distance* between the source genomes.

**Return type** `float`

## genome



`genome.product(x)`

Used to implement a product (*prod*:x) *aggregation function*.

**Parameters** **x** (*list(float)*) – The inputs to be multiplied together.

**class** `genome.DefaultGenomeConfig(params)`

Does the configuration for the `DefaultGenome` class. Has the dictionary `aggregation_function_defs`, which defines the available *aggregation functions*, and the `list` `allowed_connectivity`, which defines the available values for *initial\_connection*. Includes parameters taken from the configured gene classes, such as `genes.DefaultNodeGene`, `genes.DefaultConnectionGene`, or `iznn.IZNodeGene`.

**Parameters** **params** (*dict*) – Parameters from configuration file and `DefaultGenome` initialization (by `parse_config`).

**add\_activation(name, func)**

Adds a new *activation function*, as described in *Customizing Behavior*. Uses `ActivationFunctionSet.add`.

**Parameters**

- **name** (*str*) – The name by which the function is to be known in the *configuration file*.
- **func** (*function*) – A function meeting the requirements of *activations.validate\_activation()*.

**save(f)**

Saves the *initial\_connection* configuration and uses `config.write_pretty_params()` to write out the other parameters.

**Parameters** **f** (*file*) – The file object to be written to.

**get\_new\_node\_key** (*node\_dict*)

Finds the next unused node *key*.

**Parameters** *node\_dict* (*dict*) – A dictionary of node keys vs nodes

**Returns** A currently-unused node key.

**Return type** *int*

Changed in version 0.91-github: Moved from DefaultGenome so no longer only single-genome-instance unique.

**class** `genome.DefaultGenome` (*key*)

A *genome* for generalized neural networks. For class requirements, see *Genome Interface*. Terminology: *pin* - Point at which the network is conceptually connected to the external world; pins are either input or output. *node* - Analog of a physical neuron. *connection* - Connection between a pin/node output and a node's input, or between a node's output and a pin/node input. *key* - Identifier for an object, unique within the set of similar objects. Design assumptions and conventions. 1. Each output pin is connected only to the output of its own unique neuron by an implicit connection with weight one. This connection is permanently enabled. 2. The output pin's key is always the same as the key for its associated neuron. 3. Output neurons can be modified but not deleted. 4. The input values are applied to the input pins unmodified.

**Parameters** *key* (*int*) – *Identifier* for this individual/genome.

**classmethod** `parse_config` (*param\_dict*)

Required interface method. Provides default *node* and *connection gene* specifications (from *genes*) and uses *DefaultGenomeConfig* to do the rest of the configuration.

**Parameters** *param\_dict* (*dict*) – Dictionary of parameters from configuration file.

**Returns** Configuration object; considered opaque by rest of code, so type may vary by implementation (here, a *DefaultGenomeConfig* instance).

**Return type** *object*

**classmethod** `write_config` (*f*, *config*)

Required interface method. Saves configuration using *DefaultGenomeConfig.save()*.

**Parameters**

- *f* (*file*) – File object to write to.
- *config* (*object*) – Configuration object (here, a *DefaultGenomeConfig* instance).

**configure\_new** (*config*)

Required interface method. Configures a new genome (itself) based on the given configuration object, including genes for *connectivity* (based on *initial\_connection*) and starting *nodes* (as defined by *num\_hidden*, *num\_inputs*, and *num\_outputs* in the *configuration file*).

**Parameters** *config* (*object*) – Genome configuration object.

**configure\_crossover** (*genome1*, *genome2*, *config*)

Required interface method. Configures a new genome (itself) by *crossover* from two parent genomes. *disjoint* or *excess* genes are inherited from the fitter of the two parents, while *homologous* genes use the gene class' crossover function (e.g., *genes.BaseGene.crossover()*).

**Parameters**

- *genome1* (*object*) – The first parent genome.
- *genome2* (*object*) – The second parent genome.
- *config* (*object*) – Genome configuration object.

**mutate** (*config*)

Required interface method. *Mutates* this genome. What mutations take place are determined by configuration file settings, such as *node\_add\_prob* and *node\_delete\_prob* for the likelihood of adding or removing a *node* and *conn\_add\_prob* and *conn\_delete\_prob* for the

likelihood of adding or removing a *connection*. (Currently, more than one of these can happen with a call to `mutate`; a TODO is to add a configuration item to choose whether or not multiple mutations can happen simultaneously.) Non-structural mutations (to gene *attributes*) are performed by calling the appropriate `mutate` method(s) for connection and node genes (generally `genes.BaseGene.mutate()`).

**Parameters** `config` (*object*) – Genome configuration object.

**mutate\_add\_node** (*config*)

Takes a randomly-selected existing connection, turns its *enabled* attribute to `False`, and makes two new (enabled) connections with a new *node* between them, which join the now-disabled connection's nodes. The connection weights are chosen so as to potentially have roughly the same behavior as the original connection, although this will depend on the *activation function*, *bias*, and *response* multiplier of the new node. TODO: Particularly if the configuration is changed to only allow one structural mutation, then if there are no connections, call `mutate_add_connection()` instead of returning.

**Parameters** `config` (*object*) – Genome configuration object.

**add\_connection** (*config, input\_key, output\_key, weight, enabled*)

Adds a specified new connection; its *key* is the `tuple` of (`input_key`, `output_key`). TODO: Add validation of this connection addition.

**Parameters**

- **config** (*object*) – Genome configuration object
- **input\_key** (*int*) – *Key* of the input node.
- **output\_key** (*int*) – *Key* of the output node.
- **weight** (*float*) – The *weight* the new connection should have.
- **enabled** (*bool*) – The *enabled* attribute the new connection should have.

**mutate\_add\_connection** (*config*)

Attempts to add a randomly-selected new connection, with some filtering: 1. *input nodes* cannot be at the output end. 2. Existing connections cannot be duplicated. TODO: If a selected existing connection is not *enabled*, have some configurable chance that it will become enabled. 3. Two *output nodes* cannot be connected together. 4. If *feed\_forward* is set to `True` in the configuration file, connections cannot create *cycles*.

**Parameters** `config` (*object*) – Genome configuration object

Changed in version 0.91-github: Output nodes not allowed to be connected together.

**mutate\_delete\_node** (*config*)

Deletes a randomly-chosen (non-*output*/input) node along with its connections.

**Parameters** `config` (*object*) – Genome configuration object

**mutate\_delete\_connection** ()

Deletes a randomly-chosen connection. TODO: If the connection is *enabled*, have an option to - possibly with a *weight*-dependent chance - turn its *enabled* attribute to `False` instead.

**distance** (*other, config*)

Required interface method. Returns the *genomic distance* between this genome and the other. This distance value is used to compute genome compatibility for *speciation*. Uses (by default) the `genes.DefaultNodeGene.distance()` and `genes.DefaultConnectionGene.distance()` methods for *homologous* pairs, and the configured *compatibility\_disjoint\_coefficient* for disjoint/excess genes. (Note that this is one of the most time-consuming portions of the library; optimization - such as using `cython` - may be needed if using an unusually fast fitness function and/or an unusually large population.)

**Parameters**

- **other** (*object*) – The other `DefaultGenome` instance (genome) to be compared to.
- **config** (*object*) – The genome configuration object.

**Returns** The genomic distance.

**Return type** float

**size()**

Required interface method. Returns genome `complexity`, taken to be (number of nodes, number of enabled connections); currently only used for reporters - some retrieve this information for the highest-fitness genome at the end of each generation.

**\_\_str\_\_()**

Gives a listing of the genome's nodes and connections.

**Returns** Node and connection information.

**Return type** str

**static create\_node** (*config*, *node\_id*)

Creates a new node with the specified *id* (including for its *gene*), using the specified configuration object to retrieve the proper node gene type and how to initialize its attributes.

**Parameters**

- **config** (*object*) – The genome configuration object.
- **node\_id** (*int*) – The key for the new node.

**Returns** The new node object.

**Return type** object

**static create\_connection** (*config*, *input\_id*, *output\_id*)

Creates a new connection with the specified *id* pair as its key (including for its *gene*, as a *tuple*), using the specified configuration object to retrieve the proper connection gene type and how to initialize its attributes.

**Parameters**

- **config** (*object*) – The genome configuration object.
- **input\_id** (*int*) – The input end's key.
- **output\_id** (*int*) – The output end's key.

