NEAT-Python Documentation

Release 0.92

CodeReclaimers, LLC

Aug 01, 2017

Contents

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 example's README.md file for additional details and installation/setup instructions for the main code for each. In addition to dependencies varying with different examples, visualization of the results (via `visualize.py` modules) frequently requires graphviz and/or matplotlib. TODO: Improve README.md file information for the examples.

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:

CHAPTER 1

NEAT Overview

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.)

CHAPTER 2

Installation

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 0.92) 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 0.92 and use the example code contained in it.

You may also just get the 0.92 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 0.92.

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

To install from source, simply run:

```
python setup.py install
```

from the directory containing setup.py.

CHAPTER 3

Configuration file 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 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 (if attention is being paid to fitness for termination in the first place - see `no_fitness_termination` below).

- ***no_fitness_termination*** If this evaluates to `True`, then the `fitness_criterion` and `fitness_threshold` are ignored for termination; only valid if termination by a maximum number of generations passed to *`population.Population.run()`* is enabled, and the `found_solution`

method **is** called upon generation number termination. If it evaluates to `False`, then fitness is used to determine termination. **This defaults to "False".**

New in version 0.92.

- ***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.**

[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.**
- ***min_species_size*** The minimum number of genomes per species after reproduction. **This defaults to 2.**

[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 *aggregations*) are: `sum`, *product*, `min`, `max`, `mean`, `median`, and *maxabs* (which returns the input value with the greatest absolute value; the returned value may be positive or negative). New aggregation functions can be defined similarly to *new activation functions*. (Note that the function needs to take a `list` or other `iterable`; the `reduce` function, as in *aggregations*, may be of use in this.)

  Changed in version 0.92: Moved out of *genome* into *aggregations*; maxabs, mean, and median added; method for defining new aggregation functions added.
- ***bias_init_mean*** The mean of the normal/gaussian distribution, if it is used to *select* *bias attribute* values for new *nodes*.
- ***bias_init_stdev*** The standard deviation of the normal/gaussian distribution, if it is used to select bias values for new nodes.
- ***bias_init_type*** If set to `gaussian` or `normal`, then the initialization is to a normal/gaussian distribution. If set to `uniform`, a uniform distribution from $\max(bias\_min\_value, (bias\_init\_mean - (bias\_init\_stdev * 2)))$ to $\min(bias\_max\_value, (bias\_init\_mean + (bias\_init\_stdev * 2)))$. (Note that the standard deviation of a uniform distribution is not range/0.25, as implied by this, but the range divided by a bit over 0.288 (the square root of 12); however, this approximation makes setting the range much easier.) **This defaults to "gaussian".**

  New in version 0.92.
- ***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].
- ***enabled_rate_to_false_add*** Adds to the `enabled_mutate_rate` if the connection is currently *enabled*.
- ***enabled_rate_to_true_add*** Adds to the `enabled_mutate_rate` if the connection is currently not enabled.

  New in version 0.92: `enabled_rate_to_false_add` and `enabled_rate_to_true_add`
- ***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" stands for "Feature Selection".)
  - `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 0.92: 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, if it is used to `select` *response* multiplier *attribute* values for new *nodes*.
- ***response_init_stdev*** The standard deviation of the normal/gaussian distribution, if it is used to select response multipliers for new nodes.
- ***response_init_type*** If set to `gaussian` or `normal`, then the initialization is to a normal/gaussian distribution. If set to `uniform`, a uniform distribution from $\max(response\_min\_value, (response\_init\_mean - (response\_init\_stdev * 2)))$ to $\min(response\_max\_value, (response\_init\_mean + (response\_init\_stdev * 2)))$. (Note that the standard deviation of a uniform distribution is not range/0.25, as implied by this, but the range divided by a bit over 0.288 (the square root of 12); however, this approximation makes setting the range much easier.) **This defaults to "gaussian".**

  New in version 0.92.
- ***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).
- ***single_structural_mutation*** If this evaluates to `True`, only one structural mutation (the addition or removal of a *node* or *connection*) will be allowed per genome per generation. (If the probabilities for *conn_add_prob*, conn_delete_prob, *node_add_prob*, and node_delete_prob add up to over 1, the chances of each are proportional to the appropriate configuration value.) **This defaults to "False".**

  New in version 0.92.
- ***structural_mutation_surer*** If this evaluates to `True`, then an attempt to add a *node* to a genome lacking *connections* will result in adding a connection instead; furthermore, if an attempt to add a connection tries to add a connection that already exists, that connection will be *enabled*. If this is set to `default`, then it acts as if it had the same value as `single_structural_mutation` (above). **This defaults to "default".**

  New in version 0.92.
- ***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_init_type*** If set to `gaussian` or `normal`, then the initialization is to a normal/gaussian distribution. If set to `uniform`, a uniform distribution from $\max(weight\_min\_value, (weight\_init\_mean - (weight\_init\_stdev * 2)))$ to $\min(weight\_max\_value, (weight\_init\_mean + (weight\_init\_stdev * 2)))$. (Note that the standard deviation of a uniform distribution is not range/0.25, as implied by this, but the range divided by a bit over 0.288 (the square root of 12); however, this approximation makes setting the range much easier.) **This defaults to "gaussian".**

New in version 0.92.

- ***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

CHAPTER 4

Overview of the basic XOR example (xor2.py)

