NEORL Documentation

Release 1.8.1b

NEORL Contributors

Jul 11, 2023

CONTENTS

Latest News:

- June 5, 2022: Stable release 1.8 is out.
- Major features of 1.8 release: ensemble optimizers are added (AEO, EDEV, EPSO, HCLPSO).
- November 24, 2021: Stable release 1.7 is out.
- September 10, 2021: First NEORL stable release 1.6 is out.

Primary contact to report bugs/issues: Majdi I. Radaideh (radaideh@mit.edu)

NEORL (**N**euro**E**volution **O**ptimisation with **R**einforcement **L**earning) is a set of implementations of hybrid algorithms combining neural networks and evolutionary computation based on a wide range of machine learning and evolutionary intelligence architectures. NEORL aims to solve large-scale optimization problems relevant to operation & optimisation research, engineering, business, and other disciplines.

Github repository: https://github.com/mradaideh/neorl

NEORL paper: https://arxiv.org/abs/2112.07057

CHAPTER

ONE

COPYRIGHT

You can read the first successful application of NEORL for nuclear fuel optimisation in this News Article.

CHAPTER

TWO

USER GUIDE

2.1 General Guide

This section provides an overview of the NEORL framework and the installation steps.

2.1.1 Quick Installation

Use this guide if you are an expert Python user and aware of Python virtual environment and package management. For a safe and clean installation guide, see the *Detailed Installation* section.

Prerequisites

NEORL is tested on `python3 (3.6-3.7)` with the development headers. **Please, avoid using python 3.5 or lower** (as dictionary ordering is not preserved), or **python 3.8 or newer** (as tensorflow-1.14.0 will not be stable).

Note: NEORL supports `tensorflow` versions from 1.8.0 to 1.14.0, **we do not support tensorflow >= 2.0**. Please, make sure to uninstall `tensorflow` if already installed on your environment or have a proper version. If `tensorflow` is left in the virtual environment, NEORL will automatically force `tensorflow-1.14.0` for most stability.

Ubuntu Prerequisites

```
sudo apt-get update && sudo apt-get install cmake python3-dev
```

Windows 10 Prerequisites

To install NEORL on Windows, it is recommended to install Anaconda3 on the machine first to have some pre-installed packages, then open "Anaconda Prompt" as an administrator and use the instructions below for **Install using pip**.

Note: You can access Anaconda3 archives for all OS installers from this page https://repo.anaconda.com/archive/

Note: We typically recommend creating a new virtual environment for NEORL to avoid version conflicts and compatibility issues with other projects.

```
conda create --name neorl python=3.7
conda activate neorl
```

Install using pip

For both Ubuntu and Windows, you can install NEORL via pip

```
pip install neorl
```

2.1.2 Detailed Installation

Use this guide if you are looking for a safe and clean installation of NEORL with all Python tools and package management. If you are an expert Python user and aware of Python virtual environment and package management, see the *Quick Installation* section.

Linux/Ubuntu

Step 0: Prerequisites (Anaconda3 Installation)

Anaconda3 will provide you with OS-independent framework that hosts Python packages, including NEORL. **If you have Anaconda3 installed on your machine, move to Step 1**.

1- First download the Anaconda3 package

```
wget --no-check-certificate https://repo.continuum.io/archive/Anaconda3-2019.03-Linux-
↪x86_64.sh
```

2- Start the installer

```
bash Anaconda3-2019.03-Linux-x86_64.sh
```

3- Follow the instructions on the screen and wait until completion (See the notes below on how to respond to certain prompts)

Note: Choose the default location for installation when asked (e.g. /home/username/anaconda3)

Note: Enter **yes** when you get this prompt: **Do you wish the installer to initialize Anaconda3 by running conda init? [yes|no]**

4- You may update the setup tool packages before proceeding

```
pip install --upgrade pip setuptools wheel
```

Step 1: Create virtual environment for NEORL

NEORL is tested on `python3 (3.6-3.7)` with the development headers. **Please, avoid using python 3.5 or lower** (as dictionary ordering is not preserved), or **python 3.8 or newer** (as tensorflow-1.14.0 will not be stable).

1- Create a new python-3.7 environment with name `neorl`

```
conda create --name neorl python=3.7
```

Warning: For some machines that are not updated frequently (e.g. clusters), TensforFlow may fail to load due to outdated gcc libraries. If you encounter those errors, we typically recommend to downgrade python by using python=3.6, when creating the virtual environment.

2- Activate the environment

```
conda activate neorl
```

Warning: You need to run `conda activate neorl` every time you log in the system, therefore, it is good to add this command to your OS bashrc or environment variables for automatic activation when you log in.

Step 2: Install NEORL

Make sure `neorl` environment is activated, then run the following command:

```
pip install neorl
```

Warning: Depending on your OS, `conda` command may fail due to unknown reasons. If `conda list` command fails, then type

```
conda update -n base -c defaults conda
```

Step 3: Test NEORL

After an error-free Step 2 completion, you can test NEORL by typing on the terminal:

```
neorl
```

which yields NEORL logo

```
           NEORL: NeuroEvolution Optimisation with Reinforcement Learning
                                  NEORL
Copyright © 2021 Exelon Corporation (https://www.exeloncorp.com/) in collaboration with
             MIT Nuclear Science and Engineering (https://web.mit.edu/nse/)
                                All Rights Reserved


usage: neorl [-h] [-i INPUT] [-c CHECK] [-t] [-e] [-v]

NEORL command line API parser

optional arguments:
  -h, --help            show this help message and exit
  -i INPUT, --input INPUT
                        Name of the input ASCII file, e.g. INPUT.inp,
                        INPUT.dat (required arg)
  -c CHECK, --check CHECK
                        check input file syntax and exit
  -t, --test            run NEORL units tests
  -e, --example         print a simple NEORL script and exit
  -v, --version         show program's version number and exit
```

and you can run unit tests by running:

```
neorl --test
```

Windows 10

Step 0: Prerequisites (Anaconda3 Installation)

Anaconda3 will provide you with OS-independent framework that hosts Python packages, including NEORL. **If you have Anaconda3 installed on your machine, move to Step 1**.

1- First download the Anaconda3 package by visiting the link in the note below and search for `Anaconda3-2019.03-Windows-x86_64.exe`

Note: You can access Anaconda3 archives for all OS installers from this page https://repo.anaconda.com/archive/

---

or simply click on the link below to download:

https://repo.anaconda.com/archive/Anaconda3-2019.03-Windows-x86_64.exe

2- Start the exe installer, follow the instructions on the screen, and wait until completion. See the notes below on what options to choose.

Note:

- Choose the option "Register Anaconda as your default Python-3.7".
- For the option of "adding anaconda to your PATH variables", choose this option only if you have cleaned all previous Anaconda3 releases from your machine.

Step 1: Create virtual environment for NEORL

Search for `Anaconda Prompt` and open a new terminal as an administrator

1- Create a new python-3.7 environment with name `neorl`

```
conda create --name neorl python=3.7
```

2- Activate the environment

```
conda activate neorl
```

Step 2: Install NEORL

Make sure `neorl` environment is activated, then run the following command:

```
pip install neorl
```

Warning: Depending on your OS, `conda` command may fail due to unknown reasons. If `conda list` command fails, then type

```
conda update -n base -c defaults conda
```

Step 3: Test NEORL

After an error-free Step 2 completion, you can test NEORL by typing on the terminal:

```
neorl
```

which yields NEORL logo

```
           NEORL: NeuroEvolution Optimisation with Reinforcement Learning
                                  NEORL
Copyright © 2021 Exelon Corporation (https://www.exeloncorp.com/) in collaboration with
             MIT Nuclear Science and Engineering (https://web.mit.edu/nse/)
                                All Rights Reserved


usage: neorl [-h] [-i INPUT] [-c CHECK] [-t] [-e] [-v]

NEORL command line API parser

optional arguments:
  -h, --help            show this help message and exit
  -i INPUT, --input INPUT
                        Name of the input ASCII file, e.g. INPUT.inp,
                        INPUT.dat (required arg)
  -c CHECK, --check CHECK
                        check input file syntax and exit
  -t, --test            run NEORL units tests
  -e, --example         print a simple NEORL script and exit
  -v, --version         show program's version number and exit
```

and you can run unit tests by running:

```
neorl --test
```

Warning: You need to run `conda activate neorl` every time you log in the system, therefore, it is good to add this command to your OS environment variables for automatic activation. Similarly, make sure to connect your Jupyter notebook and Spyder IDE to `neorl` virtual environment NOT to the default `base`.

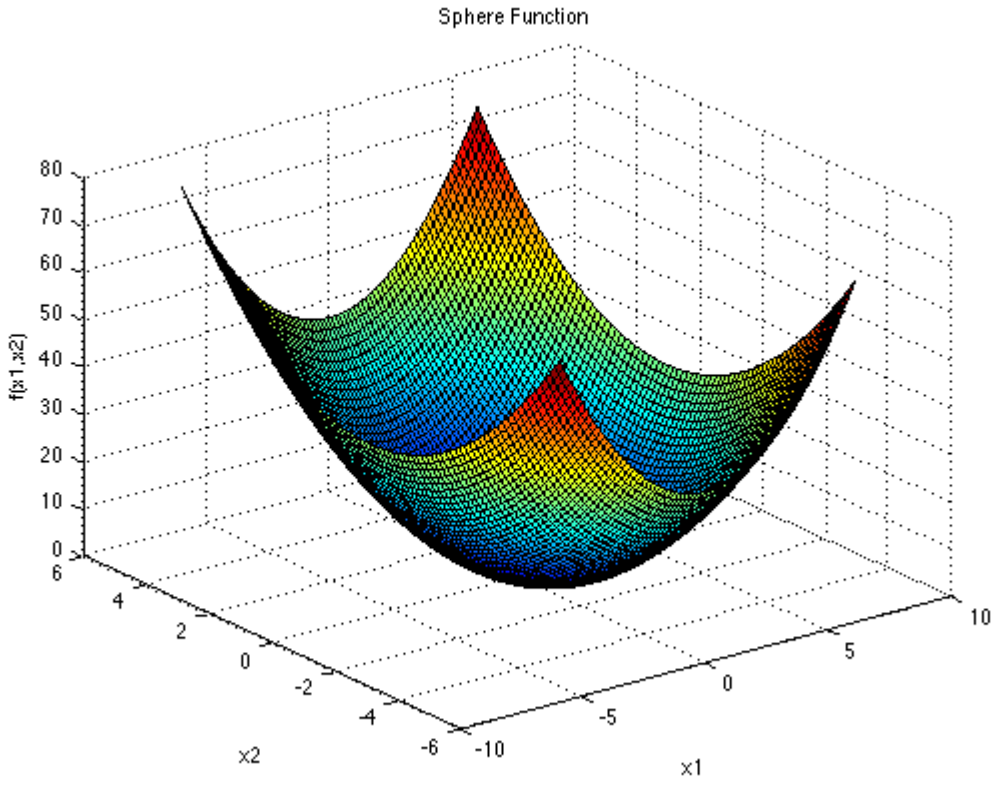
2.1.3 Getting Started

NEORL tries to follow a typical machine-learning-like syntax used in most libraries like `sklearn` and `keras`.

Here, we describe how to use NEORL to minimize the popular sphere function, which takes the form

$$f(\vec{x}) = \sum_{i=1}^{d} x_i^2$$

The sphere function is continuous, convex and unimodal. This plot shows its two-dimensional ($d = 2$) form.



The function is usually evaluated on the hypercube $x_i \in [-5.12, 5.12]$, for all $i = 1, \ldots, d$. The global minimum for the sphere function is:

$$f(\vec{x}^*) = 0, \text{ at } \vec{x}^* = [0, 0, ..., 0]$$

Here is a quick example of how to use NEORL to minimize a 5-D ($d = 5$) sphere function:

```
#-------------------------------
# Import packages
#-------------------------------
import numpy as np
import matplotlib.pyplot as plt
from neorl import DE, XNES

#-------------------------------
# Fitness
#-------------------------------
#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
            F(x) = sum_{i=1}^d xi^2
            d=1,2,3,...
            Range: [-100,100]
            Minima: 0
    """

    return sum(x**2 for x in individual)

#-------------------------------
```

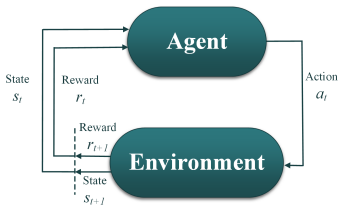
(continues on next page)

(continued from previous page)

```
# Parameter Space
#---------------------------------
#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

#---------------------------------
# DE
#---------------------------------
de=DE(mode='min', bounds=BOUNDS, fit=FIT, npop=50, CR=0.5, F=0.7, ncores=1, seed=1)
x_best, y_best, de_hist=de.evolute(ngen=120, verbose=0)
print('---DE Results---', )
print('x:', x_best)
print('y:', y_best)

#---------------------------------
# NES
#---------------------------------
x0=[-50]*len(BOUNDS)
amat = np.eye(nx)
xnes=XNES(mode='min', bounds=BOUNDS, fit=FIT, npop=50, eta_mu=0.9,
          eta_sigma=0.5, adapt_sampling=True, seed=1)
x_best, y_best, nes_hist=xnes.evolute(120, x0=x0, verbose=0)
print('---XNES Results---', )
print('x:', x_best)
print('y:', y_best)


#---------------------------------
# Plot
#---------------------------------
#Plot fitness for both methods
plt.figure()
plt.plot(de_hist['global_fitness'], label='DE')
plt.plot(np.array(nes_hist['fitness']), label='NES')
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.legend()
plt.show()
```

2.1.4 Reinforcement Learning

Reinforcement learning (RL) is a paradigm of machine learning concerned with developing intelligent systems, that know how to take actions in an environment in order to maximize cumulative reward. RL does not need labelled input/output data as other machine learning algorithms. Instead, RL collects the data on-the-fly as needed to maximize the reward. This advantage makes RL a natural choice for optimization problems, for which the search space is usually too complex and too high to generate a representative dataset.



RL algorithms, like evolutionary algorithms, focus on finding a balance between exploration (new knowledge) and exploitation (of current knowledge) to maximize the fitness/reward function. We can take this analogy to make RL intuitive in solving optimization problems through:

1- The agent: which is the optimizer. The agent is controlled by the RL algorithm that trains the agent to take proper actions. The algorithm takes the current state (s_t) and the current reward (r_t) as inputs, and decides the next action to take (a_t) as output. The action a_t in this case is a sample drawn from the parameter space for optimization ($\vec{x} = [x_1, x_2, ..., x_d]$).

2- The current state (s_t) for optimization can be set equal to the current action ($s_t \leftarrow a_t$), since we perturb the whole action space at once, and we are not marching through time.

3- The reward is similar as the fitness function in optimization. If it is a minimization problem, the user can convert to reward maximization by multiplying the final fitness value with -1.

4- The environment: takes the action provided by the agent (a_t), evaluates that action using the fitness function, assigns the next state and the next reward for taking that action (s_{t+1}, r_{t+1}), and sends them back to the RL agent. In NEORL, the user only needs to specify the fitness function and the parameter space, and NEORL can automatically create the environment class and connect that with the RL agent.

5- Steps 1-4 are repeated for sufficient time steps until the agent learns how to take the right action based on the given state such that the reward is maximized.

6-The best action taken by the agent represents the optimized input ($\vec{x}$), while the best reward is similar to the best fitness, $y = f(\vec{x})$.

Currently we have a support of some RL algorithms and hybrid neuroevolution, some are listed below

Deep Q Learning

See the *DQN* section

Proximal Policy Optimization

See the *PPO* section

Advantage Actor Critic

See the *A2C* section

Recurrent Neuroevolution of Augmenting Topologies

See the *RNEAT* section

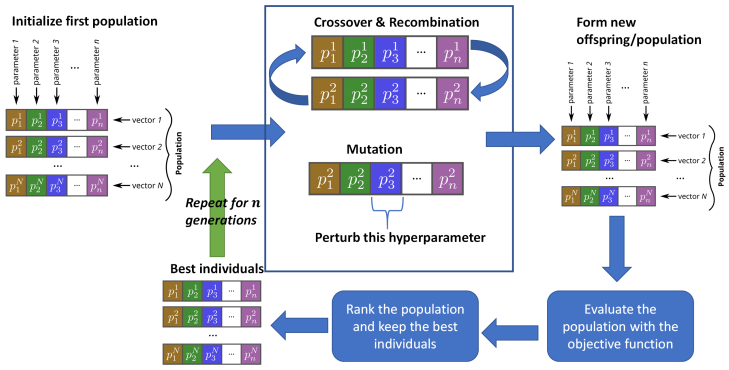
Feedforward Neuroevolution of Augmenting Topologies

See the *FNEAT* section

2.1.5 Evolutionary Algorithms

Evolutionary and Swarm algorithms are a class of computational intelligence that rely on heuristic-based approaches to solve optimization problems, which cannot be easily solved in polynomial time. These problems include NP-Hard combinatorial problems, and similar problems that require too long to process due to the fitness function complexity (e.g. engineering problems, computer simulation).

The concept of evolutionary algorithms (EA) is quite simple as it follows the process of natural selection and evolution. Genetic algorithms, evolution strategies, and differential evolution are among the most popular EA. An EA contains four overall stages: initialization, selection, genetic operators, and termination. Typically in an EA, fitter individuals that maximize/minimize the fitness function will survive and proliferate into the next generations, while weak individuals will die off and not contribute to the pool of further generations, just like natural selection processes. A typical workflow of an EA is shown in the figure below.



Swarm intelligence is also another variant of EA and nature-inspired metaheuristics, in which a population of agents are interacting locally with one another and globally with their environment. The inspiration comes from the nature, especially how biological systems interact. The agents follow simple rules and with repeated local and some random interactions between the agents, an emergence of "intelligent" global behavior can be observed among the whole population of agents. For example, this behavior can be seen in ant colonies, bee colonies, and bird flocking movement when searching for food. Among the most common swarm algorithms are particle swarm optimization, ant colony optimization, artificial bee colony, cuckoo search, and many others.

Currently we have a support of many evolutionary and swarm algorithms, some are listed below:

Evolution Strategies

See the *ES* section

Particle Swarm Optimization

See the *PSO* section

Differential Evolution

See the *DE* section

Grey Wolf Optimizer

See the *GWO* section

Whale Optimization Algorithm

See the *WOA* section

Cuckoo Search

See the *CS* section

2.1.6 Hyperparameter Tuning

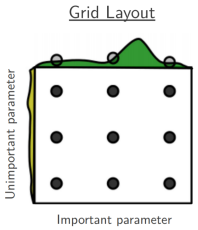
Hyperparameter tuning is the problem of choosing a set of optimal hyperparameters for a learning algorithm, which includes reinforcement learning, evolutionary, and neuroevolution algorithms of NEORL. Hyperparameter tuning is effective to maximize the efficiency of the optimization algorithm in hand. In NEORL, we provide different methods to tune hyperparameters, which are highlighted briefly here.

Grid Search

See the *Grid Search* section

Original paper: Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. Journal of machine learning research, 13(2).

Grid Search is an exhaustive search for selecting an optimal set of algorithm hyperparameters. In Grid Search, the analyst sets up a grid of hyperparameter values. A multi-dimensional full grid of all hyperparameters is constructed, which contains all possible combinations of hyperparameters. Afterwards, every combination of hyperparameter values is tested in serial/parallel, where the optimization score (e.g. fitness) is estimated. Grid search can be very expensive for fine grids as well as large number of hyperparameters to tune.



For example, to tune few hyperparameters of DQN, the following grids can be defined:

`learning_rate` =[0.0001, 0.00025, 0.0005, 0.00075, 0.001]
`batch_size` =[16, 32, 64]
`target_network_update_freq` =[100, 250, 500, 750, 1000, 1250, 1500, 1750, 2000]
`exploration_fraction` =[0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35]

The full grid has a size of = 5*3*9*7= 945 (A total of 945 hyperparameter combinations will be evaluated). Therefore, the cost of grid search is:

$$Cost = k_1 \times k_2 \times ... \times k_d,$$

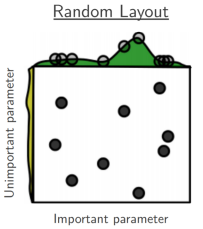
where k_i is the number of nodes in the hyperparameter i and d is the number of hyperparameters to tune.

