Holland Documentation

Release 0.0.1

Jesse Jenks and Henry Woody

Oct 13, 2018

Contents

Holland is an open source Python package for running genetic algorithms. Holland has been designed to allow arbitrary user-defined genome encodings and fitness functions.

Holland is modular enough to allow users to pick and choose submodules to meet specific optimization.

CHAPTER 1

Basic Usage

1.1 First example

Getting started with Holland is easy. Simply define your genome encoding and fitness function to get started

```
from holland import evolve
from math import cos, pi

# specify hyper-parameters for genomes
genome_parameters = {
    "gene1": {"type": "float", "min": -pi, "max": pi},
    "gene2": {"type": "float", "min": -pi, "max": pi},
}

# define a fitness function
def my_fitness_function(individual):
    return cos(inidividual.gene1) * cos(individual.gene2)


# evolve!
my_population = evolve(
    genome_parameters,
    fitness_function=my_fitness_function,
    show_fitness_plot=True,
    num_generations=100,
)
```

1.2 Intermediate Example: TSP

```
from holland import evolve
from math import sqrt

# list of cities and positions
cities = {"AZ": (1, 2), "CA": (3, 4), "NM": (5, 6), "TX": (7, 8)}

# specify hyper-parameters for genomes
# in this case there is only a single gene
genome_parameters = {
    "path": {"type": "[string]", "possible_values": cities.keys(), "mutation_function
↪": "swap"}
}


def distance(p1, p2):
    dx = p1[0] - p2[0]
    dy = p1[1] - p2[1]
    return sqrt(dx * dx + dy * dy)


# define a fitness function
# a pythonic way to find the length of a round trip
def sum_of_distances(individual):
    cities = [position[city] for city in individual["path"]]
    return sum(
        [distance(city_1, city_2) for (city_1, city_2) in zip(cities, cities[1:] +␣
↪[cities[0]])]
    )


# evolve!
my_population = evolve(
    genome_parameters,
    fitness_function=my_fitness_function,
    anneal_mutation_rate=True,
    show_fitness_plot=True,
    num_generations=100,
)
```

CHAPTER 2

Holland Reference

This page provides the core documentation reference for Holland.

2.1 Evolution

2.1.1 evolution

class holland.evolution.**Evolver**(*fitness_function*, *genome_params*, *selection_strategy*, *should_maximize_fitness=True*)

Handles evolution for a population

Parameters

- **fitness_function** (*function*) – the fitness function used to evaluate individuals; see *Fitness Function*
- **genome_params** (*dict*) – a dictionary specifying genome parameters; see *Genome Parameters*
- **selection_strategy** (*dict*) – a dictionary specifying selection parameters; see *Selection Strategy*
- **should_maximize_fitness** (*bool*) – whether fitness should be maximized or minimized

evolve(*generation_params={}*, *initial_population=None*, *stop_conditions={'n_generations': 100, 'target_fitness': inf}*, *storage_options={}*, *logging_options={'format': '%(message)s', 'level': 20}*)

The heart of Holland.

Parameters

- **generation_params** (*dict*) – a dictionary specifying how to create each generation; see *Generation Parameters*
- **initial_population** (*list*) – an initial population
- **stop_conditions** (*dict*) – conditions for stopping execution; will stop if *any* of the conditions is met; see Stop Conditions below
- **storage_options** (*dict*) – configuration options for storing fitness and genomes (should contain keys `"fitness"` and `"genomes"`); see *Fitness Storage Options* and *Genome Storage Options*
- **logging_options** (*dict*) – options for logging passed to logging.basicConfig as `kwargs`

Stop Conditions

- **n_generations** (*int*) – the number of generations to run evolution over
- **target_fitness** (*int*) – the target fitness score, will stop once the fittest individual reaches this score

Returns

- a list of fitness scores and genomes `[(fitness, genome), ...]` (fitness results); or
- a tuple of fitness results (previous bullet) and list of historical fitness statistics `(fitness_results, fitness_history)`, if `storage_options["fitness"]` has `'should_record_fitness': True` and `'format': 'memory'`

Raises

- **ValueError** – if `generation_params["n_random"] < 0` or `generation_params["n_elite"] < 0`
- **ValueError** – if `population_size < 1`
- **ValueError** – if `n_generations < 1`

Todo: If an initial population is given but does not match the given genome parameters, some kind of error should be raised

---

Todo: If an initial population is given and some genomes are missing parameters, a warning is given unless a flag is set to fill those values randomly

---

Dependencies:

- *generate_random_genomes()*
- *evaluate_fitness()*
- *generate_next_generation()*
- *update_storage()*
- *react_to_interruption()*

Example:

```
from holland import evolve
from math import cos, pi

# specify hyper-parameters for genomes
genome_parameters = {
    "gene1": {"type": "float", "min": -pi, "max": pi},
    "gene2": {"type": "float", "min": -pi, "max": pi},
}

# define a fitness function
def my_fitness_function(individual):
    return cos(inidividual.gene1) * cos(individual.gene2)


# evolve!
my_population = evolve(
    genome_parameters,
    fitness_function=my_fitness_function,
    show_fitness_plot=True,
    num_generations=100,
)
```

