pyneat Documentation

Release unknown

Adam Tupper

Jun 13, 2020

Contents

This is the documentation of **PyNEAT**, a python implementation of the NeuroEvolution of Augmenting Topologies (NEAT) algorithm created by Kenneth O'Stanley.

Note: PyNEAT began as an original implementation of NEAT, and then was modified to use the same interface as NEAT-Python to reuse various reporting and configuration features they had developed. However, the core behaviour of the NEAT algorithm implemented in PyNEAT is significantly different to that of NEAT-Python, and is much closer to the original specification of NEAT.

CHAPTER 1

Contents

1.1 Installation

If you want to draw the architectures of the networks encoded by genomes, you will need to install Graphviz:

```
sudo apt update
sudo apt install graphviz graphviz-dev
```

The remaining dependencies (listed in the requirements file) can then be installed using pip:

```
pip install -r requirements.txt
```

Finally, PyNEAT can be install via:

```
python setup.py install
```

1.2 Getting Started

PyNEAT can be imported into Python programs using:

```
import pyneat
```

1.3 License

1.4 Contributors

- Adam Tupper <adam.tupper@outlook.com>

1.5 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

1.5.1 Version 0.3.0 (13-06-2020)

This release further improves performance and substantially reduces runtime. Performance in the XOR experiments (95% success rate) now more closely matches the reported results from the original paper and the performance of Kenneth O'Stanley's C++ implementation.

Added

- Average crossover operation that averages the weights of mutual connections, instead of choosing from the parents at random. The probability of this occurring is governed by the `crossover_avg_prob` config parameter.
- Custom stagnation behaviour that doesn't preserve elite species if the number of species is above the species elitism threshold. This is more inline with the original implementation of NEAT. Previously, the stagnation behaviour of NEAT-Python was used.
- The visualise module (for visualising results) that was included with examples has now been added to the library proper.
- A `LogFileReporter` that reports the same information as the NEAT-Python `StdOutReporter`, but instead saves the logs to a log file.
- The ability to toggle on/off gene distance normalisation for the genetic distance calculation. Controlled by the `normalise_gene_dist` config parameter.
- Search refocusing if the population stagnates for 20 generations.

Changed

- Changed `crossover_prob` config parameter to `mutate_only_prob` (equivalent to 1 - `crossover_prob`).
- Added retries for add node mutations.
- Reduced the maximum weight magnitudes for the XOR example to match the original implementation of NEAT. Now weights are restricted to the range [-8., 8.].
- The `Population.run` function can now take additional keyword arguments that are passed to the provided fitness evaluation function.
- Introduced mpmath dependency to use for calculating exponentials in the activation functions (to avoid overflow errors).
- Use custom copy functions for `Genome`, `NodeGene`, and `ConnectionGene` to avoid using `deepcopy()`. This significantly reduces run time (~ 3x faster).
- Reduce the number of retries from 100 to 20 for "add connection" mutations.
- Update the XOR example solution check and config to match the original NEAT experiment.
- Remove node genes from being counted in compatibility distance calculations.
- Assign genomes to the first compatible species instead of calculating best representatives.
- Change weight replacements to use the `weight_perturb_power` instead of `weight_init_power`.

Fixed

- Fixed bug preventing `survival_threshold` from being applied.
- Species adjusted fitness is now saved to the right attribute.
- Disabled connections being used in feed-forward neural networks.
- Non-required nodes being labelled as required in some circumstances for feed-forward networks.

1.5.2 Version 0.2.0 (25-04-2020)

Added

- Support for feed-forward networks.
- A configurable probability that offspring generated through crossover are not also mutated (`crossover_only_prob`).
- Steepened sigmoid activation function.
- Markov and non-Markov double pole balancing examples.
- Non-Markov single pole balancing example.
- XOR example.