**Returns** The new connection object.

**Return type** object

**connect\_fs\_neat\_nohidden** (*config*)

Connect one randomly-chosen input to all *output nodes* (FS-NEAT without connections to *hidden nodes*, if any). Previously called `connect_fs_neat`. Implements the `fs_neat_nohidden` setting for *initial\_connection*.

**Parameters** **config** (*object*) – The genome configuration object.

**connect\_fs\_neat\_hidden** (*config*)

Connect one randomly-chosen input to all *hidden nodes* and *output nodes* (FS-NEAT with connections to hidden nodes, if any). Implements the `fs_neat_hidden` setting for *initial\_connection*.

**Parameters** **config** (*object*) – The genome configuration object.

**compute\_full\_connections** (*config*, *direct*)

Compute connections for a fully-connected feed-forward genome—each input connected to all hidden nodes (and output nodes if *direct* is set or there are no hidden nodes), each hidden node connected to all output nodes. (Recurrent genomes will also include node self-connections.)

**Parameters**

- **config** (*object*) – The genome configuration object.
- **direct** (*bool*) – Whether or not, if there are *hidden nodes*, to include links directly from input to output.

**Returns** The list of connections, as (input *key*, output key) tuples

**Return type** list(tuple(int,int))

Changed in version 0.91-github: “Direct” added to help with documentation vs program conflict; `connect_fs_neat`, `connect_full`, `connect_partial` split up.

`connect_full_nodirect` (*config*)

Create a fully-connected genome (except no direct *input* to *output* connections unless there are no *hidden nodes*).

**Parameters** `config` (*object*) – The genome configuration object.

`connect_full_direct` (*config*)

Create a fully-connected genome, including direct input-output connections.

**Parameters** `config` (*object*) – The genome configuration object.

`connect_partial_nodirect` (*config*)

Create a partially-connected genome, with (unless there are no *hidden nodes*) no direct input-output connections.

**Parameters** `config` (*object*) – The genome configuration object.

`connect_partial_direct` (*config*)

Create a partially-connected genome, possibly including direct input-output connections.

**Parameters** `config` (*object*) – The genome configuration object.

## graphs

Directed graph algorithm implementations.

`graphs.create_cycle` (*connections, test*)

Returns true if the addition of the *test connection* would create a cycle, assuming that no cycle already exists in the graph represented by *connections*. Used to avoid *recurrent* networks when a purely *feed-forward* network is desired (e.g., as determined by the `feed_forward` setting in the *configuration file*).

**Parameters**

- **connections** (*list(tuple(int, int))*) – The current network, as a list of (input, output) connection *identifiers*.
- **test** (*tuple(int, int)*) – Possible connection to be checked for causing a cycle.

**Returns** True if a cycle would be created; false if not.

**Return type** `bool`

`graphs.required_for_output` (*inputs, outputs, connections*)

Collect the *nodes* whose state is required to compute the final network output(s).

**Parameters**

- **inputs** (*list(int)*) – the *input node identifiers*; **it is assumed that the input identifier set and the node identifier set are disjoint.**
- **outputs** (*list(int)*) – the *output node identifiers*; by convention, the output node *ids* are always the same as the output index.
- **connections** (*list(tuple(int, int))*) – list of (input, output) connections in the network; should only include enabled ones.

**Returns** A list of layers, with each layer consisting of a set of node identifiers.

**Return type** `list(set(int))`

`graphs.feed_forward_layers` (*inputs, outputs, connections*)

Collect the layers whose members can be evaluated in parallel in a *feed-forward* network.



**Parameters**

- **inputs** (*list (int)*) – the network *input node identifiers*.
- **outputs** (*list (int)*) – the *output node identifiers*.
- **connections** (*list (tuple (int, int))*) – list of (input, output) connections in the network; should only include enabled ones.

**Returns** A list of layers, with each layer consisting of a set of *identifiers*; only includes nodes returned by `required_for_output`.

**Return type** `list(set(int))`

## indexer

Helps with creating new *identifiers/keys*.

**class** `indexer.Indexer` (*first*)

Initializes an Indexer instance with the internal ID counter set to `first`. This class functions to help with creating new (unique) identifiers/keys.

**Parameters** `first` (*int*) – The initial identifier (*key*) to be used.

**get\_next** (*result=None*)

If `result` is not `None`, then we return it unmodified. Otherwise, we return the next ID and increment our internal counter.

**Parameters** `result` (*int or None*) – Returned unmodified unless `None`.

**Returns** Identifier/*key* to use.

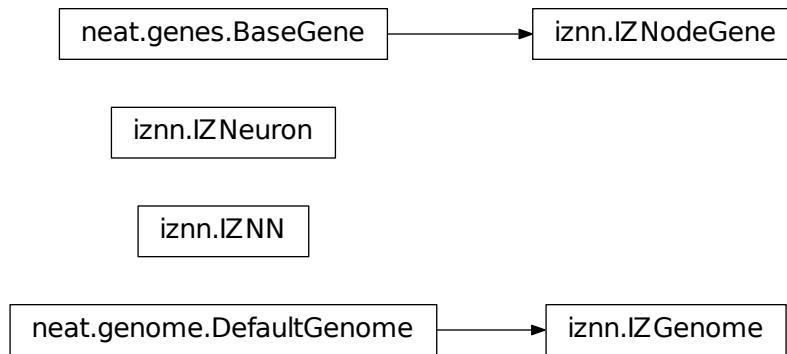
**Return type** `int`

## iznn

This module implements a spiking neural network. Neurons are based on the model described by:

Izhikevich, E. M.  
Simple Model of Spiking Neurons  
IEEE TRANSACTIONS ON NEURAL NETWORKS, VOL. 14, NO. 6, NOVEMBER 2003

See <http://www.izhikevich.org/publications/spikes.pdf>.



**class** `iznn.IZNodeGene` (*BaseGene*)

Contains attributes for the `iznn` *node* genes and determines *genomic distances*.

**class** `iznn.IZGenome` (*DefaultGenome*)

Contains the `parse_config` class method for `iznn` genome configuration.

**class** `iznn.IZNeuron` (*bias, a, b, c, d, inputs*)

Sets up and simulates the `iznn` *nodes* (neurons).

#### Parameters

- **bias** (*float*) – The bias of the neuron.
- **a** (*float*) – The time scale of the recovery variable.
- **b** (*float*) – The sensitivity of the recovery variable.
- **c** (*float*) – The after-spike reset value of the membrane potential.
- **d** (*float*) – The after-spike reset of the recovery variable.
- **inputs** (*list(tuple(int, float))*) – A list of (input key, weight) pairs for incoming connections.

**advance** (*dt\_msec*)

Advances simulation time for the neuron by the given time step in milliseconds. TODO: Currently has some numerical stability problems.

**Parameters** **dt\_msec** (*float*) – Time step in milliseconds.

**reset** ()

Resets all state variables.

**class** `iznn.IZNN` (*neurons, inputs, outputs*)

Sets up the network itself and simulates it using the connections and neurons.

#### Parameters

- **neurons** (*list*) – The *IZNeuron* instances needed.
- **inputs** (*list(int)*) – The *input node* keys.
- **outputs** (*list(int)*) – The *output node* keys.

**set\_inputs** (*inputs*)

Assigns input voltages.

**Parameters** **inputs** (*list(float)*) – The input voltages for the *input nodes*.

**reset** ()

Resets all neurons to their default state.

**get\_time\_step\_msec** ()

Returns a suggested time step; currently hardwired to 0.05. TODO: Investigate this (particularly effects on numerical stability issues).

**Returns** Suggested time step in milliseconds.

**Return type** *float*

**advance** (*dt\_msec*)

Advances simulation time for all neurons in the network by the input number of milliseconds.

**Parameters** **dt\_msec** (*float*) – How many milliseconds to advance the network.

**Returns** The values for the *output nodes*.

**Return type** *list(float)*

**static create** (*genome, config*)

Receives a genome and returns its phenotype (a neural network).

**Parameters**

- **genome** (*object*) – An IZGenome instance.
- **config** (*object*) – Configuration object.

**Returns** An IZNN instance.

**Return type** *object*

## math\_util

Contains some mathematical/statistical functions not found in the Python2 standard library, plus a mechanism for looking up some commonly used functions (such as for the *species\_fitness\_func*) by name.

**math\_util.stat\_functions**

Lookup table for commonly used {value} -> value functions; includes *max*, *min*, *mean*, and *median*. The *species\_fitness\_func* (used for *stagnation.DefaultStagnation*) is required to be one of these.