2.1.2 evaluation

class holland.evolution.**Evaluator**(*fitness_function*, *ascending=True*)

Handles evaluation of genomes

Parameters

- **fitness_function** (*func*) – a function for evaluating the fitness of each genome; see *Fitness Function*
- **ascending** (*bool*) – whether or not to sort results in ascending order of fitness

evaluate_fitness(*gene_pool*)
Evaluates the fitness of a population by applying a fitness function to each genome in the population

Parameters **gene_pool** (*list*) – a population of genomes to evaluate

Returns a sorted list of tuples of the form `(score, genome)`.

2.1.3 breeding

class holland.evolution.**PopulationGenerator**(*genome_params*, *selection_strategy*, *generation_params={}*)
Handles generating populations

Parameters

- **genome_params** (*dict*) – a dictionary specifying genome parameters; see *Genome Parameters*
- **selection_strategy** (*dict*) – a dictionary specifying selection parameters; see *Selection Strategy*
- **generation_params** (*dict*) – a dictionary specifying how to create the next generation; see *Generation Parameters*

Raises

- **ValueError** – if `n_random < 0` or `n_elite < 0`
- **ValueError** – if `n_random + n_elite > population_size`

generate_next_generation(*fitness_results*)
Generates the next generation

Parameters **fitness_results** (*list*) – a sorted list of tuples containing a fitness score in the first position and a genome in the second (returned by *evaluate_fitness()*)

Returns a list of genomes

Note: For the sake of efficiency, this method expects `fitness_results` to be sorted in order to properly select genomes on the basis of fitness. *evaluate_fitness()* returns sorted results.

Todo: Write an example for usage

Raises **ValueError** – if `n_random + n_elite > population_size`

Dependencies:

- *breed_next_generation()*
- *generate_random_genomes()*

breed_next_generation(*fitness_results*, *n_genomes*)
Generates a given number of genomes by breeding, through crossover and mutation, existing genomes

Parameters

- **fitness_results** (*list*) – a sorted list of tuples containing a fitness score in the first position and a genome in the second (returned by *evaluate_fitness()*)
- **n_genomes** (*int*) – the number of genomes to produce

Returns a list of bred genomes

Raises **ValueError** – if `n_genomes < 0`

Note: For the sake of efficiency, this method expects `fitness_results` to be sorted in order to properly select genomes on the basis of fitness. *evaluate_fitness()* returns sorted results.

Todo: Write an example for usage

Dependencies:

- *select_breeding_pool()*
- *select_parents()*
- *cross_genomes()*
- *mutate_genome()*

generate_random_genomes(*n_genomes*)

Generates a given number of genomes based on genome parameters

Parameters **n_genomes** (*int*) – the number of genomes to produce

Returns a list of randomly generated genomes

Raises **ValueError** – if `n_genomes < 0`

Todo: Write an example for usage

Dependencies:

- *bound_value()*

2.1.4 selection

class holland.evolution.**Selector**(*selection_strategy={}*)

Handles selection of genomes for breeding

Parameters **selection_strategy** – parameters for selecting a breeding pool and sets of parents; see *Selection Strategy*

Raises

- **ValueError** – if any of `top`, `mid`, `bottom`, or `random` is negative
- **ValueError** – if `n_parents < 1`

select_breeding_pool(*fitness_results*)

Selects a pool of genomes from a population from which to draw parents for breeding the next generation

Parameters **fitness_results** (*list*) – a sorted list of tuples containing a fitness score in the first position and a genome in the second (returned by *evaluate_fitness()*)

Returns a list of tuples of the form `(score, genome)` (same format as `fitness_results`)

Raises **ValueError** – if `len(fitness_results) < self.top + self.mid + self.bottom + self.random`

Note: For the sake of efficiency, this method expects `fitness_results` to be sorted in order to properly select genomes on the basis of fitness. *evaluate_fitness()* returns sorted results.

Dependencies:

- *select_from()*

select_parents(*fitness_results*)

Selects parents from the given `fitness_results` to use for breeding a new genome

Parameters **fitness_results** (*list*) – a (not necessarily sorted list of tuples containing a fitness score in the first position and a genome in the second (returned by *evaluate_fitness()*)

Returns a list of genomes (of length `self.n_parents`)

Dependencies:

- *select_random()*

2.1.5 crossover

class holland.evolution.**Crosser**(*genome_params*)

Handles genetic crossover

Parameters **genome_params** (*dict*) – a dictionary specifying genome parameters, specifcally `crossover_function` is relevant; see *Genome Parameters*

cross_genomes(*parent_genomes*)

Produces a new genome by applying crossover to multiple parent genomes

Parameters **parent_genomes** – a list of parent genomes

Returns a single genome

2.1.6 mutation

class holland.evolution.**Mutator**(*genome_params*)