Changed

- Node and connection gene keys are now generated and kept track of using an `InnovationStore` to improve crossover.
- Nodes no longer have individual biases, instead bias nodes are (optionally) used. The matches the original specification of NEAT (and reduces the search space).
- "Add connection" mutations now retry a certain number of times if a connection cannot be added (or is already present) between the selected nodes.
- Switched from Gaussian to uniform connection weight initialisation and perturbation to match the original NEAT specification.
- Now ensure that weight mutations are not performed if a structural mutation has been performed.
- Improved the method for identifying nodes that are required to compute network outputs. Previously, some unused nodes were also included.

Fixed

- Small errors in the implementation of adjusted fitness and the largest remainder allocation method that are used for calculating species offspring allocations.
- A bug causing the same parent sometimes being selected twice for crossover.
- A bug in the forward pass of RNNs that caused the current state of the outputs (instead of the previous state) to be returned.

1.5.3 Version 0.1.0 (13-02-2020)

Added

- Initial implementation based on NEAT-Python.

1.6 pyneat

1.6.1 pyneat package

Submodules

pyneat.activations module

A collection of built-in activation functions.

class pyneat.activations.**ActivationFunctionSet**
Bases: object

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

add(*name*, *function*)

get(*name*)

is_valid(*name*)

exception pyneat.activations.**InvalidActivationFunction**
Bases: TypeError

pyneat.activations.**abs_activation**(*z*)

pyneat.activations.**clamped_activation**(*z*)

pyneat.activations.**cube_activation**(*z*)

pyneat.activations.**elu_activation**(*z*)

pyneat.activations.**exp_activation**(*z*)

pyneat.activations.**gauss_activation**(*z*)

pyneat.activations.**hat_activation**(*z*)

pyneat.activations.**identity_activation**(*z*)

pyneat.activations.**inv_activation**(*z*)

pyneat.activations.**lelu_activation**(*z*)

pyneat.activations.**log_activation**(*z*)

pyneat.activations.**relu_activation**(*z*)

pyneat.activations.**selu_activation**(*z*)

pyneat.activations.**sigmoid_activation**(*z*)

pyneat.activations.**sin_activation**(*z*)

pyneat.activations.**softplus_activation**(*z*)

pyneat.activations.**square_activation**(*z*)

pyneat.activations.**steep_sigmoid_activation**(*z*)
Used in the original implementation by Stanley and Miikkulainen (2002).

pyneat.activations.**tanh_activation**(*z*)

pyneat.activations.**validate_activation**(*function*)

pyneat.config module

Customised configuration module with additional user-configurable parameters.

Modifies the config object in the default NEAT-Python config module to include the following additional parameters:

- *num_episodes*: the number of episodes each genome/agent should be evaluated for.
- *num_runs*: the number of evolutionary runs to perform.
- *checkpoint_interval*: the number of generations between checkpoint saves.
- *max_generations*: the maximum number of generations for each evolutionary run.

class pyneat.config.**CustomConfig**(*genome_type*, *reproduction_type*, *species_set_type*, *stagnation_type*, *filename*)
Bases: object

A simple custom config container for user-configurable parameters of NEAT.

To include additional top-level parameters, specify them in __params.

save(*filename*)

pyneat.genome module

This module defines the genome encoding used by NEAT.

class pyneat.genome.**ConnectionGene**(*key*, *node_in*, *node_out*, *weight*, *expressed*)
Bases: object

Defines a connection gene used in the genome encoding.

key
The innovation key for this gene.

Type int

node_in
The key of the node this connection from.

Type int

node_out
The key of the node this connection is to.

Type int

weight
The connection weight.

Type float

expressed
True if the connection is expressed (enabled) in the phenotype, False otherwise.

Type bool

copy()
Create a copy of the connection gene.

Returns A copy of itself.

Return type *ConnectionGene*

class pyneat.genome.**Genome**(*key*, *config*, *innovation_store*)
Bases: object

Defines a genome used to encode a neural network.

key
A unique identifier for the genome.

Type int

config
The genome configuration settings.

Type *GenomeConfig*

fitness
The fitness of the genome.