Random Search

See the *Random Search* section

Original paper: Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. Journal of machine learning research, 13(2).

Random search is a technique where random combinations of the hyperparameters are used to find the best solution for the algorithm used. Random search tries random combinations of the hyperparameter set, where the cost function is evaluated at these random sets in the parameter space. As indicated by the reference above, the chances of finding the optimal hyperparameters are comparatively higher in random search than grid search. This is because of the random search pattern, as the algorithm might end up being used on the optimized hyperparameters without any aliasing or wasting of resources.



For example, to tune few hyperparameters of DQN, the parameters can be defined depending on the type:

`learning_rate` = $\mathcal{U}(0.0001, 0.001)$ (Parameter type is continuous `float` uniform distribution)
`batch_size` = $\mathcal{U}\{16, 64\}$ (Parameter type is discrete `int` uniform distribution)
`target_network_update_freq` = $\mathcal{C}\{100, 500, 1000, 1500\}$ (Parameter type is categorical `grid`)
`exploration_fraction` = $\mathcal{C}\{0.05, 0.1, 0.15\}$ (Parameter type is categorical `grid`)

The cost of random search is determined by the total number of random evaluations provided by the user (`ncases`).

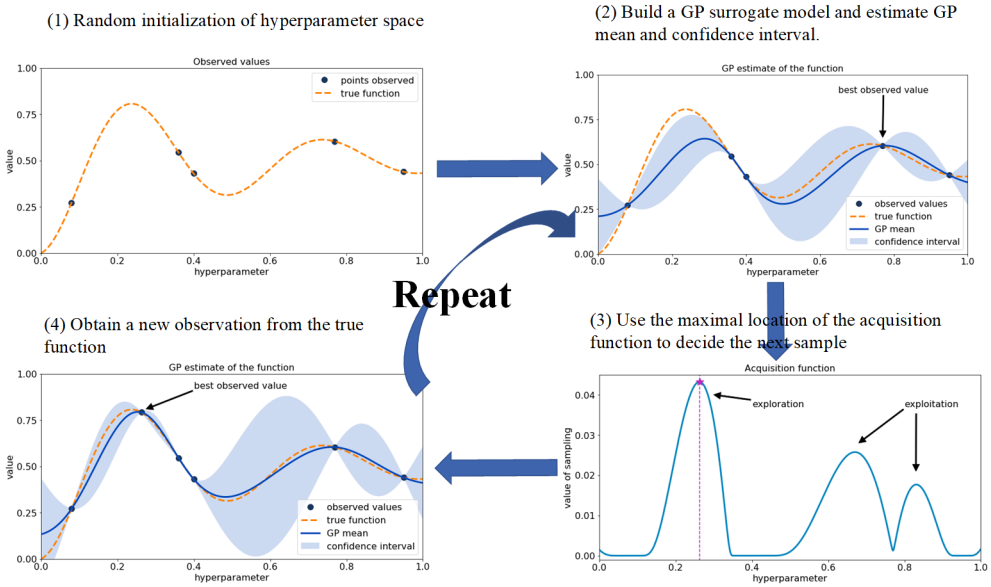
Bayesian Search

See the *Bayesian Search* section

Original paper: https://arxiv.org/abs/1012.2599

Bayesian search, in contrast to grid and random searches, keeps track of past evaluation results. Bayesian uses past evaluations to form a probabilistic model mapping hyperparameters to a probability of a score on the objective function (e.g. max/min fitness). Bayesian optimization excels when the objective functions are expensive to evaluate, when we do not have access to derivatives, or when the problem at hand is non-convex.

The heart of Bayesian optimization is Bayes theorem, which updates our prior beliefs (e.g. hyperparameter values) after new evidence/data is observed (e.g. new fitness values found by the algorithm of interest). The updated beliefs are represented by the posterior distribution, which is used to guide the next round of hyperparameter sampling. Also, Bayesian optimization combines the concepts of "surrogate" models (e.g. Gaussian processes) to accelerate the search, and the "acquisition" function to guide sampling from the posterior distribution, which both can effectively make a robust search toward the global optima of the cost function (see the Figure below). The sequential-nature of Bayesian optimization makes its parallelization complex and not natural as grid/random/evolutionary search, which is the obvious downside of Bayesian optimization.



Edited from source: https://mlconf.com/blog/lets-talk-bayesian-optimization/

For example, to tune few hyperparameters of DQN by Bayesian search, the parameter space can be defined as:

`learning_rate` = $\mathcal{U}(0.0001, 0.001)$ (Parameter type is continuous `float` uniform distribution)
`batch_size` = $\mathcal{U}\{16, 64\}$ (Parameter type is discrete `int` uniform distribution)
`target_network_update_freq` = $\mathcal{C}\{100, 500, 1000, 1500\}$ (Parameter type is categorical `grid`)
`exploration_fraction` = $\mathcal{C}\{0.05, 0.1, 0.15\}$ (Parameter type is categorical `grid`)

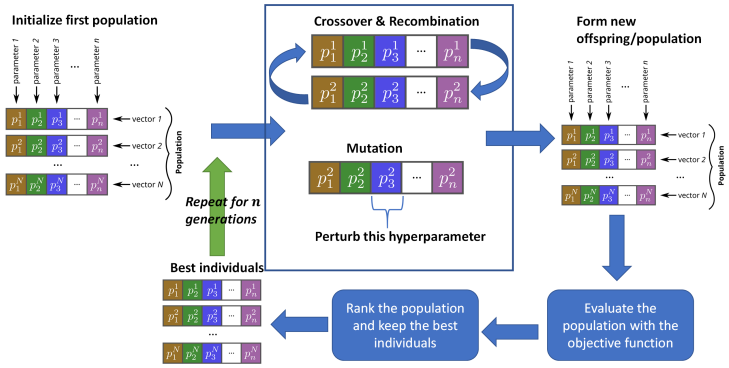
The cost of Bayesian search is determined by the total number of fitness evaluations provided by the user (`ncases`).

Evolutionary Search

See the *Evolutionary Search* section

Original paper: E. Bochinski, T. Senst and T. Sikora, "Hyper-parameter optimization for convolutional neural network committees based on evolutionary algorithms," 2017 IEEE International Conference on Image Processing (ICIP), Beijing, China, 2017, pp. 3924-3928, doi: 10.1109/ICIP.2017.8297018.

We have used a compact evolution strategy (ES) module for the purpose of tuning the hyperparameters of NEORL algorithms. See the *ES algorithm* section for more details about the (μ, λ) algorithm. To reduce the burden on the users, we specified and adapt all ES tuner hyperparameters, so the user needs to specify the hyperparameter space similar to grid, random, and Bayesian search methods. ES tuner leverages a population of individuals, where each individual represents a sample from the hyperparameter space. ES uses recombination, crossover, and mutation operations to improve the individuals from generation to the other. The best of the best individuals in all generations are reported as the top hyperparameter sets for the algorithm (See the Figure below).



Setting up the hyperparameter space for evolutionary search is quite similar to Bayesian search. Lastly, the cost of evolutionary search is determined by the total number of evaluated individuals in the population over all generations (`ngen * npop`), where `npop=10` and `ngen` is left for the user to decide.

2.2 Algorithms

This section highlights the supported NEORL algorithms and how to setup and solve a problem. The algorithms are classified in three separate categories.

2.2.1 Neural Algorithms (Reinforcement Learning)

Advantage Actor Critic (A2C)

A synchronous, deterministic variant of Asynchronous Advantage Actor Critic (A3C). It uses multiple workers to avoid the use of a replay buffer.

Original paper: https://arxiv.org/abs/1602.01783

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.rl.baselines.a2c.**A2C**(*policy*, *env*, *gamma=0.99*, *n_steps=5*, *vf_coef=0.25*, *ent_coef=0.01*, *max_grad_norm=0.5*, *learning_rate=0.0007*, *alpha=0.99*, *lr_schedule='constant'*, *verbose=0*, *seed=None*, *_init_setup_model=True*)

The A2C (Advantage Actor Critic) model class

Parameters

- **policy** – (ActorCriticPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, CnnLstmPolicy, ...)
- **env** – (NEORL environment or Gym environment) The environment to learn with PPO, either use NEORL method `CreateEnvironment` (see **below**) or construct your custom Gym environment
- **gamma** – (float) Discount factor
- **n_steps** – (int) The number of steps to run for each environment per update (i.e. batch size is n_steps * n_env where n_env is number of environment copies running in parallel)
- **vf_coef** – (float) Value function coefficient for the loss calculation
- **ent_coef** – (float) Entropy coefficient for the loss calculation
- **max_grad_norm** – (float) The maximum value for the gradient clipping
- **learning_rate** – (float) The learning rate
- **alpha** – (float) RMSProp decay parameter (default: 0.99)
- **lr_schedule** – (str) The type of scheduler for the learning rate update ('linear', 'constant', 'double_linear_con', 'middle_drop' or 'double_middle_drop')
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **seed** – (int) Seed for the pseudo-random generators (python, numpy, tensorflow). If None (default), use random seed.

learn(*total_timesteps*, *callback=None*, *log_interval=100*, *tb_log_name='A2C'*, *reset_num_timesteps=True*)

Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **callback** – (Union[callable, [callable], BaseCallback]) function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted. When the callback inherits from BaseCallback, you will have access to additional stages of the training (training start/end), please read the documentation for more details.

- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, *custom_objects=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Environment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **custom_objects** – (dict) Dictionary of objects to replace upon loading. If a variable is present in this dictionary as a key, it will not be deserialized and the corresponding item will be used instead. Similar to custom_objects in *keras.models.load_model*. Useful when you have an object in file that can not be deserialized.
- **kwargs** – extra arguments to change the model when loading

predict(*observation*, *state=None*, *mask=None*, *deterministic=False*)
Get the model's action from an observation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (np.ndarray, np.ndarray) the model's action and the next state (used in recurrent policies)

save(*save_path*, *cloudpickle=False*)
Save the current parameters to file

Parameters

- **save_path** – (str or file-like) The save location
- **cloudpickle** – (bool) Use older cloudpickle format instead of zip-archives.

class neorl.rl.make_env.**CreateEnvironment**(*method*, *fit*, *bounds*, *ncores=1*, *mode='max'*, *episode_length=50*)
A module to construct a fitness environment for certain algorithms that follow reinforcement learning approach of optimization

Parameters

- **method** – (str) the supported algorithms, choose either: `dqn`, `ppo`, `acktr`, `acer`, `a2c`.
- **fit** – (function) the fitness function
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **ncores** – (int) number of parallel processors

- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization (RL is default to `max`)
- **episode_length** – (int): number of individuals to evaluate before resetting the environment to random initial guess.

class neorl.utils.neorlcalls.**RLLogger**(*check_freq=1*, *plot_freq=None*, *n_avg_steps=10*, *pngname='history'*, *save_model=False*, *model_name='bestmodel.pkl'*, *save_best_only=True*, *verbose=False*)

Callback for logging data of RL algorathims (x,y), compatible with: A2C, ACER, ACKTR, DQN, PPO

Parameters

- **check_freq** – (int) logging frequency, e.g. 1 will record every time step
- **plot_freq** – (int) frequency of plotting the fitness progress (if `None`, plotter is deactivated)
- **n_avg_steps** – (int) if `plot_freq` is NOT `None`, then this is the number of timesteps to group to draw statistics for the plotter (e.g. 10 will group every 10 time steps to estimate min, max, mean, and std).
- **pngname** – (str) name of the plot that will be saved if `plot_freq` is NOT `None`.
- **save_model** – (bool) whether or not to save the RL neural network model (model is saved every `check_freq`)
- **model_name** – (str) name of the model to be saved if `save_model=True`
- **save_best_only** – (bool) if `save_model = True`, then this flag only saves the model if the fitness value improves.
- **verbose** – (bool) print updates to the screen

Example

Train an A2C agent to optimize the 5-D sphere function

```
from neorl import A2C
from neorl import MlpPolicy
from neorl import RLLogger
from neorl import CreateEnvironment

def Sphere(individual):
        """Sphere test objective function.
                F(x) = sum_{i=1}^d xi^2
                d=1,2,3,...
                Range: [-100,100]
                Minima: 0
        """
        return sum(x**2 for x in individual)

nx=5
bounds={}
for i in range(1,nx+1):
        bounds['x'+str(i)]=['float', -10, 10]


if __name__=='__main__':  #use this "if" block for parallel A2C!
```

(continues on next page)

(continued from previous page)

```
#create an enviroment class
env=CreateEnvironment(method='a2c', fit=Sphere,
                      bounds=bounds, mode='min', episode_length=50)
#create a callback function to log data
cb=RLLogger(check_freq=1)
#create an optimizer object based on the env object
a2c = A2C(MlpPolicy, env=env, n_steps=8, seed=1)
#optimise the enviroment class
a2c.learn(total_timesteps=2000, callback=cb)
#print the best results
print('--------------- A2C results ---------------')
print('The best value of x found:', cb.xbest)
print('The best value of y found:', cb.rbest)
```

Notes

- A2C belongs to the actor-critic family, and usually considered as the state-of-the-art in the reinforcement learning domain. A2C is parallel and supports all types of spaces.
- A2C shows sensitivity to `n_steps`, `vf_coef`, `ent_coef`, and `learning_rate`. It is always good to consider tuning these hyperparameters before using for optimization. In particular, `n_steps` is considered the most important parameter to tune for A2C. Always start with small `n_steps` and increase as needed.
- The cost of A2C equals to the `total_timesteps` in the `learn` function, where the original fitness function will be accessed `total_timesteps` times.
- See how A2C is used to solve two common combinatorial problems in *TSP* and *KP*.

Acknowledgment

Thanks to our fellows in stable-baselines, as we used their standalone RL implementation, which is utilized as a baseline to leverage advanced neuroevolution algorithms.

Hill, Ashley, et al. “Stable baselines.” (2018).

Actor-Critic with Experience Replay (ACER)

Sample Efficient Actor-Critic with Experience Replay (ACER) combines concepts of parallel agents from A2C and provides a replay memory as in DQN. ACER also includes truncated importance sampling with bias correction, stochastic dueling network architectures, and a new trust region policy optimization method.

Original paper: https://arxiv.org/abs/1611.01224

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces:
- Mixed Discrete/Continuous spaces:

Parameters

class neorl.rl.baselines.acer.**ACER**(*policy*, *env*, *gamma=0.99*, *n_steps=20*, *q_coef=0.5*, *ent_coef=0.01*, *max_grad_norm=10*, *learning_rate=0.0007*, *lr_schedule='linear'*, *buffer_size=5000*, *replay_ratio=4*, *replay_start=1000*, *verbose=0*, *seed=None*, *_init_setup_model=True*)

The ACER (Actor-Critic with Experience Replay) model class

Parameters

- **policy** – (ActorCriticPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, CnnLstmPolicy, ...)
- **env** – (NEORL environment or Gym environment) The environment to learn with PPO, either use NEORL method `CreateEnvironment` (see **below**) or construct your custom Gym environment
- **gamma** – (float) The discount value
- **n_steps** – (int) The number of steps to run for each environment per update (i.e. batch size is n_steps * n_env where n_env is number of environment copies running in parallel)
- **q_coef** – (float) The weight for the loss on the Q value
- **ent_coef** – (float) The weight for the entropy loss
- **max_grad_norm** – (float) The clipping value for the maximum gradient
- **learning_rate** – (float) The initial learning rate for the RMS prop optimizer
- **lr_schedule** – (str) The type of scheduler for the learning rate update ('linear', 'constant', 'double_linear_con', 'middle_drop' or 'double_middle_drop')
- **buffer_size** – (int) The buffer size in number of steps
- **replay_ratio** – (float) The number of replay learning per on policy learning on average, using a poisson distribution
- **replay_start** – (int) The minimum number of steps in the buffer, before experience replay starts
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **seed** – (int) Seed for the pseudo-random generators (python, numpy, tensorflow). If None (default), use random seed.

learn(*total_timesteps*, *callback=None*, *log_interval=100*, *tb_log_name='ACER'*, *reset_num_timesteps=True*)

Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **callback** – (Union[callable, [callable], BaseCallback]) function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted. When the callback inherits from BaseCallback, you will have access to additional stages of the training (training start/end), please read the documentation for more details.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, *custom_objects=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Environment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **custom_objects** – (dict) Dictionary of objects to replace upon loading. If a variable is present in this dictionary as a key, it will not be deserialized and the corresponding item will be used instead. Similar to custom_objects in *keras.models.load_model*. Useful when you have an object in file that can not be deserialized.
- **kwargs** – extra arguments to change the model when loading

predict(*observation*, *state=None*, *mask=None*, *deterministic=False*)
Get the model's action from an observation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (np.ndarray, np.ndarray) the model's action and the next state (used in recurrent policies)

save(*save_path*, *cloudpickle=False*)
Save the current parameters to file

Parameters

- **save_path** – (str or file-like) The save location
- **cloudpickle** – (bool) Use older cloudpickle format instead of zip-archives.

class neorl.rl.make_env.**CreateEnvironment**(*method*, *fit*, *bounds*, *ncores=1*, *mode='max'*, *episode_length=50*)
A module to construct a fitness environment for certain algorithms that follow reinforcement learning approach of optimization

Parameters

- **method** – (str) the supported algorithms, choose either: `dqn`, `ppo`, `acktr`, `acer`, `a2c`.

- **fit** – (function) the fitness function
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **ncores** – (int) number of parallel processors
- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization (RL is default to `max`)
- **episode_length** – (int): number of individuals to evaluate before resetting the environment to random initial guess.

class neorl.utils.neorlcalls.**RLLogger**(*check_freq=1*, *plot_freq=None*, *n_avg_steps=10*, *pngname='history'*, *save_model=False*, *model_name='bestmodel.pkl'*, *save_best_only=True*, *verbose=False*)

Callback for logging data of RL algorathims (x,y), compatible with: A2C, ACER, ACKTR, DQN, PPO

Parameters

- **check_freq** – (int) logging frequency, e.g. 1 will record every time step
- **plot_freq** – (int) frequency of plotting the fitness progress (if `None`, plotter is deactivated)
- **n_avg_steps** – (int) if `plot_freq` is NOT `None`, then this is the number of timesteps to group to draw statistics for the plotter (e.g. 10 will group every 10 time steps to estimate min, max, mean, and std).
- **pngname** – (str) name of the plot that will be saved if `plot_freq` is NOT `None`.
- **save_model** – (bool) whether or not to save the RL neural network model (model is saved every `check_freq`)
- **model_name** – (str) name of the model to be saved if `save_model=True`
- **save_best_only** – (bool) if `save_model = True`, then this flag only saves the model if the fitness value improves.
- **verbose** – (bool) print updates to the screen

Example

Train an ACER agent to optimize the 5-D discrete sphere function

```
from neorl import ACER
from neorl import MlpPolicy
from neorl import RLLogger
from neorl import CreateEnvironment

def Sphere(individual):
        """Sphere test objective function.
                F(x) = sum_{i=1}^d xi^2
                d=1,2,3,...
                Range: [-100,100]
                Minima: 0
        """
        return sum(x**2 for x in individual)
```

(continues on next page)

(continued from previous page)

```
nx=5
bounds={}
for i in range(1,nx+1):
        bounds['x'+str(i)]=['int', -100, 100]

if __name__=='__main__':  #use this "if" block for parallel ACER!

    #create an enviroment class
    env=CreateEnvironment(method='acer', fit=Sphere,
                          bounds=bounds, mode='min', episode_length=50)
    #create a callback function to log data
    cb=RLLogger(check_freq=1)
    #create an acer object based on the env object
    acer = ACER(MlpPolicy, env=env, n_steps=25, q_coef=0.55, ent_coef=0.02, seed=1)
    #optimise the enviroment class
    acer.learn(total_timesteps=2000, callback=cb)
    #print the best results
    print('--------------- ACER results ---------------')
    print('The best value of x found:', cb.xbest)
    print('The best value of y found:', cb.rbest)
```

Notes

- ACER can be observed as the parallel version of DQN with additional enhancements. ACER is also restricted to discrete spaces.
- ACER shows sensitivity to `n_steps`, `q_coef`, and `ent_coef`. It is always good to consider tuning these hyperparameters before using for optimization. In particular, `n_steps` is considered the most important parameter to tune.
- The cost of ACER equals to the `total_timesteps` in the `learn` function, where the original fitness function will be accessed `total_timesteps` times.
- See how ACER is used to solve two common combinatorial problems in *TSP* and *KP*.