**math\_util.mean** (*values*)

Returns the arithmetic mean.

**math\_util.median** (*values*)

Returns the median. (Note: For even numbers of values, does not take the mean between the two middle values.)

**math\_util.variance** (*values*)

Returns the (population) variance.

**math\_util.stdev** (*values*)

Returns the (population) standard deviation. *Note spelling.*

**math\_util.softmax** (*values*)

Compute the softmax (a differentiable/smooth approximation of the maximum function) of the given value set. The softmax is defined as follows:  $v_i = \exp(v_i)/s$ , where  $s = \sum(\exp(v_0), \exp(v_1), \dots)$ (8.1).

## nn.feed\_forward

**class** nn.feed\_forward.**FeedForwardNetwork** (*inputs, outputs, node\_evals*)

A straightforward (no pun intended) *feed-forward* neural network NEAT implementation.

### Parameters

- **inputs** (*list (int)*) – The input *keys* (IDs).
- **outputs** (*list (int)*) – The output keys.
- **node\_evals** (*list (list (object))*) – A list of *node* descriptions, with each node represented by a list.

**activate** (*inputs*)

Feeds the inputs into the network and returns the resulting outputs.

**Parameters** **inputs** (*list*) – The values for the *input nodes*.

**Returns** The values for the *output nodes*.

**Return type** *list*

**static create** (*genome, config*)

Receives a genome and returns its phenotype (a *FeedForwardNetwork*).

## nn.recurrent

**class** nn.recurrent.**RecurrentNetwork** (*inputs, outputs, node\_evals*)

A *recurrent* (but otherwise straightforward) neural network NEAT implementation.

### Parameters

- **inputs** (*list (int)*) – The input *keys* (IDs).
- **outputs** (*list (int)*) – The output keys.
- **node\_evals** (*list (list (object))*) – A list of node descriptions, with each node represented by a list.

**reset** ()

Resets all node activations to 0 (necessary due to otherwise retaining state via recurrent connections).

**activate** (*inputs*)

Feeds the inputs into the network and returns the resulting outputs.

**Parameters** **inputs** (*list*) – The values for the *input nodes*.

**Returns** The values for the *output nodes*.

**Return type** *list*

**static create** (*genome, config*)

Receives a genome and returns its phenotype (a *RecurrentNetwork*).

## parallel

Runs evaluation functions in parallel subprocesses in order to evaluate multiple genomes at once.

**class** parallel.**ParallelEvaluator** (*num\_workers, eval\_function, timeout=None*)

Runs evaluation functions in parallel subprocesses in order to evaluate multiple genomes at once.

**Parameters**

- **num\_workers** (*int*) – How many workers to have in the `Pool`.
- **eval\_function** (*function*) – The `eval_function` should take one argument - a `tuple` of (genome object, config object) - and return a single `float` (the genome's fitness) Note that this is not the same as how a fitness function is called by `Population.run`.
- **timeout** (*int or None*) – How long (in seconds) each subprocess will be given before an exception is raised (unlimited if `None`).

**\_\_del\_\_** ()

Takes care of removing the subprocesses.

**evaluate** (*genomes, config*)

Distributes the evaluation jobs among the subprocesses, then assigns each fitness back to the appropriate genome.

**Parameters**

- **genomes** (*dict (int, object)*) – A dictionary of *genome\_id* (not used) to genome objects.
- **config** (*object*) – A `config.Config` object.

## population

Implements the core evolution algorithm.

**exception** `population.CompleteExtinctionException`

Raised on complete extinction (all species removed due to stagnation) unless *reset\_on\_extinction* is set.

**class** `population.Population` (*config, initial\_state=None*)

This class implements the core evolution algorithm: 1. Evaluate fitness of all genomes. 2. Check to see if the termination criterion is satisfied; exit if it is. 3. Generate the next generation from the current population. 4. Partition the new generation into species based on genetic similarity. 5. Go to 1.

**Parameters**

- **config** (*object*) – The `Config` configuration object.
- **initial\_state** (*None or tuple(object, object, int)*) – If supplied (such as by a method of the `Checkpointter` class), a tuple of (`Population`, `Species`, generation number)

**run** (*fitness\_function, n=None*)

Runs NEAT's genetic algorithm for at most n generations. If n is `None`, run until solution is found or extinction occurs.

The user-provided `fitness_function` must take only two arguments: 1. The population as a list of (genome id, genome) tuples. 2. The current configuration object.

The return value of the fitness function is ignored, but it must assign a Python `float` to the `fitness` member of each genome.

The fitness function is free to maintain external state, perform evaluations in *parallel*, etc.

It is assumed that the fitness function does not modify the list of genomes, the genomes themselves (apart from updating the fitness member), or the configuration object.

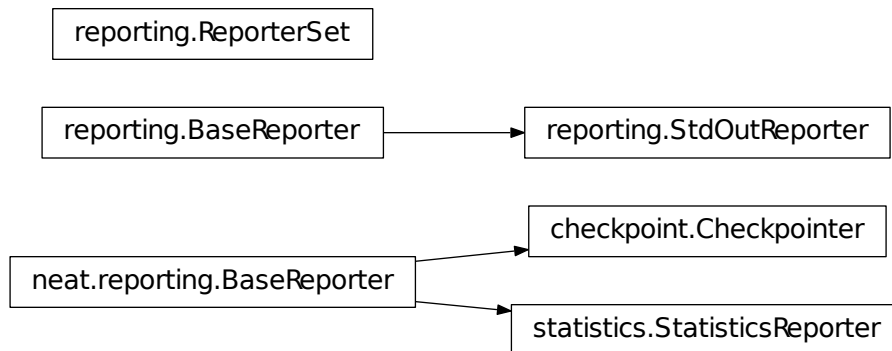
**Parameters**

- **fitness\_function** (*object*) – The fitness function to use, with arguments specified above.
- **n** (*int or None*) – The maximum number of generations to run (unlimited if *None*).

**Returns** The best genome seen.

**Return type** *object*

## reporting



**class** `reporting.ReporterSet`

Keeps track of the set of reporters and gives methods to dispatch them at appropriate points.

**add** (*reporter*)

Adds a reporter to those to be called via *ReporterSet* methods.

**Parameters** **reporter** (*object*) – A reporter instance.

**remove** (*reporter*)

Removes a reporter from those to be called via *ReporterSet* methods.

**Parameters** **reporter** (*object*) – A reporter instance.

**start\_generation** (*gen*)

Calls *start\_generation* on each reporter in the set.

**Parameters** **gen** (*int*) – The generation number.

**end\_generation** (*config, population, species*)

Calls *end\_generation* on each reporter in the set.

**Parameters**

- **config** (*object*) – *Config* configuration object.
- **population** (*dict(int, object)*) – Current population, as a dict of unique genome *ID/key* vs genome.
- **species** (*object*) – Current species set object, such as a *DefaultSpeciesSet*.

**post\_evaluate** (*config, population, species*)

Calls *post\_evaluate* on each reporter in the set.

**Parameters**

- **config** (*object*) – *Config* configuration object.
- **population** (*dict*(*int*, *object*)) – Current population, as a dict of unique genome *ID/key* vs genome.
- **species** (*object*) – Current species set object, such as a *DefaultSpeciesSet*.
- **best\_genome** (*object*) – The currently highest-fitness *genome*. (Ties are resolved pseudorandomly, by *dictionary* ordering.)

**post\_reproduction** (*config*, *population*, *species*)

Not currently called. Would call *post\_reproduction* on each reporter in the set.

**complete\_extinction** ()

Calls *complete\_extinction* on each reporter in the set.

**found\_solution** (*config*, *generation*, *best*)

Calls *found\_solution* on each reporter in the set.

**Parameters**

- **config** (*object*) – *Config* configuration object.
- **generation** (*int*) – Generation number.
- **best** (*object*) – The currently highest-fitness *genome*. (Ties are resolved pseudorandomly by *dictionary* ordering.)

**species\_stagnant** (*sid*, *species*)

Calls *species\_stagnant* on each reporter in the set.

**Parameters**

- **sid** (*int*) – The species *id/key*.
- **species** (*object*) – The *Species* object.

**info** (*msg*)

Calls *info* on each reporter in the set.

**Parameters** **msg** (*str*) – Message to be handled.

**class** reporting.**BaseReporter**

Abstract class defining the reporter interface expected by *ReporterSet*. Inheriting from it will provide a set of dummy methods to be overridden as desired, as follows:

**start\_generation** (*generation*)

Called via *ReporterSet* (by *population.Population.run()*) at the start of each generation, prior to the invocation of the fitness function.