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

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

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 0.92 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
```

CHAPTER 5

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 *Checkpointer* 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

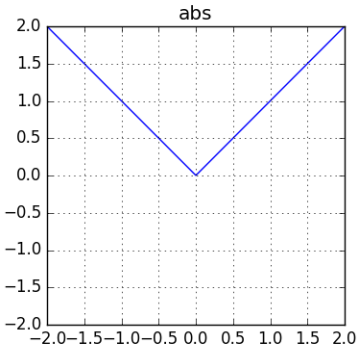
CHAPTER 6

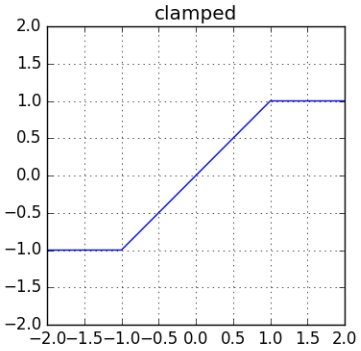
Overview of builtin activation functions

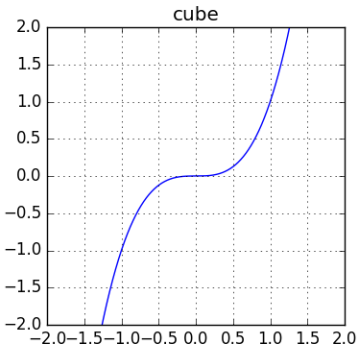
Note that some of these *functions* are scaled differently from the canonical versions you may be familiar with. The intention of the scaling is to place more of the functions' "interesting" behavior in the region $[-1, 1] \times [-1, 1]$.

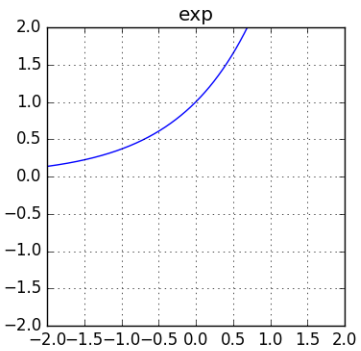
The implementation of these functions can be found in the `activations` module.

abs




clamped
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0−1.5−1.0−0.5 0.0 0.5 1.0 1.5 2.0


cube
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0−1.5−1.0−0.5 0.0 0.5 1.0 1.5 2.0

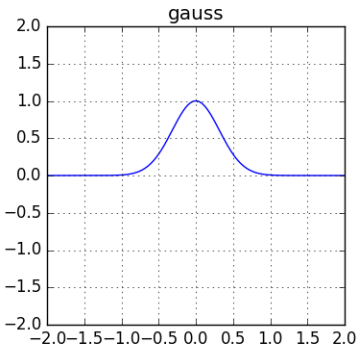

exp
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0−1.5−1.0−0.5 0.0 0.5 1.0 1.5 2.0

clamped

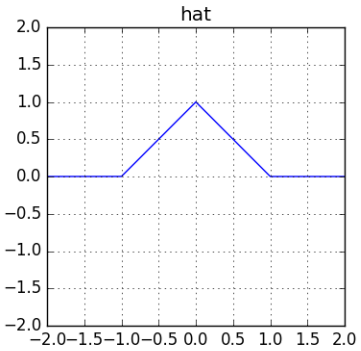
cube

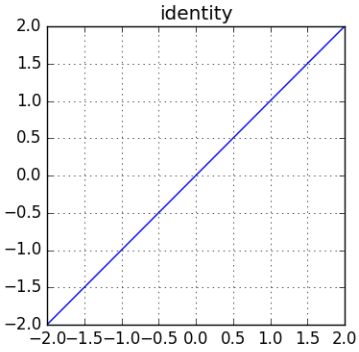
exp

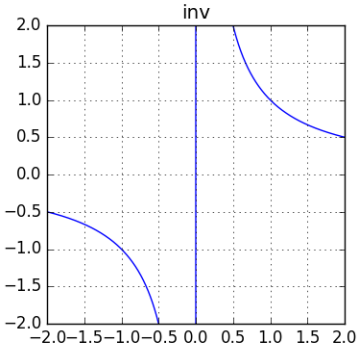
gauss

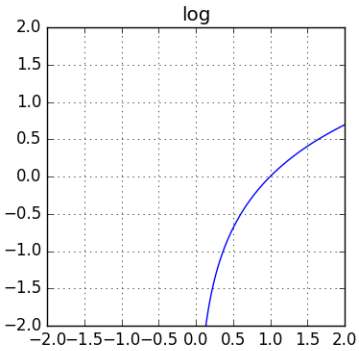


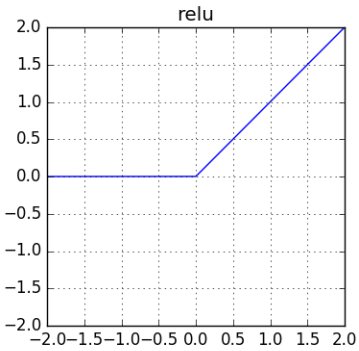
hat




identity
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0


inv
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0


log
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0


relu
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0

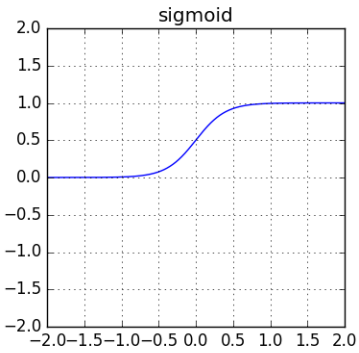
identity

inv

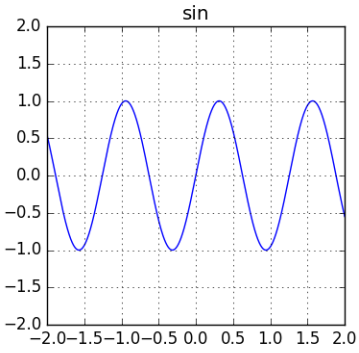
log

relu

sigmoid



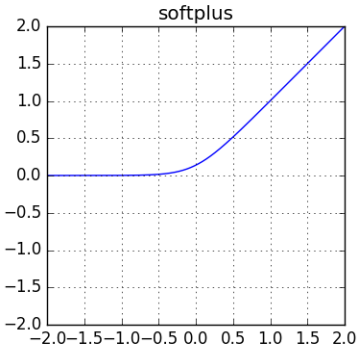
sin

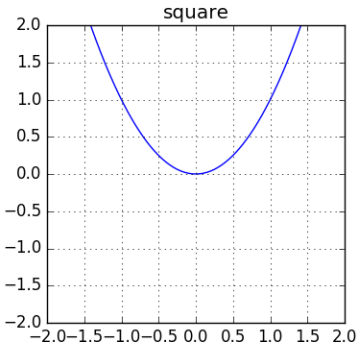


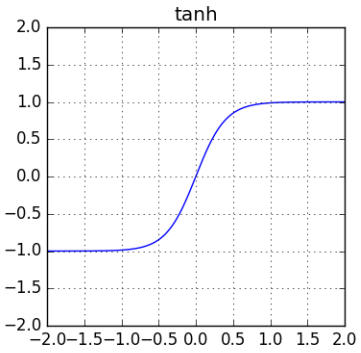
softplus

square

tanh


softplus
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0


square
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0


tanh
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0

CHAPTER 7

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:

- τ_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.
- β_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:

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

CHAPTER 8

Module summaries

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, or if an unknown activation function is requested by name via *get*.

Changed in version 0.92: 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.

Returns The function of interest

Return type 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

aggregations

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

Note: *Non-enabled connections* will, by all methods currently included in NEAT-Python, *not* be included among the numbers input to these functions, even as 0s.

---

aggregations.**product_aggregation**(*x*)
An adaptation of the multiplication function to take an iterable.

Parameters **x** (list(float) or tuple(float) or set(float)) – The numbers to be multiplied together; takes any `iterable`.

Returns $\prod(x)$

Return type float

aggregations.**sum_aggregation**(*x*)
Probably the most commonly-used aggregation function.

Parameters **x** (list(float) or tuple(float) or set(float)) – The numbers to find the sum of; takes any iterable.

Returns $\sum(x)$

Return type float

aggregations.**max_aggregation**(*x*)
Returns the maximum of the inputs.

Parameters **x** (list(float) or tuple(float) or set(float)) – The numbers to find the greatest of; takes any iterable.

Returns $\max(x)$

Return type float

aggregations.**min_aggregation**(*x*)
Returns the minimum of the inputs.

Parameters **x** (list(float) or tuple(float) or set(float)) – The numbers to find the least of; takes any iterable.

Returns $\min(x)$

Return type float

aggregations.**maxabs_aggregation**(*x*)
Returns the maximum by absolute value, which may be positive or negative. Envisioned as suitable for neural network pooling operations.

Parameters **x** (list(float) or tuple(float) or set(float)) – The numbers to find the absolute-value maximum of; takes any iterable.

Returns $x_i, i = \text{argmax}|\mathbf{x}|$

Return type float

New in version 0.92.

aggregations.**median_aggregation**(*x*)

Returns the *median* of the inputs.

Parameters **x** (list(float) or tuple(float) or set(float)) – The numbers to find the median of; takes any iterable.

Returns The median; if there are an even number of inputs, takes the mean of the middle two.

Return type float

New in version 0.92.

aggregations.**mean_aggregation**(*x*)

Returns the arithmetic mean. Potentially maintains a more stable result than `sum` for changing numbers of *enabled connections*, which may be good or bad depending on the circumstances; having both available to the algorithm is advised.

Parameters **x** (list(float) or tuple(float) or set(float)) – The numbers to find the mean of; takes any iterable.

Returns The arithmetic mean.

Return type float

New in version 0.92.

exception aggregations.**InvalidAggregationFunction**(*TypeError*)

Exception called if an aggregation function being added is invalid according to the *validate_aggregation* function, or if an unknown aggregation function is requested by name via *get*.

New in version 0.92.

aggregations.**validate_aggregation**(*function*)

Checks to make sure its parameter is a function that takes at least one argument.

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

Raises ***InvalidAggregationFunction*** – If the object does not pass the tests.

New in version 0.92.

class aggregations.**AggregationFunctionSet**

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

add(*name*, *function*)

After validating the function (via *validate_aggregation*), adds it to the available activation functions under the given name. Used by *DefaultGenomeConfig.add_activation*. TODO: Check for whether the function needs `reduce`, or at least offer a form of this function (or extra argument for it, defaulting to false) and/or its interface in *genome*, that will appropriately "wrap" the input function.

Parameters

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

New in version 0.92.

get(*name*)

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

Parameters **name** (*str*) – The name of the function.
Returns The function of interest
Return type function
Raises ***InvalidAggregationFunction*** – If the function is not known.

New in version 0.92.

__getitem__(*index*)

Present for compatibility with older programs that expect the aggregation functions to be in a dict. A wrapper for *get(index)*.

Parameters **index** (*str*) – The name of the function.
Returns The function of interest.
Return type function
Raises

- ***InvalidAggregationFunction*** – If the function is not known.
- **DeprecationWarning** – Always.

Changed in version 0.92: Originally a dictionary in *genome*.

Deprecated since version 0.92: Use *get(index)* instead.

is_valid(*name*)

Checks whether the named function is a known aggregation function.

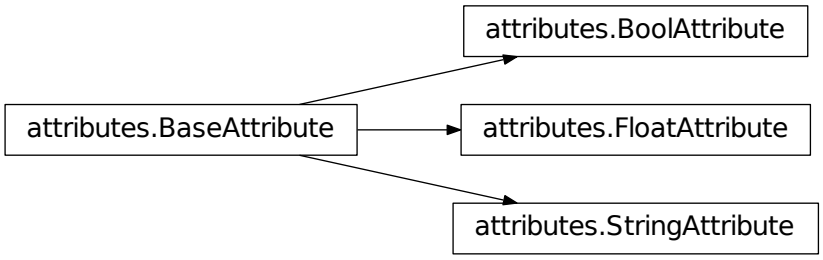
Parameters **name** (*str*) – The name of the function.
Returns Whether or not the function is known.
Return type bool

New in version 0.92.

Changed in version 0.92: Moved from *genome* and expanded to match *activations* (plus the maxabs, median, and mean functions added).

attributes

Deals with *attributes* used by *genes*.



class attributes.**BaseAttribute**(*name*, ***default_dict*)

Superclass for the type-specialized attribute subclasses, used by genes (such as via the *genes.BaseGene* implementation). Updates _config_items with any defaults supplied, then uses *config_item_name* to set up a listing of the names of configuration items using setattr.

Parameters

- **name** (*str*) – The name of the attribute, held in the instance's name attribute.
- **default_dict** (*dict(str, str)*) – An optional dictionary of defaults for the configuration items.

Changed in version 0.92: Default_dict capability added.

config_item_name(*config_item_base_name*)

Formats a configuration item's name by combining the attribute's name with the base item name.

Parameters **config_item_base_name** (*str*) – The base name of the configuration item, to be combined with the attribute's name.
Returns The configuration item's full name.
Return type str

Changed in version 0.92: Originally (as config_item_names) did not take any input and returned a list based on the _config_items subclass attribute.