Handles genetic mutation

Parameters **genome_params** (*dict*) – a dictionary specifying genome parameters; see *Genome Parameters*

mutate_genome(*genome*)

Mutates a genome

Parameters **genome** (*dict*) – the genome to mutate

Returns a mutated genome

Dependencies:

- *mutate_gene()*

mutate_gene(*gene*, *gene_params*)
Mutates a single gene

Parameters

- **gene** (*a valid gene type*) – the gene to mutate
- **gene_params** (*dict*) – parameters for a single gene; see *Genome Parameters*

Returns a mutated gene

Dependencies:

- *probabilistically_apply_mutation()*

probabilistically_apply_mutation(*target*, *gene_params*)
Either applies a mutation function to a target (gene or value of a gene) or does not, probabilistically according to the `mutation_rate`

Parameters

- **target** (*a valid, non-list, gene type*) – the target to which to apply the mutation
- **gene_params** (*dict*) – parameters for a single gene; see *Genome Parameters*

Returns either the mutated target or the original target

Dependencies:

- *bound_value()*

2.2 Library

2.2.1 fitness weighting functions

Fitness weighting functions are used by *select_parents()* to weight fitness scores and generate probabilities for selecting a genome to be a parent of a genome in the next generation. The following functions return stock weighting functions, some with configurable parameters. See *Selection Strategy* for general information.

General Example:

```
from holland.utils import select_random


weighting_function = get_some_weighting_function()

breeding_pool = select_breeding_pool(fitness_results, **selection_strategy.
↪get("pool"))
# split fitness and genomes into separate lists
fitness_scores, genomes = zip(*breeding_pool)

weighted_scores = [weighting_function(fitness) for fitness in fitness_scores]
weighted_total = sum(weighted_scores)
```

(continues on next page)

(continued from previous page)

```
selection_probabilities = [weighted_score / weighted_total for weighted_
↪score in weighted_scores]

parents = select_random(genomes, probabilities=selection_probabilities, n=2)
```

holland.library.fitness_weighting_functions.**get_uniform_weighting_function**()
Returns a function that returns a constant, regardless of input; see *Selection Strategy*

Returns a function that returns a constant

holland.library.fitness_weighting_functions.**get_linear_weighting_function**(*slope=1*)
Returns a function that weights its input linearly according to `slope`; see *Selection Strategy*

Parameters **slope** (*int/float*) – the multiplier for input

Returns a linear function

holland.library.fitness_weighting_functions.**get_polynomial_weighting_function**(*power=2*)
Returns a function that weights its input by raising the input to the `power` specified; see *Selection Strategy*

Parameters **power** (*int/float*) – the power to raise the input to

Returns a polynomial function

holland.library.fitness_weighting_functions.**get_exponential_weighting_function**(*base=2.7182818*
Returns a function that weights its input by raising the `base` to the power of the input; see *Selection Strategy*

Parameters **base** (*int/float*) – the base to raise to the power of the input

Returns a exponential function

holland.library.fitness_weighting_functions.**get_logarithmic_weighting_function**(*base=2.7182818*
Returns a function that weights its input getting the logarithm (with specified `base`) of the input; see *Selection Strategy*

Parameters **base** (*int/float*) – the base to calculate the logarithm of the input for

Returns a logarithmic function

Note: This fitness weighting function will throw an error for fitness scores less than or equal to 0.

---

holland.library.fitness_weighting_functions.**get_reciprocal_weighting_function**()
Returns a function that weights its input by raising the input to the -1 power; see *Selection Strategy*

The reciprocal weighting function is useful in cases where fitness should be minimized as the function results in granting higher selection probabilities to individuals with lower scores

Returns a function that returns `1/input`

Note: This fitness weighting function will throw an error for fitness scores equal to 0.

2.2.2 crossover functions

Crossover functions are used by *cross_genomes()* to perform crossover. The following functions return stock crossover functions, some with configurable parameters. See *Crossover Functions* for general information.

General Example:

```
crossover = get_some_crossover_function()

parent_genomes = select_parents(fitness_results)
gene_names = parent_genomes[0].keys()

offspring = {}
for gene_name in gene_names:
    parent_genes = [pg[gene_name] for pg in parent_genomes]
    offspring[gene_name] = crossover(parent_genes)
```

holland.library.crossover_functions.**get_uniform_crossover_function**()
Returns a function that applies uniform crossover (each gene value is chosen at random from the parent genes); see *Crossover Functions*

Valid For any gene type

Returns a function that accepts a list of parent genes and applies uniform crossover to them and returns a new gene

holland.library.crossover_functions.**get_point_crossover_function**(*n_crossover_points=1*)
Returns a function that applies point crossover (take gene values from one parent gene at a time until reaching a crossover point, then switch parent genes); see *Crossover Functions*