Type float

nodes
A dictionary of node key (int), node gene pairs.

Type dict

connections
A dictionary of connection gene key, connection gene pairs.

Type dict

inputs
The node keys of input nodes.

Type :list:'int'

outputs
The node keys of output nodes.

Type :list:'int'

biases
The node keys of bias nodes.

Type :list:'int'

innovation_store
The global innovation store used for tracking new structural mutations.

Type *InnovationStore*

add_bias_node(*num*)
Add a new bias node.

Parameters **num** (*int*) – A number that can uniquely identify bias nodes in the innovation store.

add_connection(*node_in*, *node_out*, *weight*, *expressed=True*)
Add a connection between two nodes.

Parameters

- **node_in** (*int*) – The key of the node that leads into the new connection.
- **node_out** (*int*) – The key of the node that the the new connection leads into.
- **weight** (*float*) – The weight of the connection. Must be a value between [0, 1].
- **expressed** (*bool*) – True if the connection should be expressed in the phenotype, False otherwise.

add_node(*node_in*, *node_out*, *node_type*)
Add a new node positioned between two other nodes. Input and output nodes are positioned between non-existent nodes.

Parameters

- **node_in** (*int*) – The key of the node that precedes this new node.
- **node_out** (*int*) – The key of the node that succeeds this new node.
- **node_type** (NodeType) – The type of node to be added.

Returns The key of the new node

Return type int

configure_crossover(*parent1*, *parent2*, *average*)
Performs crossover between two genomes.

If the two genomes have equal fitness then the joint and excess genes are inherited from the smaller genome.

Parameters

- **parent1** (Genome) – The first parent.
- **parent2** (Genome) – The second parent.
- **average** (*bool*) – Whether or not to average the weights of mutual connections or choose at random from one of the parents.

configure_new()
Configure a new genome based on the given configuration.

The initial inputs and outputs for input and output nodes are specified as negatives so that matching innovation keys are generated for corresponding input and output nodes between genomes. Inputs nodes use odd negative numbers, and output nodes use even negative numbers.

copy()
Create a copy of the genome.

Note: Copies share the same config and innovation store.

Returns

A copy of itself, but with the same config and innovation store.

Return type *Genome*

distance(*other*)
Computes the compatibility (genetic) distance between two genomes.

This is used for deciding how to speciate the population. Distance is a function of the number of disjoint and excess genes, as well as the weight/bias differences of matching genes.

Update (13.04.20): Distance is measured using the original compatibility distance measure defined by Stanley & Miikkulainen (2002).

Parameters **other** (Genome) – The other genome to compare itself to.

Returns The genetic distance between itself and the other genome.

Return type float

mutate()
Mutate the genome.

Mutates the genome according to the mutation parameter values specified in the genome configuration.

As per the original implementation of NEAT:

- If any structural mutations are performed, weight and bias mutations will not be performed.
- If an add node mutation is performed, an add connection mutation will not also be performed.

mutate_add_connection()
Performs an 'add connection' structural mutation.

A single connection with a random weight is added between two previously unconnected nodes.

Returns True if a connection was added, False otherwise.

Return type bool

mutate_add_node()
Performs an 'add node' structural mutation.

An existing connection is split and the new node is placed where the old connection used to be. The old connection is disabled and two new connection genes are added. The new connection leading into the new

node receives a weight of 1.0 and the connection leading out of the new node receives the old connection weight.

Connections from bias nodes and non-expressed nodes are not split.

Returns True is a node was added, False otherwise.

Return type bool

mutate_weights()
Mutates (perturbs) or replaces each connection weight in the genome.

Each weight is either replaced (with some probability, specified in the genome config) or perturbed.

Replaced weights and perturbations are drawn from a uniform distribution with range [-weight_perturb_power, weight_perturb_power).

classmethod parse_config(*param_dict*)
Takes a dictionary of configuration items, returns an object that will later be passed to the write_config method.

Parameters **param_dict** (*dict*) – A dictionary of configuration parameter values.

