NEvoPy

Talendar (Gabriel Nogueira)

Feb 22, 2021

CONTENTS:

NEvoPy is an open source neuroevolution framework for Python. It provides a simple and intuitive API for researchers and enthusiasts in general to quickly tackle machine learning problems using neuroevolutionary algorithms. *NEvoPy* is optimized for distributed computing and has compatibility with TensorFlow.

Currently, the neuroevolutionary algorithms implemented by *NEvoPy* are:

- **NEAT** (NeuroEvolution of Augmenting Topologies), a powerful method by Kenneth O. Stanley for evolving neural networks through complexification;
- the standard fixed-topology approach to neuroevolution, with support to TensorFlow and deep neural networks.

Note, though, that there's much more to come!

In addition to providing high-performance implementations of powerful neuroevolutionary algorithms, such as NEAT, *NEvoPy* also provides tools to help you more easily implement your own algorithms.

Neuroevolution, a form of artificial intelligence that uses evolutionary algorithms to generate artificial neural networks (ANNs), is one of the most interesting and unexplored fields of machine learning. It is a vast and expanding area of research that holds many promises for the future.

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

CHAPTER

ONE

NEVOPY OVERVIEW

1.1 Neuroevolution basics

Neuroevolution refers to the artificial evolution of neural networks using evolutionary algorithms. It's heavily inspired by the biological concept of Evolution and makes use of a population-based metaheuristic and mechanisms such as selection, reproduction, recombination and mutation to generate solutions.

A neural network is encoded, either directly or indirectly, by a *genome* (also called *genotype* or *individual*). The neural network encoded by a genome is its *phenotype*. We call a set of competing genomes a *population*. A genome's *fitness* is a measure of how well the genome performs in a given task. The goal of a neuroevolutionary algorithm is to evolve a population of genomes in order to produce genomes with a high fitness value.

The evolutionary process is divided into generations. In each generation, the population's genomes have their *fitness* calculated. Genomes with a higher fitness value have a greater chance of leaving offspring for the next generation. By favoring the reproduction of fitter genomes, the algorithm gradually increases the total fitness of the population.

If you are a beginner to neuroevolution and want to know more about this awesome area of research, here's a couple of papers and articles to get you started:

- Evolving artificial neural networks (great review paper);
- Evolving Neural Networks through Augmenting Topologies (the original paper of the NEAT algorithm);
- Neuroevolution: A different kind of deep learning (great introductory article about NE, by the creator of NEAT);
- Neuroevolution: A Primer On Evolving Artificial Neural Networks (great introductory article about NE);
- Welcoming the Era of Deep Neuroevolution (article about recent research by Uber AI Labs).

1.2 Populations and genomes in *NEvoPy*

In *NEvoPy*, a genome is an instance of a subclass that implements `BaseGenome`. Although each neuroevolutionary algorithm defines its own type of genome by implementing the `BaseGenome` class, all genomes are governed by the same general API. Note that in *NEvoPy's* API there isn't any distinction between a genome and the neural network it encodes. A genome, just like a neural network, must be capable of processing inputs based on its nodes and connections in order to produce an output. It also must be able to mutate and to generate offspring.

A population of genomes, on the other hand, is represented by the class `BasePopulation`. It defines a general API that all neuroevolutionary algorithms implemented by *NEvoPy* follow. Each algorithm makes its own implementation of that class - it's where the core of the evolutionary algorithm lives. The main method of the API is `BasePopulation.evolve()`, which triggers the evolutionary process in a population.

Most neuroevolutionary algorithms use a genetic algorithm to evolve the neural networks. What usually changes between different algorithms is how the genomes behave (how they reproduce, mutate and encode a neural network, for

example). With that in mind, *NEvoPy* implements a general-purpose genetic algorithm (see *GeneticPopulation*) that can be used as a base for most neuroevolutionary algorithms. This algorithm doesn't make strong assumptions about the genomes its evolving (it "doesn't care" if the genome encodes a network directly or indirectly, for example), so it can be used in a wide variety of scenarios. It also supports speciation.

NEvoPy currently implements the following neuroevolutionary algorithms:

- *Neuroevolution of Augmenting Topologies (NEAT)*;
- *Fixed-topology deep-neuroevolution*.

However, if you need more, implementing your own neuroevolutionary algorithm with *NEvoPy* is easy. Simply create a class that implements *BaseGenome* (thus defining how you want your genomes to behave) and let *GeneticPopulation* do the rest.

1.3 Evolving neural networks with *NEvoPy*

To evolve some neural networks with *NEvoPy*, the first thing you have to do is create a new population of genomes (represented by a class that implements *BasePopulation*). As an example, let's create a *NeatPopulation* (implements the NEAT algorithm):

```
import nevopy as ne
population = ne.neat.NeatPopulation(size=100,
                                    num_inputs=10,
                                    num_outputs=3)
```

The code above creates an instance of *NeatPopulation*, used to evolve instances of *NeatGenome* with the NEAT algorithm. The genomes are built to receive an array-like input of length 10 and to output the results as an array-like object of length 3. In *NEvoPy*, the inputs and outputs are, in most cases, instances of `numpy.ndarray` or `tensorflow.tensor`.

Now, we need to specify some routine for evaluating the population's genomes, i.e., for measuring the performance of each of the population's genomes on the task at hand in each *generation*. We call the measure of a genome's performance its *fitness* and the routine used to calculate this value a *fitness function*. Generally, a fitness function should look like this:

```
def fitness_function(genome):
    # (the genome's fitness is calculated here)
    # ...
    return fitness
```

Having created a population and defined a fitness function, we're ready to start the evolutionary process. We do that by calling the *evolve()* method:

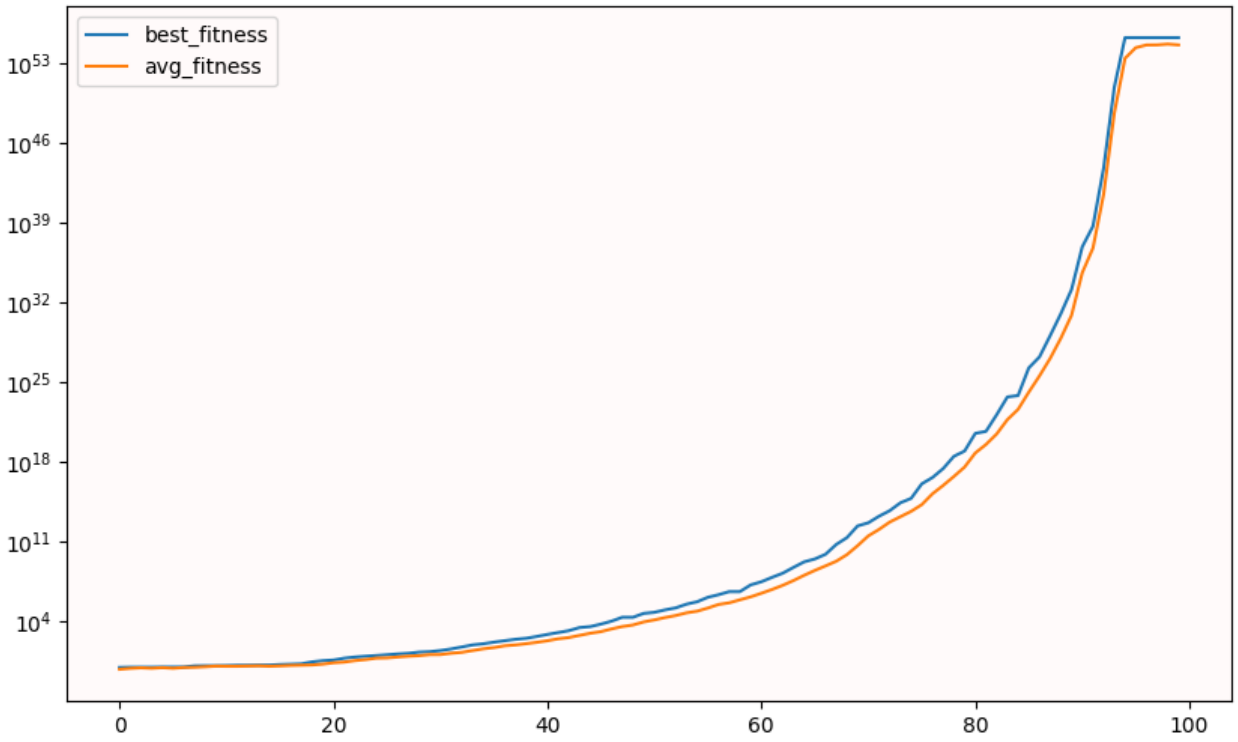
```
history = population.evolve(generations=100,
                            fitness_function=fitness_function)
```

The code above runs the NEAT algorithm for 100 generations. The *evolve()* method returns a *History* object, which contains useful statistics related to the evolutionary process. We can, for example, visualize the progression of the population's fitness by executing the following:

```
history.visualize()
```

Here is an example of a plot generated by this method:

The code bellow gets the fittest genome of the population, visualizes its topology and saves the genome:




```
best_genome = population.fittest()
best_genome.visualize()
best_genome.save("./best_genome.pkl")
```

For more information on how *NEvoPy* works, please take a look at our *docs*. For more practical examples, go to *here*.

CHAPTER

TWO

INSTALLATION

2.1 Install *NEvoPy* from PyPI using pip

To install the latest stable release of *NEvoPy* from *PyPI*, just run the following command:

```
$ pip install nevopy
```

In case you want to install *NEvoPy* with all the its most recent changes (might be unstable), you can directly install it from GitHub with *pip*:

```
$ pip install git+https://github.com/Talendar/nevopy
```

2.2 Install *NEvoPy* by cloning the project's GitHub repository

To install *NEvoPy* directly from its source code, first clone our GitHub repository by running the command:

```
$ git clone https://github.com/Talendar/nevopy
```

Then change directories to the *nevopy* folder and install the package using *pip*:

```
$ cd nevopy
$ pip install .
```

Alternatively, you can install *NEvoPy* by executing the *setup.py* script (not recommended):

```
$ cd nevopy
$ python3 setup.py install
```

CHAPTER

THREE

EXAMPLES

To learn the basics of *NEvoPy*, the XOR example is a good place to start. More examples can be found in the examples folder of the project's GitHub repo.

CHAPTER

FOUR

CALLBACKS

4.1 Introduction

A callback is a powerful tool to customize the behaviour of a population of genomes during the neuroevolutionary process. Examples include *FitnessEarlyStopping* to stop the evolution when a certain fitness has been achieved by the population, or *BestGenomeCheckpoint* to periodically save the best genome of a population during evolution. For a list with all the pre-implemented callbacks, take a look at *nevopy.callbacks*.

In this quick guide you'll learn what a *NEvoPy* callback is, what it can do, and how you can build your own.

4.2 *NEvoPy* callbacks overview

In *NEvoPy*, all callbacks subclass the *Callback* class and override a set of methods called at various stages of the evolutionary process. Callbacks are useful to get a view on internal states and statistics of a population and its genomes, as well as for modifying the behavior of the evolutionary algorithm being used.

You can pass a list of callbacks (as the keyword argument `callbacks`) to the *evolve()* method of your population.

4.3 An overview of callback methods

A callback implements one or more of the following methods (check each method's documentation for a list of accepted parameters):

- *on_generation_start*: called at the beginning of each new generation.
- *on_fitness_calculated*: called right after the fitness values of the population's genomes are calculated.
- *on_mass_extinction_counter_updated*: called right after the mass extinction counter is updated. The mass extinction counter counts how many generations have passed since the fitness of the population's best genomes improved.
- *on_mass_extinction_start*: called at the beginning of a mass extinction event. A mass extinction event occurs when the population's fitness haven't improved for a predefined number of generations. It results in the replacement of all the population's genomes (except for the fittest genome) for new randomly generated genomes.
- *on_reproduction_start*: called at the beginning of the reproductive process.
- *on_speciation_start*: called at the beginning of the speciation process. If the neuroevolutionary algorithm doesn't use speciation, this method isn't called at all.
- *on_generation_end*: called at the end of each generation.

- *on_evolution_end*: called when the evolutionary process ends.

4.4 Writing your own callbacks

To build your own callback, simply create a new class that has *Callback* as its parent class:

```
class MyCallback(Callback):
    def on_generation_start(self,
                            current_generation,
                            max_generations):
        print("This is printed at the start of every generation!")
        print(f"Starting generation {current_generation} of "
              f"{max_generations}.")
```

Then, just create a new instance of your callback and pass it to the *evolve()* of your population:

```
population.evolve(generations=100,
                  fitness_function=my_func,
                  callbacks=[MyCallback()])
```

CHAPTER

FIVE

PROCESSING

5.1 Introduction

In *NEvoPy*, most of the heavy processing involved in evolving a population of neural networks is managed by a *processing scheduler*. Processing schedulers allow the implementation of computation methods (like the use of serial or parallel processing) to be separated from the implementation of the neuroevolutionary algorithms. Examples of processing schedulers in *NEvoPy* include the *PoolProcessingScheduler*, that uses Python's `multiprocessing` module to implement parallel processing, and the *RayProcessingScheduler*, that uses the `ray` framework to implement distributed computing (it even allows you to use clusters!).

In this quick guide you'll learn what a *NEvoPy* processing scheduler is, what it can do, and how you can build your own. For a list with all the pre-implemented processing schedulers, take a look at *nevopy.processing*.

5.2 *NEvoPy* processing schedulers overview

In *NEvoPy*, all processing schedulers subclass the *ProcessingScheduler* class and override its *run()* method, which is responsible for processing a batch of items (*TProcItem*) and returning the corresponding results (*TProcResult*). The items and the results can be anything, but they usually are genomes and their fitnesses, respectively.

Processing schedulers might also be used to handle the computations associated with the reproductive process of a population.

5.3 Writing your own processing schedulers

To build your own processing scheduler, simply create a new class that has *ProcessingScheduler* as its parent class and implement the *run()* method:

```
class MyProcessingScheduler(ProcessingScheduler):

    def run(items, func):
        # ...
        return results
```

Then, just create a new instance of your new processing scheduler and pass it to the constructor of your population!

CHAPTER

SIX

NEVOPY PACKAGE

6.1 Subpackages

6.1.1 nevopy.fixed_topology package

Subpackages

nevopy.fixed_topology.layers package

Submodules

nevopy.fixed_topology.layers.base_layer module

Defines the abstract class that serves as a base for all the neural layers used by fixed-topology neuroevolutionary algorithms.

class nevopy.fixed_topology.layers.base_layer.**BaseLayer**(*config=None*, *input_shape=None*, *mutable=True*)

Bases: abc.ABC

Abstract base class that defines a neural network layer.

This abstract base class defines the general structure and behaviour of a fixed topology neural network layer in the context of neuroevolutionary algorithms.

Parameters

- **config** (*Optional[FixedTopologyConfig]*) – Settings being used in the current evolutionary session. If *None*, a config object must be assigned to the layer later on, before calling the methods that require it.
- **input_shape** (*Optional[Tuple[int, ..]]*) – Shape of the data that will be processed by the layer. If *None*, an input shape for the layer must be manually specified later or be inferred from an input sample.
- **mutable** (*Optional[bool]*) – Whether or not the layer can have its weights changed (mutation).

config

Settings being used in the current evolutionary session. If *None*, a config object hasn't been assigned to the layer yet.

Type Optional[FixedTopologyConfig]

mutable
Whether or not the layer can have its weights changed (mutation).

Type bool

abstract build(*input_shape*)
Builds the layer's weight and bias matrices.

If the layer has already been built, it will be built again (new weight and bias matrices will be generated).

Parameters **input_shape** (*Tuple[int, ..]*) – Tuple with the shape of the inputs that will be fed to the layer.

Raises **ValueError** – If the layer isn't compatible with the given input shape.

Return type None

abstract deep_copy()
Makes an exact/deep copy of the layer.

Return type *BaseLayer*

Returns An exact/deep copy of the layer, including its weights and biases.

property input_shape
The expected shape of an input for the layer.

Return type Optional[Tuple[int, ...]]

Returns A tuple with the layer's input shape or *None* if an input shape hasn't been specified yet.

classmethod load(*abs_path*)
Loads the layer from the given absolute path.

This method uses, by default, `pickle` to load the layer.

Parameters **abs_path** (*str*) – Absolute path of the saved ".pkl" file. If the given path doesn't end with the suffix ".pkl", it will be automatically added to it.

Return type *BaseLayer*

Returns The loaded layer.

abstract mate(*other*)
Mates two layers to produce a new layer (offspring).

Implements the sexual reproduction between a pair of layers. The new layer inherits information from both parents (not necessarily in an equal proportion)

Parameters **other** (*Any*) – The second layer . If it's not compatible for mating with the current layer (*self*), an exception will be raised.

Return type *BaseLayer*

Returns A new layer (the offspring born from the sexual reproduction between the current layer and the layer passed as argument. If the layer is immutable, `other` is expected to be equal to `self`, so a deep copy (*BaseLayer.deep_copy()*) of the layer is returned.

Raises ***IncompatibleLayersError*** – If the layer passed as argument to `other` is incompatible with the current layer (*self*).

abstract mutate_weights()
Randomly mutates the weights of the layer's connections.

If the layer is immutable, this method doesn't do anything.

Return type None

abstract process(*x*)
Feeds the given input(s) to the layer.

This is where the layer's logic lives. If the layer hasn't been built yet, it will be automatically built using the given input shape.

Parameters **x** (*Any*) – The input(s) to be fed to the layer. Usually a *NumPy ndarray* or a *TensorFlow tensor*.

Return type Any

Returns The output of the layer. Usually a *NumPy ndarray* or a *TensorFlow tensor*.

Raises ***InvalidInputError*** – If the shape of x doesn't match the input shape expected by the layer.

abstract random_copy()
Makes a random copy of the layer.