Valid For any list-type gene

Parameters **n_crossover_points** (*int*) – number of points at which to switch to the next parent gene (should be at least `len(parent_genes) - 1`)

Returns a function that accepts a list of parent genes and applies point crossover

Raises **ValueError** – if `n_crossover_points` is negative

Dependencies:

- *select_random()*

holland.library.crossover_functions.**get_and_crossover_function**()
Returns a function that reduces the values of the parent_genes by the logical 'and' operation; see *Crossover Functions*

Valid For `"bool"` and `"[bool]"` gene types

Returns a function that accepts a list of parent genes and applies 'and' crossover

holland.library.crossover_functions.**get_or_crossover_function**()
Returns a function that reduces the values of the parent_genes by the logical 'or' operation; see *Crossover Functions*

Valid For `"bool"` and `"[bool]"` gene types

Returns a function that accepts a list of parent genes and applies 'or' crossover

2.2.3 mutation functions

Mutation functions are used by *probabilistically_apply_mutation()* to apply mutation to a gene value. The following functions return stock mutation functions, some with configurable parameters. See *Mutation Functions* for general information.

General Example:

```
import random


mutate = get_some_mutation_function()
genome = {"gene1": [123.8, 118.2, 103.0], "gene2": [1.5, 3.7, 2.6, 1.9]}
mutation_rate = 0.01

mutated_genome = {}
for gene_name, gene in genome:
    mutated_gene = [
        mutate(value) if random.random() < mutation_rate else value  # apply␣
↪probabilistically
        for value in gene
    ]
    mutated_genome[gene_name] = mutated_gene
```

holland.library.mutation_functions.**get_flip_mutation_function**()
Returns a function that returns the negated value of the input, where the input is a boolean value; see *Mutation Functions*

Valid For "bool" and "[bool]" gene types

Returns a function that returns the negated value if its input

holland.library.mutation_functions.**get_boundary_mutation_function**(*minimum*, *maximum*)
Returns a function that pushes a value to either the minimum or maximum allowed value for a gene; see *Mutation Functions*

Valid For "int", "[int]", "float", and "[float]" gene types

Parameters

- **minimum** (*int/float*) – the minimum allowed value
- **maximum** (*int/float*) – the maximum allowed value

Returns either minimum or maximum (equally likely)

holland.library.mutation_functions.**get_uniform_mutation_function**(*minimum*, *maximum*)
Returns a function that returns a value drawn from a uniform distribution over the closed interval [minimum, maximum]; see *Mutation Functions*

Valid For any gene type

Parameters

- **minimum** (*int/float*) – the minimum allowed value
- **maximum** (*int/float*) – the maximum allowed value

Returns a sample from a uniform distribution

holland.library.mutation_functions.**get_gaussian_mutation_function**(*sigma*)
Returns a function that returns a value drawn from a gaussian (normal) distribution with mean equal to value and standard_deviation equal to sigma; see *Mutation Functions*

Valid For "int", "[int]", "float", and "[float]" gene types

Parameters **sigma** (*int/float*) – standard deviation for the gaussian distribution

Returns a sample from a gaussian distribution

2.3 Storage

2.3.1 storage manager

class holland.storage.**StorageManager**(*fitness_storage_options={}*, *genome_storage_options={}*)

Handles recording fitness statistics and genomes.

Parameters

- **fitness_storage_options** (*dict*) – options for storing fitness statistics; see *Fitness Storage Options*
- **genome_storage_options** (*dict*) – options for storing genomes and their fitness scores; see *Genome Storage Options*

update_storage(*generation_num*, *fitness_results*)

Updates storage of fitness scores and genomes (with fitness scores) when called; Decisions for whether to record or not are handled by dependencies

Parameters

- **generation_num** (*int*) – the generation number of the population that generated the `fitness_results`
- **fitness_results** (*list*) – the results of a round of evaluation (returned by *evaluate_fitness()*)

Returns `None`

Dependencies:

- *update_fitness_storage()*
- *update_genome_storage()*

react_to_interruption(*generation_num*, *fitness_results*)

Updates storage of genomes (with fitness scores) in the event of an interruption during execution if `genome_storage_options["should_record_on_interrupt"]` is set to `True`

Parameters

- **generation_num** (*int*) – the generation number of the population that generated the `fitness_results`
- **fitness_results** (*list*) – the results of a round of evaluation (returned by *evaluate_fitness()*)

Returns `None`

Dependencies:

- *record_genomes_and_fitnesses()*

update_fitness_storage(*generation_num*, *fitness_results*)

Updates storage of fitness scores if `fitness_storage_options["should_record_fitness"]` is set to `True`

Parameters

- **generation_num** (*int*) – the generation number of the population that generated the `fitness_results`

- **fitness_results** (*list*) – the results of a round of evaluation (returned by *evaluate_fitness()*)

Returns None

Dependencies:

- *record_fitness()*

update_genome_storage(*generation_num*, *fitness_results*)
Updates storage of genomes (with fitness scores) if genome_storage_options["should_record_genomes"] is set to True and the generation_num matches the recording frequency