**Parameters** **generation** (*int*) – The generation number.

**end\_generation** (*config*, *population*, *species*)

Called via *ReporterSet* (by *population.Population.run()*) at the end of each generation, after reproduction and speciation.

**Parameters**

- **config** (*object*) – *Config* configuration object.
- **population** (*dict*(*int*, *object*)) – Current population, as a dict of unique genome *ID/key* vs genome.
- **species** (*object*) – Current species set object, such as a *DefaultSpeciesSet*.

**post\_evaluate** (*config*, *population*, *species*, *best\_genome*)

Called via *ReporterSet* (by *population.Population.run()*) after the fitness function is finished.

**Parameters**

- **config** (*object*) – *Config* configuration object.
- **population** (*dict*(*int*, *object*)) – Current population, as a dict of unique genome *ID/key* vs genome.

- **species** (*object*) – Current species set object, such as a *DefaultSpeciesSet*.
- **best\_genome** (*object*) – The currently highest-fitness *genome*. (Ties are resolved pseudorandomly, by *dictionary* ordering.)

**post\_reproduction** (*config, population, species*)

Not currently called (indirectly or directly), including by either *population.Population.run()* or *reproduction.DefaultReproduction*. Note: New members of the population likely will not have a set species.

**complete\_extinction** ()

Called via *ReporterSet* (by *population.Population.run()*) if complete extinction (due to stagnation) occurs, prior to (depending on the *reset\_on\_extinction* configuration setting) a new population being created or a *population.CompleteExtinctionException* being raised.

**found\_solution** (*config, generation, best*)

Called via *ReporterSet* (by *population.Population.run()*) prior to exiting if the configured *fitness threshold* is met. (Note: Not called upon reaching the generation maximum - set when calling *population.Population.run()* - and exiting for this reason.)

**Parameters**

- **config** (*object*) – *Config* configuration object.
- **generation** (*int*) – Generation number.
- **best** (*object*) – The currently highest-fitness *genome*. (Ties are resolved pseudorandomly by *dictionary* ordering.)

**species\_stagnant** (*sid, species*)

Called via *ReporterSet* (by *reproduction.DefaultReproduction.reproduce()*) for each species considered stagnant by the stagnation class (such as *stagnation.DefaultStagnation*).

**Parameters**

- **sid** (*int*) – The species *id/key*.
- **species** (*object*) – The *Species* object.

**info** (*msg*)

Miscellaneous informational messages, from multiple parts of the library, called via *ReporterSet*.

**Parameters** **msg** (*str*) – Message to be handled.

**class** `reporting.StdoutReporter` (*show\_species\_detail*)

Uses print to output information about the run; an example reporter class.

**Parameters** **show\_species\_detail** (*bool*) – Whether or not to show additional details about each species in the population.

## reproduction

**class** `reproduction.DefaultReproduction` (*config, reporters, stagnation*)

Handles creation of genomes, either from scratch or by sexual or asexual reproduction from parents. Implements the default NEAT-python reproduction scheme: explicit fitness sharing with fixed-time species stagnation. For class requirements, see *Reproduction Interface*.

**Parameters**

- **config** (*dict*) – Configuration object, in this implementation a dictionary.
- **reporters** (*object*) – A *ReporterSet* object.



- **stagnation** (*object*) – A *DefaultStagnation* object - the current code partially depends on internals of this class (a TODO is noted to correct this).

**classmethod parse\_config** (*param\_dict*)

Required interface method. Provides defaults for elitism, survival\_threshold, and min\_species\_size parameters and updates them from the *configuration file*. TODO: Use a separate configuration class, for consistency with other types (done in the *config\_work* branch).

**Parameters** **param\_dict** (*dict*) – Dictionary of parameters from configuration file.

**Returns** Configuration object; considered opaque by rest of code, so current type returned is not required for interface.

**Return type** *dict*

**classmethod write\_config** (*f, param\_dict*)

Required interface method. Saves elitism and survival\_threshold (but not min\_species\_size) parameters to new config file. (Inconsistency re min\_species\_size fixed in the *config\_work* branch.)

**Parameters**

- **f** (*file*) – File object to write to.
- **param\_dict** (*dict*) – Dictionary of current parameters in this implementation; more generally, reproduction config object.

**create\_new** (*genome\_type, genome\_config, num\_genomes*)

Required interface method. Creates num\_genomes new genomes of the given type using the given configuration. Also initializes ancestry information (as an empty tuple).

**Parameters**

- **genome\_type** (*class*) – Genome class (such as *DefaultGenome* or *iznn.IZGenome*) of which to create instances.
- **genome\_config** (*object*) – Opaque genome configuration object.
- **num\_genomes** (*int*) – How many new genomes to create.

**Returns** A dictionary (with the unique genome identifier as the key) of the genomes created.

**Return type** *dict(int, object)*

**static compute\_spawn** (*adjusted\_fitness, previous\_sizes, pop\_size, min\_species\_size*)

Apportions desired number of members per species according to fitness (adjusted by *reproduce()* to a 0-1 scale) from out of the desired population size.

**Parameters**

- **adjusted\_fitness** (*list(float)*) – Mean fitness for species members, adjusted to 0-1 scale (see below).
- **previous\_sizes** (*list(int)*) – Number of members of species in population prior to reproduction.
- **pop\_size** (*int*) – Desired population size, as input to *reproduce()*.
- **min\_species\_size** (*int*) – Minimum number of members per species; can result in population size being above pop\_size.

**reproduce** (*config, species, pop\_size, generation*)

Required interface method. Creates the population to be used in the next generation from the given configuration instance, SpeciesSet instance, desired size of the population, and current generation number. This method is called after all genomes have been evaluated and their fitness member assigned. This method should use the stagnation instance given to the initializer to remove species deemed to have stagnated. Note: Determines relative fitnesses by transforming into (ideally) a 0-1 scale; however, if the top and bottom fitnesses are not at least 1 apart, the range may be less than 0-1, as a check against dividing by a too-small number. TODO: Make minimum difference configurable (defaulting to 1 to preserve compatibility).

**Parameters**

- **config** (*object*) – A *Config* instance.
- **species** (*object*) – A *DefaultSpeciesSet* instance. As well as depending on some of the *DefaultStagnation* internals, this method also depends on some of those of the *DefaultSpeciesSet* and its referenced species objects.
- **pop\_size** (*int*) – Population size desired, such as set in the *configuration file*.
- **generation** (*int*) – Generation count.

**Returns** New population, as a dict of unique genome *ID/key* vs *genome*.

**Return type** `dict(int, object)`

## six\_util

This Python 2/3 portability code was copied from the `six` module to avoid adding it as a dependency.

```
six_util.iterkeys(d, **kw)
```

This function returns an iterator over the keys of dict `d`.

### Parameters

- **d** (*dict*) – Dictionary to iterate over
- **kw** – The function of this parameter is unclear.

```
six_util.iteritems(d, **kw)
```

This function returns an iterator over the (key, value) pairs of dict `d`.

### Parameters

- **d** (*dict*) – Dictionary to iterate over
- **kw** – The function of this parameter is unclear.

```
six_util.itervalues(d, **kw)
```

This function returns an iterator over the values of dict `d`.

### Parameters

- **d** (*dict*) – Dictionary to iterate over
- **kw** – The function of this parameter is unclear.

## species

Divides the population into species based on *genomic distances*.

```
class species.Species(key, generation)
```

Represents a *species* and contains data about it such as members, fitness, and time stagnating. Note: *stagnation.DefaultStagnation* manipulates many of these.

### Parameters

- **key** (*int*) – *Identifier/key*
- **generation** (*int*) – Initial generation of appearance

```
update(representative, members)
```

Required interface method. Updates a species instance with the current members and most-representative member (from which *genomic distances* are measured).

### Parameters

- **representative** (*object*) – A genome instance.
- **members** – A *dictionary* of genome *id* vs genome instance.

**get\_fitnesses** ()

Required interface method (used by *stagnation.DefaultStagnation*, for instance).  
Retrieves the fitnesses of each member genome.

**Returns** List of fitnesses of member genomes.

**Return type** list(float)

**class** *species.GenomeDistanceCache* (*config*)

Caches (indexing by *genome key/id*) *genomic distance* information to avoid repeated lookups. (The *distance function* is among the most time-consuming parts of the library, although many fitness functions are likely to far outweigh this for moderate-size populations.)