Return type *BaseLayer*

Returns A new layer with the same topology of the current layer, but with newly initialized weights and biases. If the layer is immutable, a deep copy (*BaseLayer.deep_copy()*) of the layer is returned instead.

save(*abs_path*)
Saves the layer on the absolute path provided.

This method uses, by default, pickle to save the layer.

Parameters **abs_path** (*str*) – Absolute path of the saving file. If the given path doesn't end with the suffix ".pkl", it will be automatically added to it.

Return type None

abstract property weights
A list with the layer's weight matrices as numpy arrays.

For most layers, it's a tuple containing a weight matrix and a bias vector.

Return type List[ndarray]

exception nevopy.fixed_topology.layers.base_layer.**IncompatibleLayersError**
Bases: Exception

Indicates that an attempt has been made to mate (sexual reproduction) two incompatible layers.

nevopy.fixed_topology.layers.mating module

Implements some mating (sexual reproduction) functions that can be used to generate a new neural network layer from two parent layers.

nevopy.fixed_topology.layers.mating.**check_weights_compatibility**(*weight_list1*, *weight_list2*)
Checks the mating compatibility between two lists of weight matrices.

Raises ***IncompatibleLayersError*** – If one or more weight matrices in one of the lists don't have the same shape as the corresponding weight matrices in the other list.

nevopy.fixed_topology.layers.mating.**exchange_units_mating**(*layer1*, *layer2*)
Mates (sexual reproduction) two neural layers by exchanging units.

The term "unit" means different things depending on the type of the layers. For a Conv2D layer, for instance, an unit is a filter (kernel). For a Dense layer, on the other hand, an unit is a neuron (including the weights of its connections). Bias terms are also considered "units".

Generally, we can define an unit as being whatever you get when indexing a weight matrix by its last shape. Given a layer *L*, an unit of its weight matrix at index *w* is given by *L.weights[w][..., i]*, where *i* is the index of the unit (*i* is in the interval *[0, w.shape[-1][)*.

Parameters

- **layer1** (BaseLayer) – An instance of a subclass of *BaseLayer*.
- **layer2** (BaseLayer) – An instance of a subclass of *BaseLayer*.

Return type *BaseLayer*

Returns A new layer that inherits information from both parents.

Raises ***IncompatibleLayersError*** – If the weight matrices of the two given layers are not of the same shape (i.e., the layers are not compatible for mating).

nevopy.fixed_topology.layers.mating.**exchange_weights_mating**(*layer1*, *layer2*)

Mates (sexual reproduction) two neural layers by exchanging weights.

Each of the new layer's weights is randomly inherited, with equal chance, from one of the parent layers.

Parameters

- **layer1** (BaseLayer) – An instance of a subclass of *BaseLayer*.
- **layer2** (BaseLayer) – An instance of a subclass of *BaseLayer*.

Return type *BaseLayer*

Returns A new layer that randomly inherits its individual weights from its parent layers.

Raises ***IncompatibleLayersError*** – If the weight matrices of the two given layers are not of the same shape (i.e., the layers are not compatible for mating).

nevopy.fixed_topology.layers.mating.**weights_avg_mating**(*layer1*, *layer2*)

Mates (sexual reproduction) two layers by averaging their weights.

Each of the new layer's weight is the simple average (sum and divide by 2) of the parent layers weights.

Parameters

- **layer1** (BaseLayer) – An instance of a subclass of *BaseLayer*.
- **layer2** (BaseLayer) – An instance of a subclass of *BaseLayer*.

Return type *BaseLayer*

Returns A new layer whose weights are the simple average of the parent layers weights.

Raises ***IncompatibleLayersError*** – If the weight matrices of the two given layers are not of the same shape (i.e., the layers are not compatible for mating).

nevopy.fixed_topology.layers.tf_layers module

Implements subclasses of *BaseLayer* that wrap TensorFlow layers.

class nevopy.fixed_topology.layers.tf_layers.**TFConv2DLayer**(*filters*, *kernel_size*, *strides=(1, 1)*, *padding='valid'*, *activation='relu'*, *mating_func=<function exchange_units_mating>*, *config=None*, *input_shape=None*, *mutable=True*, ***tf_kwargs*)

Bases: *nevopy.fixed_topology.layers.tf_layers.TensorFlowLayer*

Wraps a *TensorFlow* 2D convolution layer.

This is a simple wrapper for tf.keras.layers.Conv2D.

class nevopy.fixed_topology.layers.tf_layers.**TFDenseLayer**(*units*, *activation=None*, *mating_func=<function exchange_weights_mating>*, *config=None*, *input_shape=None*, *mutable=True*, ***tf_kwargs*)

Bases: *nevopy.fixed_topology.layers.tf_layers.TensorFlowLayer*

Wraps a *TensorFlow* dense layer.

This is a simple wrapper for tf.keras.layers.Dense.

class nevopy.fixed_topology.layers.tf_layers.**TFFlattenLayer**(*mating_func=None*, *config=None*, *input_shape=None*, *mutable=False*, ***tf_kwargs*)

Bases: *nevopy.fixed_topology.layers.tf_layers.TensorFlowLayer*

Wraps a *TensorFlow* flatten layer.

This is a simple wrapper for tf.keras.layers.Flatten.

class nevopy.fixed_topology.layers.tf_layers.**TFMaxPool2DLayer**(*pool_size=(2, 2)*, *strides=None*, *padding='valid'*, *mating_func=None*, *config=None*, *input_shape=None*, *mutable=False*, ***tf_kwargs*)

Bases: *nevopy.fixed_topology.layers.tf_layers.TensorFlowLayer*

Wraps a *TensorFlow* 2D max pooling layer.

This is a simple wrapper for tf.keras.layers.MaxPool2D.

class nevopy.fixed_topology.layers.tf_layers.**TensorFlowLayer**(*layer_type*, *mating_func=<function exchange_units_mating>*, *config=None*, *input_shape=None*, *mutable=True*, ***tf_kwargs*)

Bases: *nevopy.fixed_topology.layers.base_layer.BaseLayer*

Wraps a *TensorFlow* layer.

This class wraps a *TensorFlow* layer, making it compatible with *NEvoPy's* neuroevolutionary algorithms. It handles the mutation and reproduction of the *TensorFlow* layer.

In most cases, there is no need to create subclasses of this class. Doing that to frequently used types of layers, however, may be desirable, since it makes using those types of layers easier (see *TFConv2DLayer* and *TFDenseLayer* as examples).

When inheriting this class, you'll usually do something like this:

```
class MyTFLayer(TensorFlowLayer):
    def __init__(self,
                 arg1, arg2,
                 activation="relu",
                 mating_func=mating.exchange_units_mating,
                 config=None,
                 input_shape=None,
                 mutable=True,
                 **tf_kwargs: Dict[str, Any]):
        super().__init__(
            layer_type=tf.keras.layers.SomeKerasLayer,
            **{k: v for k, v in locals().items()
               if k not in ["self", "tf_kwargs", "__class__"]},
            **tf_kwargs,
        )
```

Parameters

- **layer_type** (*Union[str, Type[tf.keras.layers.Layer]]*) – A reference to the *TensorFlow's* class that represents the layer (`tf.keras.layers.Dense`, for example). If it's a string, the appropriate type will be inferred (note that it must be listed in *TensorFlowLayer.KERAS_LAYERS*).
- **mating_func** (*Optional[Callable[[BaseLayer, BaseLayer], BaseLayer]]*) – Function that mates (sexual reproduction) two layers. It should receive two layers as input and return a new layer (the offspring). You can use one of the pre-built mating functions (see *fixed_topology.layers.mating*) or implement your own. If the layer is immutable, this parameter should receive *None* as argument.
- **config** (*Optional[FixedTopologyConfig]*) – Settings being used in the current evolutionary session. If *None*, a config object must be assigned to the layer later on, before calling the methods that require it.
- **input_shape** (*Optional[Tuple[int, ..]]*) – Shape of the data that will be processed by the layer. If *None*, an input shape for the layer must be manually specified later or be inferred from an input sample.
- **mutable** (*Optional[bool]*) – Whether or not the layer can have its weights changed (mutation).

- ****tf_kwargs** – Named arguments to be passed to the constructor of the *TensorFlow* layer.

KERAS_LAYERS = {'conv2D': <class 'tensorflow.python.keras.layers.convolutional.Conv2D'

build(*input_shape*)
Wrapper for `tf.keras.layers.Layer.build()`.

Return type `None`

deep_copy()
Makes an exact/deep copy of the layer.

Return type *TensorFlowLayer*

Returns An exact/deep copy of the layer, including its weights and biases.

mate(*other*)
Mates two layers to produce a new layer (offspring).

Implements the sexual reproduction between a pair of layers. The new layer inherits information from both parents (not necessarily in an equal proportion)

Parameters **other** (*Any*) – The second layer . If it's not compatible for mating with the current layer (*self*), an exception will be raised.

Return type *TensorFlowLayer*

Returns A new layer (the offspring born from the sexual reproduction between the current layer and the layer passed as argument. If the layer is immutable, `other` is expected to be equal to `self`, so a deep copy (*BaseLayer.deep_copy()*) of the layer is returned.

Raises ***IncompatibleLayersError*** – If the layer passed as argument to `other` is incompatible with the current layer (*self*).

mutate_weights(*_test_info=None*)
Randomly mutates the weights of the layer's connections.

Each weight has a chance to be perturbed by a predefined amount or to be reset. The probabilities are obtained from the settings of the current evolutionary session.

If the layer is immutable, nothing happens (the layer's weights remain unchanged).

Return type `None`

process(*x*)
Feeds the given input(s) to the layer.

This is where the layer's logic lives. If the layer hasn't been built yet, it will be automatically built using the given input shape.

Parameters **x** (*Any*) – The input(s) to be fed to the layer. Usually a *NumPy ndarray* or a *TensorFlow tensor*.

Return type `Tensor`

Returns The output of the layer. Usually a *NumPy ndarray* or a *TensorFlow tensor*.

Raises ***InvalidInputError*** – If the shape of `x` doesn't match the input shape expected by the layer.

random_copy()
Makes a random copy of the layer.

Return type *TensorFlowLayer*

Returns A new layer with the same topology of the current layer, but with newly initialized weights and biases. If the layer is immutable, a deep copy (*BaseLayer.deep_copy()*) of the layer is returned instead.

property tf_layer
The tf.keras.layers.Layer used internally.

Return type Layer

property weights
The current weight matrices of the layer.

Wrapper for `tf.keras.layers.Layer.get_weights()`.

The weights of a layer represent the state of the layer. This property returns the weight values associated with this layer as a list of Numpy arrays. In most cases, it's a list containing the weights of the layer's connections and the bias values (one for each neuron, generally).

Return type List[ndarray]

Module contents

Neural network layers to be used with *NEvoPy's* fixed-topology neuroevolutionary algorithms.

Submodules

nevopy.fixed_topology.genomes module

Implements genomes (subclasses of *BaseGenome*) that encode neural networks with a fixed topology.

class nevopy.fixed_topology.genomes.**FixedTopologyGenome**(*layers*, *config=None*, *input_shape=None*)

Bases: *nevopy.base_genome.BaseGenome*

Genome that encodes a fixed-topology multilayer neural network.

This genome directly encodes a multilayer neural network with fixed topology. The network is defined by its layers (instances of a subclass of *BaseLayer*), specified during the genome's creation.

Note: The *config* objects of individual layers are forcefully replaced by the *config* object of the genome when its assigned with a new one!

Parameters

- **layers** (*List[BaseLayer]*) – List with the layers of the network (instances of a subclass of *BaseLayer*). It's not required to set the input shape of each individual layer. If the input shapes are not set, they will be automatically set when a call to *process()* is made. There is no need to pass the *config* object to the layers (it's done automatically when this class is instantiated).
- **config** (*Optional[GeneticAlgorithmConfig]*) – Settings of the current evolutionary session. If *None*, a config object must be assigned to this genome latter.
- **input_shape** (*Optional[Tuple[int, ..]]*) – Shape of the inputs that will be fed to the genome. If a value is specified, the genome's layers are built (they have their weights initialized). If *None*, an input shape will be inferred later when an input is fed to the genome (note, however, that the weights won't be initialized until it occurs).

layers
List with the layers of the network (instances of a subclass of *BaseLayer*).

Type List[*BaseLayer*]

property config
Settings of the current evolutionary session.

If *None*, a config object hasn't been assigned to this genome yet.

Return type Optional[*GeneticAlgorithmConfig*]

deep_copy()
Makes an exact/deep copy of the genome.

Return type *FixedTopologyGenome*

Returns An exact/deep copy of the genome. It has the same topology and connections weights of the original genome.

distance(*other*)
Calculates the distance between the two genomes.

The distance is calculated based on the euclidean distance (the L2 norm of the difference) between correspondent weight matrices of the genomes layers.

Parameters **other** (FixedTopologyGenome) – The other fixed-topology genome.

Return type float

Returns A float representing the distance between the two genomes. The lower the distance, the more similar the two genomes are.

property input_shape
The input shape expected by the genome's input layer.

Return type Optional[Tuple[int, ...]]

mate(*other*)
Mates two genomes to produce a new genome (offspring).

Implements the sexual reproduction between a pair of genomes. The new genome inherits information from both parents.

Currently available mating modes for individual layers:

- *mating.exchange_weights_mating()*;
- *mating.exchange_units_mating()*;
- *mating.weights_avg_mating()*.

The mating mode of a layer is specified during its instantiation.

Parameters **other** (*Any*) – The second genome . If it's not compatible for mating with the current genome (*self*), an exception will be raised.

Return type *FixedTopologyGenome*

Returns A new genome (the offspring born from the sexual reproduction between the current genome and the genome passed as argument).

Raises ***IncompatibleGenomesError*** – If the genome passed as argument to other is incompatible with the current genome (*self*).

mutate_weights()
Randomly mutates the weights of the genome's connections.

Return type None

process(*x*)
Feeds the given input to the neural network encoded by the genome.

Parameters **x** (*Any*) – The input(s) to be fed to the neural network encoded by the genome. Usually a *NumPy ndarray* or a *TensorFlow tensor*.

Return type Any

Returns The output of the network. Usually a *NumPy ndarray* or a *TensorFlow tensor*.

Raises ***InvalidInputError*** – If the shape of X doesn't match the input shape expected by the network.

random_copy()
Makes a deep copy of the genome, but with random weights.

Return type *FixedTopologyGenome*

Returns A deep copy of the genome with the same topology of the original genome, but random connections weights.

reset()
This method doesn't do anything.

In this implementation, the default fixed topology networks do not need to reset any of its internal states before the start of a new generation.

Return type None

visualize(*show=True*, *to_file='genome.png'*, ***kwargs*)
Utility method for visualizing the genome's neural network.

This currently only works with genomes that use TensorFlow layers.

Todo: Make it possible to visualize neurons and connections.

---

show
Whether to show the generated image or not.
Type bool

to_file
Path in which the image file will be saved to.
Type str

****kwargs**
Optional named arguments to be passed to `tensorflow.keras.utils.plot_model()`.

Return type Image

Returns The generated `PIL.Image.Image` object.

Module contents

Imports core names of *nevopy.fixed_topology*.

6.1.2 nevopy.genetic_algorithm package

Submodules

nevopy.genetic_algorithm.config module

This module implements the `FixedTopologyConfig` class, used to handle the settings of *NEvoPy's* fixed-topology neuroevolutionary algorithms.

class nevopy.genetic_algorithm.config.**GeneticAlgorithmConfig**(*file_pathname=None*, ***kwargs*)

Bases: `object`

Stores the settings to be used by *GeneticPopulation*.

Individual configurations can be ignored (default values will be used), set in the arguments of this class constructor or written in a file (pathname passed as an argument).

Some parameters/attributes related to mutation chances expects a tuple with two floats, indicating the minimum and the maximum chance of the mutation occurring. A value within the given interval is chosen based on the "mass extinction factor" (mutation chances increases as the number of consecutive generations in which the population has shown no improvement increases). If you want a fixed mutation chance, just place the same value on both positions of the tuple.

Todo:

- Implementation: loading settings from a config file.
- Specify the config file organization in the docs.

Parameters

- **file_pathname** (*Optional[str]*) – The pathname of a file from where the settings should be loaded.
- ****kwargs** – Accepts any of the attributes listed for this class. When the value of an attribute isn't passed as argument, a default value is used. The default values are defined in *GeneticAlgorithmConfig.ATTRIBUTES*.

mutation_chance
Chance for a mutation to occur in a new-born genome.

Type Tuple[float, float]

weight_mutation_chance
Chance of each individual connection weight of a newborn genome being perturbed during mutation.

Type Tuple[float, float]

weight_perturbation_pc
Maximum absolute percentage of a weight's value that can be added to it during mutation. When a connection weight is being mutated, it has a chance of being perturbed. This can me summarized as follows (*p* is the weight perturbation percentage): *current weight <- current weight * (1 + random[-p, p])*.

Type Tuple[float, float]

weight_reset_chance
Chance, during mutation, for a weight to have its value reset (in which case a new random value is assigned to it).

Type Tuple[float, float]

new_weight_interval
When a weight is reset, it will be assigned with a random value in this interval.

Type Tuple[float, float]

weak_genomes_removal_pc
Percentage of the weakest genomes (those with the lowest fitness) to be removed before reproduction occurs.

Type float

mating_chance
Chance of a genome reproducing sexually, i.e., by mating / crossing-over with another genome. Decreasing this value increases the chance of a genome reproducing asexually, through binary fission (copy + mutation).

Type float

mating_mode
How the exchange of genetic material is supposed to happen during a sexual reproduction between two genomes. Options: "weights_mating" and "exchange_layers" (the new genome inherits full layers from its parents).