Acknowledgment

Thanks to our fellows in stable-baselines, as we used their standalone RL implementation, which is utilized as a baseline to leverage advanced neuroevolution algorithms.

Hill, Ashley, et al. “Stable baselines.” (2018).

Actor Critic using Kronecker-Factored Trust Region (ACKTR)

Actor Critic using Kronecker-Factored Trust Region (ACKTR) uses Kronecker-factored approximate curvature (K-FAC) for trust region optimization. ACKTR uses K-FAC to allow more efficient inversion of the covariance matrix of the gradient. ACKTR also extends the natural policy gradient algorithm to optimize value functions via Gauss-Newton approximation.

Original paper: https://arxiv.org/abs/1708.05144

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.rl.baselines.acktr.**ACKTR**(*policy*, *env*, *gamma=0.99*, *n_steps=20*, *ent_coef=0.01*, *vf_coef=0.25*, *vf_fisher_coef=1.0*, *learning_rate=0.25*, *max_grad_norm=0.5*, *kfac_clip=0.001*, *lr_schedule='linear'*, *verbose=0*, *seed=None*, *_init_setup_model=True*)

The ACKTR (Actor Critic using Kronecker-Factored Trust Region) model class4

Parameters

- **policy** – (ActorCriticPolicy or str) The policy model to use (e.g. MlpPolicy)
- **env** – (NEORL environment or Gym environment) The environment to learn with PPO, either use NEORL method `CreateEnvironment` (see **below**) or construct your custom Gym environment
- **gamma** – (float) Discount factor
- **n_steps** – (int) The number of steps to run for each environment
- **ent_coef** – (float) The weight for the entropy loss
- **vf_coef** – (float) The weight for the loss on the value function
- **vf_fisher_coef** – (float) The weight for the fisher loss on the value function
- **learning_rate** – (float) The initial learning rate for the RMS prop optimizer
- **max_grad_norm** – (float) The clipping value for the maximum gradient
- **kfac_clip** – (float) gradient clipping for Kullback-Leibler
- **lr_schedule** – (str) The type of scheduler for the learning rate update ('linear', 'constant', 'double_linear_con', 'middle_drop' or 'double_middle_drop')
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **seed** – (int) Seed for the pseudo-random generators (python, numpy, tensorflow). If None (default), use random seed.

learn(*total_timesteps*, *callback=None*, *log_interval=100*, *tb_log_name='ACKTR'*, *reset_num_timesteps=True*)

Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **callback** – (Union[callable, [callable], BaseCallback]) function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted. When the callback inherits from BaseCallback, you will have access

to additional stages of the training (training start/end), please read the documentation for more details.

- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, *custom_objects=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Environment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **custom_objects** – (dict) Dictionary of objects to replace upon loading. If a variable is present in this dictionary as a key, it will not be deserialized and the corresponding item will be used instead. Similar to custom_objects in *keras.models.load_model*. Useful when you have an object in file that can not be deserialized.
- **kwargs** – extra arguments to change the model when loading

predict(*observation*, *state=None*, *mask=None*, *deterministic=False*)
Get the model's action from an observation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (np.ndarray, np.ndarray) the model's action and the next state (used in recurrent policies)

save(*save_path*, *cloudpickle=False*)
Save the current parameters to file

Parameters

- **save_path** – (str or file-like) The save location
- **cloudpickle** – (bool) Use older cloudpickle format instead of zip-archives.

class neorl.rl.make_env.**CreateEnvironment**(*method*, *fit*, *bounds*, *ncores=1*, *mode='max'*, *episode_length=50*)
A module to construct a fitness environment for certain algorithms that follow reinforcement learning approach of optimization

Parameters

- **method** – (str) the supported algorithms, choose either: `dqn`, `ppo`, `acktr`, `acer`, `a2c`.
- **fit** – (function) the fitness function

- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **ncores** – (int) number of parallel processors
- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization (RL is default to `max`)
- **episode_length** – (int): number of individuals to evaluate before resetting the environment to random initial guess.

class neorl.utils.neorlcalls.**RLLogger**(*check_freq=1*, *plot_freq=None*, *n_avg_steps=10*, *pngname='history'*, *save_model=False*, *model_name='bestmodel.pkl'*, *save_best_only=True*, *verbose=False*)

Callback for logging data of RL algorathims (x,y), compatible with: A2C, ACER, ACKTR, DQN, PPO

Parameters

- **check_freq** – (int) logging frequency, e.g. 1 will record every time step
- **plot_freq** – (int) frequency of plotting the fitness progress (if `None`, plotter is deactivated)
- **n_avg_steps** – (int) if `plot_freq` is NOT `None`, then this is the number of timesteps to group to draw statistics for the plotter (e.g. 10 will group every 10 time steps to estimate min, max, mean, and std).
- **pngname** – (str) name of the plot that will be saved if `plot_freq` is NOT `None`.
- **save_model** – (bool) whether or not to save the RL neural network model (model is saved every `check_freq`)
- **model_name** – (str) name of the model to be saved if `save_model=True`
- **save_best_only** – (bool) if `save_model = True`, then this flag only saves the model if the fitness value improves.
- **verbose** – (bool) print updates to the screen

Example

Train an ACKTR agent to optimize the 5-D sphere function

```
from neorl import ACKTR
from neorl import MlpPolicy
from neorl import RLLogger
from neorl import CreateEnvironment

def Sphere(individual):
        """Sphere test objective function.
                F(x) = sum_{i=1}^d xi^2
                d=1,2,3,...
                Range: [-100,100]
                Minima: 0
        """
        return sum(x**2 for x in individual)

nx=5
```

(continues on next page)

(continued from previous page)

```
bounds={}
for i in range(1,nx+1):
        bounds['x'+str(i)]=['float', -10, 10]

if __name__=='__main__':  #use this "if" block for parallel ACKTR!

    #create an enviroment class
    env=CreateEnvironment(method='acktr', fit=Sphere,
                          bounds=bounds, mode='min', episode_length=50)
    #create a callback function to log data
    cb=RLLogger(check_freq=1)
    #create an acktr object based on the env object
    acktr = ACKTR(MlpPolicy, env=env, n_steps=12, seed=1)
    #optimise the enviroment class
    acktr.learn(total_timesteps=2000, callback=cb)
    #print the best results
    print('---------------- ACKTR results ----------------')
    print('The best value of x found:', cb.xbest)
    print('The best value of y found:', cb.rbest)
```

Notes

- ACKTR belongs to the actor-critic family of reinforcement learning. ACKTR uses some methods to increase the efficiency of reinforcement learning gradient-based search. ACKTR is parallel and supports all types of spaces.
- ACKTR shows sensitivity to `n_steps`, `vf_fisher_coef`, `vf_coef`, and `learning_rate`. It is always good to consider tuning these hyperparameters before using for optimization. In particular, `n_steps` is considered the most important parameter to tune for ACKTR. Always start with small `n_steps` and increase as needed.
- The cost of ACKTR equals to the `total_timesteps` in the `learn` function, where the original fitness function will be accessed `total_timesteps` times.
- See how ACKTR is used to solve two common combinatorial problems in *TSP* and *KP*.

Acknowledgment

Thanks to our fellows in stable-baselines, as we used their standalone RL implementation, which is utilized as a baseline to leverage advanced neuroevolution algorithms.

Hill, Ashley, et al. “Stable baselines.” (2018).

Deep Q Learning (DQN)

Deep Q Network (DQN) and its extensions (Double-DQN, Dueling-DQN, Prioritized Experience Replay).

Original papers:

- DQN paper: https://arxiv.org/abs/1312.5602
- Dueling DQN: https://arxiv.org/abs/1511.06581
- Double-Q Learning: https://arxiv.org/abs/1509.06461
- Prioritized Experience Replay: https://arxiv.org/abs/1511.05952

What can you use?

- Multi processing:
- Discrete spaces: ✓
- Continuous spaces:
- Mixed Discrete/Continuous spaces:

Parameters

class neorl.rl.baselines.deepq.**DQN**(*policy*, *env*, *gamma=0.99*, *learning_rate=0.0005*, *buffer_size=50000*, *exploration_fraction=0.1*, *eps_final=0.02*, *eps_init=1.0*, *train_freq=1*, *batch_size=32*, *learning_starts=1000*, *target_network_update_freq=500*, *prioritized_replay=True*, *verbose=0*, *seed=None*, *_init_setup_model=True*)

The DQN model class

Parameters

- **policy** – (DQNPolicy or str) The policy model to use (MlpPolicy, CnnPolicy, LnMlpPolicy, ...)
- **env** – (NEORL environment or Gym environment) The environment to learn with PPO, either use NEORL method `CreateEnvironment` (see **below**) or construct your custom Gym environment
- **gamma** – (float) discount factor
- **learning_rate** – (float) learning rate for adam optimizer
- **buffer_size** – (int) size of the replay buffer
- **exploration_fraction** – (float) fraction of entire training period over which the exploration rate is annealed
- **eps_final** – (float) final value of random action probability (e.g. 0.05)
- **eps_init** – (float) initial value of random action probability (e.g. 1.0)
- **train_freq** – (int) update the model every *train_freq* steps. set to None to disable printing
- **batch_size** – (int) size of a batched sampled from replay buffer for training
- **learning_starts** – (int) how many steps of the model to collect transitions for before learning starts
- **target_network_update_freq** – (int) update the target network every *target_network_update_freq* steps.
- **prioritized_replay** – (bool) if True prioritized experience replay buffer will be used.
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **seed** – (int) Seed for the pseudo-random generators (python, numpy, tensorflow). If None (default), use random seed.

learn(*total_timesteps*, *callback=None*, *log_interval=100*, *tb_log_name='DQN'*, *reset_num_timesteps=True*, *replay_wrapper=None*)

Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **callback** – (Union[callable, [callable], BaseCallback]) function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted. When the callback inherits from BaseCallback, you will have access to additional stages of the training (training start/end), please read the documentation for more details.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, *custom_objects=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Environment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **custom_objects** – (dict) Dictionary of objects to replace upon loading. If a variable is present in this dictionary as a key, it will not be deserialized and the corresponding item will be used instead. Similar to custom_objects in *keras.models.load_model*. Useful when you have an object in file that can not be deserialized.
- **kwargs** – extra arguments to change the model when loading

predict(*observation*, *state=None*, *mask=None*, *deterministic=True*)
Get the model's action from an observation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (np.ndarray, np.ndarray) the model's action and the next state (used in recurrent policies)

save(*save_path*, *cloudpickle=False*)
Save the current parameters to file

Parameters

- **save_path** – (str or file-like) The save location
- **cloudpickle** – (bool) Use older cloudpickle format instead of zip-archives.

class neorl.rl.make_env.**CreateEnvironment**(*method*, *fit*, *bounds*, *ncores=1*, *mode='max'*, *episode_length=50*)
A module to construct a fitness environment for certain algorithms that follow reinforcement learning approach of optimization

Parameters

- **method** – (str) the supported algorithms, choose either: `dqn`, `ppo`, `acktr`, `acer`, `a2c`.
- **fit** – (function) the fitness function
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **ncores** – (int) number of parallel processors
- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization (RL is default to `max`)
- **episode_length** – (int): number of individuals to evaluate before resetting the environment to random initial guess.

class neorl.utils.neorlcalls.**RLLogger**(*check_freq=1*, *plot_freq=None*, *n_avg_steps=10*, *pngname='history'*, *save_model=False*, *model_name='bestmodel.pkl'*, *save_best_only=True*, *verbose=False*)

Callback for logging data of RL algorathims (x,y), compatible with: A2C, ACER, ACKTR, DQN, PPO

Parameters

- **check_freq** – (int) logging frequency, e.g. 1 will record every time step
- **plot_freq** – (int) frequency of plotting the fitness progress (if `None`, plotter is deactivated)
- **n_avg_steps** – (int) if `plot_freq` is NOT `None`, then this is the number of timesteps to group to draw statistics for the plotter (e.g. 10 will group every 10 time steps to estimate min, max, mean, and std).
- **pngname** – (str) name of the plot that will be saved if `plot_freq` is NOT `None`.
- **save_model** – (bool) whether or not to save the RL neural network model (model is saved every `check_freq`)
- **model_name** – (str) name of the model to be saved if `save_model=True`
- **save_best_only** – (bool) if `save_model = True`, then this flag only saves the model if the fitness value improves.
- **verbose** – (bool) print updates to the screen

Example

Train a DQN agent to optimize the 5-D discrete sphere function

```
from neorl import DQN
from neorl import DQNPolicy
from neorl import RLLogger
from neorl import CreateEnvironment

def Sphere(individual):
        """Sphere test objective function.
                F(x) = sum_{i=1}^d xi^2
                d=1,2,3,...
                Range: [-100,100]
                Minima: 0
        """
```

(continues on next page)

(continued from previous page)

```
        #print(individual)
        return sum(x**2 for x in individual)

nx=5
bounds={}
for i in range(1,nx+1):
        bounds['x'+str(i)]=['int', -100, 100]

#create an enviroment class
env=CreateEnvironment(method='dqn',
                      fit=Sphere,
                      bounds=bounds,
                      mode='min',
                      episode_length=50)
#create a callback function to log data
cb=RLLogger(check_freq=1)
#create a RL object based on the env object
dqn = DQN(DQNPolicy, env=env, seed=1)
#optimise the enviroment class
dqn.learn(total_timesteps=2000, callback=cb)
#print the best results
print('--------------- DQN results ---------------')
print('The best value of x found:', cb.xbest)
print('The best value of y found:', cb.rbest)
```

Notes

- DQN is the most limited RL algorithm in the package with no multiprocessing and only restricted to discrete spaces. Nevertheless, DQN is considered the first and the heart of many deep RL algorithms.
- For parallel RL algorithm with Q-value support like DQN, use ACER.
- DQN shows sensitivity to `exploration_fraction`, `train_freq`, and `target_network_update_freq`. It is always good to consider tuning these hyperparameters before using for optimization.
- Activating `prioritized_replay` seems to improve DQN performance.
- The cost for DQN equals to the `total_timesteps` in the `learn` function, where the original fitness function will be accessed `total_timesteps` times.
- See how DQN is used to solve two common combinatorial problems in *TSP* and *KP*.

Acknowledgment

Thanks to our fellows in stable-baselines, as we used their standalone RL implementation, which is utilized as a baseline to leverage advanced neuroevolution algorithms.

Hill, Ashley, et al. “Stable baselines.” (2018).

Proximal Policy Optimisation (PPO)

The Proximal Policy Optimization algorithm combines ideas from A2C (having multiple workers) and TRPO (it uses a trust region to improve the actor). The idea is that after an update, the new policy should be not be too far from the old policy. For that, PPO uses clipping to avoid too large update.

Original paper: https://arxiv.org/abs/1707.06347

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.rl.baselines.ppo2.**PPO2**(*policy*, *env*, *gamma=0.99*, *n_steps=128*, *ent_coef=0.01*, *learning_rate=0.00025*, *vf_coef=0.5*, *max_grad_norm=0.5*, *lam=0.95*, *nminibatches=4*, *noptepochs=4*, *cliprange=0.2*, *verbose=0*, *seed=None*, *_init_setup_model=True*)

Proximal Policy Optimization algorithm

Parameters

- **policy** – (ActorCriticPolicy or str) The policy model to use (e.g. MlpPolicy)
- **env** – (NEORL environment or Gym environment) The environment to learn with PPO, either use NEORL method `CreateEnvironment` (see **below**) or construct your custom Gym environment
- **gamma** – (float) Discount factor
- **n_steps** – (int) The number of steps to run for each environment per update (i.e. batch size is n_steps * n_env where n_env is number of environment copies running in parallel)
- **ent_coef** – (float) Entropy coefficient for the loss calculation
- **learning_rate** – (float or callable) The learning rate, it can be a function
- **vf_coef** – (float) Value function coefficient for the loss calculation
- **max_grad_norm** – (float) The maximum value for the gradient clipping
- **lam** – (float) Factor for trade-off of bias vs variance for Generalized Advantage Estimator
- **nminibatches** – (int) Number of training minibatches per update. For recurrent policies, the number of environments run in parallel should be a multiple of nminibatches.
- **noptepochs** – (int) Number of epoch when optimizing the surrogate
- **cliprange** – (float or callable) Clipping parameter, it can be a function
- **verbose** – (int) the verbosity level: 0 none, 1 training information, 2 tensorflow debug
- **seed** – (int) Seed for the pseudo-random generators (python, numpy, tensorflow). If None (default), use random seed.

learn(*total_timesteps*, *callback=None*, *log_interval=1*, *tb_log_name='PPO2'*, *reset_num_timesteps=True*)
Return a trained model.

Parameters

- **total_timesteps** – (int) The total number of samples to train on
- **callback** – (Union[callable, [callable], BaseCallback]) function called at every steps with state of the algorithm. It takes the local and global variables. If it returns False, training is aborted. When the callback inherits from BaseCallback, you will have access to additional stages of the training (training start/end), please read the documentation for more details.
- **log_interval** – (int) The number of timesteps before logging.
- **tb_log_name** – (str) the name of the run for tensorboard log
- **reset_num_timesteps** – (bool) whether or not to reset the current timestep number (used in logging)

Returns (BaseRLModel) the trained model

classmethod load(*load_path*, *env=None*, *custom_objects=None*, ***kwargs*)
Load the model from file

Parameters

- **load_path** – (str or file-like) the saved parameter location
- **env** – (Gym Environment) the new environment to run the loaded model on (can be None if you only need prediction from a trained model)
- **custom_objects** – (dict) Dictionary of objects to replace upon loading. If a variable is present in this dictionary as a key, it will not be deserialized and the corresponding item will be used instead. Similar to custom_objects in *keras.models.load_model*. Useful when you have an object in file that can not be deserialized.
- **kwargs** – extra arguments to change the model when loading

predict(*observation*, *state=None*, *mask=None*, *deterministic=False*)
Get the model's action from an observation

Parameters

- **observation** – (np.ndarray) the input observation
- **state** – (np.ndarray) The last states (can be None, used in recurrent policies)
- **mask** – (np.ndarray) The last masks (can be None, used in recurrent policies)
- **deterministic** – (bool) Whether or not to return deterministic actions.