get_config_params()

Uses *config_item_name* for each configuration item to get the name, then gets the appropriate type of *config.ConfigParameter* instance for each (with any appropriate defaults being set from _config_items, including as modified by *BaseAttribute*) and returns it.

Returns A list of ConfigParameter instances.
Return type list(instance)

Changed in version 0.92: Was originally specific for the attribute subclass, since it did not pick up the appropriate type from the _config_items list; default capability also added.

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

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

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** (instance) – 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, using either a gaussian distribution with the configured mean and standard deviation, followed by *clamp* to keep the result within the desired range, or a uniform distribution, depending on the configuration setting of init_type.

Parameters **config** (instance) – The configuration object from which the mean, standard deviation, and initialization distribution type values are to be retrieved.
Returns The new value.
Return type float

Changed in version 0.92: Uniform distribution initialization option added.

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).
Parameters
- **value** (float) – The current value of the attribute.
- **config** (instance) – 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.

init_value(*config*)
Initializes the attribute's value, either using a configured default, or (if the default is "random") with a 50/50 chance of True or False.

Deprecated since version 0.92: While it is possible to use "None" as an equivalent to "random", this is too easily confusable with an actual None.

Changed in version 0.92: Ability to use "random" for a 50/50 chance of True or False added.
Parameters **config** (instance) – The configuration object from which the default parameter is to be retrieved.
Returns The new value.
Return type bool
Raises **RuntimeError** – If the default value is not recognized as standing for any of True, False, "random", or "none".

mutate_value(*value*, *config*)
With a frequency determined by the mutate_rate and rate_to_false_add or rate_to_true_add configuration parameters, replaces the value with a 50/50 chance of True or False; note that this has a 50% chance of leaving the value unchanged.
Parameters
- **value** (*bool*) – The current value of the attribute.
- **config** (instance) – The configuration object from which the mutate_rate and other parameters are to be extracted.

Returns Either the original value, if unchanged, or the new value.
Return type bool
Changed in version 0.92: Added the rate_to_false_add and rate_to_true_add parameters.

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.

init_value(*config*)
Initializes the attribute's value, either using a configured default or (if the default is "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.

Deprecated since version 0.92: While it is possible to use "None" as an equivalent to "random", this is too easily confusable with an actual None.
Parameters **config** (instance) – 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 configuration parameter, replaces the value with 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: Add configurable probabilities of which option is used. Longer-term, 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.

Parameters
- **value** (*str*) – The current value of the attribute.
- **config** (instance) – The configuration object from which the options and other parameters are to be extracted.

Returns The new value.

Return type str

Changed in version 0.92: __config_items__ changed to _config_items, since it is not a Python internal variable.

checkpoint

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

Note: The speed of this module can vary widely between python implementations (and perhaps versions).

class checkpoint.**Checkpointer**(*generation_interval=100*, *time_interval_seconds=300*, *filename_prefix='neat-checkpoint-'*)
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.) The start of the filename will be equal to filename_prefix, followed by the generation number. If there is a need to check the last generation for which a checkpoint was saved, such as to determine which file to load, access last_generation_checkpoint; if -1, none have been saved.

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.
- **filename_prefix** (*str*) – The prefix for the checkpoint file names.

save_checkpoint(*config*, *population*, *species*, *generation*)
Saves the current simulation (including randomization) state to (if using the default neat-checkpoint- for filename_prefix) neat-checkpoint-*generation*, with generation being the generation number.

Parameters
- **config** (instance) – The *config.Config* configuration instance to be used.
- **population** (dict(int, object)) – A population as created by *reproduction.DefaultReproduction.create_new()* or a compatible implementation.

- **species** (instance) – A *species.DefaultSpeciesSet* (or compatible implementation) instance.
- **generation** (int) – 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 *Population* instance that can be used with *Population.run* to restart the simulation.

Return type instance

config

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

class config.**ConfigParameter**(*name*, *value_type*, *default=None*)

Does initial handling of a particular configuration parameter.

Parameters

- **name** (*str*) – The name of the configuration parameter.
- **value_type** – The type that the configuration parameter should be; must be one of str, int, bool, float, or list.
- **default** (*str or None*) – If given, the default to use for the configuration parameter.

Changed in version 0.92: Default capability added.

__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** (instance) – The configuration parser to be used.

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

Return type str or list(str)

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(str, str)*) – Configuration parameters as output by the configuration parser.

Returns The configuration parameter value

Return type str or int or bool or float or list(str)

Raises

- **RuntimeError** – If there is a problem with the configuration parameter.

- **DeprecationWarning** – If a default is used.

Changed in version 0.92: Default capability added.

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.

Returns String version.

Return type str

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

exception config.**UnknownConfigItemError**(*NameError*)

Error for unknown configuration option(s) - partially to catch typos. TODO: *genome.DefaultGenomeConfig* does not currently check for these.

New in version 0.92.

class config.**DefaultClassConfig**(*param_dict*, *param_list*)

Replaces at least some boilerplate configuration code for reproduction, species_set, and stagnation classes.

Parameters

- **param_dict** (*dict(str, str)*) – Dictionary of configuration parameters from config file.
- **param_list** (list(instance)) – List of *ConfigParameter* instances; used to know what parameters are of interest to the calling class.

Raises ***UnknownConfigItemError*** – If a key in `param_dict` is not among the names in `param_list`.

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

Required method (inherited by calling classes). Uses *write_pretty_params()* to output parameters of interest to the calling class.

Parameters

- **f** (file) – File object to be written to.
- **config** (instance) – DefaultClassConfig instance.

New in version 0.92.

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*, which are found as some of its attributes. For a description of the configuration file, see *Configuration file description*. The __name__ attributes of the `_type` parameters are used for the titles of the configuration file sections. A Config instance's `genome_config`, `species_set_config`, `stagnation_config`, and `reproduction_config` attributes hold the configuration objects for the respective classes.

Parameters

- **genome_type** (class) – Specifies the genome class used, such as *genome.DefaultGenome* or *iznn.IZGenome*. See *Genome Interface* for the needed interface.
- **reproduction_type** (class) – Specifies the reproduction class used, such as *reproduction.DefaultReproduction*. See *Reproduction Interface* for the needed interface.