Type str

rank_prob_dist_coefficient
Coefficient α used to calculate the probability distribution used to select genomes for reproduction. Basically, the value of this constant can be interpreted as follows: the genome with the highest fitness has $\times\alpha$ more chance of being selected for reproduction than the second best genome, which, in turn, has $\times\alpha$ more chance of being selected than the third best genome, and so forth. This approach to reproduction is called rank-based selection.

Type float

predatism_chance
Chance of a newborn genome being "predated", in which case its replaced by a new randomly generated genome. This increases the genetic variability in the population.

Type float

species_distance_threshold
Minimum distance between two genomes for them to be considered as being of the same species. A lower threshold will make new species easier to appear, increasing the number of species throughout the evolutionary process.

Type float

species_elitism_threshold
Species with a number of members superior to this threshold will have one or more of their fittest members copied unchanged to the next generation.

Type int

elitism_pc
Percentage of the genomes of a big enough species to be copied unchanged to the next generation. The number of copied genomes is equal to *max(1, ceil(species_size * elitism_pc))*.

Type float

species_no_improvement_limit
If a species doesn't show improvement in its best fitness for this amount of generations, it will be removed from the species' list of the population.

Type int

mass_extinction_threshold
If the population's fitness doesn't improve for this amount of generations, the whole population, with the exception of its fittest genome, will be extinct/deleted and replaced by new randomly generated genomes. Here the fitness of a population in a given generation is considered to be equal to the fitness of its fittest genome in that generation. As the number of generations without improvements increases, the mutations chances (as specified in the settings) also increase. This simulates the increase of the evolutionary pressure acting on the population.

Type int

maex_improvement_threshold_pc
It's considered that the fitness of a population improved if, and only if, the population's fitness had an increase equivalent to this percentage. As an example, suppose that the fitness f_g of a population on generation g is 100 and that this parameter is set to *0.05* (*5%*). The fitness f_{g+1} of the population in the next generation (g + *1*) is considered to have improved if, and only if, $f_{g+1} \geq 1.05 \cdot f_g = 105$.

Type float

ATTRIBUTES = {'elitism_pc': 0.03, 'interspecies_mating_chance': 0.05, 'maex_improveme
Attributes supported by the class and their default values. Each attribute can passed as a kwarg in the class' constructor or be specified in a config file. Attributes not specified will be initialized with a default value.

MAEX_KEYS = {'mutation_chance', 'weight_mutation_chance', 'weight_perturbation_pc', 'w
Name of the attributes whose values change according to the mass extinction counter (type: Tuple[float, float]).

property maex_counter
Returns the current value stored in the config's mass extinction counter.

Return type int

update_mass_extinction(*maex_counter*)
Updates the mutation chances based on the current value of the mass extinction counter (generations without improvement).

Parameters **maex_counter** (*int*) – Current value of the mass extinction counter (generations without improvement).

Return type None

nevopy.genetic_algorithm.population module

Implements a generalizable genetic algorithm that can be used by different neuroevolution algorithms.

class nevopy.genetic_algorithm.population.**DefaultSpecies**(*creation_gen*, *members=None*)

Bases: object

Represents a species.

In the context of a genetic algorithm, a species is a set of similar (to some extent) genomes that can mate in order to generate offspring.

Parameters

- **creation_gen** (*int*) – Number of the generation in which the species is being created.
- **members** (*Optional[List[BaseGenome]]*) – Initial members of the species.

representative
Genome used to represent the species.

Type Optional[*BaseGenome*]

members
List with the genomes that belong to the species.

Type List[*BaseGenome*]

last_improvement
Generation in which the species last showed improvement of its fitness. The species fitness in a given generation is equal to the fitness of the species fittest genome on that generation.

Type int

best_fitness
The last calculated fitness of the species fittest genome.

Type Optional[float]

avg_fitness()
Returns the average fitness of the species genomes.

Return type float

compatibility(*genome*)
Returns a float indicating the compatibility of the given genome with the species.

Return type float

fittest()
Returns the fittest member of the species.

Return type *BaseGenome*

update_representative()
Chooses a new representative for the species.

This implementation follows NEAT, so a random member of the species is chosen as its representative.

Return type None

class nevopy.genetic_algorithm.population.**GeneticPopulation**(*size*, *base_genome*, *config=None*, *processing_scheduler=None*, *speciation=False*)

Bases: *nevopy.base_population.BasePopulation*

Implementation of a generalizable genetic algorithm.

This class implements a generalizable genetic algorithm that can be used by different neuroevolution algorithms. The algorithm is used to evolve a population of genomes (instances of a subclass of *BaseGenome*).

This class does not make strong assumptions about the type of genome it is dealing with, so it does not take into account the type of encoding the genome uses or how it processes input. This allows the implemented algorithm to be used in a wide range of scenarios.

The implemented genetic algorithm uses (optionally) a speciation scheme similar to the one used by the NEAT algorithm [SM02]. The computation of the distance between the genomes, however, is not implemented here, but on the subclass that implements class:.*BaseGenome*.

When subclassing this class, you probably won't need to override the *GeneticPopulation.evolve()* method, which contains the main loop of the genetic algorithm.

To better understand the default behaviour of the algorithm, it's recommended to read the docs of the methods *speciate()* and *reproduction()*.

Example

Example using *FixedTopologyGenome* as the base genome type:

```
def fitness_func(genome):
    """
    Function that takes a genome as input and returns the genome's
    fitness (a float) as output.
    """
    # ...


# Genome that's gonna serve as a model for your population:
base_genome = FixedTopologyGenome(
    layers=[TFDenseLayer(32, activation="relu"),
            TFDenseLayer(1, activation="sigmoid")],
    input_shape=my_input_shape,  # shape of your input samples
)

# Creating and evolving a population:
population = GeneticPopulation(size=100,
                               base_genome=base_genome)
history = population.evolve(generations=100,
                            fitness_function=fitness_func)

# Visualizing the evolution of the population's fitness:
history.visualize()

# Retrieving and visualizing the fittest genome of the population:
best_genome = population.fittest()
best_genome.visualize()
```

Parameters

- **size** (*int*) – Number of genomes (constant) in the population.
- **base_genome** (BaseGenome) – Instance of a subclass of *BaseGenome* that will serve as a model/base for all the population's genomes.
- **config** (*Optional[GeneticConfig]*) – The settings of the evolutionary process. If *None*, the default settings will be used.
- **processing_scheduler** (*Optional[ProcessingScheduler]*) – Processing scheduler to be used by the population. If *None*, a new instance of `RayProcessingScheduler` will be used as scheduler.
- **speciation** (*bool*) – Whether the genetic algorithm used to evolve the genomes should use speciation or not.

DEFAULT_SCHEDULER
alias of *nevopy.processing.ray_processing.RayProcessingScheduler*

property config
Config object that stores the settings used by the population.

evolve(*generations*, *fitness_function*, *callbacks=None*, *verbose=2*, ***kwargs*)
Evolves the population using a genetic algorithm.

Main method of this class. It contains the main loop of the genetic algorithm used to evolve the population of genomes.

Parameters

- **generations** (*int*) – Number of generations for the algorithm to run. A generation is completed when all the population's genomes have been processed and reproduction and speciation have occurred.
- **fitness_function** (*Callable[[BaseGenome], float]*) – Fitness function to be used to evaluate the fitness of individual genomes. It must receive a genome as input and produce a float (the genome's fitness) as output.
- **callbacks** (*Optional[List[Callback]]*) – List with instances of *Callback* that will be called during the evolutionary session. By default, a *History* callback is always included in the list. A *CompleteStdOutLogger* or a *SimpleStdOutLogger* might also be included, depending on the value passed to the `verbose` param.
- **verbose** (*int*) – Verbose level (logging on stdout). Options: 0 (no verbose), 1 (light verbose) and 2 (heavy verbose).

Return type *History*

Returns A *History* object containing useful information recorded during the evolutionary process.

static generate_offspring(*args*)
Given one or two genomes (parents), generates a new genome.

Parameters **args** (*Tuple[BaseGenome, Optional[BaseGenome], bool]*) – Tuple containing a genome in its first index, another genome or *None* in its second index and a bool in its third index. The bool indicates whether predatism will occur or not. If it's *True*, then the new genome will be randomly generated. If the second index is another genome, then the new genome will be generated by mating the two given genomes (sexual reproduction). If its *None*, the new genome will be a mutated copy (asexual reproduction / binary fission) of the genome in the first index.

Return type *BaseGenome*

Returns A new genome.

mass_extinction(*best_genome*)
All the genomes in the population (except for the best genome) are replaced by new random genomes (random copies of the population's base genome).

Return type None

reproduction()
Handles the reproduction of the population's genomes.

First, the fittest genomes of each species with more than a pre-defined number of individuals are selected to be copied unchanged to the next generation (elitism). Next, the least fit genomes of each species are discarded (reverse elitism). After that, the number of descendants of each species is calculated. The number of offspring assigned to each species is proportional to the average fitness of the species (*roulette wheel selection*). Finally, the reproduction of individuals of the same species (and, on rare occasions, between genomes of different species as well) occurs.

Genomes with a higher fitness have a higher chance of leaving offspring. Within a species, the chance of a genome reproducing is given by the position it occupies in the species fitness rank (*rank-based selection*). This means that the reproduction chance of a genome is not directly calculated from the genome's fitness, but rather from how well positioned is the genome in the fitness rank.

Some of the behaviour described above follows the original description of the NEAT algorithm [SM02].

Newborn genomes have a chance of being "eaten by a predator", in which case they are replaced by new randomly generated genomes. This technique is called *predatism*.

Return type int

Returns Number of preys (individuals replaced by a random genome).

speciate(*current_generation*)
Divides the population's genomes into species.

The algorithm follows the speciation scheme of the NEAT algorithm [SM02]:

"Each existing species is represented by a random genome inside the species from the previous generation. A given genome g in the current generation is placed in the first species in which g is compatible with the representative genome of that species. This way, species do not overlap. If g is not compatible with any existing species, a new species is created with g as its representative." - [SM02]

The degree of compatibility between two genomes is given by their distance, calculated by the *BaseGenome.distance()* method. The lower the distance the more compatible two genomes are. Two genomes are considered compatible if their distance is lower than a pre-defined number (*GeneticAlgorithmConfig.species_distance_threshold*).

Species that haven't improved their fitness for a pre-defined number of generations are extinct, i.e., they are removed from the population and aren't considered for the speciation process.

Return type None

Module contents

Imports core names of `nevopy.genetic_algorithms`.

6.1.3 nevopy.neat package

Submodules

nevopy.neat.config module

This module implements the *NeatConfig* class, used to handle the settings of the NEAT algorithm.

class nevopy.neat.config.**NeatConfig**(*file_pathname=None*, ***kwargs*)
Bases: *nevopy.genetic_algorithm.config.GeneticAlgorithmConfig*

Stores the settings of the NEAT algorithm.

Individual configurations can be ignored (default values will be used), set in the arguments of this class constructor or written in a file (pathname passed as an argument).

Some parameters/attributes related to mutation chances expects a tuple with two floats, indicating the minimum and the maximum chance of the mutation occurring. A value within the given interval is chosen based on the "mass extinction factor" (mutation chances increases as the number of consecutive generations in which the population has shown no improvement increases). If you want a fixed mutation chance, just place the same value on both positions of the tuple.

Parameters

- **file_pathname** (*Optional[str]*) – The pathname of a file from where the settings should be loaded.
- ****kwargs** – Accepts any of the attributes listed for this class. When the value of an attribute isn't passed as argument, a default value is used. The default values are defined in *NeatConfig.ATTRIBUTES*.

out_nodes_activation
Activation function to be used by the output nodes of the networks. It should receive a float as input and return a float (the resulting activation) as output.

Type Callable[[float], float]

hidden_nodes_activation
Activation function to be used by the hidden nodes of the networks. It should receive a float as input and return a float (the resulting activation) as output.

Type Callable[[float], float]

bias_value
Constant activation value to be used by the bias nodes. If *None*, bias nodes won't be used.

Type Optional[float]

weak_genomes_removal_pc
Percentage of the least fit individuals to be deleted from the population before the reproduction step.

Type float

weight_mutation_chance
Tuple containing, respectively, the minimum and maximum chance of mutating a connection gene.

Type Tuple[float, float]

new_node_mutation_chance
Tuple containing, respectively, the minimum and maximum chance of a new hidden node being added to a newly born genome.

Type Tuple[float, float]

new_connection_mutation_chance
Tuple containing, respectively, the minimum and maximum chance of a new connection being added to a newly born genome.

Type Tuple[float, float]

enable_connection_mutation_chance
Tuple containing, respectively, the minimum and maximum chance of enabling a disabled connection in a newly born genome.

Type Tuple[float, float]

disable_inherited_connection_chance
During a sexual reproduction between two genomes, this constant specifies the chance of a connection in the newly born genome being disabled if it's disabled on at least one of the parent genomes.

Type float

mating_chance
Chance of a genome reproducing sexually, i.e., by mating / crossing-over with another genome. Decreasing this value increases the chance of a genome reproducing asexually, through binary fission (copy + mutation).

Type float

interspecies_mating_chance
Chance for a sexual reproduction (mating / cross-over) to be between genomes of different species.

Type float

rank_prob_dist_coefficient
Coefficient α used to calculate the probability distribution used to select genomes for reproduction. Basically, the value of this constant can be interpreted as follows: the genome, within a species, with the highest fitness has $\times\alpha$ more chance of being selected for reproduction than the second best genome, which, in turn, has $\times\alpha$ more chance of being selected than the third best genome, and so forth. This approach to reproduction is called rank-based selection. Note that this is applied to individuals within the same species.

Type float

weight_perturbation_pc
Tuple containing, respectively, the minimum and maximum value for the maximum absolute percentage of the perturbation value of the weights. When a connection gene is being mutated, it has a chance of having a value (the perturbation) added to its weight. This can me summarized as follows (*p* is the weight perturbation percentage): *current weight <- current weight * (1 + random[-p, p])*.

Type Tuple[float, float]

weight_reset_chance
Tuple containing, respectively, the minimum and maximum chance of resetting a connection's weight during the mutation of a connection gene. The reset connection is assigned a new random weight.

Type Tuple[float, float]

new_weight_interval
Interval from which the value of a new random connection weight will be picked from.

Type Tuple[float, float]

mass_extinction_threshold
If the population's fitness doesn't improve for this amount of generations, the whole population, with the exception of its most fit genome, will be extinct/deleted and replaced by new randomly generated genomes. Here the fitness of a population in a given generation is considered to be equal to the fitness of the population's most fit genome in that generation. As the number of generations without improvements increases, the mutations chances (as specified in the settings) also increase. This simulates the increase of the evolutionary pressure acting on the population.

Type int

maex_improvement_threshold_pc
It's considered that the fitness of a population improved if, and only if, the population's fitness had an increase equivalent to this percentage. As an example, suppose that the fitness f_g of a population on generation g is 100 and that this parameter is set to *0.05* (*5%*). The fitness f_{g+1} of the population in the next generation (*g + 1*) is considered to have improved if, and only if, $f_{g+1} \geq 1.05 \cdot f_g = 105$.

Type float

infanticide_output_nodes
If *True*, newborn genomes with no enabled connections incoming to one or more output nodes will be deleted and replaced by a new randomly generated genome. Note that the term "infanticide" is being used here without any political or cultural connotation. It's used because it is the word that best describe the phenomenon at hand and is widely used in the scientific field of zoology (see this article).

Type bool

infanticide_input_nodes
If *True*, newborn genomes with no enabled connections leaving one or more input nodes will be deleted and replaced by a new randomly generated genome. Note that the term "infanticide" is being used here without any political or cultural connotation. It's used because it is the word that best describe the phenomenon at hand and is widely used in the scientific field of zoology (see this article).

Type bool

random_genome_bonus_nodes
Let *h_bonus* be the argument passed to this parameter and *h_max* the maximum number of hidden nodes within individuals of the population. When a random genome is created to replace one of the population's genomes, the number of hidden nodes in it will be a random number picked from the interval *[0, h_max + h_bonus]*.

Type int

random_genome_bonus_connections
The same as `NeatConfig.random_genome_max_bonus_hnodes`, except it refers to the number of connections involving hidden nodes in the new randomly generated genome.

Type int

excess_genes_coefficient
Used in the formula that calculates the distance between two genomes. It's the c_1 coefficient in (6.1).

Type float

disjoint_genes_coefficient
Used in the formula that calculates the distance between two genomes. It's the c_2 coefficient in (6.1).

Type float

weight_difference_coefficient
Used in the formula that calculates the distance between two genomes. It's the c_3 coefficient in (6.1).

Type float

species_distance_threshold
Minimum distance, as calculated by (6.1), between two genomes for them to be considered as being of the same species. A lower threshold will make new species easier to appear, increasing the number of species throughout the evolutionary process.

Type float

species_elitism_threshold
Species with a number of members superior to this threshold will have their fittest member copied unchanged to the next generation.

Type int

species_no_improvement_limit
If a species doesn't show improvement in its best fitness for this amount of generations, it will be extinct.

Type int

reset_innovations_period
If *None*, the innovation IDs of the new genes will never be reset. If an *int*, the innovation IDs will be reset periodically with a period (number of generations passed) equal to the value specified. As long as the id handler isn't reset, a hidden node can't be inserted more than once in a connection between two given nodes.

Type Optional[int]

allow_self_connections
Whether to allow or not connections connecting a node to itself. If a node is connected to itself, it considers its last output when calculating its new output.

Type bool

initial_node_activation
Initial activation value cached by a node when it's created or reset.

Type float

ATTRIBUTES = {'allow_self_connections': True, 'bias_value': 1, 'disable_inherited_co
Attributes supported by the class and their default values. Each attribute can passed as a kwarg in the class' constructor or be specified in a config file. Attributes not specified will be initialized with a default value.