Parameters

- **generation_num** (*int*) – the generation number of the population that generated the fitness_results
- **fitness_results** (*list*) – the results of a round of evaluation (returned by *evaluate_fitness()*)

Returns None

Dependencies:

- *should_record_genomes_now()*
- *record_genomes_and_fitnesses()*

should_record_genomes_now(*current_generation_num*)
Returns a boolean telling whether genomes should be recorded for the current_generation_num or not; Returns True if genome_storage_options["should_record_genomes"] is set to True and the generation_num matches the recording frequency, otherwise False

Parameters **generation_num** (*int*) – the generation number of the population that generated the fitness_scores

Returns a boolean telling whether or not genomes should be recorded

2.3.2 fitness

holland.storage.fitness.**record_fitness**(*generation_num*, *fitness_scores*, ***storage_options*)
Records fitness statistics for a generation to a file and returns fitness statistics

Parameters

- **generation_num** (*int*) – the generation number of the population that generated the fitness_scores
- **fitness_scores** (*list*) – the fitness scores of the generation
- **storage_options** (*dict*) – options for storing statistics, specifically file_name, format, and path are relevant; see *Fitness Storage Options*

Returns a dictionary of statistics for the fitness scores

Dependencies:

- *format_fitness_statistics()*

holland.storage.fitness.**format_fitness_statistics**(*generation_num*, *fitness_scores*)

Generate statistics on fitness scores for a generation

Parameters

- **generation_num** (*int*) – the generation number of the population that generated the fitness_scores
- **fitness_scores** (*list*) – the fitness scores of the generation

Returns a dictionary of statistics for the fitness scores

2.3.3 genomes and fitnesses

holland.storage.genomes_and_fitnesses.**record_genomes_and_fitnesses**(*generation_num*, *fitness_results*, ***storage_options*)

Records results of a round of evaluation

Parameters

- **generation_num** (*int*) – the generation number of the population that generated the fitness_scores
- **fitness_results** (*list*) – the results of a round of evaluation (returned by *evaluate_fitness()*)
- **storage_options** (*dict*) – options for selecting which results to store and how to store them, specifically `should_add_generation_suffix`, `format`, `file_name`, `path`, `top`, `mid`, `bottom` are relevant; see *Genome Storage Options*

Returns `None`

Dependencies:

- *format_genomes_and_fitnesses_for_storage()*

holland.storage.genomes_and_fitnesses.**format_genomes_and_fitnesses_for_storage**(*generation_num*, *fitness_results*, ***storage_options*)

Formats results of a round of evaluation for storage

Parameters

- **generation_num** (*int*) – the generation number of the results
- **fitness_results** (*list*) – the sorted results of a round of evaluation (returned by evaluate_fitness)
- **storage_options** (*dict*) – options for selecting which results to store, specifically `top`, `mid`, `bottom` are relevant; see *Genome Storage Options*

Returns a dictionary of the form `{"generation": generation_num, "results": selected_results}`

Note: For the sake of efficiency, this method expects `fitness_results` to be sorted in order to properly select genomes on the basis of fitness. *`evaluate_fitness()`* returns sorted results.

Dependencies:

- *`select_from()`*

2.3.4 utils

`holland.storage.utils.`**`record`**(*data*, ***storage_options*)

Records data to a file

Parameters

- **`data`** (*`list/dict`*) – the data to write to the file
- **`storage_options`** (*`dict`*) – options for writing the data to a file, specifically `format` (options: `'json'`, `'csv'`), `file_name`, and `path` are relevant; see *Fitness Storage Options* and *Genome Storage Options*

Returns `None`

Dependencies:

- *`record_to_csv()`*
- *`record_to_json()`*

`holland.storage.utils.`**`record_to_csv`**(*data*, ***storage_options*)

Writes data to a file CSV format; appends a row to an existing file; a file is created if none exists yet

Parameters

- **`data`** (*`dict`*) – the data to write to the file (with column names as keys)
- **`storage_options`** (*`dict`*) – options for writing the data to a file, specifically `file_name` and `path` are relevant; see *Fitness Storage Options* and *Genome Storage Options*

Returns `None`

Raises

- **`AssertionError`** – if storage_options["file_name"] is not specified
- **`AssertionError`** – if storage_options["path"] is not specified
- **`ValueError`** – if not all values are of type int or float

`holland.storage.utils.`**`record_to_json`**(*data*, ***storage_options*)

Writes data to a file JSON format; overwrites contents if the file already exists

Parameters

- **`data`** (*`list/dict`*) – the data to write to the file (must be valid JSON format)
- **`storage_options`** (*`dict`*) – options for writing the data to a file, specifically `file_name` and `path` are relevant; see *Fitness Storage Options* and *Genome Storage Options*

Returns `None`

Raises

- **AssertionError** – if storage_options["file_name"] is not specified
- **AssertionError** – if storage_options["path"] is not specified