**Parameters** **config** (*object*) – A genome configuration object; later used by the genome distance function.

**\_\_call\_\_** (*genome0, genome1*)

GenomeDistanceCache is called as a method with a pair of genomes to retrieve the distance.

**Parameters**

- **genome0** (*object*) – The first genome object.
- **genome1** (*object*) – The second genome object.

**Returns** The *genomic distance*.

**Return type** float

**class** *species.DefaultSpeciesSet* (*config, reporters*)

Encapsulates the default speciation scheme by configuring it and performing the speciation function (placing genomes into species by genetic similarity). *reproduction.DefaultReproduction* currently depends on this having a *species* attribute consisting of a dictionary of species keys to species.

**Parameters**

- **config** (*object*) – A configuration object (currently unused).
- **reporters** (*object*) – A *ReporterSet* instance giving reporters to be notified about *genomic distance* statistics.

**classmethod** **parse\_config** (*param\_dict*)

Required interface method. Currently, the only configuration parameter is the *compatibility\_threshold*. TODO: Use a separate configuration class, for consistency with other types (done in the *config\_work* branch).

**Parameters** **param\_dict** (*dict(str, str)*) – Dictionary of parameters from configuration file.

**Returns** Configuration object; considered opaque by rest of code, so current type returned is not required for interface.

**Return type** dict

**classmethod** **write\_config** (*f, param\_dict*)

Required interface method. Writes parameter(s) to new config file.

**Parameters**

- **f** (*file*) – File object to write to.
- **param\_dict** (*dict*) – Dictionary of current parameters in this implementation; more generally, stagnation config object.

**speciate** (*config, population, generation*)

Required interface method. Place genomes into species by genetic similarity (*genomic distance*). (The current code has a *docstring* stating that there may be a problem if all old species representatives are not dropped for each generation; it is not clear how this is consistent

with the code in `reproduction.DefaultReproduction.reproduce()`, such as for elitism.)

**Parameters**

- **config** (*object*) – *Config* object.
- **population** (*dict(int, object)*) – Population as per the output of `DefaultReproduction.reproduce`.
- **generation** (*int*) – Current generation number.

**get\_species\_id** (*individual\_id*)

Required interface method (used by `reporting.StdoutReporter`). Retrieves species *id/key* for a given genome *id/key*.

**Parameters** **individual\_id** (*int*) – Genome *id/key*.

**Returns** Species *id/key*.

**Return type** *int*

**get\_species** (*individual\_id*)

Retrieves species object for a given genome *id/key*. May become a required interface method, and useful for some fitness functions already.

**Parameters** **individual\_id** (*int*) – Genome *id/key*.

**Returns** *Species* containing the genome corresponding to the *id/key*.

**Return type** *object*

---

**Note:** TODO: Currently, depending on the settings for `species_fitness_func` and `fitness_criterion`, it is possible for a species with members **above** the `fitness_threshold` level of fitness to be considered “stagnant” (including, most problematically, because they are at the limit of fitness improvement).

---

## stagnation

**class** `stagnation.DefaultStagnation` (*config, reporters*)

Keeps track of whether species are making progress and helps remove ones that, for a *configurable number of generations*, are not.

**Parameters**

- **config** (*object*) – Configuration object; in this implementation, a *dictionary*, but should be treated as opaque outside this class.
- **reporters** (*object*) – A *ReporterSet* instance with reporters that may need activating; not currently used.

**classmethod** `parse_config` (*param\_dict*)

Required interface method. Provides defaults for `species_fitness_func`, `max_stagnation`, and `species_elitism` parameters and updates them from the configuration file. TODO: Use a separate configuration class, for consistency with other types (done in the `config_work` branch).

**Parameters** **param\_dict** (*dict(str, str)*) – Dictionary of parameters from configuration file.

**Returns** Configuration object; considered opaque by rest of code, so current type returned is not required for interface.

**Return type** *dict*

**classmethod** `write_config` (*f, param\_dict*)

Required interface method. Saves parameters to new config file. TODO: Has a default of 15 for `species_elitism`, but will be overridden by the default of 0 in `parse_config` (fixed in the `config_work` branch).

**Parameters**

- **f** (*file*) – File object to write to.
- **param\_dict** (*dict*) – Dictionary of current parameters in this implementation; more generally, stagnation config object.

**update** (*species\_set, generation*)

Required interface method. Updates species fitness history information, checking for ones that have not improved in *max\_stagnation* generations, and - unless it would result in the number of species dropping below the configured *species\_elitism* if they were removed, in which case the highest-fitness species are spared - returns a list with stagnant species marked for removal. TODO: Currently interacts directly with the internals of the *species.Species* object. Also, currently **both** checks for *num\_non\_stagnant* to stop marking stagnant **and** does not allow the top *species\_elitism* species to be marked stagnant. While the latter could admittedly help with the problem mentioned above, the ordering of species fitness is using the fitness gotten from the *species\_fitness\_func* (and thus may miss high-fitness members of overall low-fitness species, depending on the function in use).

**Parameters**

- **species\_set** (*object*) – A *species.DefaultSpeciesSet* or compatible object.
- **generation** (*int*) – The current generation.

**Returns** A list of tuples of (species *id/key*, *Species* object, *is\_stagnant*).

**Return type** `list(tuple(int, object, bool))`

Changed in version 0.91-github: Species sorted to avoid marking best-performing as stagnant even with *species\_elitism*.

## statistics

---

**Note:** There are two design decisions to be aware of: \* The most-fit genomes are based on the highest-fitness member of each generation; other genomes are not saved by this module (if they were, it would far worsen existing potential memory problems - see below), and it is assumed that fitnesses (as given by the fitness function) are not relative to others in the generation (also assumed by the use of the *fitness threshold* as a signal for exiting). Code violating this assumption (e.g., with competitive coevolution) will need to use different statistical gathering methods. \* Generally reports or records a per-generation list of values; the numeric position in the list may not correspond to the generation number if there has been a restart, such as via the *checkpoint* module. There is also a TODO item: Currently keeps accumulating information in memory, which may be a problem in long runs.

---

**class** `statistics.StatisticsReporter` (*BaseReporter*)

Gathers (via the reporting interface) and provides (to callers and/or to a file) the most-fit genomes and information on genome and species fitness and species sizes.

**post\_evaluate** (*config, population, species, best\_genome*)

Called as part of the *reporting.BaseReporter* interface after the evaluation at the start of each generation; see *BaseReporter.post\_evaluate*. Information gathered includes a copy of the best genome in each generation and the fitnesses of each member of each species.

**get\_fitness\_stat** (*f*)

Calls the given function on the genome fitness data from each recorded generation and returns the resulting list.

**Parameters** **f** (*function*) – A function that takes a list of scores and returns a summary statistic (or, by returning a list or tuple, multiple statistics) such as `mean` or `stdev`.

**Returns** A list of the results from function *f* for each generation.

**Return type** `list`

**get\_fitness\_mean()**

Gets the per-generation average fitness. A wrapper for `get_fitness_stat()` with the function being `mean`.

**Returns** List of mean genome fitnesses for each generation.

**Return type** `list(float)`

**get\_fitness\_stdev()**

Gets the per-generation standard deviation of the fitness. A wrapper for `get_fitness_stat()` with the function being `stdev`.

**Returns** List of standard deviations of genome fitnesses for each generation.

**Return type** `list(float)`

**best\_unique\_genomes(n)**

Returns the *n* most-fit genomes, with no duplication (from the most-fit genome passing unaltered to the next generation), sorted in decreasing fitness order.

**Parameters** *n* (`int`) – Number of most-fit genomes to return.

**Returns** List of *n* most-fit genomes (as genome objects).

**Return type** `list(object)`

**best\_genomes(n)**

Returns the *n* most-fit genomes, possibly with duplicates, sorted in decreasing fitness order.

**Parameters** *n* (`int`) – Number of most-fit genomes to return.

**Returns** List of *n* most-fit genomes (as genome objects).

**Return type** `list(object)`

**best\_genome()**

Returns the most-fit genome ever seen. A wrapper around `best_genomes()`.

**Returns** The most-fit genome.

**Return type** `object`

**get\_species\_sizes()**

Returns a by-generation list of lists of species sizes. Note that some values may be 0, if a species has either not yet been seen or has been removed due to *stagnation*; species without generational overlap may be more similar in *genomic distance* than the configured *compatibility\_threshold* would otherwise allow.