Returns (np.ndarray, np.ndarray) the model's action and the next state (used in recurrent policies)

save(*save_path*, *cloudpickle=False*)
Save the current parameters to file

Parameters

- **save_path** – (str or file-like) The save location
- **cloudpickle** – (bool) Use older cloudpickle format instead of zip-archives.

class neorl.rl.make_env.**CreateEnvironment**(*method*, *fit*, *bounds*, *ncores=1*, *mode='max'*, *episode_length=50*)

A module to construct a fitness environment for certain algorithms that follow reinforcement learning approach of optimization

Parameters

- **method** – (str) the supported algorithms, choose either: `dqn`, `ppo`, `acktr`, `acer`, `a2c`.
- **fit** – (function) the fitness function
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **ncores** – (int) number of parallel processors
- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization (RL is default to `max`)
- **episode_length** – (int): number of individuals to evaluate before resetting the environment to random initial guess.

class neorl.utils.neorlcalls.**RLLogger**(*check_freq=1*, *plot_freq=None*, *n_avg_steps=10*, *pngname='history'*, *save_model=False*, *model_name='bestmodel.pkl'*, *save_best_only=True*, *verbose=False*)

Callback for logging data of RL algorathims (x,y), compatible with: A2C, ACER, ACKTR, DQN, PPO

Parameters

- **check_freq** – (int) logging frequency, e.g. 1 will record every time step
- **plot_freq** – (int) frequency of plotting the fitness progress (if `None`, plotter is deactivated)
- **n_avg_steps** – (int) if `plot_freq` is NOT `None`, then this is the number of timesteps to group to draw statistics for the plotter (e.g. 10 will group every 10 time steps to estimate min, max, mean, and std).
- **pngname** – (str) name of the plot that will be saved if `plot_freq` is NOT `None`.
- **save_model** – (bool) whether or not to save the RL neural network model (model is saved every `check_freq`)
- **model_name** – (str) name of the model to be saved if `save_model=True`
- **save_best_only** – (bool) if `save_model = True`, then this flag only saves the model if the fitness value improves.
- **verbose** – (bool) print updates to the screen

Example

Train a PPO agent to optimize the 5-D sphere function

```
from neorl import PPO2
from neorl import MlpPolicy
from neorl import RLLogger
from neorl import CreateEnvironment

def Sphere(individual):
```

(continues on next page)

(continued from previous page)

```
        """Sphere test objective function.
                F(x) = sum_{i=1}^d xi^2
                d=1,2,3,...
                Range: [-100,100]
                Minima: 0
        """
        return sum(x**2 for x in individual)

nx=5
bounds={}
for i in range(1,nx+1):
        bounds['x'+str(i)]=['float', -10, 10]

if __name__=='__main__':  #use this "if" block for parallel PPO!

    #create an enviroment class
    env=CreateEnvironment(method='ppo', fit=Sphere,
                          bounds=bounds, mode='min', episode_length=50)

    #create a callback function to log data
    cb=RLLogger(check_freq=1)
    #create a RL object based on the env object
    ppo = PPO2(MlpPolicy, env=env, n_steps=12, seed=1)
    #optimise the enviroment class
    ppo.learn(total_timesteps=2000, callback=cb)
    #print the best results
    print('---------------- PPO results ----------------')
    print('The best value of x found:', cb.xbest)
    print('The best value of y found:', cb.rbest)
```

Notes

- PPO is the most popular RL algorithm due to its robustness. PPO is parallel and supports all types of spaces.
- PPO shows sensitivity to `n_steps`, `vf_coef`, `ent_coef`, and `lam`. It is always good to consider tuning these hyperparameters before using for optimization. In particular, `n_steps` is considered the most important parameter to tune for PPO. Always start with small `n_steps` and increase as needed.
- For PPO, always ensure that `ncores * n_steps` is divisible by `nminibatches`. For example, if `nminibatches=4`, then `ncores=12`/`n_steps=5` setting works, while `ncores=5`/`n_steps=5` will fail. For tuning purposes, it is recommended to choose `ncores` divisible by `nminibatches` so that you can change `n_steps` more freely.
- The cost of PPO equals to the `total_timesteps` in the `learn` function, where the original fitness function will be accessed `total_timesteps` times.
- See how PPO is used to solve two common combinatorial problems in *TSP* and *KP*.

Acknowledgment

Thanks to our fellows in stable-baselines, as we used their standalone RL implementation, which is utilized as a baseline to leverage advanced neuroevolution algorithms.

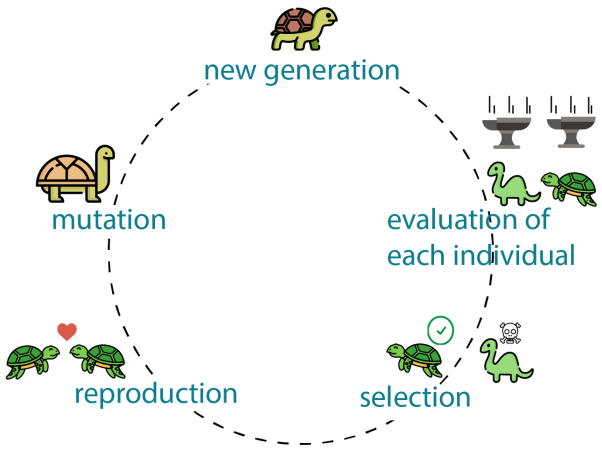
Hill, Ashley, et al. "Stable baselines." (2018).

2.2.2 Evolutionary and Swarm Algorithms

Evolution Strategies (μ, λ) (ES)

A module for the evolution strategies (μ, λ) with adaptive strategy vectors.

Original paper: Bäck, T., Fogel, D. B., Michalewicz, Z. (Eds.). (2018). Evolutionary computation 1: Basic algorithms and operators. CRC press.



What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.evolu.es.**ES**(*mode*, *bounds*, *fit*, *lambda_=60*, *mu=30*, *cxmode='cx2point'*, *alpha=0.5*, *cxpb=0.6*, *mutpb=0.3*, *smin=0.01*, *smax=0.5*, *clip=True*, *ncores=1*, *seed=None*, ***kwargs*)

Parallel Evolution Strategies

Parameters

- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization

- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **lambda_** – (int) total number of individuals in the population
- **mu** – (int): number of individuals to survive to the next generation, mu < lambda_
- **cxmode** – (str): the crossover mode, either 'cx2point' or 'blend'
- **alpha** – (float) Extent of the blending between [0,1], the blend crossover randomly selects a child in the range [x1-alpha(x2-x1), x2+alpha(x2-x1)] (Only used for cxmode='blend')
- **cxpb** – (float) population crossover probability between [0,1]
- **mutpb** – (float) population mutation probability between [0,1]
- **smin** – (float): minimum bound for the strategy vector
- **smax** – (float): maximum bound for the strategy vector
- **ncores** – (int) number of parallel processors
- **seed** – (int) random seed for sampling

evolute (*ngen*, *x0=None*, *verbose=False*)

This function evolutes the ES algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) the initial position of the swarm particles
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and a list of fitness history)

Example

```
from neorl import ES

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
            F(x) = sum_{i=1}^d xi^2
            d=1,2,3,...
            Range: [-100,100]
            Minima: 0
    """
    check=all([item >= BOUNDS['x'+str(i+1)][1] for i,item in enumerate(individual)])
↪and all([item <= BOUNDS['x'+str(i+1)][2] for i,item in enumerate(individual)])
    if not check:
        raise Exception ('--error check fails')
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]
```

(continues on next page)

(continued from previous page)

```
es=ES(mode='min', bounds=BOUNDS, fit=FIT, lambda_=80, mu=40, mutpb=0.1,
     cxmode='blend', cxpb=0.7, ncores=1, seed=1)
x_best, y_best, es_hist=es.evolute(ngen=5, verbose=1)
```

Notes

- Too large population mutation rate `mutpb` could destroy the population, the recommended range for this variable is between 0.01-0.4.
- Too large `smax` will allow the individual to be perturbed in a large rate.
- Too small `cxpb` and `mutpb` reduce ES exploration, and increase the likelihood of falling in a local optima.
- Usually, population size `lambda_` between 60-100 shows good performance along with `mu=0.5*lambda_`.
- Look for an optimal balance between `lambda_` and `ngen`, it is recommended to minimize population size to allow for more generations.
- Total number of cost evaluations for ES is `lambda_ * (ngen + 1)`.
- `cxmode='blend'` with `alpha=0.5` may perform better than `cxmode='cx2point'`.

Particle Swarm Optimisation (PSO)

A module for particle swarm optimisation with three different speed mechanisms.

Original papers:

- Kennedy, J., Eberhart, R. (1995). Particle swarm optimization. In: Proceedings of ICNN'95-international conference on neural networks (Vol. 4, pp. 1942-1948), IEEE.
- Kennedy, J., & Eberhart, R. C. (1997). A discrete binary version of the particle swarm algorithm. In: 1997 IEEE International conference on systems, man, and cybernetics. Computational cybernetics and simulation (Vol. 5, pp. 4104-4108), IEEE.
- Clerc, M., Kennedy, J. (2002). The particle swarm-explosion, stability, and convergence in a multidimensional complex space. IEEE transactions on Evolutionary Computation, 6(1), 58-73.

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.evolu.pso.**PSO**(*mode*, *bounds*, *fit*, *npar=50*, *c1=2.05*, *c2=2.05*, *speed_mech='constric'*, *ncores=1*, *seed=None*)

Parallel Particle Swarm Optimisaion (PSO) module

Parameters

- **mode** – (str) problem type, either "min" for minimization problem or "max" for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **npar** – (int) number of particles in the swarm
- **c1** – (float) cognitive speed constant
- **c2** – (float) social speed constant
- **speed_mech** – (str) type of speed mechanism to update particle velocity, choose between `constric`, `timew`, `globw`.

- **ncores** – (int) number of parallel processors
- **seed** – (int) random seed for sampling

evolute (*ngen*, *x0=None*, *verbose=False*, ***kwargs*)
This function evolutes the PSO algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) the initial position of the swarm particles
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and a list of fitness history)

Example

```
from neorl import PSO

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
                    F(x) = sum_{i=1}^d xi^2
                    d=1,2,3,...
                    Range: [-100,100]
                    Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

#setup and evolute PSO
pso=PSO(mode='min', bounds=BOUNDS, fit=FIT, c1=2.05, c2=2.1, npar=50,
                speed_mech='constric', ncores=1, seed=1)
x_best, y_best, pso_hist=pso.evolute(ngen=100, verbose=1)
```

Notes

- Always try the three speed mechanisms via `speed_mech` when you solve any problem.
- Keep c1, c2 > 2.0 when using `speed_mech='constric'`.
- `speed_mech=timew` uses a time-dependent inertia factor, where inertia `w` is annealed over PSO generations.
- `speed_mech=globw` uses a ratio of swarm global position to local position to define inertia factor, and this factor is updated every generation.
- Look for an optimal balance between `npar` and `ngen`, it is recommended to minimize particle size to allow for more generations.
- Total number of cost evaluations for PSO is `npar * (ngen + 1)`.

Heterogeneous comprehensive learning particle swarm optimization (HCLPSO)

A module for parallel heterogeneous comprehensive learning particle swarm optimization with both constriction and inertia weight support. HCLPSO leverages two subpopulations, one focuses on exploration (you) and one focuses on exploitation (your friend).

Original paper: Lynn, N., Suganthan, P. N. (2015). Heterogeneous comprehensive learning particle swarm optimization with enhanced exploration and exploitation. Swarm and Evolutionary Computation, 24, 11-24.

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.evolu.hclpso.**HCLPSO**(*mode*, *bounds*, *fit*, *g1=15*, *g2=25*, *int_transform='nearest_int'*, *ncores=1*, *seed=None*)

Heterogeneous comprehensive learning particle swarm optimization (HCLPSO)

Parameters

- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **g1** – (int): number of particles in the exploration group
- **g2** – (int): number of particles in the exploitation group (total swarm size is `g1 + g2`)
- **int_transform** – (str): method of handling int/discrete variables, choose from: `nearest_int`, `sigmoid`, `minmax`.
- **ncores** – (int) number of parallel processors (must be `<= g1+g2`)
- **seed** – (int) random seed for sampling

evolute(*ngen*, *x0=None*, *verbose=False*)

This function evolutes the HCLPSO algorithm for a number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) initial position of the particles (must be of same size as `g1 + g2`)
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and dictionary containing major search results)

Example

```
from neorl import HCLPSO

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
                    F(x) = sum_{i=1}^d xi^2
                    d=1,2,3,...
                    Range: [-100,100]
                    Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

#setup and evolute HCLPSO
hclpso=HCLPSO(mode='min', bounds=BOUNDS, g1=15, g2=25, fit=FIT, ncores=1, seed=1)
x_best, y_best, hclpso_hist=hclpso.evolute(ngen=120, verbose=1)
```

Notes

- The number of particles in the exploration subgroup (`g1`) and exploitation subgroup (`g2`) are the only hyperparameters for HCLPSO. In the original algorithm, `g1` tends to be smaller than `g2`.
- HCLPSO provides time dependent (annealing) behavior for all major PSO hyperparameters over the number of search generations (`ngen`). The cognitive speed constant (`c1`) is linearly annealed from 2.5-0.5, social speed constant (`c2`) is annealed from 0.5-2.5, inertia weight (`w`) is annealed from 0.99-0.2, while constriction coefficient (`K`) is annealed from 3-1.5. Therefore, the HCLPSO user does not need to tune these values.
- Look for an optimal balance between `g1`, `g2`, and `ngen`, it is recommended to minimize particle size to allow for more generations.
- Total number of cost evaluations for PSO is `(g1 + g2) * (ngen + 1)`.

Differential Evolution (DE)

A module for differential evolution (DE) that optimizes a problem by iteratively trying to improve a candidate solution. DE maintains the population of candidate solutions and creating new candidate solutions by combining existing ones according to simple combination methods. The candidate solution with the best score/fitness is reported by DE.

Original paper: Storn, Rainer, and Kenneth Price. "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces." Journal of global optimization 11.4 (1997): 341-359.

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.evolu.de.**DE**(*mode*, *bounds*, *fit*, *npop=50*, *F=0.5*, *CR=0.3*, *int_transform='nearest_int'*, *ncores=1*, *seed=None*, ***kwargs*)

Parallel Differential Evolution

Parameters

- **mode** – (str) problem type, either "min" for minimization problem or "max" for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **npop** – (int) number of individuals in the population
- **F** – (float) differential/mutation weight between [0,2]
- **CR** – (float) crossover probability between [0,1]
- **int_transform** – (str): method of handling int/discrete variables, choose from: `nearest_int`, `sigmoid`, `minmax`.
- **ncores** – (int) number of parallel processors
- **seed** – (int) random seed for sampling

evolute(*ngen*, *x0=None*, *verbose=False*)

This function evolutes the DE algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) the initial individuals of the population
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and a list of fitness history)

Example

```
from neorl import DE

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
            F(x) = sum_{i=1}^d xi^2
            d=1,2,3,...
            Range: [-100,100]
            Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

de=DE(mode='min', bounds=BOUNDS, fit=FIT, npop=60, F=0.5, CR=0.7, ncores=1, seed=1)
x_best, y_best, de_hist=de.evolute(ngen=100, verbose=1)
```

Notes

- Start with a crossover probability `CR` considerably lower than one (e.g. 0.2-0.3). If no convergence is achieved, increase to higher levels (e.g. 0.8-0.9)
- `F` is usually chosen between [0.5, 1].
- The higher the population size `npop`, the lower one should choose the weighting factor `F`
- You may start with `npop` =10*d, where d is the number of input parameters to optimise (degrees of freedom).
- Total number of cost evaluations for DE is `2 * npop * ngen`.

Exponential Natural Evolution Strategies (XNES)

A module for the exponential natural evolution strategies with adaptive sampling.

Original paper: Glasmachers, T., Schaul, T., Yi, S., Wierstra, D., Schmidhuber, J. (2010). Exponential natural evolution strategies. In: Proceedings of the 12th annual conference on Genetic and evolutionary computation (pp. 393-400).

What can you use?

- Multi processing: ✓
- Discrete spaces:
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces:

Parameters

class neorl.evolu.xnes.**XNES**(*mode*, *bounds*, *fit*, *A=None*, *npop=None*, *eta_mu=1.0*, *eta_sigma=None*, *eta_Bmat=None*, *adapt_sampling=False*, *ncores=1*, *seed=None*)

Exponential Natural Evolution Strategies

Parameters

- **mode** – (str) problem type, either "min" for minimization problem or "max" for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **npop** – (int) total number of individuals in the population (default: if None, it will make an approximation, see **Notes** below)
- **A** – (np.array): initial guess of the covariance matrix A (default: identity matrix, see **Notes** below)
- **eta_mu** – (float) learning rate for updating the center of the search distribution `mu` (see **Notes** below)
- **eta_sigma** – (float) learning rate for updating the step size `sigma` (default: if None, it will make an approximation, see **Notes** below)
- **eta_Bmat** – (float) learning rate for updating the normalized transformation matrix `B` (default: if None, it will make an approximation, see **Notes** below)
- **adapt_sampling** – (bool): activate the adaption sampling option
- **ncores** – (int) number of parallel processors
- **seed** – (int) random seed for sampling

evolute(*ngen*, *x0=None*, *verbose=False*)

This function evolutes the XNES algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list) initial guess for the search (must be of same size as `len(bounds)`)
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and dictionary containing major search results)

Example

```
from neorl import XNES

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
            F(x) = sum_{i=1}^d xi^2
            d=1,2,3,...
```

(continues on next page)

(continued from previous page)

```
            Range: [-100,100]
            Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

xnes=XNES(mode='min', bounds=BOUNDS, fit=FIT, npop=50, eta_mu=0.9,
          eta_sigma=0.25, adapt_sampling=True, ncores=1, seed=1)
x_best, y_best, xnes_hist=xnes.evolute(ngen=100, x0=[25,25,25,25,25], verbose=1)
```

Notes

- XNES is controlled by three major search parameters: the center of the search distribution μ (`mu`), the step size σ (`sigma`), and the normalized transformation matrix B (`B`) .
- The user provides initial guess of the covariance matrix A using the argument `A`. XNES applies $A = \sigma.B$ to determine the initial step size σ (`sigma`) and the initial transformation matrix B (`B`).
- If `A` is not provided, XNES starts from an identity matrix of size `d`, i.e. `np.eye(d)`, where `d` is the size of the parameter space.
- If `npop` is `None`, the following formula is used: $npop = Integer\{4 + [3log(d)]\}$, where `d` is the size of the parameter space.
- The center of the search distribution μ (`mu`) is updated by the learning rate `eta_mu`.
- The step size σ (`sigma`) is updated by the learning rate `eta_sigma`. If `eta_sigma` is `None`, the following formula is used: $eta\_sigma = \frac{3}{5}\frac{3+log(d)}{d\sqrt{d}}$, where `d` is the size of the parameter space.
- The normalized transformation matrix B (`B`) is updated by the learning rate `eta_Bmat`. If `eta_Bmat` is `None`, the following formula is used: $eta\_Bmat = \frac{3}{5}\frac{3+log(d)}{d\sqrt{d}}$, where `d` is the size of the parameter space.
- Activating the option `adapt_sampling` may help improving the performance of XNES.
- Look for an optimal balance between `npop` and `ngen`, it is recommended to minimize population size to allow for more generations.
- Total number of cost evaluations for XNES is `npop` * `ngen`.

Grey Wolf Optimizer (GWO)

A module for the Grey Wolf Optimizer with parallel computing support.

Original paper: Mirjalili, S., Mirjalili, S. M., & Lewis, A. (2014). Grey wolf optimizer. Advances in engineering software, 69, 46-61.

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.evolu.gwo.**GWO**(*mode*, *bounds*, *fit*, *nwolves=5*, *int_transform='nearest_int'*, *ncores=1*, *seed=None*)

Grey Wolf Optimizer

Parameters

- **mode** – (str) problem type, either "min" for minimization problem or "max" for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **nwolves** – (int): number of the grey wolves in the group
- **int_transform** – (str): method of handling int/discrete variables, choose from: `nearest_int`, `sigmoid`, `minmax`.
- **ncores** – (int) number of parallel processors (must be `<= nwolves`)
- **seed** – (int) random seed for sampling

evolute (*ngen*, *x0=None*, *verbose=False*, ***kwargs*)
This function evolutes the GWO algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) initial position of the wolves (must be of same size as `nwolves`)
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and dictionary containing major search results)

Example

```
from neorl import GWO
import matplotlib.pyplot as plt

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
            F(x) = sum_{i=1}^d xi^2
            d=1,2,3,...
            Range: [-100,100]
            Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

nwolves=5
gwo=GWO(mode='min', fit=FIT, bounds=BOUNDS, nwolves=nwolves, ncores=1, seed=1)
x_best, y_best, gwo_hist=gwo.evolute(ngen=100, verbose=1)

#-----
#or with fixed initial guess for all wolves (uncomment below)
#-----
#x0=[[-90, -85, -80, 70, 90] for i in range(nwolves)]
#x_best, y_best, gwo_hist=gwo.evolute(ngen=100, x0=x0)

plt.figure()
plt.plot(gwo_hist['alpha_wolf'], label='alpha_wolf')
plt.plot(gwo_hist['beta_wolf'], label='beta_wolf')
plt.plot(gwo_hist['delta_wolf'], label='delta_wolf')
plt.plot(gwo_hist['fitness'], label='best')
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.legend()
plt.show()
```

Notes

- GWO assigns the best fitness to the first wolf (called **Alpha**), second best fitness to **Beta** wolf, third best fitness to **Delta** wolf, while the remaining wolves in the group are called **Omega**, which follow the leadership and position of Alpha, Beta, and Delta.
- `ncores` argument evaluates the fitness of all wolves in the group in parallel. Therefore, set `ncores <= nwolves` for most optimal resource allocation.
- Look for an optimal balance between `nwolves` and `ngen`, it is recommended to minimize the number of `nwolves` to allow for more updates and more generations.
- Total number of cost evaluations for GWO is `nwolves` * `ngen`.