- **species_set_type** (class) – Specifies the species set class used, such as *species.DefaultSpeciesSet*.
- **stagnation_type** (class) – 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 `_type` classes lack a `parse_config` method.
- ***UnknownConfigItemError*** – If an option in the `NEAT` section of the configuration file is not recognized.
- **DeprecationWarning** – If a default is used for one of the `NEAT` section options.

Changed in version 0.92: Added default capabilities, UnknownConfigItemError, no_fitness_termination.

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. (A comparison of it and the input configuration file can be used to determine any default parameters of interest.)

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

ctrnn

class ctrnn.**CTRNNNodeEval**(*time_constant*, *activation*, *aggregation*, *bias*, *response*, *links*)

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

Parameters

- **time_constant** (*float*) – Controls how fast the node responds; τ_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 *ctrnn* 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(float)*) – The values for the *input nodes*.
- **advance_time** (float) – How much time to advance the network before returning the resulting outputs.
- **time_step** (float or None) – 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(float)

Raises

- **NotImplementedError** – If a time_step is not given.
- **RuntimeError** – If the number of inputs does not match the number of *input nodes*

Changed in version 0.92: Exception changed to more-specific RuntimeError.

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

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

Parameters

- **genome** (instance) – A *genome.DefaultGenome* instance.
- **config** (instance) – A *config.Config* instance.
- **time_constant** (float) – Used for the *CTRNNNodeEval* initializations.

distributed

Distributed evaluation of genomes.

Note: This module is in a **beta** state, and still *unstable* even in single-machine testing. Reliability is likely to vary, including depending on the Python version and implementation (e.g., cpython vs pypy) in use and the likelihoods of timeouts (due to machine and/or network slowness). In particular, while the code can try to reconnect between between *primary* and *secondary* nodes, as noted in the multiprocessing documentation this may not work due to data loss/corruption. Note also that this module is not responsible for starting the script copies on the different *compute nodes*, since this is very site/configuration-dependent.

About compute nodes:

The *primary compute node* (the node which creates and mutates genomes) and the *secondary compute nodes* (the nodes which evaluate genomes) can execute the same script. The role of a compute node is determined using the `mode` argument of the DistributedEvaluator. If the mode is *`MODE_AUTO`*, the *`host_is_local()`* function is used to check if the `addr` argument points to the localhost. If it does, the compute node starts as a *primary node*, and otherwise as a *secondary node*. If `mode` is *`MODE_PRIMARY`*, the compute node always starts as a primary node. If `mode` is *`MODE_SECONDARY`*, the compute node will always start as a secondary node.

There can only be one primary node per NEAT, but any number of secondary nodes. The primary node will not evaluate any genomes, which means you will always need at least two compute nodes (one primary and at least one secondary).

You can run any number of compute nodes on the same physical machine (or VM). However, if a machine has both a primary node and one or more secondary nodes, *`MODE_AUTO`* cannot be used for those secondary nodes - *`MODE_SECONDARY`* will need to be specified.

Usage:

1. Import modules and define the evaluation logic (the `eval_genome` function). (After this, check for `if __name__ == '__main__'`, and put the rest of the code inside the body of the statement, or in subroutines called from it.)
2. Load config and create a *`population`* - here, the variable `p`.
3. If required, create and add *`reporters`*.
4. Create a *`DistributedEvaluator(addr_of_primary_node, b'some_password', eval_function, mode=MODE_AUTO)`* - here, the variable `de`.
5. Call *`de.start(exit_on_stop=True)`*. The `start()` call will block on the secondary nodes and call sys.exit(0) when the NEAT evolution finishes. This means that the following code will only be executed on the primary node.
6. Start the evaluation using *`p.run(de.evaluate, number_of_generations)`*.
7. Stop the secondary nodes using *`de.stop()`*.
8. You are done. You may want to save the winning genome(s) or show some *`statistics`*.

See `examples/xor/evolve-feedforward-distributed.py` for a complete example.

Note: The below contains some (but not complete) information about private functions, classes, and similar (starting with _); this documentation is meant to help with maintaining and improving the code, not for enabling external use, and the interface may change **rapidly** with no warning.

`distributed.`**`MODE_AUTO`**

`distributed.`**`MODE_PRIMARY`**

`distributed.`**`MODE_SECONDARY`**
Values - which should be treated as constants - that are used for the `mode` argument of *`DistributedEvaluator`*. If MODE_AUTO, *`_determine_mode()`* uses *`host_is_local()`* and the specified `addr` of the *primary node* to decide the mode; the other two specify it.

`distributed._`**`STATE_RUNNING`**

distributed.**_STATE_SHUTDOWN**

distributed.**_STATE_FORCED_SHUTDOWN**
Values - which should be treated as constants - that are used to determine the current state (whether the secondaries should be continuing the run or not).

exception distributed.**ModeError**(*RuntimeError*)
An exception raised when a mode-specific method is being called without being in the mode - either a primary-specific method called by a *secondary node* or a secondary-specific method called by a *primary node*.

distributed.**host_is_local**(*hostname*, *port=22*)
Returns True if the hostname points to the localhost (including shares addresses), otherwise False.

Parameters

- **hostname** (*str*) – The hostname to be checked; will be put through socket.getfqdn.
- **port** (int) – The optional port for socket functions requiring one. Defaults to 22, the ssh port.

Returns Whether the hostname appears to be equivalent to that of the localhost.

Return type bool

distributed.**_determine_mode**(*addr*, *mode*)
Returns the mode that should be used. If mode is *MODE_AUTO*, this is determined by checking (via *host_is_local()*) if addr points to the localhost; if it does, it returns *MODE_PRIMARY*, else it returns *MODE_SECONDARY*. If mode is either MODE_PRIMARY or MODE_SECONDARY, it returns the mode argument. Otherwise, a ValueError is raised.

Parameters

- **addr** (*tuple(str, int) or bytes*) – Either a tuple of (hostname, port) pointing to the machine that has the *primary node*, or the hostname (as bytes if on 3.X).
- **mode** (*int*) – Specifies the mode to run in - must be one of *MODE_AUTO*, *MODE_PRIMARY*, or *MODE_SECONDARY*.

Raises **ValueError** – If the mode is not one of the above.

distributed.**chunked**(*data*, *chunksize*)

Splits up data and returns it as a list of chunks containing at most chunksize elements of data.

Parameters

- **data** (*list(object) or tuple(object) or set(object)*) – The data to split up; takes any iterable.
- **chunksize** (int) – The maximum number of elements per chunk.

Returns A list of chunks containing (as a list) at most chunksize elements of data.

Return type list(list(object))

Raises **ValueError** – If chunksize is not 1+ or is not an integer

class distributed.**_ExtendedManager**(*addr*, *authkey*, *mode*, *start=False*)
Manages the multiprocessing.managers.SyncManager instance. Initializes self._secondary_state to *_STATE_RUNNING*.

Parameters

- **addr** (*tuple(str, int)*) – Should be a tuple of (hostname, port) pointing to the machine running the DistributedEvaluator in primary mode. If mode is *MODE_AUTO*, the mode is determined by checking whether the hostname points to this host or not (via *_determine_mode()* and *host_is_local()*).
- **authkey** (bytes) – The password used to restrict access to the manager. All DistributedEvaluators need to use the same authkey. Note that this needs to be a bytes object for Python 3.X, and should be in 2.7 for compatibility (identical in 2.7 to a `str` object). For more information, see under *DistributedEvaluator*.