**Returns** List of lists of species sizes, ordered by species *id/key*.

**Return type** `list(list(int))`

**get\_species\_fitness(null\_value='')**

Returns a by-generation list of lists of species fitnesses; the fitness of a species is determined by the mean fitness of the genomes in the species, as with the reproduction distribution by *reproduction.DefaultReproduction*. The *null\_value* parameter is used for species not present in a particular generation (see *above*).

**Parameters** *null\_value* (`str`) – What to put in the list if the species is not present in a particular generation.

**Returns** List of lists of species fitnesses, ordered by species *id/key*.

**Return type** `list(list(float or str))`

**save\_genome\_fitness(delimiter=',', filename='fitness\_history.csv',  
with\_cross\_validation=False)**

Saves the population's best and mean fitness (using the `csv` package). At some point in the future, cross-validation fitness may be usable (via, for instance, the fitness function using alternative test situations/opponents and recording this in a `cross_fitness` attribute; this can be used for, e.g., preventing overfitting); currently, `with_cross_validation` should always be left at its `False` default.

**Parameters**

- **delimiter** (*str*) – Delimiter between columns in the file; note that the default is not ‘,’ as may be otherwise implied by the `csv` file extension (which refers to the package used).
- **filename** (*str*) – The filename to open (for writing, not appending) and write to.
- **with\_cross\_validation** (*bool*) – For future use; currently, leave at its `False` default.

**save\_species\_count** (*delimiter*=‘,’, *filename*=‘speciation.csv’)

Logs speciation throughout evolution, by tracking the number of genomes in each species. Uses `get_species_sizes()`; see that method for more information.

**Parameters**

- **delimiter** (*str*) – Delimiter between columns in the file; note that the default is not ‘,’ as may be otherwise implied by the `csv` file extension (which refers to the `csv` package used).
- **filename** (*str*) – The filename to open (for writing, not appending) and write to.

**save\_species\_fitness** (*delimiter*=‘,’, *null\_value*=‘NA’, *filename*=‘species\_fitness.csv’)

Logs species’ mean fitness throughout evolution. Uses `get_species_fitness()`; see that method for more information on, for instance, `null_value`.

**Parameters**

- **delimiter** (*str*) – Delimiter between columns in the file; note that the default is not ‘,’ as may be otherwise implied by the `csv` file extension (which refers to the `csv` package used).
- **null\_value** (*str*) – See `get_species_fitness()`.
- **filename** (*str*) – The filename to open (for writing, not appending) and write to.

**save()**

A wrapper for `save_genome_fitness()`, `save_species_count()`, and `save_species_fitness()`; uses the default values for all three.

*Table of Contents*





---

## Genome Interface

---

This is an outline of the minimal interface that is expected to be present on genome objects; example genome objects can be seen in *DefaultGenome* and *iznn.IZGenome*.

### Class Methods

*parse\_config(cls, param\_dict)*

Takes a dictionary of configuration items, returns an object that will later be passed to the *write\_config* method. This configuration object is considered to be opaque by the rest of the library.

*write\_config(cls, f, config)*

Takes a file-like object and the configuration object created by *parse\_config*. This method should write the configuration item definitions to the given file.

### Initialization/Reproduction

*\_\_init\_\_(self, key)*

Takes a unique genome instance identifier. The initializer should create the following members:

- *key*
- *connections* - (gene\_key, gene) pairs for the connection gene set.
- *nodes* - (gene\_key, gene) pairs for the node gene set.
- *fitness*

*configure\_new(self, config)*

Configure the genome as a new random genome based on the given configuration from the top-level *Config* object.

## Crossover/Mutation

*configure\_crossover(self, genome1, genome2, config)*

Configure the genome as a child of the given parent genomes.

*mutate(self, config)*

Apply mutation operations to the genome, using the given configuration.

## Speciation/Misc

*distance(self, other, config)*

Returns the genomic distance between this genome and the other. This distance value is used to compute genome compatibility for speciation.

*size(self)*

Returns a measure of genome complexity. This object is currently only given to reporters at the end of a generation to indicate the complexity of the highest-fitness genome. In the DefaultGenome class, this method currently returns (number of nodes, number of enabled connections).

---

## Reproduction Interface

---

This is an outline of the minimal interface that is expected to be present on reproduction objects. Each Population instance will create exactly one instance of the reproduction class in *Population.\_\_init\_\_* regardless of the configuration or arguments provided to *Population.\_\_init\_\_*.

### Class Methods

*parse\_config(cls, param\_dict)* - Takes a dictionary of configuration items, returns an object that will later be passed to the *write\_config* method. This configuration object is considered to be opaque by the rest of the library.

*write\_config(cls, f, config)* - Takes a file-like object and the configuration object created by *parse\_config*. This method should write the configuration item definitions to the given file.

### Initialization

*\_\_init\_\_(self, config, reporters, stagnation)* - Takes the top-level Config object, a ReporterSet instance, and a stagnation object instance.

### Other methods

*create\_new(self, genome\_type, genome\_config, num\_genomes)*: - Create *num\_genomes* new genomes of the given type using the given configuration.

*reproduce(self, config, species, pop\_size, generation)*: - Creates the population to be used in the next generation from the given configuration instance, SpeciesSet instance, desired size of the population, and current generation number. This method is called after all genomes have been evaluated and their *fitness* member assigned. This method should use the stagnation instance given to the initializer to remove species it deems to have stagnated.



**activation function****aggregation function****bias**

**response** These are the *attributes* of a *node*. They determine the output of a node as follows:  $\text{activation}(\text{bias} + (\text{response} * \text{aggregation}(\text{inputs})))$  (11.1) For available activation functions, see *Overview of builtin activation functions*; for adding new ones, see *Customizing Behavior*.

These are the properties of a *node* (such as its *activation function*) or *connection* (such as whether it is *enabled* or not) determined by its associated *gene* (in the default implementation, in the *attributes* module in combination with the gene class).

These connect between *nodes*, and give rise to the *network* in the term neural network. For non-loopback (directly *recurrent*) connections, they are equivalent to biological synapses. Connections have two *attributes*, their *weight* and whether or not they are *enabled*; both are determined by their *gene*. An example gene class for connections can be seen in *genes.DefaultConnectionGene*.

A discrete-time neural network (which should be assumed unless specified otherwise) proceeds in time steps, with processing at one *node* followed by going through *connections* to other nodes followed by processing at those other nodes, eventually giving the output. A continuous-time neural network, such as the *ctrnn* (continuous-time *recurrent* neural network) implemented in NEAT-Python, simulates a continuous process via differential equations (or other methods).

The process in sexual reproduction in which two *genomes* are combined. This involves the combination of *homologous* genes and the copying (from the highest-fitness genome) of *disjoint/excess* genes. Along with *mutation*, one of the two sources of innovation in (classical) evolution.

These are genes in NEAT not descended from a common ancestor - i.e., not *homologous*. This implementation of NEAT, like most, does not distinguish between disjoint and excess genes. For further discussion, see the *NEAT Overview*.

A neural network that is not *recurrent* is feedforward - it has no loops. (Note that this means that it has no memory - no ability to take into account past events.) It can thus be described as a **DAG (Directed Acyclic Graph)**.

The information coding (in the current implementation) for a particular aspect (*node* or *connection*) of a neural network phenotype. Contains several *attributes*, varying depending on the type of gene. Example gene classes include `genes.DefaultNodeGene`, `genes.DefaultConnectionGene`, and `iznn.IZNodeGene`; all of these are subclasses of `genes.BaseGene`.

The distance between two *homologous genes*, added up as part of the *genomic distance*. Also sometimes used as a synonym for *genomic distance*.

The set of *genes* that together code for a (neural network) phenotype. Example genome objects can be seen in `genome.DefaultGenome` and `iznn.IZGenome`, and the object interface is described in *Genome Interface*.

An approximate measure of the difference between *genomes*, used in dividing the population into *species*. For further discussion, see the *NEAT Overview*.

These are the *nodes* other than *input nodes* and *output nodes*. In the original NEAT (NeuroEvolution of Augmenting Topologies) *algorithm*, networks start with no hidden nodes, and evolve more complexity as necessary - thus “Augmenting Topologies”.