Returns The genome configuration.

Return type *GenomeConfig*

size()
Returns a measure of genome complexity.

Returns

A measure of the complexity of the genome given by (number of nodes, number of enabled connections)

Return type tuple

classmethod write_config(*filename*, *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.

Parameters

- **filename** (*str*) – The name of the file to write the genome configuration to.
- **config** (GenomeConfig) – The genome configuration to save.

class pyneat.genome.**GenomeConfig**(*params*)
Bases: object

Sets up and holds configuration information for the Genome class.

Config Parameters: num_inputs (int): The number of inputs each network should have.

num_outputs (int): The number of outputs each network should have.

num_biases (int): The number of bias nodes the network should have.

initial_conn_prob (float): The initial connection probability of each potential connection between inputs and outputs. 0.0 = no connections, i.e. all inputs are disconnected from the outputs. 1.0 = fully connected, i.e. all inputs are connected to all outputs.

activation_func (str): The name of the activation function to be used by hidden and output nodes. Must be present in the set of possible activation functions.

compatibility_disjoint_coefficient (float): The disjoint and excess coefficient to be used when calculating genome distance.

compatibility_weight_coefficient (float): The weight and bias coefficient to be used when calculation genome distance.

normalise_gene_dist (bool): Whether or not normalise the gene dist (for genetic distance calculations) for large genomes.

feed_forward (bool): False if recurrent connections are allowed, True otherwise.

conn_add_prob (float): The probability of adding a new connection when performing mutations.

node_add_prob (float): The probability of adding a new node when performing mutations.

weight_mutate_prob (float): The probability of mutating the connection weights of a genome when performing mutations.

weight_replace_prob (float): The probability of replacing, instead of perturbing, a connection weight when performing weight mutations.

weight_init_power (float): Sets the range of possible values for weight replacements and new weight initialisations.

weight_perturb_power (float): Sets the range of possible values for weight perturbations.

weight_min_value (float): Sets the minimum allowed value for connection weights.

weight_max_value (float): Sets the maximum allowed value for connection weights.

gene_disable_prob (float): The probability of disabling a gene in the child that is disabled in either of the parents when performing crossover.

save(*filename*)
Save the genome configuration.

Parameters **filename** (*str*) – The filename to write the configuration to.

class pyneat.genome.**NodeGene**(*key*, *type*, *activation*)
Bases: object

Defines a node gene used in the genome encoding.

key
The innovation key (also the node key) for this gene.

Type int

type
The type of the node (either input, output or hidden).

Type *NodeType*

activation
The node activation function. Note that input and bias nodes should not have an activation function (i.e. it is the identity function).

Type function

copy()
Create a copy of the node gene.

Returns A copy of itself.

Return type *NodeGene*

class pyneat.genome.**NodeType**
Bases: enum.Enum

Define the types for nodes in the network.

BIAS = 3

HIDDEN = 1

INPUT = 0

OUTPUT = 2

pyneat.graph_utils module

Utility functions for various graph operations.

pyneat.graph_utils.**creates_cycle**(*connections*, *test*)

Checks to see if adding the test connection to the network would create a cycle.

Parameters

- **connections** (*list*) – A list of (node in key, node out key) pairs for each connection.
- **test** (*tuple*) – A tuple of the form (node in key, node out key) that represents the connection trying to be added that needs to be tested.

Returns True if 'test' creates a cycle, False otherwise.

Return type bool

pyneat.graph_utils.**find_path**(*sources*, *goals*, *connections*)

Try to find a path between the any of the start nodes and any of the goal nodes.

Parameters

- **sources** (*list*) – A list of node keys that the path may start from.
- **goals** (*list*) – A list of node keys that the path may finish at.
- **connections** (*list*) – A list of tuples that specify the input and output node keys of each enabled connection.

Returns A list of each node along the discovered path.