2.4 Utils

2.4.1 utility functions

holland.utils.utils.**bound_value**(*value*, *minimum=-inf*, *maximum=inf*, *to_int=False*)
Bounds a value between a minimum and maximum

Parameters

- **value** (*int/float*) – the value to bound
- **minimum** (*int/float*) – the lower bound
- **maximum** (*int/float*) – the upper bound
- **to_int** (*bool*) – whether or not to cast the result to an int

Returns the bounded value

holland.utils.utils.**select_from**(*values*, *top=0*, *mid=0*, *bottom=0*, *random=0*)
Selects elements from a (sorted) list without replacement

Parameters

- **values** (*list*) – the list of values to select from
- **top** (*int*) – number of elements to select from the top (end) of the list
- **mid** (*int*) – number of elements to select from the middle of the list
- **bottom** (*int*) – number of elements to select from the bottom (start) of the list
- **random** (*int*) – number of elements to select randomly from the list

Returns a list of selected elements

holland.utils.utils.**select_random**(*choices*, *probabilities=None*, *n=1*, *should_replace=False*)
Selects random elements from a list

Parameters

- **choices** (*list*) – list of elements to select from
- **probabilities** (*list*) – list of probabilities for selecting each element in `choices`; if not specified, uniform probability is used
- **n** (*int*) – number of elements to select from `choices`
- **should_replace** (*bool*) – specifies if selection should be done with replacement or not

Returns a list of length `n` of elements selected randomly from `choices`

Raises

- **ValueError** – if `probabilities` is given but `len(probabilities) != len(choices)`
- **ValueError** – if any element of `probabilities` is negative

- **ValueError** – if `sum(probabilities) > 1`
- **ValueError** – if `should_replace` is `False` but `n > len(choices)`

`holland.utils.utils.`**is_numeric_type**(*gene_params*)

Determines if a gene is of a numeric type or not (whether list type or not); e.g. returns `False` if type is `"bool"` or `"[bool]"`, but `True` if type is `"float"` or `"[float]"`

Parameters **gene_params** (*dict*) – a dictionary of parameters for a single gene; see *Genome Parameters*

Returns a boolean indiciating whether the gene is of a numeric type or not

`holland.utils.utils.`**is_list_type**(*gene_params*)

Determines if a gene is of a list type or not; e.g. returns `False` if type is `"float"` but `True` if type is `"[float]"`

Parameters **gene_params** (*dict*) – a dictionary of parameters for a single gene; see *Genome Parameters*

Returns a boolean indicating whether the gene is of a list type or not

CHAPTER 3

Configuration

This page provides information on configuring Holland, specifically initializing the `Evolver` class and using its `evolve()` method.

3.1 Fitness Function

The fitness function is a user-written function that maps genomes to fitness scores, which in turn, are used in breeding the next generation. A fitness function must accept a single genome and must return either:

- an integer or float corresponding to the fitness of the given genome (Darwinian Evolution); or
- a tuple/list with the fitness score in the first position and a modified genome in the second (Lamarckian Evolution).

See `evaluate_fitness()` for details on how the fitness function is used.

Holland is designed to be application-agnostic, so a fitness function can evaluate a genome in any way so long as the input and output match what is expected. A fitness function might simply plug in different values from a genome's

genes into a formula or it might create an instance of some class according to the parameters specified in the genome and then run a simulation for that individual.

Example:

```
def darwinian_fitness_function(genome):
    """Evaluates an ImageClassifier instance (that uses a neural network with␣
↪weight vectors w1 and w2)"""
    individual = ImageClassifier(w1=genome["w1"], w2=genome["w2"])
    fitness = 0

    for label, image in labeled_images:
        classification = individual.classify(image)
        if classification == label:
            fitness += 100

    return fitness


def lamarckian_fitness_function(genome):
    """Evaluates an ImageClassifier instance (that uses a neural network with␣
↪weight vectors w1 and w2)"""
    individual = ImageClassifier(w1=genome["w1"], w2=genome["w2"])
    fitness = 0

    for label, image in labeled_images:
        classification = individual.classify(image)
        if classification == label:
            fitness += 100
        else:
            individual.back_propagate()

    final_genome = individual.get_weights()

    return fitness, final_genome
```

3.2 Genome Parameters

In order to generate initial and random genomes, perform crossover on genomes, and mutate genomes, `genome_params` are required to be specifed. The structure of genomes for populates are determined by these parameters. `genome_params` is a dictionary whose keys correspond to individual genes, where the dictionary contained at each key specifies parameters for that gene.

Each gene must have a specified type. There are two broad categories of gene types: list-types and value-types. List-type genes are lists of a set length and containing only elements of a single type. Value-type genes are single values. List-type genes use the notation `"[type]"` while value-type genes use the notation `"type"`.