MAEX_KEYS = {'enable_connection_mutation_chance', 'new_connection_mutation_chance', 'n
Name of the attributes whose values change according to the mass extinction counter (type: Tuple[float, float]).

nevopy.neat.genes module

Implements the nodes (neurons) and edges (connections) of a genome.

class nevopy.neat.genes.**ConnectionGene**(*cid*, *from_node*, *to_node*, *weight*, *enabled=True*)
Bases: object

A connection between two nodes.

A connection gene represents/encodes a connection (edge) between two nodes (neurons) of a neural network (phenotype of a genome).

Parameters

- **cid** (*int*) – The innovation number of the connection. As described in the original NEAT paper [SM02], this serves as a historical marker for the gene, helping to identify homologous genes.

- **from_node** (NodeGene) – Node from where the connection is originated. The source node of the connection.
- **to_node** (NodeGene) – Node to where the connection is headed. The destination node of the connection.
- **weight** (*float*) – The weight of the connection.
- **enabled** (*bool*) – Whether the initial state of the newly created connection should enabled or disabled.

weight
The weight of the connection.

Type float

enabled
Whether the connection is enabled or not. A disabled connection won't be considered during the computations of the neural network.

Type bool

property from_node
Node where the connection is originated (source node).

Return type *NodeGene*

property id
Innovation number of the connection gene.

As described in the original NEAT paper [SM02], this value serves as a historical marker for the gene, helping to identify homologous genes. Although must of the identification is based on the nodes that form the connection, this ID is helpful to increase the speed of certain comparisons.

Return type int

self_connecting()
Returns *True* if the connection is connecting a node to itself and *False* otherwise.

Return type bool

property to_node
Node to where the connection is headed (destination node).

Return type *NodeGene*

exception nevopy.neat.genes.**ConnectionIdException**
Bases: Exception

Indicates that an attempt has been made to assign a new ID to a connection gene that already has an ID.

class nevopy.neat.genes.**NodeGene**(*node_id*, *node_type*, *activation_func*, *initial_activation*)
Bases: object

A gene that represents/encodes a neuron (node) in a neural network.

A *NodeGene* is the portion of a *NeatGenome* that encodes a neuron (node) of the neural network encoded by the *NeatGenome*. It has an activation function, which is applied to inputs received from other nodes of the network.

Parameters

- **node_id** (*int*) – The node's identifier / innovation number.
- **node_type** (NodeGene.Type) – The node's type.

- **activation_func** (*Callable[[float], float]*) – Activation function to be used by the node. It should receive a float as input and return a float (the resulting activation) as output.
- **initial_activation** (*float*) – initial value of the node's activation (used when processing recurrent connections between nodes).

in_connections
List with the connections (*ConnectionGene*) leaving this node, i.e., connections that have this node as the source.

Type List[*ConnectionGene*]

out_connections
List with the connections (*ConnectionGene*) coming to this node, i.e., connections that have this node as the destination.

Type List[*ConnectionGene*]

class Type(*value*)
Bases: `enum.Enum`

Specifies the possible types of node genes.

BIAS = 1

HIDDEN = 2

INPUT = 0

OUTPUT = 3

activate(*x*)
Applies the node's activation function to the given input.

The node's activation value, i.e., the node's cached output, is updated by this call and can be later be accessed through the property *activation*.

Return type `None`

Returns None. The node's output is updated internally.

property activation
The node's cached activation value, i.e., the node's output when it was last processed.

Return type `float`

property id
Innovation ID of the gene.

This ID is used to mate genomes and to calculate their difference.

"The innovation numbers are historical markers that identify the original historical ancestor of each gene. New genes are assigned new increasingly higher numbers." - [SM02]

Return type `int`

reset_activation()
Resets the node's activation value (it's cached output) to its initial value.

Return type `None`

simple_copy()
Makes and returns a simple copy of this node.

Wraps a call to this class' constructor.

The copied node shares the same values for all the attributes of the source node, except for the connections. The copied node is created without any connections.

Return type *NodeGene*

Returns A copy of this node without any connection.

property type
Type of the node (input, bias, hidden or output).

Return type *Type*

exception nevopy.neat.genes.**NodeIdException**
Bases: Exception

Indicates that an attempt has been made to assign a new ID to a gene node that already has an ID.

exception nevopy.neat.genes.**NodeParentsException**
Bases: Exception

Indicates that an attempt has been made to get the parents of a non-hidden node.

nevopy.neat.genes.**align_connections**(*con_list1*, *con_list2*, *print_alignment=False*)
Aligns the matching connection genes of the given lists.

In the context of NEAT [SM02], aligning homologous connections genes is required both to compare the similarity of a pair of genomes and to perform sexual reproduction. Two connection genes are said to match or to be homologous if they have the same innovation ID, meaning that they represent the same structure.

Genes that do not match are either disjoint or excess, depending on whether they occur within or outside the range of the other parent's innovation numbers. They represent a structure that is not present in the other genome.

Parameters

- **con_list1** (*List[ConnectionGene]*) – The first list of connection genes.
- **con_list2** (*List[ConnectionGene]*) – The second list of connection genes.
- **print_alignment** (bool) – Whether to print the generated alignment or not. Used for debugging.

Return type Tuple[List[Optional[*ConnectionGene*]], List[Optional[*ConnectionGene*]]]

Returns

A tuple containing two lists of the same size. Index 0 corresponds to the first list and index 1 to the second list. The returned lists contain connection genes or *None*. The order of the genes is preserved in the returned lists (but not their indices!).

If, given a position, there are two genes (one in each list), the genes match. On the other hand, if, in the position, there is only one gene (on one of the lists) and a *None* value (on the other list), the genes are either disjoint or excess.

nevopy.neat.genomes module

Implements the genome and its main operations.

A genome is a collection of genes that encode a neural network (the genome's phenotype). In this implementation, there is no distinction between a genome and the network it encodes. In NEAT, the genome is the entity subject to evolution.

exception nevopy.neat.genomes.**ConnectionExistsError**
Bases: Exception

Exception that indicates that a connection between two given nodes already exists.

exception nevopy.neat.genomes.**ConnectionToBiasNodeError**
Bases: Exception

Exception that indicates that an attempt has been made to create a connection containing a bias node as destination.

class nevopy.neat.genomes.**FixTopNeatGenome**(*fito_genome*, *num_neat_inputs*, *num_neat_outputs*, *config*, *initial_neat_connections=True*)
Bases: *nevopy.neat.genomes.NeatGenome*

Integration of a NEAT genome with a fixed topology genome.

This class defines a new type of NEAT genome that integrates the default NeatGenome with a :class:.FixedTopologyGenome`. It can be used with *NeatPopulation*.

When an input is received, it's first processed by the layers of the fixed topology genome. The output is, then, processed using NEAT, which generates the final output.

Note: This class is useful when the inputs that will be fed to the genome have high dimensions. Since NEAT doesn't scale well with such lengthy inputs (like images), a fixed topology genome (that can contain, for instance, convolutional layers) can be used to reduce the dimensionality of the input before feeding it to NEAT's nodes.

Parameters

- **fito_genome** (FixedTopologyGenome) – Instance of *FixedTopologyGenome* to be used to pre-process the inputs. It will also be evolved.
- **num_neat_inputs** (*int*) – Length of the flattened outputs of the fixed topology genome. It's also the number of input nodes of the NEAT genome.
- **num_neat_outputs** (*int*) – Number of output nodes of the NEAT genome.
- **config** (NeatConfig) – Settings of the current evolutionary session.
- **initial_neat_connections** (*bool*) – Whether to create connections connecting each input node of the NEAT genome to each of its output nodes.

deep_copy()
Makes an exact/deep copy of the genome.

All the nodes and connections (including their weights) of the parent genome are copied to the new genome.

Return type *FixTopNeatGenome*

Returns An exact/deep copy of the genome.

distance(*other*)
Sums, to the default distance calculated by *NeatGenome.distance()*, the sum of the absolute difference between the fixed topology layers weights.

Return type float

mate(*other*)
Mates two genomes to produce a new genome (offspring).

Sexual reproduction. Follows the idea described in the original paper of the NEAT algorithm:

"When crossing over, the genes in both genomes with the same innovation numbers are lined up. These genes are called matching genes. (...). Matching genes are inherited randomly, whereas disjoint genes (those that do not match in the middle) and excess genes (those that do not match in the end) are inherited from the more fit parent. (...) [If the parents fitness are equal] the disjoint and excess genes are also inherited randomly. (...) there's a preset chance that an inherited gene is disabled if it is disabled in either parent." - [SM02]

Parameters **other** (NeatGenome) – The second genome. Currently, *NeatGenome* is only compatible for mating with instances of *NeatGenome* or of one of its subclasses.

Return type *NeatGenome*

Returns A new genome (the offspring born from the sexual reproduction between the current genome and the genome passed as argument.

Raises ***IncompatibleGenomesError*** – If the genome passed as argument to other is incompatible with the current genome (*self*).

mutate_weights()
Randomly mutates the weights of the genome's connections.

Each connection gene in the genome has a chance to be perturbed, reset or to remain unchanged.

Return type None

process(*x*)
Feeds the input to the fixed topology genome and uses the output as input to the NEAT genome.

Return type ndarray

random_copy()
Makes a deep copy of the genome, but with random weights.

Return type *FixTopNeatGenome*

Returns A deep copy of the genome with the same topology of the original genome, but random connections weights.

simple_copy()
Makes a simple copy of the genome.

Wraps a call to this class' constructor. The new genome's is initialized without a fixed topology genome (fito_genome) - the value of this attribute is *None*.

Return type *FixTopNeatGenome*

Returns A copy of the genome without any of its connections (including the ones between input and output nodes) and hidden nodes. The attribute fito_genome is set to None.

class nevopy.neat.genomes.**NeatGenome**(*num_inputs*, *num_outputs*, *config*, *initial_connections=True*)
Bases: *nevopy.base_genome.BaseGenome*

Linear representation of a neural network's connectivity.

In the context of NEAT, a genome is a collection of genes that encode a neural network (the genome's phenotype). In this implementation, there is no distinction between a genome and the network it encodes. A genome processes inputs based on its nodes and connections in order to produce an output, emulating a neural network.

Note: The instances of this class are the entities subject to evolution by the NEAT algorithm.

Note: The encoded networks are Graph Neural Networks (GNNs), connectionist models that capture the dependence of graphs via message passing between the nodes of graphs.

Note: When declaring a subclass of this class, you should always override the methods *simple_copy()*, *deep_copy()* and *random_copy()*, so that they return an instance of your subclass and not of *NeatGenome*. It's recommended (although optional) to also override the methods *distance()* and *mate()*.

Parameters

- **num_inputs** (*int*) – Number of input nodes in the network.
- **num_outputs** (*int*) – Number of output nodes in the network.
- **config** (NeatConfig) – Settings of the current evolution session.
- **initial_connections** (*bool*) – If True, connections between the input nodes and the output nodes of the network will be created.

species_id
Indicates the species to which the genome belongs.

Type int

fitness
The last calculated fitness of the genome.

Type float

adj_fitness
The last calculated adjusted fitness of the genome.

Type float

hidden_nodes
List with all the node genes of the type *NodeGene.Type.HIDDEN* in the genome.

Type list of *NodeGene*

connections
List with all the connection genes in the genome.

Type list of *ConnectionGene*

_existing_connections_dict
Used as a fast lookup table to consult existing connections in the network. Given a node N, it maps N's ID to the IDs of all the nodes that have a connection with N as the source.

Type Dict[int, Set]

add_connection(*cid*, *src_node*, *dest_node*, *enabled=True*, *weight=None*)
Adds a new connection gene to the genome.

Parameters

- **cid** (*int*) – ID of the connection. It's used as a historical marker of the connection's creation, acting as an "innovation number".
- **src_node** (NodeGene) – Node from where the connection leaves (source node).
- **dest_node** (NodeGene) – Node to where the connection is headed (destination node).
- **enabled** (*bool*) – Whether the new connection should be enabled or not.
- **weight** (*Optional[float]*) – The weight of the connection. If *None*, a random value (within the interval specified in the settings) will be chosen.

Raises

- ***ConnectionExistsError*** – If the connection *src_node->dest_node* already exists in the genome.
- ***ConnectionToBiasNodeError*** – If *dest_node* is an input or bias node (nodes of these types do not process inputs!).

Return type None

add_random_connection(*id_handler*)
Adds a new connection between two random nodes in the genome.

This is an implementation of the *add connection mutation*, described in the original NEAT paper [SM02].

Parameters **id_handler** (IdHandler) – ID handler that will be used to assign an ID to the new connection. The handler's internal cache of existing connections will be updated accordingly.

Return type Optional[Tuple[*NodeGene*, *NodeGene*]]

Returns A tuple containing the source node and the destination node of the connection, if a new connection was successfully created. *None*, if there is no space in the genome for a new connection.

add_random_hidden_node(*id_handler*)
Adds a new hidden node to the genome in a random position.

This method implements the *add node mutation* procedure described in the original NEAT paper:

"An existing connection is split and the new node placed where the old connection used to be. The old connection is disabled and two new connections are added to the genome. The new connection leading into the new node receives a weight of 1, and the new connection leading out receives the same weight as the old connection." - [SM02]

Only currently enabled connections are considered eligible to "host" the new hidden node.

Parameters **id_handler** (IdHandler) – ID handler that will be used to assign an ID to the new hidden node. The handler's internal cache of existing nodes and connections will be updated accordingly.

Return type Optional[*NodeGene*]

Returns The new hidden node, if it was successfully created. *None* if it wasn't possible to find a connection to "host" the new node. This usually happens when the ID handler hasn't been reset in a while.

property config
Settings of the current evolutionary session.

If *None*, a config object hasn't been assigned to this genome yet.

Return type Any

connection_exists(*src_id*, *dest_id*)
Checks whether a connection between the given nodes exists.

Parameters

- **src_id** (*int*) – ID of the connection's source node.
- **dest_id** (*int*) – ID of the connection's destination node.

Return type bool

Returns *True* if the specified connection exists in the genome's network and *False* otherwise.

deep_copy()
Makes an exact/deep copy of the genome.

All the nodes and connections (including their weights) of the parent genome are copied to the new genome.

Return type *NeatGenome*

Returns An exact/deep copy of the genome.

distance(*other*)
Calculates the distance between two genomes.

The shorter the distance between two genomes, the greater the similarity between them is. In the context of NEAT, the similarity between genomes increases as:

1) the number of matching connection genes increases;
2) the absolute difference between the matching connections weights decreases;

The distance between genomes is used for speciation and for sexual reproduction (mating).

The formula used is shown below. It's the same as the one presented in the original NEAT paper [SM02]. All the coefficients are configurable.

$$\delta = c_1 \cdot \frac{E}{N} + c_2 \cdot \frac{D}{N} + c_3 \cdot W \tag{6.1}$$

Parameters **other** (NeatGenome) – The other genome (an instance of *NeatGenome* or one of its subclasses).

Return type float

Returns The distance between the genomes.

enable_random_connection()
Randomly activates a disabled connection gene.

Return type None

info()
Returns a string with the genome's nodes activations and connections. Used mostly for debugging purposes.

Return type str

property input_shape
Number of input nodes in the genome.

Return type int

mate(*other*)
Mates two genomes to produce a new genome (offspring).

Sexual reproduction. Follows the idea described in the original paper of the NEAT algorithm:

"When crossing over, the genes in both genomes with the same innovation numbers are lined up. These genes are called matching genes. (...). Matching genes are inherited randomly, whereas disjoint genes (those that do not match in the middle) and excess genes (those that do not match in the end) are inherited from the more fit parent. (...) [If the parents fitness are equal] the disjoint and excess genes are also inherited randomly. (...) there's a preset chance that an inherited gene is disabled if it is disabled in either parent." - [SM02]

Parameters **other** (NeatGenome) – The second genome. Currently, *NeatGenome* is only compatible for mating with instances of *NeatGenome* or of one of its subclasses.

Return type *NeatGenome*

Returns A new genome (the offspring born from the sexual reproduction between the current genome and the genome passed as argument.

Raises ***IncompatibleGenomesError*** – If the genome passed as argument to other is incompatible with the current genome (*self*).

mutate_weights()
Randomly mutates the weights of the genome's connections.

Each connection gene in the genome has a chance to be perturbed, reset or to remain unchanged.

Return type None

nodes()
Returns all the genome's node genes. Order: inputs, bias, outputs and hidden.

Return type List[*NodeGene*]

property output_shape
Number of output nodes in the genome.

Return type int

process(*x*)
Feeds the given input to the neural network.

In this implementation, there is no distinction between a genome and the neural network it encodes. The genome will emulate a neural network (its phenotype) in order to process the given input. The encoded network is a Graph Neural Networks (GNN).

Note: The processing is done recursively, starting from the output nodes (top-down approach). Because of that, nodes not connected to at least one of the network's output nodes won't be processed.

Parameters **x** (*Sequence[float]*) – A sequence object (like a list or numpy array) containing the inputs to be fed to the neural network input nodes. It represents a single training sample. The value in the index *i* of *X* will be fed to the i^{th} input node of the neural network.

Return type ndarray

Returns A numpy array containing the outputs of the network's output nodes. The index *i* contains the activation value of the i^{th} output node of the network.

Raises ***InvalidInputError*** – If the number of elements in *X* doesn't match the number of input nodes in the network.

process_node(*n*)

Recursively processes the activation of the given node.

Unless it's a bias or input node (that have a fixed output), a node must process the input it receives from other nodes in order to produce an activation. This is done recursively: if *n* receives input from a node *m* that haven't had its activation calculated yet, the activation of *m* will be calculated recursively before the activation of *n* is computed. Recurrences are solved by using the previous activation of the "problematic" node.

Let w_i be the weight of the i^{th} connection that has *n* as destination node. Let a_i be the current cached output of the source node of c_i. Let σ be the activation function of *n*. The activation (output) *a* of *n* is computed as follows:

$a = \sigma(\sum_i w_i \cdot a_i)$

Parameters **n** (NodeGene) – The node to be processed.