Return type list

pyneat.graph_utils.**group_layers**(*inputs*, *outputs*, *biases*, *connections*, *nodes*)

Group nodes together into layers that can be evaluated in parallel by a feed-forward neural network.

i.e. nodes in the same layer are independent conditional on the nodes in previous layers.

Parameters

- **inputs** (*list*) – The keys of the input nodes.
- **outputs** (*list*) – The keys of the output nodes.
- **biases** (*list*) – The keys of the bias nodes.
- **connections** (*list*) – A list of (node in key, node out key) pairs for each expressed connection.
- **nodes** (*set*) – The set of nodes required for calculating the output value.

Returns

A list of sets that contain the node keys for the nodes in each layer.

Return type list

pyneat.graph_utils.**required_for_output**(*inputs*, *biases*, *outputs*, *connections*, *nodes*)
Check to see if a node is required for computing the output of the network.

A hidden node h in a NN is required if the following hold:

a) there is a path from h to an output node

b) there is a path from an input node to h

Shortcuts can be taken if there is a path from h1 to h2 and h1 has been marked as required.

Parameters

- **inputs** (*list*) – The keys of input nodes.
- **biases** (*list*) – The keys of bias nodes.
- **outputs** (*list*) – The keys of output nodes.
- **connections** (*list*) – A list of tuples that specify the input and output node keys of each enabled connection.
- **nodes** (*list*) – The keys of all nodes in the network.

Returns The set of nodes required for computing the output of the network.

Return type set

pyneat.innovation module

Module for tracking structural innovations in genomes during evolution.

class pyneat.innovation.**InnovationRecord**(*key*, *innov_type*, *node_in*, *node_out*)
Bases: object

Record information about a structural mutations for comparison against other mutations.

key
A unique identifier for the record.

Type int

innov_type
The type of structural mutation that occurred.

Type *InnovationType*

node_in
The incoming node for the new connection or node (at the time of the mutation occurring in the case of new node mutations).

Type int

node_out
The outgoing node for the new connection or node (at the time of the mutation occurring in the case of new node mutations).

Type int

class pyneat.innovation.**InnovationStore**
Bases: object

Store records of new node and connection mutations for lookup. Also responsible for generating unique innovation keys.

key_to_record
A dictionary containing innovation records for each new structural mutation, indexed by innovation keys.

Type dict

mutation_to_key
A dictionary containing innovation keys, indexed by mutations (node_in, node_out, innovation_type).

Type dict

_innovation_key_generator
Generates the next innovation key.

Type generator

get_innovation_key(*node_in*, *node_out*, *innovation_type*)
Get a new or existing innovation key for a structural mutation.

Parameters

- **node_in** (*int*) – The input node to the new node/connection.
- **node_out** (*int*) – The output node to the new node/connection.
- **innovation_type** (InnovationType) – The type of structural mutation.

Returns The innovation key for the mutation.

Return type int

class pyneat.innovation.**InnovationType**
Bases: enum.Enum

Define the types of structural innovations.

NEW_BIAS = 2

NEW_CONNECTION = 1

NEW_NODE = 0

pyneat.population module

Implements the core of the evolutionary algorithm.

exception pyneat.population.**CompleteExtinctionException**
Bases: Exception

class pyneat.population.**Population**(*config*, *initial_state=None*)
Bases: object

This class implements the core evolution algorithm.

The steps of the algorithm are as follows:

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.

reporters
The set of reporters used for logging.

Type ReporterSet

config
The global configuration settings for the entire algorithm.

Type *CustomConfig*

reproduction
The reproduction scheme for generating genomes.

Type *Reproduction*

innovation_store
The store for innovation records for tracking structural mutations.

Type *InnovationStore*

fitness_criterion
The fitness function to assess the population with to test for termination.

Type function

population
The population of individuals. A dictionary of genome key, genome pairs.

Type dict

species
The speciation scheme for dividing the population into species.

Type *SpeciesSet*

generation
The generation number.

Type int

best_genome
The best genome discovered so far (according to fitness).