Simulated Annealing (SA)

A module for parallel Simulated Annealing. A Synchronous Approach with Occasional Enforcement of Best Solution.

Original paper: Onbaşoğlu, E., Özdamar, L. (2001). Parallel simulated annealing algorithms in global optimization. Journal of global optimization, 19(1), 27-50..

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.evolu.sa.**SA**(*mode*, *bounds*, *fit*, *cooling='fast'*, *chain_size=10*, *Tmax=10000*, *Tmin=1*, *chi=0.1*, *move_func=None*, *reinforce_best='soft'*, *lmbda=1.5*, *alpha=1.5*, *threshold=10*, *ncores=1*, *seed=None*)

Parallel Simulated Annealing

Parameters

- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **cooling** – (str) cooling schedule, choose `fast`, `boltzmann`, `cauchy`,``equilibrium``. The `equilibrium` mode is only valid with `ncores > 1` (See **Notes** below)
- **chain_size** – (int) number of individuals to evaluate in the chain every generation (e.g. like `npop` for other algorithms)
- **Tmax** – (int) initial/maximum temperature
- **Tmin** – (int) final/minimum temperature
- **chi** – (float or list of floats) probability of perturbing every attribute of the input `x`, ONLY used if `move_func=None`. For `ncores > 1`, if a scalar is provided, constant value is used across all `ncores`. If a list of size `ncores` is provided, each core/chain uses different value of `chi` (See **Notes** below)
- **move_func** – (function) custom self-defined function that controls how to perturb the input space during annealing (See **Notes** below)
- **reinforce_best** – (str) an option to control the starting individual of the chain at every generation. Choose `None`, `hard`, `soft` (See **Notes** below).
- **lmbda** – (float) ONLY used if `cooling = equilibrium`, control the cooling rate, and the speed at which the algorithm converges.
- **alpha** – (float) ONLY used if `cooling = equilibrium`, control the initial temperature of the cooling schedule.
- **threshold** – (float) ONLY used if `cooling = equilibrium`. The threshold (in %) for the acceptance rate of solution under which the algorithm stops running.
- **ncores** – (int) number of parallel processors (`ncores > 1` is required for `cooling = equilibrium`)
- **seed** – (int) random seed for sampling

evolute(*ngen*, *x0=None*, *verbose=False*)

This function evolutes the SA algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) initial samples to start the evolution (`len(x0)` must be equal to `ncores`)
- **verbose** – (int) print statistics to screen

Returns (tuple) (best individual, best fitness, and dictionary containing major search results)

Example

```
from neorl import SA
import matplotlib.pyplot as plt
import random

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

#define a custom moving function
def my_move(x, **kwargs):
    #-----
    #this function selects two random indices in x and perturb their values
    #-----
    x_new=x.copy()
    indices=random.sample(range(0,len(x)), 2)
    for i in indices:
        x_new[i] = random.uniform(BOUNDS['x1'][1],BOUNDS['x1'][2])

    return x_new

#setup and evolute a serial SA
sa=SA(mode='min', bounds=BOUNDS, fit=FIT, chain_size=50, chi=0.2, Tmax=10000,
      move_func=my_move, reinforce_best='soft', cooling='boltzmann', ncores=1, seed=1)

#setup and evolute parallel SA with `equilibrium` cooling
#sa=SA(mode='min', bounds=BOUNDS, fit=FIT, chain_size=20, chi=0.2, Tmax=10000,␣
↪threshold = 1, lmbda=0.05,
#      move_func=my_move, reinforce_best='soft', cooling='equilibrium', ncores=8,␣
↪seed=1)

x_best, y_best, sa_hist=sa.evolute(ngen=100, verbose=1)

#plot different statistics
plt.figure()
plt.plot(sa_hist['accept'], '-o', label='Acceptance')
plt.plot(sa_hist['reject'], '-s', label='Rejection')
plt.plot(sa_hist['improve'], '-^', label='Improvement')
plt.xlabel('Generation')
plt.ylabel('Rate (%)')
plt.legend()
plt.show()
```

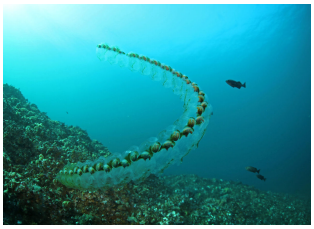
Notes

- Temperature is annealed between `Tmax` and `Tmin` if any of the following `cooling` schedules is used: `fast`, `cauchy`, and `boltzmann`.
- A special cooling schedule `equilibrium` is supported, which is associated with the following arguments:
  1. The initial temperature is determined by $T_0 = Max(\alpha * STD(\vec{E}_0), Tmin)$, where $STD(\vec{E}_0)$ is the standard deviation of the finesses of the initial chains. A tolerance is applied to the temperature if the standard deviation converged to zero.
  2. `alpha` is the parameter above that controls the initial temperature of SA.
  3. `lmbda` expresses the cooling rate or the speed of the temperature decay. Larger values lead to faster cooling.
  4. The `threshold` (in %) expresses the acceptance rate threshold under which the SA stops running. For example, if `threshold=10`, when the mean of acceptance rate of all chains falls below 10%, SA terminates. For zero threshold, SA will terminate when all chains no longer accept any new solution.
  5. The `equilibrium` cooling option is activated for parallel SA chains only, i.e. when `ncores > 1`.
- Custom `move_func` is allowed by following the input/output format in the example above. If `None` the default moving function is used, which is controlled by the `chi` parameter. Therefore, `chi` is used ONLY if `move_func=None`.
- `chi` controls the probability of perturbing an attribute of the individual. For example, for `d=4`, $\vec{x} = [x_1, x_2, x_3, x_4]$, for every x_i, a uniform random number $U[0, 1]$ is compared to `chi`, if `U[0,1] < chi`, the attribute is perturbed. Otherwise, it remains fixed.
- For every generation, a total of `chain_size` individuals are executed. Therefore, look for an optimal balance between `chain_size` and `ngen`.
- Option `reinforce_best` allows enforcing a solution from the previous generation to use as a chain startup in the next generation. Three options are available for this argument:
  1. `None`: No solution is enforced. The last chain state is preserved to the next generation.
  2. `hard`: the best individual in the chain is used as the initial starting point.
  3. `soft`: an energy-based sampling approach is utilized to draw an individual from the chain to start the next generation.
- Total number of cost evaluations for SA is `chain_size * (ngen + 1)`.
- If `ncores > 1`, parallel SA chains are initialized to accelerate the calculations.
- If `ncores > 1` and `move_func=None`, parallel SA chains can have different `chi` values, provided as a list/vector.

Salp Swarm Algorithm (SSA)

A module for the Salp Swarm Algorithm with parallel computing support.

Original paper: Mirjalili, S., Gandomi, A. H., Mirjalili, S. Z., Saremi, S., Faris, H., & Mirjalili, S. M. (2017). Salp Swarm Algorithm: A bio-inspired optimizer for engineering design problems. Advances in Engineering Software, 114, 163-191.



What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.evolu.ssa.**SSA**(*mode*, *bounds*, *fit*, *nsalps=5*, *int_transform='nearest_int'*, *ncores=1*, *seed=None*)

Salp Swarm Algorithm

Parameters

- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **nsalps** – (int): number of salps in the swarm
- **int_transform** – (str): method of handling int/discrete variables, choose from: `nearest_int`, `sigmoid`, `minmax`.
- **ncores** – (int) number of parallel processors (must be `<= nsalps`)
- **seed** – (int) random seed for sampling

evolute(*ngen*, *x0=None*, *c1=None*, *verbose=False*)

This function evolutes the SSA algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) initial position of the salps (must be of same size as `nsalps`)
- **c1** – (float/list): a scalar value or a list of values with size `ngen` for the coefficient that controls exploration/exploitation. If `None`, default annealing formula for `c1` is used (see **Notes** below for more info).
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and dictionary containing major search results)

Example

```
from neorl import SSA

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
                    F(x) = sum_{i=1}^d xi^2
                    d=1,2,3,...
                    Range: [-100,100]
                    Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

nsalps=20
#setup and evolute SSA
ssa=SSA(mode='min', bounds=BOUNDS, fit=FIT, nsalps=nsalps, ncores=1, seed=1)
x_best, y_best, ssa_hist=ssa.evolute(ngen=100, c1=None, verbose=1)
```

Notes

- SSA mimics the swarming behavior of salps when navigating and foraging in oceans. SSA cares mostly about the leading salp, where its position is optimized to achieve better food source (i.e. fitness).
- Salp leader is mostly controlled by the coefficient `c1`, which balances SSA exploration and exploitation. The default formula for $c_1 = 2e^{-(4g/ngen)^2}$, where g is the current generation (goes from 1 to `ngen`), and $ngen$ is the total number of generations to evolute (`ngen`). Therefore, `c1` is typically annealed from a large value at the beginning to increase exploration, to a very small value toward the end of evolution to prioritize exploitation.
- The user can also provide a scalar/fixed value for `c1` to overwrite the default annealing formula described above. Also, the user can provide a schedule for `c1` generated by another formula in a `list` form. The size of the list MUST equal to `ngen`. For example, for `ngen=5`, the user can provide `c1=[5, 0.5, 0.05, 0.005, 0.0005]`, where for every generation, the corresponding `c1` value is used.
- Therefore, if `c1=None`, the user should notice that `ngen` value used within the `.evolute` function has an impact on the `c1` value and hence on SSA overall performance.
- `ncores` argument evaluates the fitness of all salps in the swarm in parallel. Therefore, set `ncores <= nsalps` for most optimal resource allocation.
- Look for an optimal balance between `nsalps` and `ngen`, it is recommended to minimize the number of `nsalps` to allow for more updates and more generations.
- Total number of cost evaluations for SSA is `nsalps * (ngen + 1)`.

Whale Optimization Algorithm (WOA)

A module for the Whale Optimization Algorithm with parallel computing support.

Original paper: Mirjalili, S., Lewis, A. (2016). The whale optimization algorithm. Advances in engineering software, 95, 51-67.

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.evolu.woa.**WOA**(*mode*, *bounds*, *fit*, *nwhales=5*, *a0=2*, *b=1*, *int_transform='nearest_int'*, *ncores=1*, *seed=None*)

Whale Optimization Algorithm

Parameters

- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **nwhales** – (int): number of whales in the population

- **a0** – (float): initial value for coefficient `a`, which is annealed from `a0` to 0 (see **Notes** below for more info).
- **b** – (float): constant for defining the shape of the logarithmic spiral
- **int_transform** – (str): method of handling int/discrete variables, choose from: `nearest_int`, `sigmoid`, `minmax`.
- **ncores** – (int) number of parallel processors (must be `<= nwhales`)
- **seed** – (int) random seed for sampling

evolute(*ngen*, *x0=None*, *verbose=False*, ***kwargs*)

This function evolutes the WOA algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) initial position of the whales (must be of same size as `nwhales`)
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and dictionary containing major search results)

Example

```
from neorl import WOA
import matplotlib.pyplot as plt

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
                    F(x) = sum_{i=1}^d xi^2
                    d=1,2,3,...
                    Range: [-100,100]
                    Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

nwhales=20
#setup and evolute WOA
woa=WOA(mode='min', bounds=BOUNDS, fit=FIT, nwhales=nwhales, a0=1.5, b=1, ncores=1,␣
↪seed=1)
x_best, y_best, woa_hist=woa.evolute(ngen=100, verbose=1)

plt.figure()
plt.plot(woa_hist['a'], label='a')
plt.plot(woa_hist['A'], label='A')
plt.xlabel('generation')
plt.ylabel('coefficient')
plt.legend()
plt.show()
```

Notes

- WOA mimics the social behavior of humpback whales, which is inspired by the bubble-net hunting strategy.
- The whale leader is controlled by multiple coefficients, where `a` is considered the most important. The coefficient `a` balances WOA exploration and exploitation. The value of `a` is annealed "linearly" from `a0 > 0` to 0 over the course of `ngen`. Typical values for `a0` are 1, 1.5, 2, and 4.
- Therefore, the user should notice that `ngen` value used within the `.evolute` function has an impact on the `a` value and hence on WOA overall performance.
- `ncores` argument evaluates the fitness of all whales in the population in parallel. Therefore, set `ncores <= nwhales` for most optimal resource allocation.
- Look for an optimal balance between `nwhales` and `ngen`, it is recommended to minimize the number of `nwhales` to allow for more updates and more generations.
- Total number of cost evaluations for WOA is `nwhales * (ngen + 1)`.

Moth-flame Optimization (MFO)

A module for the Moth-flame Optimization with parallel computing support.

Original paper: Mirjalili, S. (2015). Moth-flame optimization algorithm: A novel nature-inspired heuristic paradigm. Knowledge-based systems, 89, 228-249.

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.evolu.mfo.**MFO**(*mode*, *bounds*, *fit*, *nmoths=50*, *b=1*, *int_transform='nearest_int'*, *ncores=1*, *seed=None*)

Moth-flame Optimization (MFO)

Parameters

- **mode** – (str) problem type, either "min" for minimization problem or "max" for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`

- **fit** – (function) the fitness function
- **nmoths** – (int) number of moths in the population
- **b** – (float) constant for defining the shape of the logarithmic spiral
- **int_transform** – (str): method of handling int/discrete variables, choose from: `nearest_int`, `sigmoid`, `minmax`.
- **ncores** – (int) number of parallel processors
- **seed** – (int) random seed for sampling

evolute (*ngen*, *x0=None*, *verbose=False*, ***kwargs*)

This function evolutes the MFO algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) the initial individuals of the population
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and a list of fitness history)

Example

```
from neorl import MFO

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
                    F(x) = sum_{i=1}^d xi^2
                    d=1,2,3,...
                    Range: [-100,100]
                    Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

#setup and evolute MFO
mfo=MFO(mode='min', bounds=BOUNDS, fit=FIT, nmoths=50, ncores=1, seed=1)
x_best, y_best, mfo_hist=mfo.evolute(ngen=200, verbose=1)
```

Notes

- MFO mimics the navigation behavior of moths in nature. Moths fly in night by maintaining a fixed angle with respect to the moon to travel in a straight line for long distances. However, the moths may get trapped in a deadly spiral path around artificial lights (i.e. called flames). This algorithm models this behavior to perform optimization by escaping the local/deadly regions during search.
- MFO creates two equal arrays of moth and flame positions. The moths are actual search agents that move around the search space, whereas flames are the best position of moths that obtains so far. Therefore, the flame can be seen as checkpoint of the best solutions found by the moths during the search.
- A logarithmic spiral is used as the main update mechanism of moths, which is controlled by the parameter `b`.
- MFO emphasizes exploitation through annealing an internal parameter `r` between -1 and -2. The value of `r` plays a factor in convergence as the moths prioritize their best solutions as we approach the value of `ngen`.
- `ncores` argument evaluates the fitness of all moths in parallel. Therefore, set `ncores <= nmoths` for most optimal resource allocation.
- Look for an optimal balance between `nmoths` and `ngen`, it is recommended to minimize the number of `nmoths` to allow for more updates and more generations.
- Total number of cost evaluations for MFO is `nmoths` * `ngen`.

JAYA Algorithm

A module for the JAYA Algorithm with parallel computing support.

Original paper: Rao, R. (2016). Jaya: A simple and new optimization algorithm for solving constrained and unconstrained optimization problems. International Journal of Industrial Engineering Computations, 7(1), 19-34.

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.evolu.jaya.**JAYA**(*mode*, *bounds*, *fit*, *npop=50*, *int_transform='nearest_int'*, *ncores=1*, *seed=None*)

JAYA algorithm

Parameters

- **mode** – (str) problem type, either "min" for minimization problem or "max" for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **npop** – (int) number of individuals in the population

- **int_transform** – (str): method of handling int/discrete variables, choose from: `nearest_int`, `sigmoid`, `minmax`.
- **ncores** – (int) number of parallel processors
- **seed** – (int) random seed for sampling

evolute (*ngen*, *x0=None*, *verbose=False*)

This function evolutes the MFO algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) the initial individuals of the population
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and a list of fitness history)

Example

```
from neorl import JAYA

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
                    F(x) = sum_{i=1}^d xi^2
                    d=1,2,3,...
                    Range: [-100,100]
                    Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

#setup and evolute JAYA
jaya=JAYA(mode='min', bounds=BOUNDS, fit=FIT, npop=60, ncores=1, seed=1)
x_best, y_best, jaya_hist=jaya.evolute(ngen=200, verbose=1)
```

Notes

- JAYA concept is very simple that any optimization algorithm should look for solutions that move towards the best solution and should avoid the worst solution. Therefore, JAYA keeps tracking of both the best and worst solutions and varies the population accordingly.
- JAYA is free of special hyperparameters, therefore, the user only needs to specify the size of the population `npop`.
- `ncores` argument evaluates the fitness of all individuals in the population in parallel. Therefore, set `ncores <= npop` for most optimal resource allocation.
- Look for an optimal balance between `npop` and `ngen`, it is recommended to minimize the number of `npop` to allow for more updates and more generations.

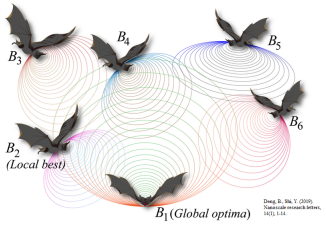
- Total number of cost evaluations for JAYA is `npop * (ngen + 1)`.

Bat Algorithm (BAT)

A module for the bat optimization algorithm with differential operator, Levy flights trajectory, and parallel computing support.

Original papers:

- Xie, J., Zhou, Y., Chen, H. (2013). A novel bat algorithm based on differential operator and Lévy flights trajectory. Computational intelligence and neuroscience, 2013.
- Yang, X. S. (2010). A new metaheuristic bat-inspired algorithm. In Nature inspired cooperative strategies for optimization (NICSO 2010) (pp. 65-74). Springer, Berlin, Heidelberg.