Descended from a common ancestor; two genes in NEAT from different genomes are either homologous or *dis-joint/excess*. In NEAT, two genes that are homologous will have the same *key/id*. For *node* genes, the key is an `int` incremented with each newly-created node; for *connection* genes, the key is a `tuple` of the keys of the nodes being connected. For further discussion, see the *NEAT Overview*.

Various of the objects used by the library are indexed by an key (id); for most, this is an `int`, which is either unique in the library as a whole (as with *species* and *genomes*), or within a genome (as with *node genes*). For *connection* genes, this is a `tuple` of two `ints`, the keys of the connected nodes.

These are the *nodes* through which the network receives inputs. They cannot be deleted (although *connections* from them can be), cannot be the output end of a *connection*, and have: no *aggregation function*; a fixed *bias* of 0; a fixed *response* multiplier of 1; and a fixed *activation function* of *identity*. Note: In the *genome* module, they are not in many respects treated as actual nodes, but simply as *keys* for input ends of connections. Sometimes known as an input *pin*.

The process in which the *attributes* of a *gene* (or the genes in a *genome*) are (randomly, with likelihoods determined by configuration parameters) altered. Along with *crossover*, one of the two sources of innovation in (classical) evolution.

Also known as a neuron (as in a *neural network*). They are of three types: *input*, *hidden*, and *output*. Nodes have one or more *attributes*, such as an *activation function*; all are determined by their *gene*. Classes of node genes include `genes.DefaultNodeGene` and `iznn.IZNodeGene`.

These are the *nodes* to which the network delivers outputs. They cannot be deleted (although *connections* to them can be) but can otherwise be *mutated* normally. The output of this node is connected to the corresponding output *pin* with an implicit *weight*-1, *enabled* connection.

Point at which the network is effectively connected to the external world. Pins are either input (aka *input nodes*) or output (connected to an *output node* with the same *key* as the output pin).

A recurrent neural network has cycles in its topography. These may be a *node* having a *connection* back to itself, with (for a *discrete-time* neural network) the prior time period’s output being provided to the node as one of its inputs. They may also have longer cycles, such as with output from node A going into node B (via a connection) and an output from node B going (via another connection) into node A. (This gives it a possibly-useful memory - an ability to take into account past events - unlike a *feedforward* neural network; however, it also makes it harder to work with in some respects.)

Subdivisions of the population into groups of similar (by the *genomic distance* measure) individuals (*genomes*), which compete among themselves but share fitness relative to the rest of the population. This is, among other things, a mechanism to try to avoid the quick elimination of high-potential topological mutants that have an initial poor fitness prior to smaller “tuning” changes. For further discussion, see the *NEAT Overview*.

These are the *attributes* of a *connection*. If a connection is enabled, then the input to it (from a *node*) is multiplied by the weight then sent to the output (to a node - possibly the same node, for a *recurrent* neural network). If a connection is not enabled, then the output is 0; genes for such connections are the equivalent of *pseudogenes* that, as in *in vivo* evolution, can be reactivated at a later time. TODO: Some versions of NEAT give a chance, such as 25%, that a disabled connection will be enabled during *crossover*; in the future, this should be an option.

*Table of Contents*





## CHAPTER 12

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



**a**

activations, 31  
attributes, 32

**c**

checkpoint, 34  
config, 35  
ctrnn, 36

**g**

genes, 37  
genome, 39  
graphs, 44

**i**

indexer, 45  
iznn, 45

**m**

math\_util, 47

**n**

nn.feed\_forward, 47  
nn.recurrent, 48

**p**

parallel, 48  
population, 49

**r**

reporting, 50  
reproduction, 52

**s**

six\_util, 54  
species, 54  
stagnation, 56  
statistics, 57



## Symbols

`__call__()` (species.GenomeDistanceCache method), 55  
`__del__()` (parallel.ParallelEvaluator method), 49  
`__lt__()` (genes.BaseGene method), 38  
`__repr__()` (config.ConfigParameter method), 35  
`__str__()` (genes.BaseGene method), 38  
`__str__()` (genome.DefaultGenome method), 43

## A

`activate()` (nn.feed\_forward.FeedForwardNetwork method), 48  
`activate()` (nn.recurrent.RecurrentNetwork method), 48  
activation function, 9, 19, 23, 31, 40, 65  
ActivationFunctionSet (class in activations), 31  
activations (module), 31  
`add()` (activations.ActivationFunctionSet method), 31  
`add()` (reporting.ReporterSet method), 50  
`add_activation()` (genome.DefaultGenomeConfig method), 40  
`add_connection()` (genome.DefaultGenome method), 42  
`advance()` (ctrnn.CTRNN method), 37  
`advance()` (iznn.IZNeuron method), 46  
`advance()` (iznn.IZNN method), 47  
aggregation function, 9, 40, 65  
attributes, 9–11  
attributes (module), 32

## B

BaseAttribute (class in attributes), 32  
BaseGene (class in genes), 38  
BaseReporter (class in reporting), 51  
`best_genome()` (statistics.StatisticsReporter method), 58  
`best_genomes()` (statistics.StatisticsReporter method), 58  
`best_unique_genomes()` (statistics.StatisticsReporter method), 58  
bias, 9, 65  
BoolAttribute (class in attributes), 33

## C

checkpoint (module), 34

Checkpointner (class in checkpoint), 34  
`clamp()` (attributes.FloatAttribute method), 32  
compatibility\_disjoint\_coefficient, 9, 42  
compatibility\_threshold, 9, 55  
compatibility\_weight\_coefficient, 9, 39  
`complete_extinction()`, 49  
`complete_extinction()` (reporting.BaseReporter method), 52  
`complete_extinction()` (reporting.ReporterSet method), 51  
CompleteExtinctionException, 49  
`compute_full_connections()` (genome.DefaultGenome method), 43  
`compute_spawn()` (reproduction.DefaultReproduction static method), 53  
Config (class in config), 36  
config (module), 35  
`config_item_names()` (attributes.BaseAttribute method), 32  
ConfigParameter (class in config), 35  
`configure_crossover()` (genome.DefaultGenome method), 41  
`configure_new()` (genome.DefaultGenome method), 41  
`connect_fs_neat_hidden()` (genome.DefaultGenome method), 43  
`connect_fs_neat_nohidden()` (genome.DefaultGenome method), 43  
`connect_full_direct()` (genome.DefaultGenome method), 44  
`connect_full_nodirect()` (genome.DefaultGenome method), 43  
`connect_partial_direct()` (genome.DefaultGenome method), 44  
`connect_partial_nodirect()` (genome.DefaultGenome method), 44  
connection, 10, 11, 39, 42, 42, 43  
continuous-time, 29, 37  
`copy()` (genes.BaseGene method), 39  
`create()` (ctrnn.CTRNN static method), 37  
`create()` (iznn.IZNN static method), 47

`create()` (`nn.feed_forward.FeedForwardNetwork` static method), 48  
`create()` (`nn.recurrent.RecurrentNetwork` static method), 48  
`create_connection()` (`genome.DefaultGenome` static method), 43  
`create_new()` (`reproduction.DefaultReproduction` method), 53  
`create_node()` (`genome.DefaultGenome` static method), 43  
`creates_cycle()` (in module `graphs`), 44  
`crossover`, **3**, **39**, **41**  
`crossover()` (`genes.BaseGene` method), 39  
`ctrnn`, **29**  
`CTRNN` (class in `ctrnn`), 37  
`ctrnn` (module), 36  
`CTRNNNodeEval` (class in `ctrnn`), 36

## D

`DefaultConnectionGene` (class in `genes`), 39  
`DefaultGenome`, 9, 20, 61  
`DefaultGenome` (class in `genome`), 41  
`DefaultGenomeConfig` (class in `genome`), 40  
`DefaultNodeGene` (class in `genes`), 39  
`DefaultReproduction`, 8, 20  
`DefaultReproduction` (class in `reproduction`), 52  
`DefaultSpeciesSet` (class in `species`), 55  
`DefaultStagnation`, 8  
`DefaultStagnation` (class in `stagnation`), 56  
`disjoint`, **3**, **9**  
`distance()` (`genes.DefaultConnectionGene` method), 39  
`distance()` (`genes.DefaultNodeGene` method), 39  
`distance()` (`genome.DefaultGenome` method), 42

## E

`elitism`, **8**, **53**, **53**  
`enabled`, 10  
`enabled_default`, **10**, **10**  
`end_generation()`, 49  
`end_generation()` (`reporting.BaseReporter` method), 51  
`end_generation()` (`reporting.ReporterSet` method), 50  
`evaluate()` (`parallel.ParallelEvaluator` method), 49  
`excess`, **3**