Type *Genome*

add_reporter(*reporter*)
Add a new reporter to the reporter set.

Parameters **reporter** (*Reporter*) – The reporter to add to the reporter set.

remove_reporter(*reporter*)
Remove a reporter from the reporter set.

Parameters **reporter** (*Reporter*) – The reporter to remove from the reporter set.

run(*fitness_function*, *n=None*, ***kwargs*)
Runs NEAT's genetic algorithm for at most n generations. If n is None, run until solution is found or extinction occurs.

The user-provided fitness_function must take only two arguments:

1. The population as a list of (genome id, genome) tuples.
2. The current configuration object.

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

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

It is assumed that 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 assess genomes with.
- **n** (*int*) – The maximum number of generations to run for.
- ****kwargs** – Extra arguments that are passed to the fitness function.

Returns The best genome found during the run(s).

Return type *Genome*

pyneat.reporting module

Additional reporters (on top of what are offered by NEAT-Python) that are triggered on particular events.

class pyneat.reporting.**LogFileReporter**(*filename*, *log_level=20*, *show_species_detail=True*)

Bases: neat.reporting.BaseReporter

Write the same information as the StdOutReporter, but instead of printing to standard out, the outputs are written to a log file.

filename
The name of the log file to use.

Type str

log_level
The log level (either DEBUG, INFO, WARNING, ERROR, or CRITICAL). Note: All LogFileReporter logs are INFO level.

Type logging.level

show_species_detail
Whether to show detailed species information.

Type bool

generation
The current generation.

Type int

generation_start_time
The start time of the current generation.

Type int

generation_times
The length of time to complete each generation.

Type list

num_extinctions
The number of extinctions that have occurred.

Type int

complete_extinction()
The log message to write after a complete extinction has occurred.

end_generation(*config*, *population*, *species_set*)
The log message to write when a generation is ended.

Parameters

- **config** (CustomConfig) – The global configuration settings for the entire algorithm.
- **population** (*dict*) – The population of individuals. A dictionary of genome key, genome pairs.
- **species_set** (SpeciesSet) – The speciation scheme for dividing the population into species.

found_solution(*config*, *generation*, *best*)
The log message to write after a solution has been found.

info(*msg*)
Write a custom log message.

Parameters **msg** (*str*) – The log message to write.

post_evaluate(*config*, *population*, *species*, *best_genome*)
The log message to write after evaluating the fitness of the population.

Parameters

- **config** (CustomConfig) – The global configuration settings for the entire algorithm.
- **population** (*dict*) – The population of individuals. A dictionary of genome key, genome pairs.
- **species** (SpeciesSet) – The speciation scheme for dividing the population into species.
- **best_genome** (Genome) – The best genome from the population.

species_stagnant(*sid*, *species*)
The log message to write when a species has stagnated.

start_generation(*generation*)
The log message to write when a generation is started.

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

pyneat.reproduction module

Implement the NEAT reproduction scheme.

The reproduction scheme is specifies the behaviour for creating, mutating and in any way altering the population of genomes during evolution.

class pyneat.reproduction.**Reproduction**(*config*, *reporters*, *stagnation*)
Bases: object

Implements the NEAT reproduction scheme.

TODO: Decide which attributes should be private.

reproduction_config
The configuration for reproduction hyperparameters.

Type *ReproductionConfig*

reporters
The set of reporters to log events via.

Type ReporterSet

genome_key_generator
Keeps track of the next genome key when generating offspring.

Type generator

stagnation
Keeps track of which species have stagnated.

Type *Stagnation*

ancestors
A dictionary that stores the parents of each offspring produced.

Type dict

static compute_num_offspring(*remaining_species*, *pop_size*)
Compute the number of offspring per species (proportional to fitness).

Note: The largest remainder method is used to ensure the population size is maintained (https://en.wikipedia.org/wiki/Largest_remainder_method).