What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.evolu.bat.**BAT**(*mode*, *bounds*, *fit*, *nbats=50*, *fmin=0*, *fmax=1*, *A=0.5*, *r0=0.5*, *alpha=1.0*, *gamma=0.9*, *levy='False'*, *int_transform='nearest_int'*, *ncores=1*, *seed=None*)

BAT Algorithm

Parameters

- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['float', 0.1, 0.8], 'x2': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **nbats** – (int): number of bats in the population
- **fmin** – (float): minimum value of the pulse frequency
- **fmax** – (float): maximum value of the pulse frequency
- **A** – (float) initial value of the loudness rate
- **r0** – (float) asymptotic value of the pulse rate

- **alpha** – (float) decay factor of loudness `A`, i.e. A approaches 0 by the end of evolution if `alpha < 1`
- **gamma** – (float) exponential factor of the pulse rate `r`, i.e. `r` increases abruptly at the beginning and then converges to `r0` by the end of evolution
- **levy** – (bool): a flag to activate Levy flight steps of the bat to increase bat diversity
- **int_transform** – (str): method of handling int/discrete variables, choose from: `nearest_int`, `sigmoid`, `minmax`.
- **ncores** – (int) number of parallel processors (must be `<= nbats`)
- **seed** – (int) random seed for sampling

evolute (*ngen*, *x0=None*, *verbose=False*)

This function evolutes the BAT algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) initial position of the bats (must be of same size as `nbats`)
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and dictionary containing major search results)

Example

```
from neorl import BAT

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
                    F(x) = sum_{i=1}^d xi^2
                    d=1,2,3,...
                    Range: [-100,100]
                    Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

#setup and evolute BAT
bat=BAT(mode='min', bounds=BOUNDS, fit=FIT, nbats=40,
        fmin=0, fmax=1, A=1.0, r0=0.7,
        ncores=1, seed=1)
x_best, y_best, bat_hist=bat.evolute(ngen=100, verbose=1)
```

Notes

- BAT mimics the echolocation behavior of bats in nature. The bats emit a very loud and short sound pulse; the echo that reflects back from the surrounding objects is received by their big ears. This feedback information of echo is analyzed by the bats, which reveals the direction of the flight pathway. The echo also helps the bats to distinguish different insects and obstacles to hunt prey, where the search for the prey here is analogous to the search for global optima.
- The bats start with loudness value of `A`, and decay it by a factor of `alpha`. If the user chooses `alpha=1`, fixed value of `A` is used.
- Bats fly randomly to search for the prey with frequency varying between `fmin` and `fmax`.
- The bats emit pulses with emission rate represented by the asymptotic value (`r0`). The value of emission rate is updated in generation `i` according to $r_i = r_0(1 - exp(-\gamma i))$, where `gamma` is the exponential factor of the pulse rate. `r` typically decreases abruptly at the beginning and then converges back to `r0` by the end of the evolution.
- We provide a flexible BAT implemetation that can handle continuous (`float`), discrete (`int`), and categorical (`grid`) spaces and their mix. The user can control the type of discrete transformation via the argument `int_transform`.
- `ncores` argument evaluates the fitness of all bats in parallel. Therefore, set `ncores <= nbats` for most optimal resource allocation.
- Look for an optimal balance between `nbats` and `ngen`, it is recommended to minimize the number of `nbats` to allow for more updates and more generations.
- Total number of cost evaluations for BAT is `3*nbats * (ngen + 1)`.

Harris Hawks Optimization (HHO)

A module for the Harris Hawks Optimization with parallel computing support and mixed discrete/continuous optimization ability.

Original paper: Heidari, A. A., Mirjalili, S., Faris, H., Aljarah, I., Mafarja, M., & Chen, H. (2019). Harris hawks optimization: Algorithm and applications. Future generation computer systems, 97, 849-872.

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.evolu.hho.**HHO**(*mode*, *bounds*, *fit*, *nhawks*, *int_transform='nearest_int'*, *ncores=1*, *seed=None*)

Harris Hawks Optimizer

Parameters

- **mode** – (str) problem type, either "min" for minimization problem or "max" for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **nhawks** – (int): number of the hawks in the group
- **int_transform** – (str): method of handling int/discrete variables, choose from: `nearest_int`, `sigmoid`, `minmax`.
- **ncores** – (int) number of parallel processors (must be `<= nhawks`)
- **seed** – (int) random seed for sampling

evolute(*ngen*, *x0=None*, *verbose=False*, ***kwargs*)

This function evolutes the HHO algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) initial position of the hawks (must be of same size as `nhawks`)
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and dictionary containing major search results)

Example

```
from neorl import HHO

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
                    F(x) = sum_{i=1}^d xi^2
                    d=1,2,3,...
                    Range: [-100,100]
                    Minima: 0
```

(continues on next page)

(continued from previous page)

```
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

#setup and evolute HHO
hho=HHO(mode='min', bounds=BOUNDS, fit=FIT, nhawks=20, ncores=1, seed=1)
x_best, y_best, hho_hist=hho.evolute(ngen=200, verbose=1)
```

Notes

- HHO is inspired by the cooperative behavior and chasing style of Harris' hawks in nature, which is called surprise pounce. Several hawks cooperatively pounce a prey from different directions in an attempt to surprise it. The prey here can be a rabbit, which is a representative of the global optima.
- HHO employs different exploration and exploitation strategies in form of soft and hard besieges as well as rapid dives before attacking the prey. These strategies are parameter-free, as only `nhawks` needs to be specified by the user.
- We provide a flexible HHO implemetation that can handle continuous (`float`), discrete (`int`), and categorical (`grid`) and their mix. The user can control the type of discrete transformation via the argument `int_transform`.
- `ncores` argument evaluates the fitness of all hawks in the swarm in parallel after the position update. Therefore, set `ncores <= nhawks` for most optimal resource allocation.
- Look for an optimal balance between `nhawks` and `ngen`, it is recommended to minimize the number of `nhawks` to allow for more updates and more generations.
- Total number of cost evaluations for HHO is `2*nhawks * ngen` (this is an upper bound estimate as there is randomness in whether some of the hawks are evaluated or not).

Ant Colony Optimization (ACO)

A module for the Ant Colony Optimization with parallel computing support and continuous optimization ability.

Original paper: Socha, K., & Dorigo, M. (2008). Ant colony optimization for continuous domains. European journal of operational research, 185(3), 1155-1173.

What can you use?

- Multi processing: ✓
- Discrete spaces:
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces:

Parameters

class neorl.evolu.aco.**ACO**(*mode*, *fit*, *bounds*, *nants=40*, *narchive=10*, *Q=0.5*, *Z=1.0*, *ncores=1*, *seed=None*)

Ant Colony Optimization

Parameters

- **mode** – (str) problem type, either "min" for minimization problem or "max" for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **nants** – (int) number of total ants
- **narchive** – (int) size of archive of best ants (recommended `narchive < nants`)
- **Q** – (float) diversification/intensification factor (see **Notes** below)
- **Z** – (float) deviation-distance ratio or pheromone evaporation rate, high Z leads to slow convergence (see **Notes** below).
- **ncores** – (int) number of parallel processors
- **seed** – (int) random seed for sampling

evolute(*ngen*, *x0=None*, *verbose=False*)

This function evolutes the ACO algorithm for number of generations

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) the initial individuals of the population
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and list of fitness history)

Example

```
from neorl import ACO
import random

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
                    F(x) = sum_{i=1}^d xi^2
                    d=1,2,3,...
                    Range: [-100,100]
                    Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

nants=40
x0=[[random.uniform(-100,100)]*nx for item in range(nants)]
acor = ACO(mode='min', fit=FIT, bounds=BOUNDS, nants=nants, narchive=10,
           Q=0.5, Z=1, ncores=1, seed=1)
x_best, y_best, acor_hist=acor.evolute(ngen=100, x0=x0, verbose=1)
```

Notes

- ACO is inspired from the cooperative behavior and food search of ants. Several ants cooperatively search for food in different directions in an attempt to find the global optima (richest food source).
- For ACO, the archive of best ants is `narchive`, where it must be less than the population size `nants`.
- The factor `Q` controls the rate of exploration/exploitation of ACO. When `Q` is small, the best-ranked solutions are strongly preferred next (more exploitation), and when `Q` is large, the probability of all solutions become more uniform (more exploration).
- The factor `Z` is the pheromone evaporation rate, which controls search behavior. As `Z` increases, the search becomes less biased towards the points of the search space that have been already explored, which are kept in the archive. In general, the higher the value of `Z`, the lower the convergence speed of ACO.
- `ncores` argument evaluates the fitness of all ants in parallel after the position update. Therefore, set `ncores <= nants` for most optimal resource allocation.
- Look for an optimal balance between `nants` and `ngen`, it is recommended to minimize the number of `nants` to allow for more updates and more generations.
- Total number of cost evaluations for ACO is `nants * ngen`.

Cuckoo Search (CS)

A module for the Cuckoo Search Algorithm with parallel computing support.

Original paper: Yang, X. S., & Deb, S. (2009, December). Cuckoo search via Lévy flights. In 2009 World congress on nature & biologically inspired computing (NaBIC) (pp. 210-214). IEEE.

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.evolu.cs.**CS**(*mode*, *bounds*, *fit*, *ncuckoos=15*, *pa=0.25*, *int_transform='nearest_int'*, *ncores=1*, *seed=None*)

Cuckoo Search Algorithm

Parameters

- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **ncuckoos** – (int) number of cuckoos or nests in the population: one cuckoos per nest. Default value is 15.
- **pa** – (float) a scalar value for the coefficient that controls exploration/exploitation, i.e. fraction of the cuckoos/nests that will be replaced by the new cuckoos/nests.
- **int_transform** – (str) method of handling int/discrete variables, choose from: `nearest_int`, `sigmoid`, `minmax`.
- **ncores** – (int) number of parallel processors (must be `<= ncuckoos`)
- **seed** – (int) random seed for sampling

evolute(*ngen*, *x0=None*, *verbose=False*)

This function evolutes the CS algorithm for number of generations

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) initial position of the cuckoos (must be of same size as `ncuckoos`)

- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and dictionary containing major search results)

Example

```
from neorl import CS

#Define the fitness function
def Sphere(individual):
    """Sphere test objective function.
                    F(x) = sum_{i=1}^d xi^2
                    d=1,2,3,...
                    Range: [-100,100]
                    Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y
#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

#setup and evolute CS
cs = CS(mode = 'min', bounds = BOUNDS, fit = Sphere, ncuckoos = 40, pa = 0.25, seed=1)
x_best, y_best, cs_hist=cs.evolute(ngen = 200, verbose=True)
```

Notes

- CS algorithm is based on the obligate brood parasitic behavior of some cuckoo species in combination with the Levy flight behavior of some birds and fruit flies. CS assumes each cuckoo lays one egg at a time, and dump its egg in randomly chosen nest. Also, the best nests with high quality of eggs will carry over to the next generations.
- `pa` controls exploration/exploitation of the algorithm, it is the fraction of the cuckoos/nests that will be replaced by new cuckoos/nests. In this case, the host bird can either throw the egg away or abandon the nest, and build a completely new nest.
- `ncores` argument evaluates the fitness of all cuckoos in the population in parallel. Therefore, set `ncores <= ncuckoos` for most optimal resource allocation.
- Look for an optimal balance between `ncuckoos` and `ngen`, it is recommended to minimize the number of `ncuckoos` to allow for more updates and more generations.
- Total number of cost evaluations for CS is `2*ncuckoos * (ngen + 1)`.

Tabu Search (TS)

A module for the tabu search with long-term memory for discrete/combinatorial optimization.

Original papers:

- Glover, F. (1989). Tabu search—part I. ORSA Journal on computing, 1(3), 190-206.
- Glover, F. (1990). Tabu search—part II. ORSA Journal on computing, 2(1), 4-32.

What can you use?

- Multi processing:
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.evolu.ts.**TS**(*mode*, *bounds*, *fit*, *tabu_tenure=6*, *penalization_weight=0.8*, *swap_mode='perturb'*, *ncores=1*, *seed=None*)

Tabu Search Algorithm

Parameters

- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['int', -10, 10], 'x3': ['int', -100, 100]}`
- **fit** – (function) the fitness function
- **tabu_tenure** – (int): Timestep under which a certain list of solution cannot be accessed (for diversification). Default value is 6.
- **penalization_weight** – (float): a scalar value for the coefficient that controls exploration/exploitation, i.e. importance of the frequency of a certain action performed in the search. The higher the value, the least likely is an action to be performed again after multiple attempts.
- **swap_mode** – (str): either "swap" for swapping two elements of the input or "perturb" to perturb each input within certain bounds (see **Notes** below)
- **ncores** – (int) number of parallel processors (only `ncores=1` is supported now)
- **seed** – (int) random seed for sampling

evolute(*ngen*, *x0=None*, *verbose=False*)

This function evolutes the TS algorithm for number of generations

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list) initial position of the tabu (vector size must be of same size as `len(bounds)`)
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and dictionary containing major search results)

Example

```
from neorl import TS

#Define the fitness function
def Sphere(individual):
    """Sphere test objective function.
                    F(x) = sum_{i=1}^d xi^2
                    d=1,2,3,...
                    Range: [-100,100]
                    Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y


#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(nx):
    BOUNDS['x'+str(i)]=['int', -100, 100]

#setup and evolute TS
x0=[-25,50,100,-75,-100]  #initial guess
ts=TS(mode = "min", bounds = BOUNDS, fit = Sphere, tabu_tenure=60,
      penalization_weight = 0.8, swap_mode = "perturb", ncores=1, seed=1)
x_best, y_best, ts_hist=ts.evolute(ngen = 700, x0=x0, verbose=1)
```

Notes

- Tabu search (TS) is a metaheuristic algorithm that can be used for solving combinatorial optimization problems (problems where an optimal ordering and selection of options is desired). Also, we adapted TS to solve bounded discrete problems for which the candidate solution needs to be perturbed and bounded.
- For `swap_mode`, choose `perturb` for problems that have lower/upper bounds at which the individual is perturbed between them to find optimal solution (e.g. Sphere function). Choose `swap` for combinatorial problems where the elements of the individual are swapped (not perturbed) to find the optimal solution (e.g. Travel Salesman, Job Scheduling).
- `tabu_tenure` refers to the number of timesteps to perform to enable any particular update to happen again (i.e. swapping of two entries x_i, x_j or perturbation of an entry x_i). For example, if tabu_tenure=6 and x_i of a candidate solution is perturbed, within 6 additional timesteps, x_i can be perturbed if and only if the resulting perturbation yields to a solution better than the current best one.
- `penalization_weight` represents the importance/frequency of a certain action performed in the search. Large values of `penalization_weight` reduces the frequency of using the same action again in the search.

2.2.3 Hybrid and Neuroevolution Algorithms

Feedforward Neuroevolution of Augmenting Topologies (FNEAT)

Neuroevolution of Augmenting Topologies (NEAT) uses evolutionary genetic algorithms to evolve neural architectures, where the best optimized neural network is selected according to certain criteria. For NEORL, NEAT tries to build a neural network that minimizes or maximizes an objective function by following {action, state, reward} terminology of reinforcement learning. In FNEAT, genetic algorithms evolve Feedforward neural networks for optimization purposes in a reinforcement learning context.

Original paper: Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. Evolutionary computation, 10(2), 99-127.

What can you use?

- Multi processing: ✓
- Discrete spaces:
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces:

Parameters

class neorl.hybrid.fneat.**FNEAT**(*mode*, *fit*, *bounds*, *config*, *ncores=1*, *seed=None*)
Feedforward NeuroEvolution of Augmenting Topologies (FNEAT)

Parameters

- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization (RL is default to `max`)
- **fit** – (function) the fitness function
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **config** – (dict) dictionary of RNEAT hyperparameters, see **Notes** below for available hyperparameters to change
- **ncores** – (int) number of parallel processors
- **seed** – (int) random seed for sampling

evolute(*ngen*, *x0=None*, *save_best_net=False*, *checkpoint_itv=None*, *startpoint=None*, *verbose=False*)
This function evolutes the FNEAT algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list) initial position of the NEAT (must have same size as the `x` variable)
- **save_best_net** – (bool) save the winner neural network to a pickle file
- **checkpoint_itv** – (int) generation frequency to save checkpoints for restarting purposes (e.g. 1: save every generation, 10: save every 10 generations)

- **startpoint** – (str) name/path to the checkpoint file to use to start the search (the checkpoint file can be saved by invoking the argument `checkpoint_itv`)
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and dictionary containing major search results)

Example

Train a FNEAT agent to optimize the 5-D sphere function

```
from neorl import FNEAT
import numpy as np

def Sphere(individual):
    """Sphere test objective function.
            F(x) = sum_{i=1}^d xi^2
            d=1,2,3,...
            Range: [-100,100]
            Minima: 0
    """
    return sum(x**2 for x in individual)

nx=5
lb=-100
ub=100
bounds={}
for i in range(1,nx+1):
        bounds['x'+str(i)]=['float', -100, 100]

# modify your own NEAT config
config = {
    'pop_size': 50,
    'num_hidden': 1,
    'activation_mutate_rate': 0.1,
    'survival_threshold': 0.3,
    }

fneat=FNEAT(fit=Sphere, bounds=bounds, mode='min', config= config, ncores=1, seed=1)
#some random guess (just one individual)
x0 = np.random.uniform(lb,ub,nx)
x_best, y_best, fneat_hist=fneat.evolute(ngen=5, x0=x0, verbose=True,
                                         checkpoint_itv=None, startpoint=None)
```

Notes

- **The following major hyperparameters can be changed when you define the `config` dictionary:**

Hyperparameter	Description
– pop_size – num_hidden – elitism – survival_threshold – min_species_size – activation_mutate_rate – aggregation_mutate_rate – weight_mutate_rate – bias_mutate_rate	– The number of individuals in each generation (30) – The number of hidden nodes to add to each genome in the initial population (1) – The number of individuals to survive from one generation to the next (1) – The fraction for each species allowed to reproduce each generation(0.3) – The minimum number of genomes per species after reproduction (2) – The probability that mutation will replace the node's activation function (0.05) – The probability that mutation will replace the node's aggregation function (0.05) – The probability that mutation will change the connection weight by adding a random value (0.5) – The probability that mutation will change the bias of a node by adding a random value (0.7)

Acknowledgment

Thanks to our fellows in NEAT-Python, as we have used their NEAT implementation to leverage our optimization classes.

https://github.com/CodeReclaimers/neat-python

Recurrent Neuroevolution of Augmenting Topologies (RNEAT)

Neuroevolution of Augmenting Topologies (NEAT) uses evolutionary genetic algorithms to evolve neural architectures, where the best optimized neural network is selected according to certain criteria. For NEORL, NEAT tries to build a neural network that minimizes or maximizes an objective function by following {action, state, reward} terminology of reinforcement learning. In RNEAT, genetic algorithms evolve Recurrent neural networks for optimization purposes in a reinforcement learning context.

Original paper: Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. Evolutionary computation, 10(2), 99-127.

What can you use?

- Multi processing: ✓
- Discrete spaces:
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces:

Parameters

class neorl.hybrid.rneat.**RNEAT**(*mode*, *fit*, *bounds*, *config*, *ncores=1*, *seed=None*)

Recurrent NeuroEvolution of Augmenting Topologies (RNEAT)

Parameters

- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization (RL is default to `max`)
- **fit** – (function) the fitness function
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **config** – (dict) dictionary of RNEAT hyperparameters, see **Notes** below for available hyperparameters to change
- **ncores** – (int) number of parallel processors
- **seed** – (int) random seed for sampling

evolute(*ngen*, *x0=None*, *save_best_net=False*, *checkpoint_itv=None*, *startpoint=None*, *verbose=False*)

This function evolutes the RNEAT algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list) initial position of the NEAT (must have same size as the `x` variable)
- **save_best_net** – (bool) save the winner neural network to a pickle file
- **checkpoint_itv** – (int) generation frequency to save checkpoints for restarting purposes (e.g. 1: save every generation, 10: save every 10 generations)
- **startpoint** – (str) name/path to the checkpoint file to use to start the search (the checkpoint file can be saved by invoking the argument `checkpoint_itv`)
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and dictionary containing major search results)

Example

Train a RNEAT agent to optimize the 5-D sphere function

```
from neorl import RNEAT
import numpy as np

def Sphere(individual):
    """Sphere test objective function.
            F(x) = sum_{i=1}^d xi^2
            d=1,2,3,...
            Range: [-100,100]
            Minima: 0
    """
    return sum(x**2 for x in individual)
```

(continues on next page)

(continued from previous page)

```
nx=5
lb=-100
ub=100
bounds={}
for i in range(1,nx+1):
        bounds['x'+str(i)]=['float', -100, 100]

# modify your own NEAT config
config = {
    'pop_size': 50,
    'num_hidden': 1,
    'activation_mutate_rate': 0.1,
    'survival_threshold': 0.3,
    }

# model config
rneat=RNEAT(fit=Sphere, bounds=bounds, mode='min', config= config, ncores=1, seed=1)
#A random initial guess (provide one individual)
x0 = np.random.uniform(lb,ub,nx)
x_best, y_best, rneat_hist=rneat.evolute(ngen=200, x0=x0,
                                         verbose=True, checkpoint_itv=None,
                                         startpoint=None)
```

Notes

- **The following major hyperparameters can be changed when you define the `config` dictionary:**

Hyperparameter	Description
– pop_size – num_hidden – elitism – survival_threshold – min_species_size – activation_mutate_rate – aggregation_mutate_rate – weight_mutate_rate – bias_mutate_rate	– The number of individuals in each generation (30) – The number of hidden nodes to add to each genome in the initial population (1) – The number of individuals to survive from one generation to the next (1) – The fraction for each species allowed to reproduce each generation(0.3) – The minimum number of genomes per species after reproduction (2) – The probability that mutation will replace the node's activation function (0.05) – The probability that mutation will replace the node's aggregation function (0.05) – The probability that mutation will change the connection weight by adding a random value (0.5) – The probability that mutation will change the bias of a node by adding a random value (0.7)

Acknowledgment

Thanks to our fellows in NEAT-Python, as we have used their NEAT implementation to leverage our optimization classes.

https://github.com/CodeReclaimers/neat-python

Prioritized replay Evolutionary and Swarm Algorithm (PESA)

A module for the parallel hybrid PESA algorithm with prioritized experience replay from reinforcement learning. This is the classical PESA that hybridizes PSO, ES, and SA modules within NEORL.