This is an example of `genome_params`:

```
{
    "gene1": {
        "type": "int",
        "max": 100000,
        "min": -100000,
        "initial_distribution": lambda: random.uniform(-100000, 100000),
        "crossover_function": get_uniform_crossover_function(),
```

(continues on next page)

(continued from previous page)

```
        "mutation_function": get_gaussian_mutation_function(100),
        "mutation_rate": 0.01
    },
    "gene2": {
        "type": "[float]",
        "size": 100,
        "max": 100000,
        "min": -100000,
        "initial_distribution": lambda: random.uniform(-100000, 100000),
        "crossover_function": get_point_crossover_function(n_crossover_points=3),
        "mutation_function": get_gaussian_mutation_function(100),
        "mutation_rate": 0.01
    },
    "gene3": {
        "type": "bool",
        "initial_distribution": lambda: random.random() < 0.5,
        "crossover_function": get_uniform_crossover_function(),
        "mutation_function": get_flip_mutation_function(),
        "mutation_rate": 0.05
    },
    "gene4": {
        "type": "[str]",
        "size": 5,
        "initial_distribution": lambda: random.sample(list_of_words, 1)[0],
        "crossover_function": get_uniform_crossover_function(),
        "mutation_function": rotate_order,
        "mutation_level": "gene",
        "mutation_rate": 0.05
    }
}
```

The significance of these values is as follows:

- **type** (*str*) – specifies the type of the gene; if the gene is just a single value, use the plain type, but if the gene is a list of values, use the type in brackets; options:
  - `"float"`, `"[float]"`
  - `"int"`, `"[int]"`
  - `"bool"`, `"[bool]"`
  - `"str"`, `"[str]"`
- **size** (*int*) – specifies the length of the gene if list-type
- **max** (*int/float*) – specifies the maximum allowed value for the gene or any element of the gene if of a numeric type
- **min** (*int/float*) – specifies the minimum allowed value for the gene or any element of the gene if of a numeric type
- **initial_distribution** (*func*) – a function for initializing a random gene with values; must not accept any positional arguments
- **crossover_function** (*func*) – a function to cross multiple parent genes; see *Crossover Functions* for more
- **mutation_function** (*func*) – a function that mutates either the whole gene or a single value of the gene (depending on `mutation_level`); see *Mutation Functions* for more

- **mutation_level** (*str*) – specifies how to apply the `mutation_funtion`: either to the gene as a whole, or just individual values; default is `"value"` (options: `"value"`, `"gene"`); irrelevant for value-type genes
- **mutation_rate** (*int/float*) – probability (`0` to `1`) that each value of the gene gets mutated (by applying the `mutation_function`)

3.3 Crossover Functions

Crossover functions are used to splice parent genes together to form a gene for an offspring. Crossover functions can be custom made, but Holland offers a few common crossover functions built in, these are described in the *crossover functions* subsection of *Library*. If you write or find a novel crossover function that you find useful, consider contributing it to the Holland library!

Crossover functions act on, and are specified for, individual genes, rather than entire genomes. Since Holland supports reproduction between an arbitrary number of individuals (parents) crossover functions must accept a single argument: a list containing parent gene(s). The length of this list is determined by the number of parents as specified in the `selection_strategy` (see *Selection Strategy*). Crossover functions must return a single gene.

Example:

```
def crossover(parent_genes):
    """Take each value by alternating between parent genes"""
    num_parents = len(parent_genes)
    gene_length = len(parent_genes[0])
    return [parent_genes[i % num_parents][i] for i in range(gene_length)]
```

3.4 Mutation Functions

Mutation functions are used to modify gene values. Like *Crossover Functions*, mutation functions can be custom made, but Holland offers a few common mutation functions built in, these are described in the *mutation functions* subsection of *Library*. If you write or find a novel mutation function that you find useful, consider contributing it to the Holland library!

Mutation functions can act on either individual values of a gene or an entire gene, but not the whole genome. Mutation functions are specified for each gene. To have a mutation function applied to a whole gene (when the gene is a list-type), the option `"mutation_level"` should be set to `"gene"` instead of `"value"` (see *Genome Parameters* for more detail); for value-type genes this distinction does not matter. For most applications of the Genetic Algorithm a `"mutation_level"` of `"value"` should be appropriate, but some applications—e.g. Travelling Salesman—require mutations be applied at the gene level.

A mutation function is applied probabilistically (by *`probabilistically_apply_mutation()`*), and, therefore, need not consider the `mutation_rate` of the gene. Mutation functions must return the mutated value or gene.

Example:

```
import random


def mutate_value(value):
    """Randomly doubles or halves a value -- applied at "value" level"""
    if random.random() < 0.5:
        return value * 2
    return value / 2
```

(continues on next page)

(continued from previous page)

```
def mutate_gene(gene):
    """Shuffle a gene -- applied at "gene" level"""
    random.shuffle(gene)
    return gene
```

3.5 Selection Strategy

The selection strategy for breeding the next generation of indviduals is specified in the `selection_strategy` dictionary. The strategy is ultimately used by the functions *`select_breeding_pool()`*, which uses information contained in the `"pool"` section of the selection strategy, and *`select_parents()`*, which uses information contained in `"parents"`.