TODO: Investigate a more efficient implementation of offspring allocation

Parameters

- **remaining_species** (*dict*) – A dictionary ({species key: species}) of the remaining species after filtering for stagnation.
- **pop_size** (*int*) – The specified size of the population.

Returns

A dictionary of the number of offspring allowed for each species of the form {species key: number of offspring}.

Return type dict

create_new(*genome_type*, *genome_config*, *num_genomes*, *innovation_store*)
Create a brand new population.

Note: This is a required interface method.

Parameters

- **genome_type** (Genome) – The type of the genome to create individuals using.
- **genome_config** (GenomeConfig) – The genome configuration.
- **num_genomes** (*int*) – The number of genomes to create (population size).
- **innovation_store** (InnovationStore) – The population-wide innovation store used for tracking new structural mutations.

Returns

A dictionary of genome key, genome pairs that make up the new population.

Return type dict

generate_parent_pools(*remaining_species*)
Culls the lowest performing members of each remaining species

Parameters **remaining_species** (*dict*) – Species key/species pairs for the remaining species after stagnated species have been removed.

Returns

The parent genomes for each species. A dictionary of the form species key, genomes.

Return type dict

classmethod parse_config(*param_dict*)

Takes a dictionary of configuration items, returns an object that will later be passed to the write_config method.

Note: This is a required interface method.

Parameters **param_dict** (*dict*) – A dictionary of configuration parameter values.

Returns The reproduction configuration.

Return type *ReproductionConfig*

reproduce(*config*, *species*, *pop_size*, *generation*, *innovation_store*, *refocus*)

Produces the next generation of genomes.

Note: This is a required interface method.

The steps are broadly as follows:

1. Filter stagnant species.
2. Compute the number of offspring for each remaining species.
3. **Generate the parent pool for each remaining species (eliminate** the lowest performing members).
4. Generate the new population.

Parameters

- **config** (*Config*) – The experiment configuration.
- **species** (SpeciesSet) – The current allocation of genomes to species.
- **pop_size** (*int*) – The desired size of the population.
- **generation** (*int*) – The number of the next generation.
- **innovation_store** (InnovationStore) – The population-wide innovation store used for tracking new structural mutations.

Returns

A dictionary of genome key, genome pairs that make up the new population.

Return type dict

classmethod write_config(*filename*, *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.

Note: This is a required interface method.

Parameters

- **filename** (*str*) – The filename of the file to write the configuration to.
- **config** (ReproductionConfig) – The reproduction config to save.

class pyneat.reproduction.**ReproductionConfig**(*params*)
Bases: object

Sets up and hold configuration information for the Reproduction class.

Config Parameters:

mutate_only_prob (float): The probability that a child is generated through mutation alone. Crossover is only an option if there is more than one remaining parent in the parent pool for the species in question.

crossover_avg_prob (float): The probability that the weights of mutual connections are averaged from both parents instead of chosen at random from one or the other.

crossover_only_prob (float): The probability that a child generated via crossover is not also mutated.

inter_species_crossover_prob (float): The probability (given crossover) that the child is instead generating using parents from different species. Relies on their being more than one species.

num_elites (int): The number of elites from each species to be copied to the next generation. The size of a species must surpass the elitism_threshold for elitism to occur.

elitism_threshold (int): Elitism will only be applied for a species if the number of remaining parents exceeds this threshold.

survival_threshold (float): The proportion of members of each species that are added to the parent pool and are allowed to reproduce. The fittest members are kept.

save(*filename*)
Save the reproduction configuration.

Parameters **filename** (*str*) – The filename to write the configuration to.

pyneat.species module

Divides a population into species based on genetic distance.

class pyneat.species.**GenomeDistanceCache**(*config*)
Bases: object

class pyneat.species.**Species**(*key*, *generation*)
Bases: object

Encapsulates all information about a particular species.

key
A unique identifier for the species.

Type int

created
The generation in which the species was created.

Type int

last_improved
The last generation where the fitness of the species improved.

Type int

members
A dictionary of {genome ID: genome} pairs for each genome in the species.