Original paper: Radaideh, M. I., & Shirvan, K. (2022). PESA: Prioritized experience replay for parallel hybrid evolutionary and swarm algorithms-Application to nuclear fuel. Nuclear Engineering and Technology.

https://doi.org/10.1016/j.net.2022.05.001

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.hybrid.pesa.**PESA**(*mode*, *bounds*, *fit*, *npop*, *mu=None*, *memory_size=None*, *alpha_init=0.1*, *alpha_end=1*, *alpha_backdoor=0.1*, *Tmax=10000*, *chi=0.1*, *cxpb=0.7*, *mutpb=0.1*, *c1=2.05*, *c2=2.05*, *speed_mech='constric'*, *ncores=1*, *seed=None*)

PESA Major Parameters

Parameters

- **mode** – (str) problem type, either "min" for minimization problem or "max" for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **npop** – (int) total number of individuals in each group. So for ES, PSO, and SA, full population is `npop*3`.
- **mu** – (int) number of individuals to survive to the next generation. Also, `mu` equals to the number of individuals to sample from the memory. If None, `mu=int(npop/2)`. So 1/2 of PESA population comes from previous generation, and 1/2 comes from the replay memory (See **Notes** below for more info)
- **memory_size** – (int) max size of the replay memory (if None, `memory_size` is built to accommodate all samples during search)
- **alpha_init** – (float) initial value of the prioritized replay coefficient (See **Notes** below)

- **alpha_end** – (float) final value of the prioritized replay coefficient (See **Notes** below)
- **alpha_backdoor** – (float) backdoor greedy replay rate/probability to sample from the memory for SA instead of random-walk (See **Notes** below)

PESA Auxiliary Parameters (for the internal algorithms)

Parameters

- **cxpb** – (float) for **ES**, population crossover probability between [0,1]
- **mutpb** – (float) for **ES**, population mutation probability between [0,1]
- **c1** – (float) for **PSO**, cognitive speed constant
- **c2** – (float) for **PSO**, social speed constant
- **speed_mech** – (str) for **PSO**, type of speed mechanism for to update particle velocity, choose between `constric`, `timew`, `globw`.
- **Tmax** – (float) for **SA**, initial/max temperature to start the annealing process
- **chi** – (float) for **SA**, probability to perturb an attribute during SA annealing (occurs when `rand(0,1) < chi`).

PESA Misc. Parameters

Parameters

- **ncores** – (int) number of parallel processors
- **seed** – (int) random seed for sampling

evolute (*ngen*, *x0=None*, *warmup=100*, *verbose=0*)
This function evolutes the PESA algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) initial samples to start the replay memory (`len(x0)` must be equal or more than `npop`)
- **warmup** – (int) number of random warmup samples to initialize the replay memory and must be equal or more than `npop` (only used if `x0=None`)
- **verbose** – (int) print statistics to screen, 0: no print, 1: PESA print, 2: detailed print

Returns (tuple) (best individual, best fitness, and a list of fitness history)

Example

```
from neorl import PESA

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
            F(x) = sum_{i=1}^d xi^2
            d=1,2,3,...
            Range: [-100,100]
            Minima: 0
    """
    y=sum(x**2 for x in individual)
```

(continues on next page)

(continued from previous page)

```
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

npop=60
pesa=PESA(mode='min', bounds=BOUNDS, fit=FIT, npop=npop, mu=40, alpha_init=0.2,
          alpha_end=1.0, alpha_backdoor=0.1, ncores=1)
x0=[[50,50,50,50,50] for i in range(npop)]  #initial guess
x_best, y_best, pesa_hist=pesa.evolute(ngen=50, x0=x0, verbose=1)
```

Notes

- PESA is symmetric, meaning population size is equal between PSO, ES, and SA, which is helpful to ensure that all algorithms can update the memory with similar computing time. For example, if the user sets `npop=60`, then in every generation, the swarm of PSO has 60 particles, ES population has 60 individuals, and SA chain has length of 60.
- `mu` defines the number of individuals from `npop` to survive to the next generation, and also the number of samples to replay from the memory. This is applicable to PSO and ES alone as SA does not have the concept of population. For example, by setting `mu=40` and `npop=60`, then after every generation, the top 40 individuals in PSO and ES survive. Then the replay memory feeds 40 individuals to ES, which form the new pool of 80 individuals that go through offspring processes that produce again `npop=60`. For PSO, the replay memory provides `60-40=20` particles to form a new swarm of `npop=60`.
- For complex problems and limited memory, we recommend to set `memory_size ~ 5000`. When the memory gets full, old samples are overwritten by new ones. Allowing a large memory for complex problems may slow down PESA as handling large memories is more computationally exhaustive. If `memory_size = None`, the memory size will be set to maximum value of `ngen*npop*3`.
- For parallel computing of PESA, pick `ncores` divisible by 3 (e.g. 6, 18, 30) to ensure equal computing power across the internal algorithms.
- If `ncores=1`, serial calculation of PESA is executed.
- Example on how to assign computing resources for PESA. Lets assume a generation of `npop=60` individuals and `ncores=30`. Then, `icores=int(ncores/3)` or `icores=10` cores are assigned to each algorithm of PSO, ES, and SA. In this case, the ES population has size `npop=60` and it is evaluated in parallel with 10 cores. PSO swarm also has `npop=60` particles evaluated with 10 cores. SA releases 10 parallel chains, each chain evaluates 6 individuals.
- Check the sections of *PSO*, *ES*, and *SA* for notes on the internal algorithms and the auxiliary parameters of PESA.
- Start the prioritized replay with a small fraction for `alpha_init < 0.1` to increase randomness earlier to improve PESA exploration. Choose a high fraction for `alpha_end > 0.9` to increase exploitation by the end of evolution.
- The rate of `alpha_backdoor` replaces the regular random-walk sample of SA with the best individual in the replay memory to keep SA chain up-to-date. For example, `alpha_backdoor=0.1` implies that out of 10 individuals in the SA chain, 1 comes from the memory and the other 9 come from classical random-walk. Keep the value of `alpha_backdoor` small enough, e.g. `alpha_backdoor < 0.2`, to avoid SA divergence.

- Look for an optimal balance between `npop` and `ngen`, it is recommended to minimize population size to allow for more generations.
- Total number of cost evaluations for PESA is `ngen*npop*3 + warmup`.

Modern PESA (PESA2)

A module for the parallel hybrid PESA2 algorithm with prioritized experience replay from reinforcement learning. Modern PESA2 combines GWO, WOA, and DE modules in NEORL.

Original paper: Radaideh, M. I., & Shirvan, K. (2022). PESA: Prioritized experience replay for parallel hybrid evolutionary and swarm algorithms-Application to nuclear fuel. Nuclear Engineering and Technology.

https://doi.org/10.1016/j.net.2022.05.001

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.hybrid.pesa2.**PESA2**(*mode*, *bounds*, *fit*, *R_frac=0.5*, *memory_size=None*, *alpha_init=0.1*, *alpha_end=1*, *nwolves=5*, *npop=50*, *CR=0.7*, *F=0.5*, *nwhales=10*, *int_transform='nearest_int'*, *ncores=1*, *seed=None*)

Prioritized replay for Evolutionary Swarm Algorithms: PESA 2 (Modern Version) A hybrid algorithm of GWO, DE, and WOA

PESA2 Major Parameters

Parameters

- **mode** – (str) problem type, either "min" for minimization problem or "max" for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **R_frac** – (int) fraction of `npop`, `nwolves`, `nwhales` to survive to the next generation. Also, `R_frac` equals to the number of individuals to replay from the memory
- **memory_size** – (int) max size of the replay memory (if None, `memory_size` is built to accommodate all samples during search)
- **alpha_init** – (float) initial value of the prioritized replay coefficient (See **Notes** below)
- **alpha_end** – (float) final value of the prioritized replay coefficient (See **Notes** below)

PESA2 Auxiliary Parameters (for the internal algorithms)

Parameters

- **npop** – (int) for **DE**, total number of individuals in DE population
- **CR** – (float) for **DE**, crossover probability between [0,1]
- **F** – (float) for **DE**, differential/mutation weight between [0,2]
- **nwolves** – (float) for **GWO**, number of wolves for GWO
- **nwhales** – (float) for **WOA**, number of whales in the population of WOA

PESA2 Misc. Parameters

Parameters

- **int_transform** – (str): method of handling int/discrete variables, choose from: `nearest_int`, `sigmoid`, `minmax`.
- **ncores** – (int) number of parallel processors
- **seed** – (int) random seed for sampling

evolute (*ngen*, *x0=None*, *replay_every=1*, *warmup=100*, *verbose=0*)

This function evolutes the PESA2 algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) initial samples to start the replay memory (`len(x0)` must be equal or more than `npop`)
- **replay_every** – (int) perform memory replay every number of generations, default: replay after every generation
- **warmup** – (int) number of random warmup samples to initialize the replay memory and must be equal or more than `npop` (only used if `x0=None`)
- **verbose** – (int) print statistics to screen, 0: no print, 1: PESA print, 2: detailed print

Returns (tuple) (best individual, best fitness, and a list of fitness history)

Example

```
from neorl import PESA2

#Define the fitness function
def FIT(individual):
        """Sphere test objective function.
                F(x) = sum_{i=1}^d xi^2
                d=1,2,3,...
                Range: [-100,100]
                Minima: 0
        """
        y=sum(x**2 for x in individual)
        return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
        BOUNDS['x'+str(i)]=['float', -100, 100]
```

(continues on next page)

(continued from previous page)

```
pesa2=PESA2(mode='min', bounds=BOUNDS, fit=FIT, npop=50, nwolves=5, nwhales=5,␣
↪ncores=1)
x_best, y_best, pesa2_hist=pesa2.evolute(ngen=50, replay_every=2, verbose=2)
```

Notes

- PESA2 is symmetric, meaning population size is equal between DE, WOA, and GWO, which is helpful to ensure that all algorithms can update the memory with similar computing time. Since GWO/WOA have typically smaller population than DE, i.e. `nwolves < npop`, `nwhales < npop`, PESA2 adjusts number of internal generations for GWO/WOA to ensure similar fitness calculations per individual algorithm.
- For example, if the user sets `npop=60` for DE, `nwovles=6` for GWO, and `nwhales=10` for WOA, then GWO and WOA are executed internally for 10 and 6 generations, respectively, to have a total of 60 evaluations per individual algorithm.
- `R_frac` defines the fraction of individuals from `npop`, `nwovles`, and `nwhales` to survive to the next generation, and also the number of samples to replay from the memory. For example, if the user sets `R_frac=0.5`, `npop=60` for DE, `nwovles=6` for GWO, and `nwhales=10` for WOA, then after every generation, the top 30 individuals in DE, the best 3 wolves, and the best 5 whales survive to the next generation. Then the replay memory feeds 30 individuals to DE, three new wolves to GWO, and 5 new whales to WOA.
- For complex problems and limited memory, we recommend to set `memory_size ~ 5000`. When the memory gets full, old samples are overwritten by new ones. Allowing a large memory for complex problems may slow down PESA2 as handling large memories is more computationally exhaustive. If `memory_size = None`, the memory size will be set to maximum value of `ngen*npop*3`.
- For parallel computing of PESA2, pick `ncores` divisible by 3 (e.g. 6, 18, 30) to ensure equal computing power across the internal algorithms.
- If `ncores=1`, serial calculation of PESA2 is executed.
- Check the sections of *GWO*, *WOA*, and *DE* for notes on the internal algorithms and the auxiliary parameters of PESA2.
- Start the prioritized replay with a small fraction for `alpha_init < 0.1` to increase randomness earlier to improve PESA exploration. Choose a high fraction for `alpha_end > 0.9` to increase exploitation by the end of evolution.
- Look for an optimal balance between `npop` and `ngen`, it is recommended to minimize population size to allow for more generations.
- Total number of cost evaluations for PESA2 is `ngen*npop*3 + warmup`.

RL-informed Evolution Strategies (PPO-ES)

The Proximal Policy Optimization algorithm starts the search to collect some individuals given a fitness function through a RL environment. In the second step, the best PPO individuals are used to guide evolution strategies (ES), where RL individuals are randomly introduced into the ES population to enrich their diversity. The user first runs PPO search followed by ES, the best results of both stages are reported to the user.

Original papers:

- Radaideh, M. I., & Shirvan, K. (2021). Rule-based reinforcement learning methodology to inform evolutionary algorithms for constrained optimization of engineering applications. Knowledge-Based Systems, 217, 106836.
- Radaideh, M. I., Forget, B., & Shirvan, K. (2021). Large-scale design optimisation of boiling water reactor bundles with neuroevolution. Annals of Nuclear Energy, 160, 108355.

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.hybrid.ppoes.**PPOES**(*mode*, *fit*, *env*, *bounds*, *npop=60*, *npop_rl=6*, *init_pop_rl=True*, *hyperparam={}*, *seed=None*)

A PPO-informed ES Neuroevolution module

Parameters

- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization
- **fit** – (function) the fitness function to be used with ES
- **env** – (NEORL environment or Gym environment) The environment to learn with PPO, either use NEORL method `CreateEnvironment` (see **below**) or construct your custom Gym environment.
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **npop** – (int): population size of ES
- **npop_rl** – (int): number of RL/PPO individuals to use in ES population (`npop_rl < npop`)
- **init_pop_rl** – (bool) flag to initialize ES population with PPO individuals
- **hyperparam** – (dict) dictionary of ES hyperparameters (`cxpb`, `cxmode`, `mutpb`, `alpha`, `mu`, `smin`, `smax`) and PPO hyperparameters (`n_steps`, `gamma`, `learning_rate`, `ent_coef`, `vf_coef`, `lam`, `cliprange`, `max_grad_norm`, `nminibatches`, `noptephocs`)
- **seed** – (int) random seed for sampling

evolute(*ngen*, *ncores=1*, *verbose=False*)

This function evolutes the ES algorithm for number of generations with guidance from RL individuals.

Parameters

- **ngen** – (int) number of generations to evolute
- **ncores** – (int) number of parallel processors to use with ES
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and a list of fitness history)

learn(*total_timesteps*, *rl_filter=100*, *verbose=False*)

This function starts the learning of PPO algorithm for number of timesteps to create individuals for evolutionary search

Parameters

- **total_timesteps** – (int) number of timesteps to run

- **rl_filter** – (int) number of top individuals to keep from the full RL search
- **verbose** – (bool) print statistics to screen

Returns (dataframe) dataframe of individuals/fitness sorted from best to worst

class neorl.rl.make_env.**CreateEnvironment**(*method*, *fit*, *bounds*, *ncores=1*, *mode='max'*, *episode_length=50*)

A module to construct a fitness environment for certain algorithms that follow reinforcement learning approach of optimization

Parameters

- **method** – (str) the supported algorithms, choose either: `dqn`, `ppo`, `acktr`, `acer`, `a2c`.
- **fit** – (function) the fitness function
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **ncores** – (int) number of parallel processors
- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization (RL is default to `max`)
- **episode_length** – (int): number of individuals to evaluate before resetting the environment to random initial guess.

Example

Train a PPO-ES agent to optimize the 5-D sphere function

```
from neorl import PPOES
from neorl import CreateEnvironment

def Sphere(individual):
    """Sphere test objective function.
            F(x) = sum_{i=1}^d xi^2
            d=1,2,3,...
            Range: [-100,100]
            Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y


#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]


if __name__=='__main__':  #use this block for parallel PPO!
    #create an enviroment class for RL/PPO
    env=CreateEnvironment(method='ppo', fit=Sphere, ncores=1,
                          bounds=BOUNDS, mode='min', episode_length=50)

    #change hyperparameters of PPO/ES if you like (defaults should be good to start␣
↪with)
```

(continues on next page)

(continued from previous page)

```
    h={'cxpb': 0.8,
       'mutpb': 0.2,
       'n_steps': 24,
       'lam': 1.0}

    #Important: `mode` in CreateEnvironment and `mode` in PPOES must be consistent
    #fit is needed to be passed again for ES, must be same as the one used in env
    ppoes=PPOES(mode='min', fit=Sphere,
                env=env, npop_rl=4, init_pop_rl=True,
                bounds=BOUNDS, hyperparam=h, seed=1)
    #first run RL for some timesteps
    rl=ppoes.learn(total_timesteps=2000, verbose=True)
    #second run ES, which will use RL data for guidance
    ppoes_x, ppoes_y, ppoes_hist=ppoes.evolute(ngen=20, ncores=1, verbose=True)
↪#ncores for ES
```

RL-informed Differential Evolution (ACKTR-DE)

The Actor Critic using Kronecker-Factored Trust Region (ACKTR) algorithm starts the search to collect some individuals given a fitness function through a RL environment. In the second step, the best ACKTR individuals are used to guide differential evolution (DE), where RL individuals are randomly introduced into the DE population to enrich their diversity by replacing the worst DE individuals. The user first runs ACKTR search followed by DE, the best results of both stages are reported to the user.

Original papers:

- Radaideh, M. I., & Shirvan, K. (2021). Rule-based reinforcement learning methodology to inform evolutionary algorithms for constrained optimization of engineering applications. Knowledge-Based Systems, 217, 106836.

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.hybrid.ackde.**ACKDE**(*mode*, *fit*, *env*, *bounds*, *npop=60*, *npop_rl=6*, *init_pop_rl=True*, *hyperparam={}*, *seed=None*)

A ACKTR-informed DE Neuroevolution module

Parameters

- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization
- **fit** – (function) the fitness function to be used with DE
- **env** – (NEORL environment or Gym environment) The environment to learn with ACKTR, either use NEORL method `CreateEnvironment` (see **below**) or construct your custom Gym environment.

- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **npop** – (int): population size of DE
- **npop_rl** – (int): number of RL/ACKTR individuals to use in DE population (`npop_rl < npop`)
- **init_pop_rl** – (bool) flag to initialize DE population with ACKTR individuals
- **hyperparam** – (dict) dictionary of DE hyperparameters (`F`, `CR`) and ACKTR hyperparameters (`n_steps`, `gamma`, `learning_rate`, `ent_coef`, `vf_coef`, `vf_fisher_coef`, `kfac_clip`, `max_grad_norm`, `lr_schedule`)
- **seed** – (int) random seed for sampling

evolute(*ngen*, *ncores=1*, *verbose=False*)

This function evolutes the DE algorithm for number of generations with guidance from RL individuals.

Parameters

- **ngen** – (int) number of generations to evolute
- **ncores** – (int) number of parallel processors to use with DE
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and a list of fitness history)

learn(*total_timesteps*, *rl_filter=100*, *verbose=False*)

This function starts the learning of ACKTR algorithm for number of timesteps to create individuals for evolutionary search

Parameters

- **total_timesteps** – (int) number of timesteps to run
- **rl_filter** – (int) number of top individuals to keep from the full RL search
- **verbose** – (bool) print statistics to screen

Returns (dataframe) dataframe of individuals/fitness sorted from best to worst

class neorl.rl.make_env.**CreateEnvironment**(*method*, *fit*, *bounds*, *ncores=1*, *mode='max'*, *episode_length=50*)

A module to construct a fitness environment for certain algorithms that follow reinforcement learning approach of optimization

Parameters

- **method** – (str) the supported algorithms, choose either: `dqn`, `ppo`, `acktr`, `acer`, `a2c`.
- **fit** – (function) the fitness function
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **ncores** – (int) number of parallel processors
- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization (RL is default to `max`)
- **episode_length** – (int): number of individuals to evaluate before resetting the environment to random initial guess.