Return type float

Returns The activation value (output) of the node.

random_copy()

Makes a deep copy of the genome, but with random weights.

Return type *NeatGenome*

Returns A deep copy of the genome with the same topology of the original genome, but random connections weights.

reset()

Wrapper for *reset_activations()*.

Return type None

reset_activations()

Resets cached activations of the genome's nodes.

It restores the current activation value of all the nodes in the network to their initial value.

Return type None

simple_copy()

Makes a simple copy of the genome.

Wraps a call to this class' constructor.

Return type *NeatGenome*

Returns A copy of the genome without any of its connections (including the ones between input and output nodes) and hidden nodes.

valid_in_nodes()

Checks if all the genome's input nodes are valid.

An input node is considered to be valid if it has at least one enabled connection leaving it, i.e., its activation is used as input by at least one other node.

Return type bool

Returns *True* if all the genome's input nodes are valid and *False* otherwise.

valid_out_nodes()
Checks if all the genome's output nodes are valid.

An output node is considered to be valid if it receives, during its processing, at least one input, i.e., the node has at least one enabled incoming connection. Invalid output nodes simply outputs a fixed default value and are, in many cases, undesirable.

Return type bool

Returns *True* if all the genome's output nodes have at least one enabled incoming connection and *False* otherwise. Self-connecting connections are not considered.

visualize(***kwargs*)
Simple wrapper for the *nevopy.neat.visualization.visualize_genome()* function. Please refer to its documentation for more information.

Return type None

visualize_activations(***kwargs*)
Simple wrapper for the *nevopy.neat.visualization.visualize_activations()* function. Please refer to its documentation for more information.

Return type Any

nevopy.neat.id_handler module

This module implements the ID handler, use to assign IDs to species, genomes, hidden nodes and connections. In the case of nodes and connections genes, the ID can also be interpreted as an innovation number.

class nevopy.neat.id_handler.**IdHandler**(*num_inputs*, *num_outputs*, *has_bias*)
Bases: object

Handles the assignment of IDs.

An ID handler manages the assignment of IDs to species, genomes, hidden nodes and connections. In the case of nodes and connections genes, the ID can also be interpreted as an innovation number.

"The innovation numbers are historical markers that identify the original historical ancestor of each gene. New genes are assigned new increasingly higher numbers." - [SM02]

The ID handler implements the following solution:

"A possible problem is that the same structural innovation will receive different innovation numbers in the same generation if it occurs by chance more than once. However, by keeping a list of the innovations that occurred in the current generation, it is possible to ensure that when the same structure arises more than once through independent mutations in the same generation, each identical mutation is assigned the same innovation number. Thus, there is no resultant explosion of innovation numbers." - [SM02]

In *NEvoPy*, it's possible to configure the rate at which innovation numbers are reset (see *NeatConfig.reset_innovations_period*).

Warning: This class isn't compatible with parallel processing.

Parameters

- **num_inputs** (*int*) – Number of input nodes in the genomes.
- **num_outputs** (*int*) – Number of output nodes in the genomes.

- **has_bias** (*bool*) – Whether the genomes have a bias node.

next_connection_id(*src_id*, *dest_id*)
Returns an ID / innovation number for a connection gene.

The new connection is identified through the IDs of its source and destination nodes. While the ID handler isn't reset, connections that have the same source and destination nodes will be assigned the same ID.

Parameters

- **src_id** (*int*) – ID of the new connection's source node.
- **dest_id** (*int*) – ID of the new connection's destination node.

Return type int

Returns An ID for the new connection.

next_hidden_node_id(*src_id*, *dest_id*)
Returns an ID / innovation number for a hidden node.

A hidden node is created by breaking an existing connection of the genome in two. Consider two nodes *A* and *B*, both of which are present in multiple genomes of the population. While the ID handler isn't reset, hidden nodes created by breaking the connection *A->B* will be assigned the same ID / innovation number.

Parameters

- **src_id** (*int*) – ID of the source node of the connection being broken to create the new hidden node.
- **dest_id** (*int*) – ID of the destination node of the connection being broken to create the new hidden node.

Return type int

Returns An ID for the new hidden node.

next_species_id()
Returns a new unique ID for a species.

reset()
Resets the cache of new nodes and connections.

This resets the handler's cached innovations.

Return type None

nevopy.neat.population module

Implementation of the main mechanisms of the NEAT algorithm.

This is the main module of *NEvoPy's* implementation of the NEAT algorithm. It implements the *NeatPopulation* class, which handles the evolution of a population/community of NEAT genomes.

class nevopy.neat.population.**NeatPopulation**(*size*, *num_inputs=None*, *num_outputs=None*, *base_genome=None*, *config=None*, *processing_scheduler=None*)
Bases: *nevopy.base_population.BasePopulation*

Population of individuals (genomes) to be evolved by the NEAT algorithm.

Main class of *NEvoPy's* implementation of the NEAT algorithm. It represents a population of individuals (genomes) to be evolved. The correct term, in NEAT's case, is actually "community" (group of populations of two or more different species) rather than "population" (subset of individuals of one species), since NEAT

divides its genomes into species. However, to maintain consistency with the neuroevolution literature, the term "population" is used.

To use NEAT, most users will need to use only this class. It's main method, *evolve()*, starts the evolutionary process. By providing a processing scheduler, the user is able to specify how the computation of the fitness of the population's genomes will occur (whether to use serial or parallel processing, CPU or GPU, etc).

By default, a *PoolProcessingScheduler* is used. It implements parallel processing using (by default) all the CPU cores of the machine where the program is running. Alternatively, if you want to run the evolution process on multiple machines (cluster) you should check out the *RayProcessingScheduler*.

Example

Suppose you have already defined a function called *fitness_func* that takes a genome as input and calculates its fitness. If the networks take 10 input values and outputs 3 values, here is how you can proceed to create and evolve a population of 100 genomes using the default settings and processing scheduler:

```
def fitness_func(genome):
    """
    Function that takes a genome as input and returns the genome's
    fitness (a float) as output.
    """
    # ...


# Creating and evolving a population:
population = NeatPopulation(size=100,
                            num_inputs=10,
                            num_outputs=3)
history = population.evolve(generations=100,
                            fitness_function=fitness_func)

# Visualizing the progression of the population's fitness:
history.visualize()

# Retrieving and visualizing the fittest genome of the population:
best_genome = population.fittest()
best_genome.visualize()
```

Parameters

- **size** (*int*) – Number of genomes in the population (constant value).
- **num_inputs** (*Optional[int]*) – Number of input nodes in each genome. If *None*, the number of inputs will be inferred from the base genome.
- **num_outputs** (*Optional[int]*) – Number of output nodes in each genome. If *None*, the number of outputs will be inferred from the base genome.
- **base_genome** (*Optional[NeatGenome]*) – Genome that will serve as a base for the randomly generated genomes of the population. If *None*, a new genome of the class *NeatGenome* will be used as the base genome.
- **config** (*NeatConfig*) – The settings of the evolutionary process. If *None* the default settings will be used.
- **processing_scheduler** (*Optional[ProcessingScheduler]*) – Processing scheduler to be used to compute the fitness of the population's genomes. If *None*, the default scheduler will be used *PoolProcessingScheduler*.

species
List with the currently alive species in the population.

Type List[*NeatSpecies*]

DEFAULT_SCHEDULER
alias of *nevopy.processing.pool_processing.PoolProcessingScheduler*

property config
Config object that stores the settings used by the population.

Return type Any

evolve(*generations*, *fitness_function*, *callbacks=None*, *verbose=2*, ***kwargs*)
Evolves the population of genomes using the NEAT algorithm.

Parameters

- **generations** (*int*) – Number of generations for the algorithm to run. A generation is completed when all the population's genomes have been processed and reproduction and speciation has occurred.
- **fitness_function** (*Callable[[NeatGenome], float]*) – Fitness function to be used to evaluate the fitness of individual genomes. It must receive a genome as input and produce a float (the genome's fitness) as output.
- **callbacks** (*Optional[List[Callback]]*) – List with instances of *Callback* that will be called during the evolutionary session. By default, a *History* callback is always included in the list. A *CompleteStdOutLogger* or a *SimpleStdOutLogger* might also be included, depending on the value passed to the verbose param.
- **verbose** (*int*) – Verbose level (logging on stdout). Options: 0 (no verbose), 1 (light verbose) and 2 (heavy verbose).

Return type *History*

Returns A *History* object containing useful information recorded during the evolutionary process.

generate_offspring(*species*, *rank_prob_dist*)
Generates a new genome from one or more genomes of the species.

The offspring can be generated either by mating two randomly chosen genomes (sexual reproduction) or by cloning a single genome (asexual reproduction / binary fission). After the newly born genome is created, it has a chance of mutating. The possible mutations are:

. Enabling a disabled connection;
. Changing the weights of one or more connections;
. Creating a new connection between two random nodes;
. Creating a new random hidden node.

Parameters

- **species** (NeatSpecies) – Species from which the offspring will be generated.
- **rank_prob_dist** (*Sequence*) – Sequence (usually a numpy array) containing the chances of each of the species genomes being the first parent of the newborn genome.

Return type *NeatGenome*

Returns A newly generated genome.

info()
Returns a string containing relevant information about the population.

Return type str

offspring_proportion(*num_offspring*)
Calculates the number of descendants each species will leave for the next generation.

Every species is assigned a potentially different number of offspring in proportion to the sum of adjusted fitnesses of its member organisms [SM02]. This selection method is called *roulette wheel selection*.

Parameters **num_offspring** (*int*) – Number of genomes to be generated by all the species combined.

Returns A dictionary mapping the ID of each of the population's species to the number of descendants it will leave for the next generation.

Return type Dict[int, int]

reproduction()
Handles the reproduction of the population's genomes.

This method implements the reproduction mechanism described in the original paper of the NEAT algorithm [SM02].

First, the most fit genomes of each species with more than a pre-defined number of individuals are selected to be passed unchanged to the next generation (elitism). Next, the least fit genomes of each species are discarded (reverse elitism). After that, the number of descendants of each species is calculated based on the proportion between the total fitness of the population and the adjusted fitness of the species (roulette wheel selection). Finally, the reproduction of individuals of the same species (and, on rare occasions, between genomes of different species as well) occurs.

Genomes with a higher fitness have a higher chance of leaving offspring. Within a species, the chance of a genome reproducing is given by the position it occupies in the species fitness rank (rank-based selection). This means that the reproduction chance of a genome is not directly calculated from the genome's fitness, but rather from how well positioned is the genome in the fitness rank.

Most of the behaviour described above can be adjusted by changing the settings of the evolutionary process (see *NeatConfig*).

Return type None

speciation(*generation*)
Divides the population's genomes into species.

The importance of speciation for NEAT:

"Speciating the population allows organisms to compete primarily within their own niches instead of with the population at large. This way, topological innovations are protected in a new niche where they have time to optimize their structure through competition within the niche. The idea is to divide the population into species such that similar topologies are in the same species." - [SM02]

The distance (compatibility) between a pair of genomes is calculated based on to the number of excess and disjoint genes between them. See *NeatGenome.distance()* for more information.

About the speciation process:

"Each existing species is represented by a random genome inside the species from the previous generation. A given genome g in the current generation is placed in the first species in which g is compatible with the representative genome of that species. This way, species do not overlap. If g is not compatible with any existing species, a new species is created with g as its representative." - [SM02]

Species that haven't improved their fitness for a pre-defined number of generations are extinct, i.e., they are removed from the population and aren't considered for the speciation process. This number is configurable.

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

Return type None

nevopy.neat.species module

Implementation of the *NeatSpecies* class.

class nevopy.neat.species.**NeatSpecies**(*species_id*, *generation*)

Bases: object

Represents a species within NEAT's evolutionary environment.

Parameters

- **species_id** (*int*) – Unique identifier of the species.
- **generation** (*int*) – Current generation. The generation in which the species is born.

representative

Genome used to represent the species.

Type Optional[*NeatGenome*]

members

List with the genomes that belong to the species.

Type List[*NeatGenome*]

last_improvement

Generation in which the species last showed improvement of its fitness. The species fitness in a given generation is equal to the fitness of the species most fit genome on that generation.

Type int

best_fitness

The last calculated fitness of the species most fit genome.

Type Optional[float]

avg_fitness()

Returns the average fitness of the species genomes.

Return type float

fittest()

Returns the fittest member of the species.

Return type *NeatGenome*

property id

Unique identifier of the species.

Return type int

random_representative()

Randomly chooses a new representative for the species.

Return type None

nevopy.neat.visualization module

This module implements visualization utilities related to the NEAT algorithm.

class nevopy.neat.visualization.**NodeVisualizationInfo**(*label=''*, *activation_threshold=0.5*, *mode='greater'*, *equality_precision=0.01*)

Bases: object

Stores information about an input or output node of a *NeatGenome* to be visualized with the neat.visualize_activations() function.

Parameters

- **label** (*str*) – Label to be drawn next to the node.
- **activation_threshold** (*float*) – Value to be taken as reference when checking if the node is activated or not.
- **mode** (*str*) – Name of the method to be used to check if the node is activated or not. Currently available modes: "greater", "less", "equal" and "diff".

is_activated(*activation*)

Checks whether the node is activated or not.

Return type bool

nevopy.neat.visualization.**columns_graph_layout**(*genome*, *width*, *height*, *node_size*, *horizontal_pad_pc=(0.03, 0.03)*, *vertical_pad_pc=(0.03, 0.03)*, *ideal_h_nodes_per_col=4*, *consider_bias_node=True*)

Positions the network's nodes in columns.

The input nodes are placed in the left-most column and the output nodes are placed in the right-most columns. The hidden nodes are placed in columns located between those two columns. For big networks, try using a smaller node size for better quality.

Parameters

- **genome** (NeatGenome) – The genome to be visualized.
- **width** (*float*) – Width of the figure / surface.
- **height** (*float*) – Height of the figure / surface.
- **node_size** (*float*) – Size of the drawn nodes.
- **horizontal_pad_pc** (*Tuple[float, float]*) – Tuple containing the size of the padding on the left and on the right of the surface. Unit: the width of the surface.
- **vertical_pad_pc** (*Tuple[float, float]*) – Tuple containing the size of the padding below and above the surface. Unit: the height of the surface.
- **ideal_h_nodes_per_col** (*int*) – Preferred number of hidden nodes per column (the algorithm will try to draw columns with this amount of hidden nodes when possible).
- **consider_bias_node** (*bool*) – Whether the bias node should be considered or not when calculating the positions.

Return type Dict[int, Tuple[float, float]]

Returns Dictionary mapping the ID of each node to a tuple containing its position in the figure.

nevopy.neat.visualization.**visualize_activations**(*genome*, *surface_size=(700, 450)*, *node_radius=14*, *node_deactivated_color=(190, 190, 190)*, *node_activated_color=(2, 68, 144)*, *bias_node_color='yellow'*, *node_border_color='black'*, *edge_activated_color=(0, 120, 233)*, *edge_deactivated_color=(100, 100, 100)*, *activated_edge_width=2*, *deactivated_edge_width=1*, *horizontal_pad_pc=(0.015, 0.015)*, *vertical_pad_pc=(0.015, 0.015)*, *hidden_activation_threshold=0.5*, *input_visualization_info=None*, *output_visualization_info=None*, *output_activate_greatest_only=True*, *show_input_values=False*, *show_output_values=False*, *labels_color='white'*, *labels_config=None*, *show_activation_light=True*, *activation_light_color=(104, 179, 235)*, *activation_light_radius_pc=2*, *ideal_h_nodes_per_col=4*, *background_color='black'*, *node_border_thickness=2*, *draw_bias_node=False*, *return_rgb_array=False*)

Draws the network using different colors for activated and deactivated nodes and edges.

Note: This method requires `pygame` installed. You can install it using the command:

```
$ pip install pygame
```

Parameters

- **genome** (NeatGenome) – The genome to be visualized.
- **surface_size** (*Tuple[int, int]*) – Width and height of the pygame surface to be drawn.
- **node_radius** (*float*) – Radius (size) of the drawn nodes.
- **node_deactivated_color** (*Union[str, Tuple[int, int, int]]*) – Color of deactivated nodes.
- **node_activated_color** (*Union[str, Tuple[int, int, int]]*) – Color of activated nodes.
- **bias_node_color** (*Union[str, Tuple[int, int, int]]*) – Color of the bias node.
- **node_border_color** (*Union[str, Tuple[int, int, int]]*) – Color of the nodes' borders.