## F

`feed-forward`, *see* `feedforward`  
`feed_forward`, **10**, **42**, 44  
`feed_forward_layers()` (in module `graphs`), 44  
`feedforward`, 10, 44  
`FeedForwardNetwork` (class in `nn.feed_forward`), 48  
`fitness function`, **3**, **13**, **49**, **53**, **57**  
`fitness_criterion`, **7**, **35**, **49**, **56**  
`fitness_threshold`, **7**, **35**, **49**, **52**, **56**  
`FloatAttribute` (class in `attributes`), 32

`format()` (`config.ConfigParameter` method), 35  
`found_solution()`, 7, **49**, **52**  
`found_solution()` (`reporting.BaseReporter` method), 52  
`found_solution()` (`reporting.ReporterSet` method), 51

## G

`gene`, **38**, **46**  
`genes` (module), 37  
`genetic distance`, **39**, **42**  
`genome`, 9, 20, 46, 53, **61**  
`genome` (module), 39  
`GenomeDistanceCache` (class in `species`), 55  
`genomic distance`, **3**, **9**, **39**, **42**, **54**, **55**, **62**  
`get()` (`activations.ActivationFunctionSet` method), 31  
`get_config_params()` (`attributes.BoolAttribute` method), 33  
`get_config_params()` (`attributes.FloatAttribute` method), 32  
`get_config_params()` (`attributes.StringAttribute` method), 34  
`get_config_params()` (`genes.BaseGene` class method), 38  
`get_fitness_mean()` (`statistics.StatisticsReporter` method), 58  
`get_fitness_stat()` (`statistics.StatisticsReporter` method), 57  
`get_fitness_stdev()` (`statistics.StatisticsReporter` method), 58  
`get_fitnesses()` (`species.Species` method), 55  
`get_new_node_key()` (`genome.DefaultGenomeConfig` method), 40  
`get_next()` (`indexer.Indexer` method), 45  
`get_species()` (`species.DefaultSpeciesSet` method), 56  
`get_species_fitness()` (`statistics.StatisticsReporter` method), 58  
`get_species_id()` (`species.DefaultSpeciesSet` method), 56  
`get_species_sizes()` (`statistics.StatisticsReporter` method), 58  
`get_time_step_msec()` (`iznn.IZNN` method), 47  
`graphs` (module), 44

## H

`hidden node`, 10, 41  
`homologous`, **3**, **9**

## I

`id`, *see* `key`  
`Indexer` (class in `indexer`), 45  
`indexer` (module), 45  
`info()`, **53**, **55**  
`info()` (`reporting.BaseReporter` method), 52  
`info()` (`reporting.ReporterSet` method), 51  
`init_attributes()` (`genes.BaseGene` method), 39  
`init_value()` (`attributes.BoolAttribute` method), 33  
`init_value()` (`attributes.FloatAttribute` method), 33

init\_value() (attributes.StringAttribute method), 34  
 initial\_connection, 10, 10, 40, 41, 43  
 input node, 10, 41  
 interpret() (config.ConfigParameter method), 35  
 InvalidActivationFunction, 31  
 is\_valid() (activations.ActivationFunctionSet method), 31  
 iteritems() (in module six\_util), 54  
 iterkeys() (in module six\_util), 54  
 itervalues() (in module six\_util), 54  
 IZGenome (class in iznn), 46  
 IZNeuron (class in iznn), 46  
 IZNN (class in iznn), 46  
 iznn (module), 45  
 IZNodeGene (class in iznn), 46

## K

key, 3, 38, 40, 41, 45, 51, 53, 54

## M

math\_util (module), 47  
 max\_stagnation, 8, 56  
 mean() (in module math\_util), 47  
 median() (in module math\_util), 47  
 min\_species\_size, 9, 53  
 mutate, *see* mutation  
 mutate() (genes.BaseGene method), 39  
 mutate() (genome.DefaultGenome method), 41  
 mutate\_add\_connection() (genome.DefaultGenome method), 42  
 mutate\_add\_node() (genome.DefaultGenome method), 42  
 mutate\_delete\_connection() (genome.DefaultGenome method), 42  
 mutate\_delete\_node() (genome.DefaultGenome method), 42  
 mutate\_value() (attributes.BoolAttribute method), 33  
 mutate\_value() (attributes.FloatAttribute method), 33  
 mutate\_value() (attributes.StringAttribute method), 34  
 mutation, 9–11, 33, 34, 39, 41

## N

nn.feed\_forward (module), 47  
 nn.recurrent (module), 48  
 node, 9–11, 39, 42, 43, 46

## O

output node, 11, 41

## P

parallel (module), 48  
 ParallelEvaluator (class in parallel), 48  
 parse() (config.ConfigParameter method), 35  
 parse\_config() (genes.BaseGene class method), 38

parse\_config() (genome.DefaultGenome class method), 41  
 parse\_config() (reproduction.DefaultReproduction class method), 53  
 parse\_config() (species.DefaultSpeciesSet class method), 55  
 parse\_config() (stagnation.DefaultStagnation class method), 56  
 pin, 41  
 pop\_size, 8, 35, 53, 53  
 Population (class in population), 49  
 population (module), 49  
 post\_evaluate(), 49  
 post\_evaluate() (reporting.BaseReporter method), 51  
 post\_evaluate() (reporting.ReporterSet method), 50  
 post\_evaluate() (statistics.StatisticsReporter method), 57  
 post\_reproduction() (reporting.BaseReporter method), 52  
 post\_reproduction() (reporting.ReporterSet method), 51  
 product() (in module genome), 40

## R

recurrent, 29, 37, 44  
 RecurrentNetwork (class in nn.recurrent), 48  
 remove() (reporting.ReporterSet method), 50  
 ReporterSet (class in reporting), 50  
 reporting, 19  
 reporting (module), 50  
 reproduce() (reproduction.DefaultReproduction method), 53  
 reproduction, 8, 20  
 reproduction (module), 52  
 required\_for\_output() (in module graphs), 44  
 reset() (ctrnn.CTRNN method), 37  
 reset() (iznn.IZNeuron method), 46  
 reset() (iznn.IZNN method), 47  
 reset() (nn.recurrent.RecurrentNetwork method), 48  
 reset\_on\_extinction, 8, 35, 49  
 response, 11, 65  
 restore\_checkpoint() (checkpoint.Checkpointer static method), 35  
 run() (population.Population method), 49

## S

save() (config.Config method), 36  
 save() (genome.DefaultGenomeConfig method), 40  
 save() (statistics.StatisticsReporter method), 59  
 save\_checkpoint() (checkpoint.Checkpointer static method), 34  
 save\_genome\_fitness() (statistics.StatisticsReporter method), 58  
 save\_species\_count() (statistics.StatisticsReporter method), 59  
 save\_species\_fitness() (statistics.StatisticsReporter method), 59

- set\_inputs() (iznn.IZNN method), 46
- six\_util (module), 54
- size() (genome.DefaultGenome method), 43
- softmax() (in module math\_util), 47
- speciate() (species.DefaultSpeciesSet method), 55
- species, 9, 20
- Species (class in species), 54
- species (module), 54
- species\_elitism, 8, 56
- species\_fitness\_func, 8, 47, 56
- species\_stagnant(), 53
- species\_stagnant() (reporting.BaseReporter method), 52
- species\_stagnant() (reporting.ReporterSet method), 51
- stagnation, 8, 20, 47, 53
- stagnation (module), 56
- start\_generation(), 49
- start\_generation() (reporting.BaseReporter method), 51
- start\_generation() (reporting.ReporterSet method), 50
- stat\_functions (in module math\_util), 47
- statistics (module), 57
- StatisticsReporter (class in statistics), 57
- stdev() (in module math\_util), 47
- StdOutReporter (class in reporting), 52
- StringAttribute (class in attributes), 34
- survival\_threshold, 8, 53, 53

## U

- update() (species.Species method), 54
- update() (stagnation.DefaultStagnation method), 57

## V

- validate\_activation() (in module activations), 31
- variance() (in module math\_util), 47

## W

- weight, 11
- write\_config() (genome.DefaultGenome class method), 41
- write\_config() (reproduction.DefaultReproduction class method), 53
- write\_config() (species.DefaultSpeciesSet class method), 55
- write\_config() (stagnation.DefaultStagnation class method), 56
- write\_pretty\_params() (in module config), 35