The fitness weighting function determines how to weight fitness scores in order to translate into probabilities for selection of a genome as a parent for an individual in the next generation. For cases in which fitness is sought to be maximized, an increasing fitness weighting function should be used, whereas cases in which fitness should be minimized (e.g. fitness represents error) should employ a decreasing fitness weighting function. In both cases a uniform weighting function will suffice. In the case of minimizing fitness, a reciprocal weighting function, linear weighting function with negative slope, or polynomial weighting function with negative power will work. See *fitness weighting functions* for stock fitness weighting functions.

The dictionary `selection_strategy` should have the below form. The example values shown here are the defaults and any parameters that are not specified will use these values as defaults:

```
{
    "pool": {
        "top": 0,
        "mid": 0,
        "bottom": 0,
        "random": 0
    },
    "parents": {
        "weighting_function": lambda x: 1,
        "n_parents": 2
    }
}
```

The significance of these values is as follows:

- **pool**
  - **top** (*int*) – number of genomes to select from the top (end) of the pack (by fitness)
  - **mid** (*int*) – number of genomes to select from the middle of the pack (by fitness)
  - **bottom** (*int*) – number of genomes to select from the bottom (start) of the pack (by fitness)
  - **random** (*int*) – number of genomes to select at random
- **parents**
  - **weighting_function** (*func*) – function for converting a fitness score into a probability for selecting an individual as a parent (default is uniform weighting); higher weights indicate a higher probability of being selected
  - **n_parents** (*int*) – number of parents to select for each offspring

Note: It is recommended that the `weighting_function` return only positive values. While Holland can handle weighting functions that return negative values, this presents an ambiguous case in terms of converting weighted scores to probabilities. Current handling of this case aims to minimally distort probabilities, but results may not be exactly what you expect.

3.6 Generation Parameters

When creating the population for the next generation, a few optional parameters can be set:

- **n_random** (*int*) – number of fully random genomes to introduce to the population in each generation
- **n_elite** (*int*) – number of (most fit) genomes to preserve for the next generation
- **population_size** (*int*) – size of the population in each generation (required if an initial population is not given)

These values should be placed in the `generation_params` dictionary.

3.7 Fitness Storage Options

To measure performance improvements over the generations, fitness statistics can be stored for each generation. If enabled, the statistics recorde are max, min, mean, median, and standard deviation. Values can be stored either to a file (csv) or in memory and returned by `evolve()`. By default fitness statistics are not recorded.

The following options are available:

- **should_record_fitness** (*bool*) – determines whether or not to record fitness
- **format** (*str*) – file format (options: 'csv', 'memory'); if 'memory', stats are returned as second element of tuple in `evolve()`
- **file_name** (*str*) – name of the file to write to
- **path** (*str*) – location of the file to write

See the *fitness* subsection of *Storage* for more on how these values are used.

3.8 Genome Storage Options

To record snapshots of the population over the generations genomes and their corresponding fitness scores (in the same format returned by `evaluate_fitness()`) can be recorded. If enabled, individuals will be selected according to the specified strategy and stored to a file (json). Additionally, by setting `should_record_on_interrupt` to `True` (which is independent of the value of `should_record_genomes`), genomes will be recorded if an unhandled exception is thrown during execution. By default genomes are not recorded.

The following options are available:

- **should_record_genomes** (*bool*) – determines wether or not to record genomes at all
- **record_every_n_generations** (*int*) – recording frequency
- **should_record_on_interrupt** (*bool*) – determines wether or not to record genomes if an unhandled exception (including KeyboardInterrupt) is raised
- **format** (*str*) – file format (options: 'json')

- **file_name** (*str*) – name of the file to write to
- **path** (*str*) – location of the file to write
- **should_add_generation_suffix** (*bool*) – determines whether or not to append '-generation_{n}' to the end of file_name
- **top** (*int*) – number of genomes and scores to select from the top of the pack (by fitness)
- **mid** (*int*) – number of genomes and scores to select from the middle of the pack (by fitness)
- **bottom** (*int*) – number of genomes and scores to select from the bottom of the pack (by fitness)

See the *genomes and fitnesses* subsection of *Storage* for more on how these values are used.

CHAPTER 4

Contributing

4.1 Reporting Bugs

If you encouter a bug, please open an issue on our github page.

4.2 Contribution Guide

To contribute a new feature, please fork our repository and submit a pull request.

4.2.1 Style Guide

General Code Style

All code should be written in python and formatted with black.

Updating or Writing Methods

When updating methods, make sure to update the *Dependencies* list.

Updating Examples

When updating an example, use grep to make sure all references are updated if necessary. e.g. updating lines of highlighted code

CHAPTER 5

Indices and tables

- genindex
- search

Python Module Index

h

Index

P

R

S

U