- **mode** (*int*) – Specifies the mode to run in - must be one of *MODE_AUTO*, *MODE_PRIMARY*, or *MODE_SECONDARY*. Processed by *_determine_mode()*.
- **start** (*bool*) – Whether to call the *start()* method after initialization.

__reduce__()
Used by `pickle` to serialize instances of this class. TODO: Appears to assume that `start` (for initialization) should be true; perhaps `self.manager` should be checked? (This may require :py:meth::*stop()* to set `self.manager` to `None`, incidentally.)

Returns Information about the class instance; a tuple of (class name, tuple(addr, authkey, mode, True)).

Return type tuple(str, tuple(tuple(str, int), bytes, int, bool))

start()
Starts (if in *MODE_PRIMARY*) or connects to (if in *MODE_SECONDARY*) the manager.

stop()
Stops the manager using shutdown . TODO: Should this set `self.manager` to None?

set_secondary_state(*value*)
Sets the value for the `secondary_state`, shared between the nodes via multiprocessing.managers.Value.

Parameters **value** (*int*) – The desired secondary state; must be one of *_STATE_RUNNING*, *_STATE_SHUTDOWN*, or *_STATE_FORCED_SHUTDOWN*.

Raises
- **ValueError** – If the `value` is not one of the above.
- **RuntimeError** – If the manager has not been *started*.

secondary_state
The property `secondary_state` - whether the secondary nodes should still be processing elements.

get_inqueue()
Returns the inqueue.

Returns The incoming queue.

Return type instance

Raises **RuntimeError** – If the manager has not been *started*.

get_outqueue()
Returns the outqueue.

Returns The outgoing queue.

Return type instance

Raises **RuntimeError** – If the manager has not been *started*.

get_namespace()
Returns the manager's namespace instance.

Returns The namespace.

Return type instance
Raises **RuntimeError** – If the manager has not been *started*.

class distributed.**DistributedEvaluator**(*addr*, *authkey*, *eval_function*, *secondary_chunksize=1*, *num_workers=None*, *worker_timeout=60*, *mode=MODE_AUTO*)

An evaluator working across multiple machines (*compute nodes*).

Warning: See Authentication Keys for more on the `authkey` parameter, used to restrict access to the manager.

Parameters

- **addr** (*tuple(str, int)*) – Should be a tuple of (hostname, port) pointing to the machine running the DistributedEvaluator in primary mode. If mode is *MODE_AUTO*, the mode is determined by checking whether the hostname points to this host or not (via *host_is_local()*).
- **authkey** (bytes) – The password used to restrict access to the manager. All DistributedEvaluators need to use the same authkey. Note that this needs to be a bytes object for Python 3.X, and should be in 2.7 for compatibility (identical in 2.7 to a `str` object).
- **eval_function** (`function`) – The eval_function should take two arguments - a genome object and a 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*, nor by *ParallelEvaluator* (although it is more similar to the latter).
- **secondary_chunksize** (int) – The number of *genomes* that will be sent to a *secondary node* at any one time.
- **num_workers** (int or None) – The number of worker processes per *secondary node*, used for evaluating genomes. If None, will use multiprocessing.cpu_count() to determine the number of processes (see further below regarding this default). If 1 (for a secondary node), including if there is no usable result from `multiprocessing.cpu_count()`, then the process creating the DistributedEvaluator instance will also do the evaluations.
- **worker_timeout** (float or None) – specifies the timeout (in seconds) for a secondary node getting the results from a worker subprocess; if None, there is no timeout.
- **mode** (*int*) – Specifies the mode to run in - must be one of *MODE_AUTO* (the default), *MODE_PRIMARY*, or *MODE_SECONDARY*.

Raises **ValueError** – If the mode is not one of the above.

Note: Whether the default for `num_workers` is appropriate can vary depending on the evaluation function (e.g., whether cpu-bound, memory-bound, i/o-bound...), python implementation, and other factors; if unsure and maximal per-machine performance is critical, experimentation will be required.

---

is_primary()

Returns True if the caller is the *primary node*; otherwise False.

Returns `True` if primary, `False` if *secondary*

Return type bool

is_master()
A backward-compatibility wrapper for *is_primary()*.
Returns True if primary, False if *secondary*
Return type bool
Raises **DeprecationWarning** – Always.
Deprecated since version 0.92.

start(*exit_on_stop=True*, *secondary_wait=0*, *reconnect=False*)
If the DistributedEvaluator is in primary mode, starts the manager process and returns. If the DistributedEvaluator is in secondary mode, it connects to the manager and waits for tasks.
Parameters
- **exit_on_stop** (bool) – If a secondary node, whether to exit if (unless reconnect is True) the connection is lost, the primary calls for a shutdown (via *stop()*), or - even if reconnect is True - the primary calls for a forced shutdown (via calling *stop()* with force_secondary_shutdown set to True).
- **secondary_wait** (float) – Specifies the time (in seconds) to sleep before actually starting, if a *secondary node*.
- **reconnect** (*bool*) – If a secondary node, whether it should try to reconnect if the connection is lost.

Raises
- **RuntimeError** – If already started.
- **ValueError** – If the mode is invalid.

stop(*wait=1*, *shutdown=True*, *force_secondary_shutdown=False*)
Stops all secondaries.
Parameters
- **wait** (float) – Time (in seconds) to wait after telling the secondaries to stop.
- **shutdown** (bool) – Whether to shutdown the multiprocessing.managers.SyncManager also (after the wait, if any).
- **force_secondary_shutdown** (*bool*) – Causes secondaries to shutdown even if started with reconnect true (via setting the secondary_state to *_STATE_FORCED_SHUTDOWN* instead of *_STATE_SHUTDOWN*).

Raises
- ***ModeError*** – If not the *primary node* (not in *MODE_PRIMARY*).
- **RuntimeError** – If not yet *started*.

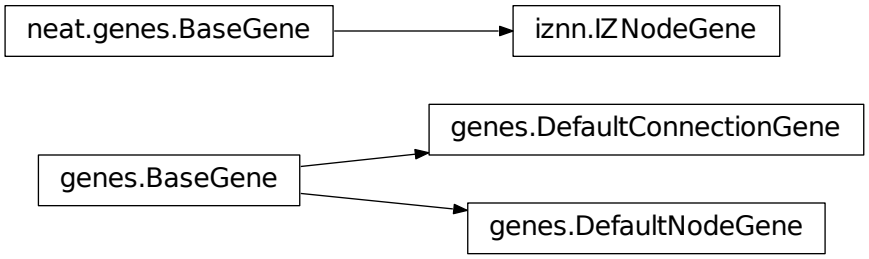
evaluate(*genomes*, *config*)
Evaluates the genomes. Distributes the genomes to the secondary nodes, then gathers the fitnesses from the secondary nodes and assigns them to the genomes. Must not be called by *secondary nodes*. TODO: Improved handling of errors from broken connections with the secondary nodes may be needed.
Parameters
- **genomes** (dict(int, instance)) – Dictionary of (*genome_id*, genome)
- **config** (instance) – Configuration object.

Raises ***ModeError*** – If not the *primary node* (not in *MODE_PRIMARY*).

New in version 0.92.

genes

Handles node and connection 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 or tuple(int, int)) – The gene *identifier*. Note: For connection genes, determining whether they are *homologous* (for *genomic distance* and *crossover* determination) uses the (ordered) identifiers of the connected nodes.

__str__()

Converts gene attributes into a printable format.

Returns Stringified gene instance.

Return type str

__lt__(*other*)

Allows sorting genes by *keys*.

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

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 *BaseAttribute.get_config_params*). 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(instance)

Raises **DeprecationWarning** – If the gene class uses `__gene_attributes__` instead of `_gene_attributes`

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** (instance) – 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** (instance) – 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 instance

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** (instance) – 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 instance

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

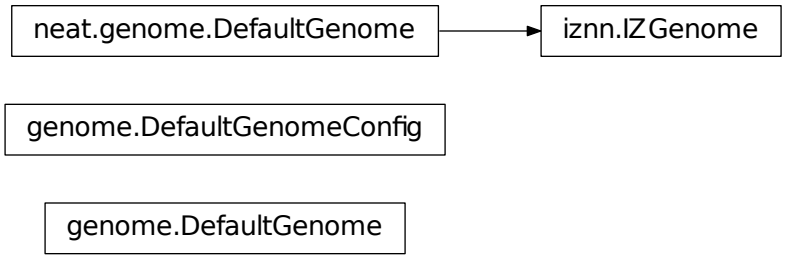
Returns The contribution of this pair to the *genomic distance* between the source genomes.

Return type float

Changed in version 0.92: __gene_attributes__ changed to _gene_attributes, since it is not a Python internal variable. Updates also made due to addition of default capabilities to *attributes*.

genome

Handles genomes (individuals in the population).



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

Does the configuration for the DefaultGenome class. Has 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*. The *activations.ActivationFunctionSet* instance is available via its `activation_defs` attribute, and the *aggregations.AggregationFunctionSet* instance is available via its `aggregation_defs` - or, for compatibility, `aggregation_function_defs` - attributes. TODO: Check for unused configuration parameters from the config file.