- **edge_activated_color** (*Union[str, Tuple[int, int, int]]*) – Color of activated edges.
- **edge_deactivated_color** (*Union[str, Tuple[int, int, int]]*) – Color of deactivated edges.
- **activated_edge_width** (*int*) – The width/thickness of activated edges.
- **deactivated_edge_width** (*int*) – The width/thickness of deactivated edges.
- **horizontal_pad_pc** (*Tuple[float, float]*) – Tuple containing the size of the padding on the left and on the right of the surface. Unit: the width of the surface.
- **vertical_pad_pc** (*Tuple[float, float]*) – Tuple containing the size of the padding below and above the surface. Unit: the height of the surface.
- **hidden_activation_threshold** (*float*) – Activation threshold for hidden nodes. If the activation value of a hidden node is greater than this threshold, the node is considered to be activated.
- **(Optional[Union[** (*output_visualization_info*) – List[NodeVisualizationInfo], List[str]]]): If it's a list of strings, each string will be the label of an input node and default settings will be used to determine if an input node is activated or not. If it's a list of *NodeVisualizationInfo* objects, then the information provided in the objects will be used instead. If `None`, default settings will be used to determine if an input node is activated or not and no labels will be drawn for the input nodes.
- **(Optional[Union[** – List[NodeVisualizationInfo], List[str]]]): If it's a list of strings, each string will be the label of an output node and default settings will be used to determine if an output node is activated or not. If it's a list of *NodeVisualizationInfo* objects, then the information provided in the objects will be used instead. If `None`, default settings will be used to determine if an output node is activated or not and no labels will be drawn for the output nodes.
- **show_input_values** (*bool*) – If `True` and `input_visualization_info` is not `None`, then the input values will be drawn next to each input node.
- **show_output_values** (*bool*) – If `True` and `output_visualization_info` is not `None`, then the output values will be drawn next to each output node.
- **output_activate_greatest_only** (*bool*) – If `True`, only one output node can be activated at a time (the node with the greatest activation value). Otherwise, more than one node can be activated at a time.
- **labels_color** (*Union[str, Tuple[int, int, int]]*) – Color of the labels.
- **labels_config** (*Dict[str, Any]*) – Keyword arguments to be passed to the `pygame.SysFont()` constructor.
- **show_activation_light** (*bool*) – Whether or not to show a "light ring" around activated nodes.
- **activation_light_color** (*Optional[Union[str, Tuple[int, int, int]]]*) – The color of the light ring to be shown around activated nodes.
- **activation_light_radius_pc** (*float*) – Radius of the light ring to be drawn around activated nodes. Unit: the node's radius.
- **ideal_h_nodes_per_col** (*int*) – Preferred number of hidden nodes per column (the algorithm will try to draw columns with this amount of hidden nodes whenever possible).

- **background_color** (*Union[str, Tuple[int, int, int]]*) – The background color of the surface.
- **node_border_thickness** (*Optional[float]*) – Thickness of the nodes' borders. If `None` or *0*, no border will be drawn.
- **draw_bias_node** (*bool*) – Whether to draw the network's bias node or not.
- **return_rgb_array** (*bool*) – If `True`, returns a numpy array with the generated image instead of a pygame surface.

Return type `Union[ForwardRef, ndarray]`

Returns

If `return_rgb_array` is `False`, an instance of `pygame.Surface` with the drawings is returned. You can display it using `pygame`:

```
screen_size = 700, 450
display = pygame.display.set_mode(screen_size)
# ...
s = genome.visualize_activations(surface_size=screen_size)
display.blit(s, [0, 0])
pygame.display.update()
```

If `return_rgb_array` is `True`, a numpy array with the generated image is returned instead.

Raises **ModuleNotFoundError** – If `pygame` is not found.

nevopy.neat.visualization.**visualize_genome**(*genome*, *layout_name='columns'*, *layout_kwargs=None*, *show=True*, *block_thread=True*, *save_to=None*, *save_transparent=False*, *figsize=(10, 6)*, *node_size=300*, *pad=1*, *legends=True*, *nodes_ids=True*, *node_id_color='black'*, *edge_curviness=0.1*, *edges_ids=False*, *edge_id_color='black'*, *background_color='snow'*, *legend_box_color='honeydew'*, *input_color='deepskyblue'*, *output_color='mediumseagreen'*, *hidden_color='silver'*, *bias_color='khaki'*)

Plots the neural network (phenotype) encoded by the genome.

The network is drawn as a graph, with nodes and edges. An edge's color is chosen according to the edge's weight. Edges with greater weights are drawn with more intense / stronger colors. Edges connecting a node to itself aren't be drawn.

This method uses NetworkX to handle the drawings. It positions the network's nodes according to a layout, whose name you can specify in the parameter `layout_name`. The currently available layouts are:

- All the standard *NetworkX's* layouts available in this link;
- The *graphviz* layout; it's really good, but to use it you must have *Graphviz-Dev* and *pygraphviz* installed on your machine;
- The *columns* layout (used by default), implemented exclusively for *NEvoPy*; it positions the nodes in columns (see `NeatGenome.columns_graph_layout()`, specially the parameter `ideal_h_nodes_per_col`).

For the colors parameters, it's possible to pass a string with the color HEX value or a string with the color's name (names available here: https://matplotlib.org/3.1.0/gallery/color/named_colors.html).

Parameters

- **genome** (*NeatGenome*) – The genome to be visualized.
- **layout_name** (*str*) – The name of the layout to be used to position the network's nodes.
- **layout_kwargs** (*Optional[Dict[str, Any]]*) – Keyed arguments to be passed to the layout. Check each layout documentation for more information about the accepted arguments.
- **show** (*bool*) – Whether to show the generated image or not. If True, a window will be created by *matplotlib* to show the image.
- **block_thread** (*bool*) – Whether to block the execution's thread while showing the image. Useful for visualizing multiple networks at once. In this case, you should call *NeatGenome.visualize()* with this parameter set to *False* on all genomes except for the last one, so all the windows are shown simultaneously.
- **save_to** (*Optional[str]*) – Path to save the image. If *None*, the image won't be automatically saved.
- **save_transparent** (*bool*) – Whether the saved image should have a transparent background or not.
- **figsize** (*Tuple[int, int]*) – Size of the matplotlib figure.
- **node_size** (*int*) – Size of the drawn nodes, in *points**2* (the area of each node). Default size is 300. See the parameter *s* of matplotlib.axes.Axes.scatter for more information.
- **pad** (*int*) – The image's padding (distance between the figure of the network and the image's border).
- **legends** (*bool*) – If *True*, a box with legends describing the nodes colors will be drawn.
- **nodes_ids** (*bool*) – If *True*, the nodes will have their ID drawn inside them.
- **node_id_color** (*str*) – Color of the drawn nodes ids.
- **edge_curviness** (*float*) – Angle, in radians, of the edges arcs. A value of 0 indicates a straight line.
- **edges_ids** (*bool*) – If *True*, each connection/edge will have its ID drawn on it. Keep in mind that some labels might overlap with each other, making only one of them visible.
- **edge_id_color** (*str*) – Color of the drawn connections/edges ids.
- **background_color** (*str*) – Color of the figure's background.
- **legend_box_color** (*str*) – Color of the legend box.
- **input_color** (*str*) – Color of the input nodes.
- **output_color** (*str*) – Color of the output nodes.
- **hidden_color** (*str*) – Color of the hidden nodes.
- **bias_color** (*str*) – Color of the bias node.

Raises **RuntimeError** – If both *show* and *save_to* parameters are set to *False* (in which case the function wouldn't be doing anything but wasting computation).

Return type None

Module contents

Imports core names of *nevopy.neat*.

6.1.4 nevopy.processing package

Submodules

nevopy.processing.base_scheduler module

Defines a common interface for processing schedulers.

This module contains a base model for a processing scheduler, the entity responsible for managing the computation of a population's fitness in *NEvoPy's* algorithms. Schedulers allow the implementation of the computation methods (like the use of serial or parallel processing) to be separated from the implementation of the neuroevolutionary algorithms.

nevopy.processing.base_scheduler.**TProcItem**
TypeVar indicating an item to be scheduled for processing by a *ProcessingScheduler*. Alias for TypeVar("TProcItem").

Type TypeVar

nevopy.processing.base_scheduler.**TProcResult**
TypeVar indicating the result of processing a *TProcItem*. Alias for TypeVar("TProcResult").

Type TypeVar

class nevopy.processing.base_scheduler.**ProcessingScheduler**
Bases: abc.ABC

Defines a common interface for processing schedulers.

In *NEvoPy*, a processing scheduler is responsible for managing the computation of the fitness of a population of individuals being evolved. This abstract class defines a common interface for processing schedulers used by different algorithms. Schedulers allow the implementation of the computation methods (like the use of serial or parallel processing) to be separated from the implementation of the neuroevolutionary algorithms.

Implementing your own processing scheduler is useful when you want to customize the computation of the population's fitness. You can, for example, implement a scheduler that makes use of multiple CPU cores or GPUs (parallel processing).

abstract run(*items*, *func*)
Processes the given items and returns a result.

Main function of the scheduler. Call it to make the scheduler manage the processing of a batch of items.

Parameters

- **items** (*Sequence[TProcItem]*) – Iterable containing the items to be processed.
- **func** (*Optional[Callable[[TProcItem], TProcResult]]*) – Callable (usually a function) that takes one item *TProcItem* as input and returns a result *TProcResult* as output. Generally, *TProcItem* is an individual in the population and *TProcResult* is the individual's fitness. Since some scenarios requires the fitness of the population's individuals to be calculated together, at once, the use of this parameter is not mandatory (this decision is a implementation particularity of each sub-classed scheduler). If additional arguments must be passed to the callable you want to use, it's possible to use Python's functools.partial or to just wrap it with a simple function.

Return type List[~TProcResult]

Returns A list containing the results of the processing of each item. It is guaranteed that the ordering of the items in the returned list follows the order in which the items are yielded by the iterable passed as argument.

nevopy.processing.networked_scheduler module

Implementation of a processing scheduler take makes use of workers hosted in different machines in a network.

Todo: Implementation of the scheduler.

nevopy.processing.pool_processing module

Implements a processing scheduler that uses `multiprocessing.Pool`.

`multiprocessing.Pool` is a built-in Python class that facilitates parallel processing on a single machine. Note that it requires compatibility with *pickle*.

class nevopy.processing.pool_processing.**PoolProcessingScheduler**(*num_processes=None*, *chunksize=None*)

Bases: *nevopy.processing.base_scheduler.ProcessingScheduler*

Processing scheduler that uses Python's `multiprocessing.Pool`.

This scheduler implements parallel processing (on a single machine) using Python's built-in module `multiprocessing`, specifically, the class `Pool`.

Note: `Pool` uses, internally, *pickle* as the serialization method. This might be a source of errors due to incompatibility. Make sure you read the docs carefully before using this scheduler.

Note: When the processing of individual items isn't a very resource demanding task (e.g., learning the 2 variable XOR), using this scheduler might yield significantly better performance than using *RayProcessingScheduler* (due to *ray's* greater overhead). However, in most situations, the performance difference is negligible and using *RayProcessingScheduler* as the processing scheduler is preferable to using this class, since *ray* is safer, scales better and allows clustering.

Parameters

- **num_processes** (*Optional[int]*) – Number of worker processes to use. If *None*, then the number returned by `os.cpu_count()` is used.
- **chunksize** (*Optional[int]*) – `Pool.map()`, used internally by the scheduler, chops the input iterable into a number of chunks which it submits to the process pool as separate tasks. This parameter specifies the (approximate) size of these chunks.

close()
Calls the equivalent method on the scheduler's pool object.

join()
Calls the equivalent method on the scheduler's pool object.

run(*items*, *func*)
Processes the given items and returns a result.

Main function of the scheduler. Call it to make the scheduler manage the parallel processing of a batch of items using `multiprocessing.Pool`.

Note: Make sure that both *items* and *func* are serializable with pickle.

Parameters

- **items** (*Sequence[TProcItem]*) – Iterable containing the items to be processed.
- **func** (*Callable[[TProcItem], TProcResult]*) – Callable (usually a function) that takes one item *TProcItem* as input and returns a result *TProcResult* as output. Generally, *TProcItem* is an individual in the population and *TProcResult* is the individual's fitness.

Return type List[~TProcResult]

Returns A list containing the results of the processing of each item. It is guaranteed that the ordering of the items in the returned list follows the order in which the items are yielded by the iterable passed as argument.

terminate()
Calls the equivalent method on the scheduler's pool object.

nevopy.processing.ray_processing module

Implements a processing scheduler that uses the *ray* framework.

By using *ray* (https://github.com/ray-project/ray), the scheduler is able to implement parallel processing, either on a single machine or on a cluster.

class nevopy.processing.ray_processing.**RayProcessingScheduler**(*address=None*, *num_cpus=None*, *num_gpus=None*, *worker_gpu_frac=None*, ***kwargs*)
Bases: *nevopy.processing.base_scheduler.ProcessingScheduler*

Scheduler that uses *ray* to implement parallel processing.

Ray is an open source framework that provides a simple, universal API for building distributed applications. This scheduler uses it to implement parallel processing. It's possible to either run *ray* on a single machine or on a cluster. For more information regarding the *ray* framework, checkout the project's GitHub page: https://github.com/ray-project/ray.

It's possible to view the *ray's* dashboard at http://127.0.0.1:8265. It contains useful information about the distribution of work and usage of resources by *ray*.

When this class is instantiated, a new *ray* runtime is created. You should close existing *ray* runtimes before creating a new *ray* scheduler, to avoid possible conflicts. If, for some reason, you want to use a currently running *ray* runtime instead of creating a new one, pass *True* as argument to *ignore_reinit_error*.

This class is, basically, a simple *wrapper* for *ray*. If you're an advanced user and this scheduler doesn't meet your needs, it's recommended that you implement your own scheduler by inheriting *ProcessingScheduler*.

Parameters

- **address** (*Optional[str]*) – The address of the Ray cluster to connect to. If this address is not provided, then this command will start Redis, a raylet, a plasma store, a plasma manager, and some workers. It will also kill these processes when Python exits. If the driver is running on a node in a Ray cluster, using *auto* as the value tells the driver to detect the the cluster, removing the need to specify a specific node address.
- **num_cpus** (*Optional[int]*) – Number of CPUs the user wishes to assign to each raylet. By default, this is set based on virtual cores (value returned by os.cpu_count()).
- **num_gpus** (*Optional[int]*) – Number of GPUs the user wishes to assign to each raylet. By default, this is set based on detected GPUs. If you are using TensorFlow, it's recommended for you to execute the following piece of code before importing the module:

```
import os
os.environ["TF_FORCE_GPU_ALLOW_GROWTH"] = "true"
```

  This will prevent individual TensorFlow's sessions from allocating the entire GPU memory available.
- **worker_gpu_frac** (*Optional[float]*) – Minimum fraction of a GPU a worker needs in order to use it. If there isn't enough GPU resources available for a worker when a task is assigned to it, it will not use any GPU resources. Here we consider the number of workers as being equal to the number of virtual CPU cores available. By default, this fraction is set to `num_gpus / num_cpus`, which means that all workers will use the GPUs, each being able to access an equal fraction of them. Note that this might be a source of *out of memory errors*, since the GPU fraction assigned to each worker might be too low. It's usually better to manually select a fraction.
- ****kwargs** – Optional named arguments to be passed to *ray.init()*. For a complete list of the parameters of *ray.init()*, check *ray's* official docs (https://docs.ray.io/en/master/package-ref.html).

run (*items*, *func*)

Processes the given items and returns a result.

Main function of the scheduler. Call it to make the scheduler manage the parallel processing of a batch of items using *ray*.

Parameters

- **items** (*Sequence[TProcItem]*) – Sequence containing the items to be processed.
- **func** (*Callable[[TProcItem], TProcResult]*) – Callable (usually a function) that takes one item *TProcItem* as input and returns a result *TProcResult* as output. Generally, *TProcItem* is an individual in the population and *TProcResult* is the individual's fitness. If additional arguments must be passed to the callable you want to use, it's possible to use Python's `functools.partial` or to just wrap it with a simple function. The callable doesn't need to be annotated with *ray.remote*, this is handled for you.

Return type `List`[~TProcResult]

Returns A list containing the results of the processing of each item. It is guaranteed that the ordering of the items in the returned list follows the order in which the items are yielded by the iterable passed as argument.

nevopy.processing.serial_processing module

Implements a simple wrapper for the serial processing of items.

class nevopy.processing.serial_processing.**SerialProcessingScheduler**
Bases: *nevopy.processing.base_scheduler.ProcessingScheduler*

Simple wrapper for the serial processing of items.

This is scheduler is just a wrapper for the serial processing of items, i.e., the processing of one item at a time. It doesn't involve any explicit parallel processing.

run(*items*, *func*)
Sequentially processes the input items.

Parameters

- **items** (*Sequence[TProcItem]*) – Iterable containing the items to be processed.
- **func** (*Callable[[TProcItem], TProcResult]*) – Callable (usually a function) that takes one item *TProcItem* as input and returns a result *TProcResult* as output. Generally, *TProcItem* is an individual in the population and *TProcResult* is the individual's fitness. If additional arguments must be passed to the callable you want to use, it's possible to use Python's `functools.partial` or to just wrap it with a simple function.

Return type List[~TProcResult]

Returns A list containing the results of the processing of each item. It is guaranteed that the ordering of the items in the returned list follows the order in which the items are yielded by the iterable passed as argument.

Module contents

Imports core names of *nevopy.processing*.

6.1.5 nevopy.utils package

Subpackages

nevopy.utils.gym_utils package

Submodules

nevopy.utils.gym_utils.callbacks module

This module defines an interface for callbacks to be used with *GymFitnessFunction*.

class nevopy.utils.gym_utils.callbacks.**BatchObsGymCallback**
Bases: *nevopy.utils.gym_utils.callbacks.GymCallback*

Simple callback that expands the dimensions of the observations yielded by a `gym.Env` before feeding them to a genome.

Simply turns the observation into a batch of one item (the observation itself), so it can be fed to genomes that require batched inputs (like genomes that use TensorFlow, for example).

on_obs_processing(*wrapped_obs*)
Called right BEFORE the observation yielded by the environment is fed to the genome.

Changing the observation stored by the wrapper will have effects on the fitness function.

Subclasses should override this method for any actions to run.

Parameters **wrapped_obs** (*nevopy.utils.MutableWrapper[Any]*) – Mutable wrapper around he observation yielded by the gym environment.

Return type None

class nevopy.utils.gym_utils.callbacks.**GymCallback**
Bases: object

Interface for callbacks to be used with *GymFitnessFunction*.

Each of the callback's method is called at a different point during the evaluation of a genome's fitness by a *GymFitnessFunction*.

It's not required for a subclass to implement all the methods of this class (you can implement only those that will be useful for your case).

on_action_chosen(*wrapped_action*)
Called right AFTER an action is chosen by the genome.

Changing the action stored by the wrapper will have effects on the fitness function.

Subclasses should override this method for any actions to run.