Type dict

representative
The genome that is the representative of the species, against which new genomes will be compared to see if they belong in this species.

Type *Genome*

fitness
The species fitness.

Type float

adjusted_fitness
The sum of the adjusted fitnesses for each genome in the species.

Type float

fitness_history
All previous fitness values. One for each generation this species has survived for.

Type :list:'float'

get_fitnesses()
Get the fitnesses of each genome that belongs to this species.

Returns The fitness of each genome that belongs to this species.

Return type list

update(*representative*, *members*)
Replace the current individuals with a new set of individuals.

Parameters

- **representative** (Genome) – The genome that is the new representative
- **this species.** (*for*) –
- **members** (*dict*) – A dictionary of genome ID and genome pairs of the new members of the species.

class pyneat.species.**SpeciesSet**(*config*, *reporters*)
Bases: neat.config.DefaultClassConfig

Encapsulates the speciation scheme.

species_set_config
The speciation configuration.

Type DefaultClassConfig

reporters
The set of reporters that log events.

Type ReporterSet

species_key_generator
Keeps track of the next species ID.

Type generator

species
A dictionary of species ID, species pairs.

Type dict

genome_to_species
A dictionary of genome ID, species ID pairs.

Type dict

get_species(*genome_key*)
Get the species to given individual belongs to.

Parameters **genome_key** (*int*) – The unique key of the genome to check.

Returns The species the individual belongs to.

Return type *Species*

get_species_id(*genome_key*)
Get the species ID of the species the given individual belongs to.

Parameters **genome_key** (*int*) – The unique key of the genome to check.

Returns The key of the species the individual belongs to.

Return type int

classmethod parse_config(*param_dict*)
Parse the speciation parameter values

Parameters **param_dict** (*dict*) – A dictionary of parameter values.

Returns The speciation configuration.

Return type DefaultClassConfig

speciate(*config*, *population*, *generation*)
Speciate the population.

Parameters

- **config** (*Config*) – The global NEAT configuration.
- **population** (*dict*) – A dictionary of genome ID, genome pairs.
- **generation** (*int*) – The current generation.

pyneat.stagnation module

Track the progress of species and remove those that have stalled.

TODO: Add stagnation tests. TODO: Update module docstrings, and clean up comments and code.

class pyneat.stagnation.**Stagnation**(*config*, *reporters*)
Bases: neat.config.DefaultClassConfig

Track the progress of species and remove those that have stalled.

classmethod parse_config(*param_dict*)
Parses the stagnation configuration parameters.

Config Parameters:

species_fitness_func (str): The function (mean, max) for aggregating the fitnesses of the members of each species.

max_stagnation (int): The maximum number of generations a species can stall for before being deemed stagnant.

species_elitism (int): The minimum number of species that should be retained.

TODO: Refactor *species_elitism* as *min_species*

Parameters **param_dict** – ...

Returns

...

Return type DefaultClassConfig

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 parameter if they were removed, in which case the highest-fitness species are spared - returns a list with stagnant species marked for removal.

Parameters

- **species_set** – ...
- **generation** – ...

Returns

...

Return type list

pyneat.visualise module

Modified visualisation functions from NEAT-Python example.

TODO: Document modifications.

pyneat.visualise.**draw_net**(*genome*, *view=False*, *filename=None*, *node_names=None*, *show_disabled=True*, *prune_unused=False*, *node_colors=None*, *fmt='svg'*)

Receives a genome and draws a neural network with arbitrary topology.

pyneat.visualise.**plot_species**(*statistics*, *view=False*, *filename='speciation.svg'*)

Visualizes speciation throughout evolution.

pyneat.visualise.**plot_spikes**(*spikes*, *view=False*, *filename=None*, *title=None*)

Plots the trains for a single spiking neuron.

pyneat.visualise.**plot_stats**(*statistics*, *ylog=False*, *view=False*, *filename='avg_fitness.svg'*)

Plots the population's average and best fitness.

Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W