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

Raises **RuntimeError** – If `initial_connection` or *structural_mutation_surer* is invalid.

Changed in version 0.92: Aggregation functions moved to *aggregations*; additional configuration parameters added.

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()*.

add_aggregation(*name*, *func*)

Adds a new *aggregation function*. Uses *AggregationFunctionSet.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 *aggregations.validate_aggregation()*.

New in version 0.92.

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.

Raises **RuntimeError** – If the value for a *partial-connectivity configuration* is not in [0.0,1.0].

get_new_node_key(*node_dict*)

Finds the next unused node *key*. TODO: Explore using the same *node* key if a particular connection is replaced in more than one genome in the same generation (use a *reporting.BaseReporter.end_generation()* method to wipe a dictionary of connection tuples versus node keys).

Parameters **node_dict** (dict(int, instance)) – A dictionary of node keys vs nodes

Returns A currently-unused node key.

Return type int

Raises **AssertionError** – If a newly-created id is already in the node_dict.

Changed in version 0.92: Moved from DefaultGenome so no longer only single-genome-instance unique.

check_structural_mutation_surer()

Checks vs *structural_mutation_surer* and, if necessary, `single_structural_mutation` to decide if changes from the former should happen.

Returns If should have a structural mutation under a wider set of circumstances.

Return type bool

New in version 0.92.

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(str, str)*) – 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 instance

classmethod write_config(*f*, *config*)

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

Parameters

- **f** (file) – File object to write to.
- **config** (instance) – 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** (instance) – 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** (instance) – The first parent genome.
- **genome2** (instance) – The second parent genome.
- **config** (instance) – Genome configuration object; currently ignored.

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*. Checks *single_structural_mutation* for whether more than one structural mutation should be permitted per call. 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** (instance) – Genome configuration object.

Changed in version 0.92: `single_structural_mutation` config parameter added.

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. If there are no connections available, may call `mutate_add_connection()` instead, depending on the result from `check_structural_mutation_surer`.

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

Changed in version 0.92: Potential addition of connection instead added.

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 further validation of this connection addition?

Parameters

- **config** (instance) – Genome configuration object.
- **input_key** (`int`) – *Key* of the connection's input-side node.
- **output_key** (`int`) – Key of the connection's output-side 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. (If an existing connection is selected, it may be *enabled* depending on the result from `check_structural_mutation_surer`.) 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** (instance) – Genome configuration object

Changed in version 0.92: Output nodes not allowed to be connected together. Possibility of enabling existing connection added.

mutate_delete_node(*config*)

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

Parameters **config** (instance) – 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** (instance) – The other DefaultGenome instance (genome) to be compared to.
- **config** (instance) – 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.

Returns Genome complexity

Return type tuple(int, int)

__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** (instance) – The genome configuration object.
- **node_id** (*int*) – The key for the new node.

Returns The new node instance.

Return type instance

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** (instance) – The genome configuration object.
- **input_id** (*int*) – The input end node's key.
- **output_id** (*int*) – The output end node's key.

Returns The new connection instance.

Return type instance

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** (instance) – The genome configuration object.

Changed in version 0.92: Connect_fs_neat, connect_full, connect_partial split up - documentation vs program conflict.

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** (instance) – The genome configuration object.

Changed in version 0.92: Connect_fs_neat, connect_full, connect_partial split up - documentation vs program conflict.

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** (instance) – 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 0.92: "Direct" added to help with documentation vs program conflict for `initial_connection` of `full` or `partial`.

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

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

Changed in version 0.92: Connect_fs_neat, connect_full, connect_partial split up - documentation vs program conflict.

connect_full_direct(*config*)
Create a fully-connected genome, including direct input-output connections even if there are hidden nodes.

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

Changed in version 0.92: Connect_fs_neat, connect_full, connect_partial split up - documentation vs program conflict.

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

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

Changed in version 0.92: Connect_fs_neat, connect_full, connect_partial split up - documentation vs program conflict.

connect_partial_direct(*config*)
Create a partially-connected genome, possibly including direct input-output connections even if there are hidden nodes.

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

Changed in version 0.92: Connect_fs_neat, connect_full, connect_partial split up - documentation vs program conflict.

graphs

Directed graph algorithm implementations.

graphs.**creates_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 set of node identifiers.

Return type 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*.

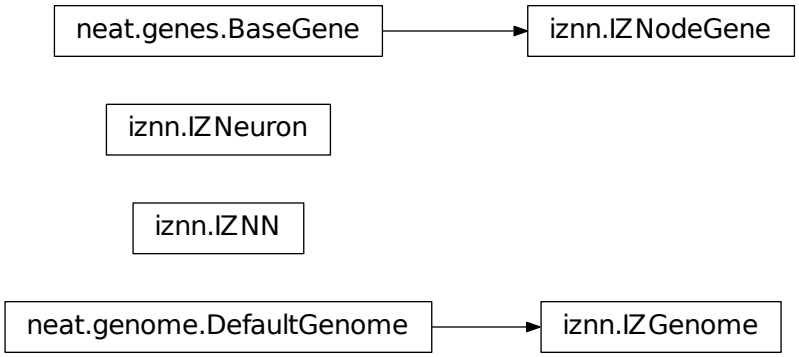
Return type list(set(int))

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.



iznn.**REGULAR_SPIKING_PARAMS**

iznn.**INTRINSICALLY_BURSTING_PARAMS**

iznn.**CHATTERING_PARAMS**

iznn.**FAST_SPIKING_PARAMS**

iznn.**THALAMO_CORTICAL_PARAMS**

iznn.**RESONATOR_PARAMS**

iznn.**LOW_THRESHOLD_SPIKING_PARAMS**
Parameter sets (for `a`, `b`, `c`, and `d`, described below) producing known types of spiking behaviors.

class iznn.**IZNodeGene**(*BaseGene*)
Contains attributes for the iznn *node* genes and determines *genomic distances*. TODO: Genomic distance currently does not take into account the node's *bias*.

distance(*other*, *config*)
Determines the *genomic distance* between this node gene and the other node gene.

Parameters
- **other** (instance) – The other IZNodeGene instance.
- **config** (instance) – Configuration object, in this case a *genome.DefaultGenomeConfig* instance.

class iznn.**IZGenome**(*DefaultGenome*)
Contains the parse_config class method for iznn genome configuration, which returns a *genome.DefaultGenomeConfig* instance.

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.

Raises **RuntimeError** – If the number of inputs does not match the number of input nodes.

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(instance)) – 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** (instance) – An IZGenome instance.
- **config** (instance) – Configuration object, in this implementation a *config.Config* instance.

Returns An IZNN instance.
Return type instance

Changed in version 0.92: __gene_attributes__ changed to _gene_attributes, since it is not a Python internal variable.

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, namely `max`, `min`, *mean*, *median*, and *median2*. The *species_fitness_func* (used for *stagnation.DefaultStagnation*) is required to be one of these.

Changed in version 0.92: *median2* added.

math_util.**mean**(*values*)
Returns the arithmetic mean.