Parameters **wrapped_action** (*nevopy.utils.MutableWrapper[Any]*) – Mutable wrapper around the action chosen by the genome.

Return type None

on_env_built(*env*, *genome*)
Called right AFTER the gym environment is built.

This method is called right after the gym environment is built, i.e., right after a call to gym.make() is made.

Parameters

- **env** (*gym.Env*) – The gym environment that's going to be used by the fitness function.
- **genome** (*nevopy.BaseGenome*) – The genome currently being evaluated by the fitness function.

Return type None

on_env_close()
Called right BEFORE the environment is closed and the function returns the fitness of the genome.

Subclasses should override this method for any actions to run.

Return type None

on_episode_start(*current_eps*, *total_eps*)
Called at the start of a new episode, before the env is reset.

Subclasses should override this method for any actions to run.

Parameters

- **current_eps** (*int*) – Number of the current episode.
- **total_eps** (*int*) – Total number of episodes to run during the current session.

Return type None

on_obs_processing(*wrapped_obs*)
Called right BEFORE the observation yielded by the environment is fed to the genome.

Changing the observation stored by the wrapper will have effects on the fitness function.

Subclasses should override this method for any actions to run.

Parameters **wrapped_obs** (*nevopy.utils.MutableWrapper[Any]*) – Mutable wrapper around he observation yielded by the gym environment.

Return type None

on_step_start(*current_step*, *max_steps*)
Called at the start of a new step.

Subclasses should override this method for any actions to run.

Parameters

- **current_step** (*int*) – Number of the current step.
- **max_steps** (*int*) – Maximum number of steps allowed in each episode.

Return type None

on_step_taken(*obs*, *reward*, *done*, *info*, *total_reward*, *force_stop_eps*)
Called right AFTER the environment's step() method is called.

Subclasses should override this method for any actions to run.

Parameters

- **obs** (*Any*) – The observation yielded by the environment.
- **reward** (*float*) – The reward yielded by the environment.
- **done** (*bool*) – Whether or not the episode has finished.
- **info** (*Dict[str, Any]*) – Extra information yielded by the environment.
- **total_reward** (*float*) – Total reward obtained by the genome so far.
- **force_stop_eps** (*nevopy.utils.MutableWrapper[bool]*) – Setting the value on this wrapper to *True* will forcefully stop the current episode.

Return type None

on_visualization()
Called right BEFORE the rendering of the environment occurs.

This method is only called when True is passed to the visualize parameter of GymFitnessFunction.__call__().

Subclasses should override this method for any actions to run.

Return type None

nevopy.utils.gym_utils.fitness_function module

This module implements a generalizable fitness function that can be used with most gym environments.

class nevopy.utils.gym_utils.fitness_function.**GymFitnessFunction**(*make_env*, *env_renderer=None*, *callbacks=None*, *default_num_episodes=1*, *default_max_steps=None*, *num_obs_skip=0*)

Bases: object

Wrapper for a fitness function to be used with gym.

This utility class implements a generalizable fitness function compatible with different gym environments.

Parameters

- **make_env** (*Callable[[], gym.Env]*) – Callable that creates the environment to be used. It should receive no arguments and return an instance of gym.Env.
- **env_renderer** (*Optional[GymRenderer]*) – Instance of *GymRenderer* (or a subclass) to be used to render the environment. By default, a new instance of *GymRenderer* is created (default rendering of the environment).
- **callbacks** (*Optional[List[GymCallback]]*) – List with callbacks to be called at different stages of the evaluation of the genome's fitness.
- **default_num_episodes** (*int*) – Default number of episodes ran in each call to the fitness function. This can be overridden during the call to the fitness function.
- **default_max_steps** (*Optional[int]*) – Default maximum number of steps allowed in each episode. By default, there is no limit to the number of steps. This can be overridden during the call to the fitness function.
- **num_obs_skip** (*int*) – Number of observations to be skipped during an episode. As an example, consider this value is set to 3. In this case, for each sequence of 4 observations yielded by the environment, only the 1st one will be fed to the genome. When, during a step, no observation is fed to the genome, the genome's last output is used to advance the environment's state.

env_renderer
Instance of *GymRenderer* (or a subclass) to be used to render the environment.

Type *GymRenderer*

callbacks
List with callbacks to be called at different stages of the evaluation of the genome's fitness.

Type List[*GymCallback*]

num_obs_skip
Number of observations to be skipped during an episode.

Type int

nevopy.utils.gym_utils.renderers module

This module implements the entities responsible for rendering a `gym.Env` during the evaluation of a genome's fitness by a *GymFitnessFunction*.

class nevopy.utils.gym_utils.renderers.**GymRenderer**(*fps=45*)

Bases: `object`

Defines the entity responsible for rendering a `gym.Env` during the evaluation of a genome's fitness by a *GymFitnessFunction*.

Parameters **fps** (*int*) – Frames per second.

fps

Frames per second.

Type int

flush()

Flushes the internal buffers of the renderer.

Doesn't do anything by default. Subclasses should override this method in order for any action to occur.

Return type `None`

render(*env*, *genome*)

Renders the environment in "human mode".

Parameters

- **env** (*gym.Env*) – Environment to be rendered.
- **genome** (BaseGenome) – Genome currently being evaluated.

Return type `None`

class nevopy.utils.gym_utils.renderers.**NeatActivationsGymRenderer**(*out_path='./gym_videos'*, *fps=30*, *play_video=True*, ***kwargs*)

Bases: *nevopy.utils.gym_utils.renderers.GymRenderer*

Gym env renderer that renders a NEAT genome's neural network while the genome is interacting with the environment.

Three videos will be generated: one containing the recording of the genome's interactions with the environment, another containing the genome's neural network states during the interactions and another containing the concatenation of the two previously videos, side by side.

Note: Compatible with *NeatGenome* only!

Note: This renderer requires you to have the `opencv-python` and `scikit-video` packages installed. You can install them using *pip*. You'll also need *FFmpeg*.

Parameters

- **out_path** (*str*) – Path to the output directory.
- **fps** (*int*) – Frames per second of the generated videos.

- **play_video** (*bool*) – If `True`, the concatenated videos will be automatically played after the rendering is done.
- ****kwargs** (*Dict[str, Any]*) – Named arguments to be passed to `genome.visualize_activations()`.

flush()
Generates the videos from the images in the cache, closes the necessary resources and clears the image cache.

Return type `None`

static play_video(*video_file*, *fps*)
Plays a video from a file. ?????

Return type `None`

render(*env*, *genome*)
Renders the environment in "human mode".

Parameters

- **env** (*gym.Env*) – Environment to be rendered.
- **genome** (`BaseGenome`) – Genome currently being evaluated.

Return type `None`

Module contents

Exposes the core functionalities of `gym_utils`.

Submodules

nevopy.utils.deprecation module

Implements a decorator that can be used to mark functions, methods or classes as being deprecated.

nevopy.utils.deprecation.**deprecated**(*decorated_item=None*, ***, *version=None*, *instructions=None*, *warn_once=True*)
Decorator for marking functions, methods or classes deprecated.

This decorator logs a deprecation warning whenever the decorated item is called. It has the following format:

> From {call info}: {function/method/class} (from {module}) is deprecated and will be removed in the future. Instructions for updating: {instructions}

The field {function/method/class} will contain:

- the function's name, if the decorated item is a function;
- the method's name and the method's class name, if the decorated item is a method;
- the class' name, if the decorated item is a class.

This decorator also edits the docstring of the decorated item. A deprecation notice is added to the start of the docstring.

Parameters

- **decorated_item** (*Optional[Any]*) – The item being decorated. Having this parameter allows the decorator to be used without arguments.

- **instructions** (*Optional[str]*) – Instructions on how to update the code using the deprecated item.
- **version** (*Optional[str]*) – Version in which the item was deprecated.
- **warn_once** (*bool*) – If *True*, a warning will be printed only the first time the decorated item is called. Otherwise, every call will log a warning.

Return type Any

Returns The decorated function, method or class.

nevopy.utils.utils module

This module implements useful utility functions.

class nevopy.utils.utils.**Comparable**
Bases: object

Indication of a "comparable" type.

class nevopy.utils.utils.**MutableWrapper**(*value*)
Bases: Generic[typing._T]

Simple class for wrapping immutable objects so they can be passed by reference to a callable.

nevopy.utils.utils.**align_lists**(*lists*, *getkey=None*, *placeholder=None*)
Aligns the given lists based on their common values.

Repeated entries within a single list are discarded.

Example

```
>>> align_lists(([1, 2, 3, 6], [1, 3, 4, 5]))
[[1, 2, 3, None, None, 6], [1, None, 3, 4, 5, None]]
```

Parameters

- **lists** (*Iterable[List[_T]]*) – Iterable that yields lists containing the objects to be aligned.
- **getkey** (*Optional[Callable[[_T], Comparable]]*) – Optional function to be passed to sorted() to retrieve comparable keys from the objects to be aligned.
- **placeholder** (*Optional[Any]*) – Value to be used as a placeholder when an item doesn't match with any other (see the example above, where *None* is the placeholder).

Return type List[List[~_T]]

Returns A list containing the aligned lists. The items in the aligning lists will be sorted in ascending order.

nevopy.utils.utils.**chance**(*p*)
Randomly returns *True* or *False*.

Parameters **p** (*float*) – Float between 0 and 1. Specifies the chance of the function returning True.

Return type bool

Returns A randomly chosen boolean value (*True* or *False*).

nevopy.utils.utils.**clear_output**()
Clears the output.

Should work on Windows and Linux terminals and on Jupyter notebooks. On PyCharm, it simply prints a bunch of new lines.

Return type None

nevopy.utils.utils.**is_jupyter_notebook**()
Checks whether the program is running on a jupyter notebook.

> **Warning:** This function is not guaranteed to work! It simply checks if `IPython.get_ipython()` returns *None*.

Return type bool

Returns *True* if the program is running on a jupyter notebook and *False* otherwise.

nevopy.utils.utils.**make_table_row**(*name*, *current*, *past*, *abs_format='.2E'*, *inc_format='+0.2E'*, *pc_format='+0.2%'*, *show_inc_pc=True*, *colors=True*, *positive_color='green'*, *negative_color='red'*, *neutral_color='white'*)
Makes a row for a *columnar* table.

Information in the row: name of the attribute; current value of the attribute; past value of the attribute; how much the attribute increased (absolute and percentage).

Return type List[str]

nevopy.utils.utils.**make_xor_data**(*num_variables=2*)
Builds data using the *XOR* logic function.

The generated inputs are all the possible combinations of input values with the specified number of variables. Each variable is a bit (0 or 1). The generated outputs are the results (a single bit each) of the *XOR* function applied to all the inputs.

Example

```
>>> xor_in, xor_out = make_xor_data(num_variables=2)
>>> for x, y in zip(xor_in, xor_out):
...     print(f"{x} -> {y}")
...
[0 0] -> 0
[0 1] -> 1
[1 0] -> 1
[1 1] -> 0
```

Parameters **num_variables** (*int*) – Number of input variables for the *XOR* logic function.

Return type Tuple[ndarray, ndarray]

Returns A tuple with two numpy arrays. The first array contains the input values, and the second array contains the output of the *XOR* function.

nevopy.utils.utils.**min_max_norm**(*values*)
Applies min-max normalization to the given values.

Return type array

nevopy.utils.utils.**pickle_load**(*abs_path*)
Loads an object from the given absolute path.

Simple wrapper around the *pickle* package.

Parameters **abs_path** (*str*) – Absolute path of the saved ".pkl" file. If the given path doesn't end with the suffix ".pkl", it will be automatically added.

Return type Any

Returns The loaded object.

nevopy.utils.utils.**pickle_save**(*obj*, *abs_path*)
Saves the given object to the given absolute path.

Simple wrapper around the *pickle* package.

Parameters

- **obj** (*Any*) – Object to be saved.
- **abs_path** (*str*) – Absolute path of the saving file. If the given path doesn't end with the suffix ".pkl", it will be automatically added.

Return type None

nevopy.utils.utils.**rank_prob_dist**(*size*, *coefficient*, *min_prob=1e-09*)
Calculates a probability distribution that associates a probability to each position of a rank with the given size.

Parameters

- **size** (*int*) – Size of the rank (and of the probability distribution).
- **coefficient** (*float*) – This constant (let's call it *c*) can be interpreted as follows: the position *p* of the rank is assigned a probability that is *c* times higher than the position *p + 1* of the rank. If c = 2, here is an example of a probability distribution generated by this function: *[0.5, 0.25, 0.125, 0.0675, ...]*.
- **min_prob** (*float*) – Probabilities with a value lower than the value passed to this parameter will be converted to 0. This prevents the occurrence of an arithmetic underflow.

Return type ndarray

Returns A numpy array with the probability distribution. The value in the index *i* of the array represents the probability of the position *i* of the rank.

nevopy.utils.utils.**round_proportional_distribution**(*to_distribute*, *values*)
Given an integer *A* and a sequence *S* of arbitrary size *k*, this function divides *A* into *k* integers. The proportion that the *i-th* integer represents of *A* is approximately equal to the proportion that *S[i]* represents of *sum(S[i])*.

Example

```
>>> round_proportional_distribution(100, [1.22, 2.78, 0.26, 5.74])
[12, 28, 3, 57]
```

Parameters

- **to_distribute** (*int*) – Integer to be distributed.
- **values** (*Sequence[float]*) – Values that will serve as a reference.

Return type List[int]

Returns A list with the same size as `values` containing integers that sum to `to_distribute`.

Module contents

Exposes the main utility functions and classes within this package.

6.2 Submodules

6.3 nevopy.activations module

This module implements some activation functions.

Todo: Make all activation functions compatible with numpy arrays.

nevopy.activations.**linear**(*x*)
Linear activation function (simply returns the input, unchanged).

Return type float

nevopy.activations.**sigmoid**(*x*, *clip_value=64*)
Numeric stable implementation of the sigmoid function.

Estimated lower-bound precision with a clip value of 64: 10^(-28).

Return type float

nevopy.activations.**steepened_sigmoid**(*x*, *step=4.9*)
Steepened version of the sigmoid function.

The original NEAT paper used a steepened version of the sigmoid function with a step value of 4.9.

"We used a modified sigmoidal transfer function, (x) = 1 / (1 + exp(4.9x)), at all nodes. The steepened sigmoid allows more fine tuning at extreme activations. It is optimized to be close to linear during its steepest ascent between activations 0.5 and 0.5." - [SM02]

Return type float

6.4 nevopy.base_genome module

Declares the base abstract class that defines the behaviour of a genome.

In the context of neuroevolution, a genome is the entity subject to the evolutionary process. It encodes a neural network (the genome's phenotype), either directly or indirectly.

This module declares the base abstract class that must be inherited by all the different classes of genomes used by the neuroevolutionary algorithms in *NEvoPy*.

class nevopy.base_genome.**BaseGenome**
Bases: abc.ABC

Defines the general behaviour of a genome in *NEvoPy*.

This class must be inherited by all the different classes of genomes present in *NEvoPy*

In the context of neuroevolution, a genome is the entity subject to the evolutionary process. It encodes a neural network (the genome's phenotype), either directly or indirectly.

As pointed out by [SM02], direct encoding schemes, employed in most cases, specify in the genome every connection and node that will appear in the phenotype. In contrast, indirect encodings usually only specify rules for constructing a phenotype. These rules can be layer specifications or growth rules through cell division.

One of the goals of this base abstract class is to abstract those details for the user, defining a general interface for the different types of genomes used by the different neuroevolutionary algorithms in *NEvoPy*. Generally, for *NEvoPy*, there is no distinction between a genome and the network it encodes.

A genome must be capable of processing inputs based on its nodes and connections in order to produce an output, emulating a neural network. It also must be able to mutate and to generate offspring, in order to evolve.

fitness
The current fitness value of the genome.

Type float

abstract property config
Settings of the current evolutionary session.

If *None*, a config object hasn't been assigned to this genome yet.

Return type `Any`

abstract deep_copy()
Makes an exact/deep copy of the genome.

Return type *BaseGenome*

Returns An exact/deep copy of the genome. It has the same topology and connections weights of the original genome.

abstract distance(*other*)
Calculates the distance between two genomes.

Parameters **other** (`BaseGenome`) – The other genome.

Return type `float`

Returns A float representing the distance between the two genomes. The lower the distance, the more similar the two genomes are.

abstract property input_shape
The expected shape of the inputs that will be fed to the genome.

Return type `Union[int, Tuple[int, ...], None]`

Returns

- `None`, if an input shape has not been defined yet;
- An `int`, if the expected inputs are one-dimensional;
- A `tuple` with the expected inputs' dimensions, if they're multi-dimensional.

classmethod load(*abs_path*)
Loads the genome from the given absolute path.

This method uses, by default, `pickle` to load the genome.

Parameters **abs_path** (*str*) – Absolute path of the saved ".pkl" file.

Return type *BaseGenome*

Returns The loaded genome.

abstract mate(*other*)
Mates two genomes to produce a new genome (offspring).

Implements the sexual reproduction between a pair of genomes. The new genome inherits information from both parents (not necessarily in an equal proportion)

Parameters **other** (*Any*) – The second genome. If it's not compatible for mating with the current genome (*self*), an exception will be raised.

Return type *BaseGenome*

Returns A new genome (the offspring born from the sexual reproduction between the current genome and the genome passed as argument).

Raises ***IncompatibleGenomesError*** – If the genome passed as argument to `other` is incompatible with the current genome (*self*).

abstract mutate_weights()
Randomly mutates the weights of the genome's connections.

Return type `None`

abstract process(*x*)
Feeds the given input to the neural network encoded by the genome.

Parameters **x** (*Any*) – The input(s) to be fed to the neural network encoded by the genome. Usually a *NumPy ndarray* or a *TensorFlow tensor*.

Return type `Any`

Returns The output of the network. Usually a *NumPy ndarray* or a *TensorFlow tensor*.