Parameters **values** (*list(float) or set(float) or tuple(float)*) – Numbers to take the mean of.

Returns The arithmetic mean.

Return type float

math_util.**median**(*values*)
Returns the median for odd numbers of values; returns the higher of the middle two values for even numbers of values.

Parameters **values** (*list(float) or set(float) or tuple(float)*) – Numbers to take the median of.

Returns The median.

Return type float

math_util.**median2**(*values*)
Returns the median for odd numbers of values; returns the mean of the middle two values for even numbers of values.

Parameters **values** (*list(float) or set(float) or tuple(float)*) – Numbers to take the median of.

Returns The median.

Return type float

New in version 0.92.

math_util.**variance**(*values*)
Returns the (population) variance.

Parameters **values** (*list(float) or set(float) or tuple(float)*) – Numbers to get the variance of.

Returns The variance.

Return type float

math_util.**stdev**(*values*)
Returns the (population) standard deviation. *Note spelling.*

Parameters **values** (*list(float) or set(float) or tuple(float)*) – Numbers to get the standard deviation of.

Returns The standard deviation.

Return type float

math_util.**softmax**(*values*)
Compute the softmax (a differentiable/smooth approximization of the maximum function) of the given value set. (See the Wikipedia entry for more on softmax. Envisioned as useful for postprocessing of network output.)

Parameters **values** (*list(float) or set(float) or tuple(float)*) – Numbers to get the softmax of.

Returns $v_i = \exp(v_i)/s$, where $s = \sum(\exp(v_0), \exp(v_1), \dots)$(8.1)

Return typelist(float)

Changed in version 0.92: Previously not functional on Python 3.X due to changes to map.

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(float)*) – The values for the *input nodes*.
Returns The values for the *output nodes*.
Return type list(float)
Raises **RuntimeError** – If the number of inputs is not the same as the number of input nodes.

static create(*genome*, *config*)
Receives a genome and returns its phenotype.
Parameters
- **genome** (instance) – Genome to return phenotype for.
- **config** (instance) – Configuration object.

Returns A *FeedForwardNetwork* instance.
Return type instance

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(float)*) – The values for the *input nodes*.
Returns The values for the *output nodes*.
Return type list(float)
Raises **RuntimeError** – If the number of inputs is not the same as the number of input nodes.

static create(*genome*, *config*)
Receives a genome and returns its phenotype.
Parameters
- **genome** (instance) – Genome to return phenotype for.
- **config** (instance) – Configuration object.

Returns A *RecurrentNetwork* instance.
Return type instance

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. The analogous *threaded* is probably preferable for python implementations without a GIL (Global Interpreter Lock); note that neat-python is not currently tested vs any such implementations.

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*, nor by *ThreadedEvaluator* (although it is more similar to the latter).
- **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** (list(tuple(int, instance))) – A list of tuples of *genome_id* (not used), genome.
- **config** (instance) – A *config.Config* instance.

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** (instance) – The *Config* configuration object.
- **initial_state** (None or tuple(instance, instance, int)) – If supplied (such as by a method of the *Checkpointer* class), a tuple of (Population, Species, generation number)

Raises **RuntimeError** – If the *fitness_criterion* function is invalid.

run(*fitness_function*, *n=None*)
Runs NEAT's genetic algorithm for at most n generations. If n is None, run until a solution is found or total 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** (function) – 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 instance

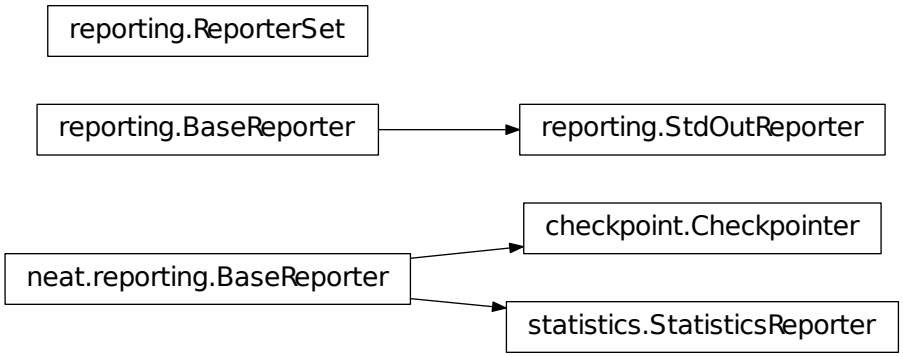
Raises

- **RuntimeError** – If None for n but *no_fitness_termination* is True.
- ***CompleteExtinctionException*** – If all species go extinct due to *stagnation* but *reset_on_extinction* is False.

Changed in version 0.92: *no_fitness_termination* capability added.

reporting

Makes possible reporter classes, which are triggered on particular events and may provide information to the user, may do something else such as checkpointing, or may do both.



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** (instance) – A reporter instance.

remove(*reporter*)

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

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

post_evaluate(*config*, *population*, *species*)

Calls *post_evaluate* on each reporter in the set.

Parameters

- **config** (instance) – *Config* configuration instance.
- **population** (dict(int, instance)) – Current population, as a dict of unique genome *ID/key* vs genome.
- **species** (instance) – Current species set object, such as a *DefaultSpeciesSet* instance.
- **best_genome** (instance) – 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** (instance) – *Config* configuration instance.
- **generation** (*int*) – Generation number.
- **best** (instance) – 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** (instance) – The *Species* instance.

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

post_evaluate(*config*, *population*, *species*, *best_genome*)
Called via *ReporterSet* (by *population.Population.run()*) after the fitness function is finished.
Parameters
- **config** (instance) – *Config* configuration instance.
- **population** (dict(int, instance)) – Current population, as a dict of unique genome *ID/key* vs genome.
- **species** (instance) – Current species set object, such as a *DefaultSpeciesSet* instance.
- **best_genome** (instance) – 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, unless *no_fitness_termination* is set; if it is set, then called upon reaching the generation maximum - set when calling `population.Population.run()` - and exiting for this reason.)

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

Changed in version 0.92: *no_fitness_termination* capability added.

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** (instance) – The `Species` instance.

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

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

class reproduction.**DefaultReproduction** (*config*, *reporters*, *stagnation*)
Implements the default NEAT-python reproduction scheme: explicit fitness sharing with fixed-time species stagnation. Inherits from `config.DefaultClassConfig` the required class method `write_config`. TODO: Provide some sort of optional cross-species performance criteria, which are then used to control stagnation and possibly the mutation rate configuration. This scheme should be adaptive so that species do not evolve to become "cautious" and only make very slow progress.

Parameters
- **config** (instance) – Configuration object, in this implementation a `config.DefaultClassConfig` instance.
- **reporters** (instance) – A `ReporterSet` instance.
- **stagnation** (instance) – A `DefaultStagnation` instance - the current code partially depends on internals of this class (a TODO is noted to correct this).

Changed in version 0.92: Configuration changed to use DefaultClassConfig, instead of a dictionary, and inherit write_config.

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*, in this implementation using *config.DefaultClassConfig*.

Parameters **param_dict** (*dict(str, str)*) – Dictionary of parameters from configuration file.

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

Return type DefaultClassConfig instance

Changed in version 0.92: Configuration changed to use DefaultClassConfig instead of a dictionary.

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** (instance) – 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, instance)

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()* and *set* in the configuration file.
- **min_species_size** (*int*) – Minimum number of members per species, set via the *min_species_size* configuration parameter (or the *elitism* configuration parameter, if higher); 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** (instance) – A *Config* instance.
- **species** (instance) – 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, instance)