Raises ***InvalidInputError*** – If the shape of `X` doesn't match the input shape expected by the network.

abstract random_copy()
Makes a deep copy of the genome, but with random weights.

Return type *BaseGenome*

Returns A deep copy of the genome with the same topology of the original genome, but random connections weights.

abstract reset()
Prepares the genome for a new generation.

In this method, relevant actions related to the reset of a genome's internal state, in order to prepare it to a new generation, are implemented. The implementation of this method is not mandatory.

Return type `None`

save(*abs_path*)
Saves the genome to the given absolute path.

This method uses, by default, `pickle` to save the genome.

Parameters **abs_path** (*str*) – Absolute path of the saving file. If the given path doesn't end with the suffix ".pkl", it will be automatically added.

Return type `None`

abstract visualize(***kwargs*)
Utility method for visualizing the genome's neural network.

Return type `None`

exception nevopy.base_genome.**IncompatibleGenomesError**
Bases: Exception

Indicates that an attempt has been made to mate (sexual reproduction) two incompatible genomes.

exception nevopy.base_genome.**InvalidInputError**
Bases: Exception

Indicates that the a given input isn't compatible with a given neural network.

6.5 nevopy.base_population module

Implementation of the base abstract class that defines a population of genomes, each of which encodes a neural network.

class nevopy.base_population.**BasePopulation**(*size*, *processing_scheduler*)
Bases: abc.ABC, Generic[typing.TGenome]

Base abstract class that defines a population of genomes (neural nets).

This base abstract class defines a population of genomes (each of which encodes a neural network) to be evolved through neuroevolution. It's in this class' subclasses where the core of *NEvoPy's* neuroevolutionary algorithms are implemented.

Parameters

- **size** (*int*) – Number of genomes (constant) in the population.
- **processing_scheduler** (ProcessingScheduler) – Processing scheduler to be used by the population.

scheduler
Processing scheduler used by the population. It's responsible for abstracting the details on how the processing is done (whether it's sequential or distributed, local or networked, etc).

Type *ProcessingScheduler*

genomes
List with the genomes (neural networks being evolved) currently in the population.

Type List[TGenome]

stop_evolving
Flag that when set to *True* stops the evolutionary process being executed by the *evolve()* method.

Type bool

DEFAULT_SCHEDULER = None
Default processing scheduler to be used by the population.

average_fitness()
Returns the average fitness of the population's genomes.

Return type float

abstract property config
Config object that stores the settings used by the population.

Return type Any

abstract evolve(*generations*, *fitness_function*, *callbacks=None*, ***kwargs*)
Evolves the population of genomes through neuroevolution.

This is the main method of this class. It's here where the main loop of the neuroevolutionary algorithm implemented is located.

Parameters

- **generations** (*int*) – Maximum number of evolutionary generations.
- **fitness_function** (*Callable[[TGenome], float]*) – Fitness function used to compute the fitness of a genome. It must take an instance of class:.*BaseGenome* as input and return the genome's fitness (float).
- **callbacks** (*Optional[List["ne.callbacks.Callback"]]*) – List with instances of *Callback*. The callbacks will be called during different stages of an evolutionary generation. They can be used to customize the algorithm's behaviour.

Return type *History*

Returns An instance of *nevopy.callbacks.History* containing relevant information about the evolutionary session.

fittest()
Returns the most fit genome in the population.

Return type ~TGenome

classmethod load(*abs_path*, *scheduler=None*)
Loads the population from the given absolute path.

This method uses, by default, `pickle` to load the population.

Parameters

- **abs_path** (*str*) – Absolute path of the saved ".pkl" file.
- **scheduler** (*Optional[ProcessingScheduler]*) – Processing scheduler to be used by the population. If *None*, the default one will be used.

Return type *BasePopulation*

Returns The loaded population.

save(*abs_path*)
Saves the population on the absolute path provided.

This method uses, by default, `pickle` to save the population. The processing scheduler used by the population won't be saved (a new one will have to be assigned to the population when it's loaded again).

Parameters **abs_path** (*str*) – Absolute path of the saving file. If the given path doesn't end with the suffix ".pkl", it will be automatically added to it.

Return type `None`

property size
Size of the population.

Return type `int`

6.6 nevopy.callbacks module

Defines a base interface for all callbacks and implements simple callbacks.

For NEvoPy, callbacks are utilities called at certain points during the evolution of a population. They are a powerful tool to customize the behavior of a neuroevolutionary algorithm.

Example

To implement your own callback, simply create a class that inherits from *Callback* and pass an instance of it to *BasePopulation.evolve()*.

```
class MyCallback(Callback):
    def on_generation_start(self,
                            current_generation,
                            max_generations):
        print("This is printed at the start of every generation!")
        print(f"Starting generation {current_generation} of "
              f"{max_generations}.")

# ...

population.evolve(generations=100,
                  fitness_function=my_func,
                  callbacks=[MyCallback()])
```

class nevopy.callbacks.**BestGenomeCheckpoint**(*output_path=None*, *min_improvement_pc=-inf*)

Bases: *nevopy.callbacks.Callback*

Saves the best genome of the population (checkpoint) at different moments of the evolutionary process.

Parameters

- **output_path** (*Optional[str]*) – Path of the output files. By default, the checkpoints are saved to a new directory named according to the current date and time.
- **min_improvement_pc** (*float*) – Minimum improvement (percentage) in the population's best fitness, since the last checkpoint, necessary for a new checkpoint. If `float('-inf')` (default) the best genomes of all generations will be saved.

output_path
Path of the output files.

Type str

min_improvement_pc
Minimum improvement (percentage) in the population's best fitness, since the last checkpoint, necessary for a new checkpoint.

Type float

on_fitness_calculated(*best_fitness*, *avg_fitness*, ***kwargs*)
Called right after the fitness values of the population's genomes are calculated.

Subclasses should override this method for any actions to run.

Parameters

- **best_fitness** (*float*) – Fitness of the fittest genome in the population.

- **avg_fitness** (*float*) – Average fitness of the population's genomes.

Return type None

on_generation_start(*current_generation*, *max_generations*, ***kwargs*)
Called at the beginning of each new generation.

Subclasses should override this method for any actions to run.

Parameters

- **current_generation** (*int*) – Number of the current generation.
- **max_generations** (*int*) – Maximum number of generations.

Return type None

class nevopy.callbacks.**Callback**
Bases: abc.ABC

Abstract base class used to build new callbacks.

This class defines the general structure of the callbacks used by *NEvoPy's* neuroevolutionary algorithms. It's not required for a subclass to implement all the methods of this class (you can implement only those that will be useful for your case).

population
Reference to the instance of a subclass of Population being evolved by one of *NEvoPy's* neuroevolutionary algorithms.

Type *BasePopulation*

on_evolution_end(*total_generations*, ***kwargs*)
Called when the evolutionary process ends.

Parameters **total_generations** (*int*) – Total number of generations processed during the evolutionary process. Might not be the maximum number of generations specified by the user, if some sort of early stopping occurs.

Return type None

on_fitness_calculated(*best_fitness*, *avg_fitness*, ***kwargs*)
Called right after the fitness values of the population's genomes are calculated.

Subclasses should override this method for any actions to run.

Parameters

- **best_fitness** (*float*) – Fitness of the fittest genome in the population.
- **avg_fitness** (*float*) – Average fitness of the population's genomes.

Return type None

on_generation_end(*current_generation*, *max_generations*, ***kwargs*)
Called at the end of each generation.

Subclasses should override this method for any actions to run.

Parameters

- **current_generation** (*int*) – Number of the current generation.
- **max_generations** (*int*) – Maximum number of generations.

Return type None

on_generation_start(*current_generation*, *max_generations*, ***kwargs*)
Called at the beginning of each new generation.

Subclasses should override this method for any actions to run.

Parameters

- **current_generation** (*int*) – Number of the current generation.
- **max_generations** (*int*) – Maximum number of generations.

Return type None

on_mass_extinction_counter_updated(*mass_extinction_counter*, ***kwargs*)
Called right after the mass extinction counter is updated.

Subclasses should override this method for any actions to run.

Parameters **mass_extinction_counter** (*int*) – Current value of the mass extinction counter.

Return type None

on_mass_extinction_start(***kwargs*)
Called at the beginning of a mass extinction event.

Subclasses should override this method for any actions to run.

Note: When this is called, *on_reproduction_start()* is usually not called (depends on the algorithm).

Return type None

on_reproduction_start(***kwargs*)
Called at the beginning of the reproductive process.

Subclasses should override this method for any actions to run.

Note: When this is called, *on_mass_extinction_start()* is usually not called (depends on the algorithm).

Return type None

on_speciation_start(***kwargs*)
Called at the beginning of the speciation process.

Called after the reproduction or mass extinction have occurred and immediately before the speciation process. If the neuroevolution algorithm doesn't implement speciation, this method won't be called.

Subclasses should override this method for any actions to run.

Return type None

class nevopy.callbacks.**CompleteStdOutLogger**(*colored_text=True*, *output_cleaner=<function clear_output>*)
Bases: *nevopy.callbacks.Callback*

Callback that prints info to the standard output.

Note: This callback is heavily verbose / wordy! Consider using the reduced logger (*SimpleStdOutLogger*) if you don't like too much text on your screen.

SEP_SIZE = 80

TAB_ARGS = {'justify': 'c', 'min_column_width': 14, 'no_borders': False}

on_evolution_end(*total_generations*, ***kwargs*)
Called when the evolutionary process ends.

Parameters **total_generations** (*int*) – Total number of generations processed during the evolutionary process. Might not be the maximum number of generations specified by the user, if some sort of early stopping occurs.

Return type None

on_fitness_calculated(*best_fitness*, *avg_fitness*, ***kwargs*)
Called right after the fitness values of the population's genomes are calculated.

Subclasses should override this method for any actions to run.

Parameters

- **best_fitness** (*float*) – Fitness of the fittest genome in the population.
- **avg_fitness** (*float*) – Average fitness of the population's genomes.

Return type None

on_generation_end(*current_generation*, *max_generations*, ***kwargs*)
Called at the end of each generation.

Subclasses should override this method for any actions to run.

Parameters

- **current_generation** (*int*) – Number of the current generation.
- **max_generations** (*int*) – Maximum number of generations.

Return type None

on_generation_start(*current_generation*, *max_generations*, ***kwargs*)
Called at the beginning of each new generation.

Subclasses should override this method for any actions to run.

Parameters

- **current_generation** (*int*) – Number of the current generation.
- **max_generations** (*int*) – Maximum number of generations.

Return type None

on_mass_extinction_counter_updated(*mass_extinction_counter*, ***kwargs*)
Called right after the mass extinction counter is updated.

Subclasses should override this method for any actions to run.

Parameters **mass_extinction_counter** (*int*) – Current value of the mass extinction counter.

Return type None

on_mass_extinction_start(***kwargs*)
Called at the beginning of a mass extinction event.

Subclasses should override this method for any actions to run.

Note: When this is called, *on_reproduction_start()* is usually not called (depends on the algorithm).

Return type None

on_reproduction_start(***kwargs*)
Called at the beginning of the reproductive process.

Subclasses should override this method for any actions to run.

Note: When this is called, *on_mass_extinction_start()* is usually not called (depends on the algorithm).

Return type None

on_speciation_start(***kwargs*)
Called at the beginning of the speciation process.

Called after the reproduction or mass extinction have occurred and immediately before the speciation process. If the neuroevolution algorithm doesn't implement speciation, this method won't be called.

Subclasses should override this method for any actions to run.

Return type None

class nevopy.callbacks.**FitnessEarlyStopping**(*fitness_threshold*, *min_consecutive_generations*)
Bases: *nevopy.callbacks.Callback*

Stops the evolution if a given fitness value is achieved.

This callback is used to halt the evolutionary process when a certain fitness value is achieved by the population's best genome for a given number of consecutive generations.

Parameters

- **fitness_threshold** (*float*) – Fitness to be achieved for the evolution to stop.
- **min_consecutive_generations** (*int*) – Number of consecutive generations with a fitness equal or higher than fitness_threshold for the early stopping to occur.

fitness_threshold
Fitness to be achieved for the evolution to stop.

Type float

min_consecutive_generations
Number of consecutive generations with a fitness equal or higher than fitness_threshold for the early stopping to occur.

Type int

stopped_generation
Generation in which the early stopping occurred. *None* if the early stopping never occurred.

Type Optional[int]

on_fitness_calculated(*best_fitness*, *avg_fitness*, ***kwargs*)
Called right after the fitness values of the population's genomes are calculated.

Subclasses should override this method for any actions to run.

Parameters

- **best_fitness** (*float*) – Fitness of the fittest genome in the population.
- **avg_fitness** (*float*) – Average fitness of the population's genomes.

Return type None

on_generation_end(*current_generation*, *max_generations*, ***kwargs*)
Called at the end of each generation.

Subclasses should override this method for any actions to run.

Parameters

- **current_generation** (*int*) – Number of the current generation.
- **max_generations** (*int*) – Maximum number of generations.

Return type None

class nevopy.callbacks.**History**
Bases: *nevopy.callbacks.Callback*

Callback that records events during the evolutionary process.

Besides the regular attributes in the methods signature, the caller can also pass other attributes through "kwargs". All the attributes passed to the methods will have they value stored in the *history* dictionary.

history
Dictionary that maps an attribute's name to a list with the attribute's values along the evolutionary process.

Type Dict[str, List[Any]]

on_evolution_end(*total_generations*, ***kwargs*)
Called when the evolutionary process ends.

Parameters **total_generations** (*int*) – Total number of generations processed during the evolutionary process. Might not be the maximum number of generations specified by the user, if some sort of early stopping occurs.

Return type None

on_fitness_calculated(*best_fitness*, *avg_fitness*, ***kwargs*)
Called right after the fitness values of the population's genomes are calculated.

Subclasses should override this method for any actions to run.

Parameters

- **best_fitness** (*float*) – Fitness of the fittest genome in the population.
- **avg_fitness** (*float*) – Average fitness of the population's genomes.

Return type None

on_generation_end(*current_generation*, *max_generations*, ***kwargs*)
Called at the end of each generation.

Subclasses should override this method for any actions to run.

Parameters

- **current_generation** (*int*) – Number of the current generation.
- **max_generations** (*int*) – Maximum number of generations.

Return type None

on_generation_start(*current_generation*, *max_generations*, ***kwargs*)
Called at the beginning of each new generation.

Subclasses should override this method for any actions to run.

Parameters

- **current_generation** (*int*) – Number of the current generation.
- **max_generations** (*int*) – Maximum number of generations.

Return type None

on_mass_extinction_counter_updated(*mass_extinction_counter*, ***kwargs*)
Called right after the mass extinction counter is updated.

Subclasses should override this method for any actions to run.

Parameters **mass_extinction_counter** (*int*) – Current value of the mass extinction counter.

Return type None

on_mass_extinction_start(***kwargs*)
Called at the beginning of a mass extinction event.

Subclasses should override this method for any actions to run.

Note: When this is called, *on_reproduction_start()* is usually not called (depends on the algorithm).

Return type None

on_reproduction_start(***kwargs*)
Called at the beginning of the reproductive process.

Subclasses should override this method for any actions to run.

Note: When this is called, *on_mass_extinction_start()* is usually not called (depends on the algorithm).

Return type None

on_speciation_start(***kwargs*)
Called at the beginning of the speciation process.

Called after the reproduction or mass extinction have occurred and immediately before the speciation process. If the neuroevolution algorithm doesn't implement speciation, this method won't be called.

Subclasses should override this method for any actions to run.

Return type None

visualize(*attrs=('best_fitness', 'avg_fitness')*, *figsize=(10, 6)*, *log_scale=True*)
Simple utility method for plotting the recorded information.

This method is a simple wrapper around *matplotlib*. It isn't suited for advanced plotting.

attrs
Tuple with the names of the attributes to be plotted. If "all", all plottable attributes are plotted.
Type Union[Tuple[str, ..], str]

log_scale
Whether or not to use a logarithmic scale on the y-axis.
Type bool

Return type None

class nevopy.callbacks.**SimpleStdOutLogger**
Bases: *nevopy.callbacks.Callback*

Callback that prints minimal info to the standard output.

on_evolution_end(*total_generations*, ***kwargs*)
Called when the evolutionary process ends.

Parameters **total_generations** (*int*) – Total number of generations processed during the evolutionary process. Might not be the maximum number of generations specified by the user, if some sort of early stopping occurs.

Return type None

on_fitness_calculated(*best_fitness*, *avg_fitness*, ***kwargs*)
Called right after the fitness values of the population's genomes are calculated.

Subclasses should override this method for any actions to run.

Parameters

- **best_fitness** (*float*) – Fitness of the fittest genome in the population.
- **avg_fitness** (*float*) – Average fitness of the population's genomes.

Return type None

on_generation_start(*current_generation*, *max_generations*, ***kwargs*)
Called at the beginning of each new generation.

Subclasses should override this method for any actions to run.

Parameters

- **current_generation** (*int*) – Number of the current generation.
- **max_generations** (*int*) – Maximum number of generations.

Return type None

on_mass_extinction_counter_updated(*mass_extinction_counter*, ***kwargs*)
Called right after the mass extinction counter is updated.

Subclasses should override this method for any actions to run.

Parameters **mass_extinction_counter** (*int*) – Current value of the mass extinction counter.

Return type None

on_mass_extinction_start(***kwargs*)
Called at the beginning of a mass extinction event.

Subclasses should override this method for any actions to run.

Note: When this is called, `on_reproduction_start()` is usually not called (depends on the algorithm).

Return type `None`

6.7 Module contents

Imports the core names of NEvoPy.

CHAPTER

SEVEN

BIBLIOGRAPHY

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[SM02] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. URL: http://nn.cs.utexas.edu/?stanley:ec02.

PYTHON MODULE INDEX

n

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W