Changed in version 0.92: Previously, the minimum and maximum relative fitnesses were determined (contrary to the comments in the code) including members of species being removed due to stagnation; it is now determined using only the non-stagnant species. The minimum size of species was (and is) the greater of the *min_species_size* and *elitism* configuration parameters; previously, this was not taken into account for *compute_spawn()*; this made it more likely to have a population size above the *configured population size*.

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** (instance) – A genome instance.
- **members** (dict(int, instance)) – 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*, memoized by this class, 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** (instance) – A genome configuration instance; 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** (instance) – The first genome instance.
- **genome1** (instance) – The second genome instance.

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. Inherits from *config.DefaultClassConfig* the required class method *write_config*.

Parameters
- **config** (instance) – A configuration object, in this implementation a *config.Config* instance.
- **reporters** (instance) – A *ReporterSet* instance giving reporters to be notified about *genomic distance* statistics.

Changed in version 0.92: Configuration changed to use DefaultClassConfig, instead of a dictionary, and inherit write_config.

classmethod parse_config(*param_dict*)
Required interface method. Currently, the only configuration parameter is the *compatibility_threshold*; this method provides a default for it and updates it from the configuration file, in this implementation using *config.DefaultClassConfig*.
Parameters **param_dict** (*dict(str, str)*) – Dictionary of parameters from configuration file.
Returns SpeciesSet configuration object; considered opaque by rest of code, so current type returned is not required for interface.
Return type DefaultClassConfig instance
Changed in version 0.92: Configuration changed to use DefaultClassConfig instead of a dictionary.

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

Required interface method. Place genomes into species by genetic similarity (*genomic distance*). TODO: 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*. TODO: Check if sorting the unspeciated genomes by fitness will improve speciation (by making the highest-fitness member of a species its representative).

Parameters

- **config** (instance) – `Config` instance.
- **population** (dict(int, instance)) – 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 instance

stagnation

Keeps track of whether species are making progress and helps remove ones that are not.

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).

---

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. Inherits from `config.DefaultClassConfig` the required class method `write_config`.

Parameters

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

Changed in version 0.92: Configuration changed to use DefaultClassConfig, instead of a dictionary, and inherit write_config.

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, in this implementation using `config.DefaultClassConfig`.

Parameters **param_dict** (*dict(str, str)*) – Dictionary of parameters from configuration file.

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

Return type DefaultClassConfig instance

Changed in version 0.92: Configuration changed to use DefaultClassConfig instead of a dictionary.

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

Returns A list of tuples of (species *id/key*, *Species* instance, is_stagnant).

Return type list(tuple(int, instance, bool))

Changed in version 0.92: Species sorted (by the species fitness according to the `species_fitness_func`) 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 mean 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_median()
Gets the per-generation median fitness. A wrapper for *get_fitness_stat()* with the function being *median2*. Not currently used internally.

New in version 0.92.

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 instances).
Return type list(instance)

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 instances).
Return type list(instance)

best_genome()
Returns the most-fit genome ever seen. A wrapper around *best_genomes()*.
Returns The most-fit genome.
Return type instance

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.

threaded

Runs evaluation functions in parallel threads (using the python library module threading) in order to evaluate multiple genomes at once. Probably preferable to *parallel* for python implementations without a GIL (Global Interpreter Lock); note, however, that neat-python is not currently tested on any such implementation.

class threaded.**ThreadedEvaluator**(*num_workers*, *eval_function*)
Runs evaluation functions in parallel threads in order to evaluate multiple genomes at once.

Parameters

- **num_workers** (*int*) – How many worker threads to use.
- **eval_function** (function) – The eval_function should take two arguments - a genome object and a 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*, nor by *ParallelEvaluator* (although it is more similar to the latter).

__del__()
Attempts to take care of removing each worker thread, but deliberately calling self.stop() in the threads may be needed. TODO: Avoid reference cycles to ensure this method is called. (Perhaps use weakref, depending on what the cycles are? Note that weakref is not compatible with saving via pickle, so all of them will need to be removed prior to any save.)

start()
Starts the worker threads, if in the primary thread.

stop()
Stops the worker threads and waits for them to finish.

_worker():
The worker function.

evaluate(*genomes*, *config*)
Starts the worker threads if need be, queues the evaluation jobs for the worker threads, then assigns each fitness back to the appropriate genome.

Parameters

- **genomes** (list(tuple(int, instance))) – A list of tuples of *genome_id*, genome instances.
- **config** (instance) – A *config.Config* instance.

New in version 0.92.

Table of Contents

CHAPTER 9

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).

CHAPTER 10

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.

CHAPTER 11

Glossary

activation function

aggregation function

bias

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

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).

Using the `distributed` module, genomes can be evaluated on multiple machines (including virtual machines) at once. Each such machine/host is called a `compute node`. These are of two types, *primary nodes* and *secondary nodes*.

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`.

This implementation of NEAT uses, like most, multiple semi-separated generations (some genomes may survive multiple generations via *elitism*). In terms of generations, the steps are as follows: generate the next generation from the current population; partition the new generation into *species* based on *genetic similarity*; evaluate fitness of all genomes; check if a/the termination criterion is satisfied; if not, repeat. (The ordering in the `population` module is somewhat different.) Generations are numbered, and a limit on the number of generations is one type of termination criterion.

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 *disjoint*/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. For *input nodes* (or input *pins*), it is the input's (list or tuple) index plus one, then multiplied by negative one; for *output nodes*, it is equal to the output's (list or tuple) index.

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`. (They should not be confused with *compute nodes*, host machines on which `distributed` evaluations of *genomes* are performed.)

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).

If using the `distributed` module, you will need one primary *compute node* and at least one *secondary node*. The primary node creates and mutates genomes, then distributes them to the secondary nodes for evaluation. (It does not do any evaluations itself; thus, at least one secondary node is required.)

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.)

If using the `distributed` module, you will need at least one secondary *compute node*, as well as a *primary node*. The secondary nodes evaluate genomes, distributed to them by the primary node.

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

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

T

U

V

W

X