Example

Train a ACKTR-DE agent to optimize the 5-D sphere function

```
from neorl import ACKDE
from neorl import CreateEnvironment

def Sphere(individual):
    """Sphere test objective function.
            F(x) = sum_{i=1}^d xi^2
            d=1,2,3,...
            Range: [-100,100]
            Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y


#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

if __name__=='__main__':  #use this block for parallel ACKTR!
    #create an enviroment class for RL/ACKTR
    env=CreateEnvironment(method='acktr', fit=Sphere, ncores=1,
                          bounds=BOUNDS, mode='min', episode_length=50)

    #change hyperparameters of ACKTR/DE if you like (defaults should be good to start␣
↪with)
    h={'F': 0.5,
       'CR': 0.3,
       'n_steps': 20,
       'learning_rate': 0.001}

    #Important: `mode` in CreateEnvironment and `mode` in ACKDE must be consistent
    #fit is needed to be passed again for DE, must be same as the one used in env
    ackde=ACKDE(mode='min', fit=Sphere, npop=60,
                env=env, npop_rl=6, init_pop_rl=False,
                bounds=BOUNDS, hyperparam=h, seed=1)
    #first run RL for some timesteps
    rl=ackde.learn(total_timesteps=2000, verbose=True)
    #second run DE, which will use RL data for guidance
    ackde_x, ackde_y, ackde_hist=ackde.evolute(ngen=100, ncores=1, verbose=True)
↪#ncores for DE
```

Neural Genetic Algorithms (NGA)

A module for the surrogate-based genetic algorithms trained by offline data-driven tri-training approach. The surrogate model used is radial basis function networks (RBFN).

Original paper: Huang, P., Wang, H., & Jin, Y. (2021). Offline data-driven evolutionary optimization based on tri-training. Swarm and Evolutionary Computation, 60, 100800.

What can you use?

- Multi processing:
- Discrete spaces:
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces:

Parameters

class neorl.hybrid.nga.**NGA**(*mode*, *bounds*, *fit*, *npop*, *num_warmups=None*, *hidden_shape=None*, *kernel='gaussian'*, *ncores=1*, *seed=None*)

Neural Genetic Algorithm

Parameters

- **mode** – (str) problem type, either "min" for minimization problem or "max" for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **npop** – (int) population size of genetic algorithms
- **num_warmups** – (int) number of warmup samples to train the surrogate which will be evaluated by the real fitness `fit` (if `None`, `num_warmups=20*len(bounds)`)
- **hidden_shape** – (int) number of hidden layers in the RBFN network (if `None`, `hidden_shape=int(sqrt(int(num_warmups/3)))`)
- **kernel** – (str) kernel type for the RBFN network (choose from `gaussian`, `reflect`, `mul`, `inmul`)
- **ncores** – (int) number of parallel processors (currently only `ncores=1` is supported)
- **seed** – (int) random seed for sampling

evolute(*ngen*, *verbose=False*)

This function evolutes the NGA algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **verbose** – (bool) print statistics to screen

Returns (tuple) (list of best individuals, list of best fitnesses)

Example

```
from neorl import NGA

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
                    F(x) = sum_{i=1}^d xi^2
                    d=1,2,3,...
                    Range: [-100,100]
                    Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

nga = NGA(mode='min', bounds=BOUNDS, fit=FIT, npop=40, num_warmups=200,
          hidden_shape=10, seed=1)
individuals, surrogate_fit = nga.evolute(ngen=100, verbose=False)

#make evaluation of the best individuals using the real fitness function
real_fit=[FIT(item) for item in individuals]

#print the best individuals/fitness found
min_index=real_fit.index(min(real_fit))
print('------------------------ Final Summary --------------------------')
print('Best real individual:', individuals[min_index])
print('Best real fitness:', real_fit[min_index])
print('-----------------------------------------------------------------')
```

Notes

- Tri-training concept uses semi-supervised learning to leverage surrogate models that approximate the real fitness function to accelerate the optimization process for expensive fitness functions. Three RBFN models are trained, which are used to determine the best individual from one generation to the next, which is added to retrain the three surrogate models. The real fitness function `fit` is ONLY used to evaluate `num_warmups`. Afterwards, the three RBFN models are used to guide the genetic algorithm optimizer.
- For `num_warmups`, choose a reasonable value to accommodate the number of design variables `x` in your problem. If `None`, the default value of warmup samples is 20 times the size of `x`.
- For `hidden_shape`, large number of hidden layers can slow down surrogate training, small number can lead to underfitting. If `None`, the default value of `hidden_shape` is $int(\sqrt{int(num_{warmups}/3)})$.
- The `kernel` can play a significant role in surrogate training. Four options are available: Gaussian function (`gaussian`), Reflected function (`reflect`), Multiquadric function (`mul`), and Inverse multiquadric function (`inmul`).
- Total number of cost evaluations via the real fitness function `fit` for NGA is `num_warmups`.
- Total number of cost evaluations via the surrogate model for NGA is `npop` * `ngen`.

Neural Harris Hawks Optimization (NHHO)

A module for the surrogate-based Harris Hawks Optimization trained by offline data-driven tri-training approach. The surrogate model used is feedforward neural networks constructed from tensorflow.

Original paper: Huang, P., Wang, H., & Jin, Y. (2021). Offline data-driven evolutionary optimization based on tri-training. Swarm and Evolutionary Computation, 60, 100800.

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.hybrid.nhho.**NHHO**(*mode*, *bounds*, *fit*, *nhawks*, *num_warmups=None*, *int_transform='nearest_int'*, *nn_params={}*, *ncores=1*, *seed=None*)

Neural Harris Hawks Optimizer

Parameters

- **mode** – (str) problem type, either "min" for minimization problem or "max" for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **nhawks** – (int): number of the hawks in the group
- **num_warmups** – (int) number of warmup samples to train the surrogate which will be evaluated by the real fitness `fit` (if `None`, `num_warmups=20*len(bounds)`)
- **int_transform** – (str): method of handling int/discrete variables, choose from: `nearest_int`, `sigmoid`, `minmax`.
- **nn_params** – (dict) parameters for building the surrogate models in dictionary form. Keys are: `test_split`, `learning_rate`, `activation`, `num_nodes`, `batch_size`, `epochs`, `save_models`, `verbose`, `plot`. See **Notes** below for descriptions.
- **ncores** – (int) number of parallel processors to train the three surrogate models (only `ncores=1` or `ncores=3` are allowed)
- **seed** – (int) random seed for sampling

evolute(*ngen*, *x0=None*, *verbose=False*)

This function evolutes the NHHO algorithm for number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **x0** – (list of lists) initial position of the hawks (must be of same size as `nhawks`)

- **verbose** – (bool) print statistics to screen

Returns (tuple) (list of best individuals, list of best fitnesses)

Example

```
from neorl import NHHO
import time
import sys

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
                    F(x) = sum_{i=1}^d xi^2
                    d=1,2,3,...
                    Range: [-100,100]
                    Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

nn_params = {}
nn_params['num_nodes'] = [60, 30, 15]
nn_params['learning_rate'] = 8e-4
nn_params['epochs'] = 100
nn_params['plot'] = False #will accelerate training
nn_params['verbose'] = False #will accelerate training
nn_params['save_models'] = False  #will accelerate training

try:
    ngen=int(sys.argv[1])  #get ngen as external argument for testing
except:
    ngen=50      #or use default ngen

t0=time.time()
nhho = NHHO(mode='min', bounds=BOUNDS, fit=FIT, nhawks=20,
            nn_params=nn_params, ncores=3, seed=1)
individuals, fitnesses = nhho.evolute(ngen=ngen, verbose=True)
print('Comp Time:', time.time()-t0)

#make evaluation of the best individuals using the real fitness function
real_fit=[FIT(item) for item in individuals]

#print the best individuals/fitness found
min_index=real_fit.index(min(real_fit))
print('------------------------ Final Summary --------------------------')
print('Best real individual:', individuals[min_index])
print('Best real fitness:', real_fit[min_index])
print('-----------------------------------------------------------------')
```

Notes

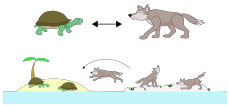
- Tri-training concept uses semi-supervised learning to leverage surrogate models that approximate the real fitness function to accelerate the optimization process for expensive fitness functions. Three feedforward neural network models are trained, which are used to determine the best individual from one generation to the next, which is added to retrain the three surrogate models. The real fitness function `fit` is ONLY used to evaluate `num_warmups`. Afterwards, the three neural network models are used to guide the Harris hawks optimizer.
- For `num_warmups`, choose a reasonable value to accommodate the number of design variables `x` in your problem. If `None`, the default value of warmup samples is 20 times the size of `x`.
- Total number of cost evaluations via the real fitness function `fit` for NHHO is `num_warmups`.
- Total number of cost evaluations via the surrogate model for NHHO is `2 * nhawks` * `ngen`.
- The following variables can be used in `nn_params` dictionary to construct the surrogate model

Hyperparameter	Description
– num_nodes – learning_rate – batch_size – activation – test_split – epochs – verbose – save_models – plot	– List of number of nodes, e.g. [64, 32] creates two layer-network with 64 and 32 nodes (default: [100, 50, 25]) – The learning rate of Adam optimizer (default: 6e-4) – The minibatch size (default: 32) – Activation function type (default: `relu`) – Fraction of test data or test split (default: 0.2) – Number of training epochs (default: 20) – Flag to print different surrogate error to screen (default: True) – Flag to save the neural network models (default: True) – Flag to generate plots for surrogate training loss and surrogate prediction accuracy (default: True)

Animorphic Ensemble Optimization (AEO)

A module for Animorphic Ensemble Optimization. A hybrid island model with different evolutionary populations evolved in islands.

Original paper: in progress



What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.hybrid.aeo.**AEO**(*bounds*, *fit*, *optimizers*, *gen_per_cycle*, *mode='min'*, *seed=None*, ***kwargs*)

Animorphoc Ensemble Optimizer

Parameters

- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **optimizers** – (list) list of optimizer instances to be included in the ensemble
- **gen_per_cycle** – (int) number of generations performed in evolution phase per cycle
- **mode** – (str) problem type, either "min" for minimization problem or "max" for maximization
- **seed** – (int) random seed for sampling

evolute(*Ncyc*, *npop0=None*, *x0=None*, *pop0=None*, *stop_criteria=None*, *verbose=False*)

This function evolutes the AEO algorithm for a number of cycles. Either (`npop0`) or (`x0` and `pop0`) are required.

Parameters

- **Ncyc** – (int) number of cycles to evolute
- **pop0** – (list of ints) number of individuals in starting population for each optimizer
- **x0** – (list of lists) initial positions of individuals in problem space
- **pop0** – (list of ints) population assignments for `x0`, integer corresponding to assigned population ordered according to self.optimize
- **stop_criteria** – (None or callable) function which returns condition if evolution should continue, can be used to stop evolution at certain number of function evaluations

Returns (tuple) (best individual, best fitness, xarray.Dataset of various algorithm parameters)

Example

```
import matplotlib.pyplot as plt
from neorl import AEO

from neorl import DE
from neorl import ES
from neorl import GWO
from neorl import PSO
from neorl import WOA
from neorl import MFO
from neorl import SSA
from neorl import JAYA

#define the fitness function
def FIT(individual):
    """Sphere test objective function.
            F(x) = sum_{i=1}^d x_i^2
            d=1,2,3,...
            Range: [-100,100]
            Minima: 0
    """
    y = sum(x**2 for x in individual)

    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

#Define algorithms to be used in enembles
#   parameters not directly describing population size
#   are carried into the AEO algorithm. See de2 for an
#   example of this.
es  =  ES(mode='min', fit=FIT, bounds=BOUNDS)
gwo = GWO(mode='min', fit=FIT, bounds=BOUNDS)
woa = WOA(mode='min', fit=FIT, bounds=BOUNDS)
mfo = MFO(mode='min', fit=FIT, bounds=BOUNDS)
ssa = SSA(mode='min', fit=FIT, bounds=BOUNDS)
de1 =  DE(mode='min', fit=FIT, bounds=BOUNDS)
de2 =  DE(mode='min', fit=FIT, bounds=BOUNDS, F=0.5, CR=0.5)
pso = PSO(mode='min', fit=FIT, bounds=BOUNDS)
jaya=JAYA(mode='min', fit=FIT, bounds=BOUNDS)

ensemble = [es, gwo, woa, mfo, ssa, de1, de2, pso, jaya]

#initialize an intance of aeo
aeo = AEO(mode='min', fit=FIT, bounds=BOUNDS, optimizers=ensemble,
        gen_per_cycle=2)

#perform evolution
best_x, best_y, log = aeo.evolute(15)

print('Best x')
print(best_x)
```

(continues on next page)

(continued from previous page)

```
print('Best y')
print(best_y)

plt.figure()
for p in log.coords['pop']:
    plt.plot(log.coords['cycle'], log['nmembers'].sel({'pop' : p}),
             label = p.values)

plt.xlabel("Cycle")
plt.ylabel("Number of Members")
plt.legend()
plt.show()
```

Notes

- Only valid in `mode='min'`.
- AEO supports `ES`, `GWO`, `WOA`, `MFO`, `SSA`, `DE`, `PSO` and `JAYA`.
- Algorithm objects must be defined prior to their inclusion in `AEO` (see example)
- Parameters such as `F` in `DE` or `mu` in `ES` are carried into `AEO` after initialization of these algorithms.
- Population size parameters such as `nwolves` in `GWO` are used to determine the starting populations but are changed as the algorithm progresses.
- `fit`, `bounds` and `mode` should be consistent across algorithms passed into `AEO`.
- The total number of function evaluation changes depending on the algorithms in the ensemble and the distribution of members.
- Information on the returned *log* can be found in the code for the `AEO` class.
- Extra options around the migration process can be accessed through the `kwargs` paramter.

Ensemble of Differential Evolution Variants (EDEV)

A powerful hybrid ensemble of three differential evolution variants: JADE (adaptive differential evolution with optional external archive), CoDE (differential evolution with composite trial vector generation strategies and control parameters), and EPSDE (differential evolution algorithm with ensemble of parameters and mutation strategies).

Original paper: Wu, G., Shen, X., Li, H., Chen, H., Lin, A., Suganthan, P. N. (2018). Ensemble of differential evolution variants. Information Sciences, 423, 172-186.

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.hybrid.edev.**EDEV**(*mode*, *bounds*, *fit*, *npop=100*, *int_transform='nearest_int'*, *ncores=1*, *seed=None*)

Ensemble of differential evolution variants

Parameters

- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **npop** – (int): total size of the full population, which will be divided into three subpopulations
- **int_transform** – (str): method of handling int/discrete variables, choose from: `nearest_int`, `sigmoid`, `minmax`.
- **ncores** – (int) number of parallel processors (must be `<= npop`)
- **seed** – (int) random seed for sampling

evolute(*ngen*, *ng=20*, *x0=None*, *verbose=False*)

This function evolutes the EDEV algorithm for a number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **ng** – (int) the period or number of generations to determine the best performing DE variant and reward subpopulation assignment, `ng < ngen` (see **Notes** below for more info).
- **x0** – (list of lists) initial position of the individuals (must be of same size as `npop`)
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and dictionary containing major search results)

Example

```
from neorl import EDEV

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
                    F(x) = sum_{i=1}^d xi^2
                    d=1,2,3,...
                    Range: [-100,100]
                    Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
for i in range(1,nx+1):
```

(continues on next page)

(continued from previous page)

```
    BOUNDS['x'+str(i)]=['float', -100, 100]

#setup and evolute EDEV
edev=EDEV(mode='min', bounds=BOUNDS, fit=FIT, npop=100, ncores=1, seed=1)
x_best, y_best, edev_hist=edev.evolute(ngen=100, ng=10, verbose=1)
```

Notes

- Choosing the value of `npop` and `lambda_` should be careful for EDEV. The third variant (EPSDE) performs mutation and crossover on 6 different individuals. Therefore, make sure that the second and third sub-populations have more than 6 individuals for optimal performance, or simply ensure that `int(npop * lambda_) > 6`.
- Increasing `lambda_` value will make the size of the three sub-populations comparable. In the original paper, one population is bigger than the others, so for `lambda_ = 0.1` and `npop=100`, the three sub-populations have sizes 80, 10, and 10.
- The parallelization of EDEV is bottlenecked by the size of the sub-populations. For example, for sub-populations of size 80, 10, and 10, using `ncores = 80` will ensure that the first sub-population is executed in one round, but the other two sub-populations will be evaluated in sequence with 10 cores only.
- Unlike standalone DE, for EDEV, the values of the hyperparameters `F` and `CR` are automatically adapted.
- The parameter `ng` in the `evolute` function helps to determine the frequency/period at which the three sub-populations are swapped between the DE variants based on their prior performance. For example, if `ngen = 100` and `ng=20`, five updates will occur during the full evolution process.
- Look for an optimal balance between `npop` and `ngen`, it is recommended to keep population size optimal to allow for more generations, but also sufficient to keep the three sub-populations active.
- As EDEV is a bit complex, the total number of cost evaluations for EDEV can be accessed via the returned dictionary key: `edev_hist['F-Evals']` in the example above.

Ensemble Particle Swarm Optimization (EPSO)

A powerful hybrid ensemble of five particle swarm optimization variants: classical inertia weight particle swarm optimization (PSO), self-organizing hierarchical particle swarm optimizer withtime-varying acceleration coefficients (HPSO-TVAC), Fitness-Distance-Ratio based PSO (FDR-PSO), Distance-based locally informed PSO (LIPS), and Comprehensive Learning PSO (CLPSO).

Original paper: Lynn, N., Suganthan, P. N. (2017). Ensemble particle swarm optimizer. Applied Soft Computing, 55, 533-548.

What can you use?

- Multi processing: ✓
- Discrete spaces: ✓
- Continuous spaces: ✓
- Mixed Discrete/Continuous spaces: ✓

Parameters

class neorl.hybrid.epso.**EPSO**(*mode*, *bounds*, *fit*, *g1=15*, *g2=25*, *int_transform='nearest_int'*, *ncores=1*, *seed=None*)

Ensemble Particle Swarm Optimization (EPSO)

Parameters

- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization
- **bounds** – (dict) input parameter type and lower/upper bounds in dictionary form. Example: `bounds={'x1': ['int', 1, 4], 'x2': ['float', 0.1, 0.8], 'x3': ['float', 2.2, 6.2]}`
- **fit** – (function) the fitness function
- **g1** – (int): number of particles in the exploration group
- **g2** – (int): number of particles in the exploitation group (total swarm size is `g1 + g2`)
- **int_transform** – (str): method of handling int/discrete variables, choose from: `nearest_int`, `sigmoid`, `minmax`.
- **ncores** – (int) number of parallel processors (must be `<= g1+g2`)
- **seed** – (int) random seed for sampling

evolute(*ngen*, *LP=3*, *x0=None*, *verbose=False*)

This function evolutes the EPSO algorithm for a number of generations.

Parameters

- **ngen** – (int) number of generations to evolute
- **LP** – (int) number of generations before updating the success and failure memories for the ensemble variants (i.e. learning period)
- **x0** – (list of lists) initial position of the particles (must be of same size as `g1 + g2`)
- **verbose** – (bool) print statistics to screen

Returns (tuple) (best individual, best fitness, and dictionary containing major search results)

Example

```
from neorl import EPSO

#Define the fitness function
def FIT(individual):
    """Sphere test objective function.
                    F(x) = sum_{i=1}^d xi^2
                    d=1,2,3,...
                    Range: [-100,100]
                    Minima: 0
    """
    y=sum(x**2 for x in individual)
    return y

#Setup the parameter space (d=5)
nx=5
BOUNDS={}
```

(continues on next page)

(continued from previous page)

```
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

#setup and evolute EPSO
epso=EPSO(mode='min', bounds=BOUNDS, g1=15, g2=25, fit=FIT, ncores=1, seed=None)
x_best, y_best, epso_hist=epso.evolute(ngen=100, LP=3, verbose=1)
```

Notes

- The number of particles in the exploration subgroup (`g1`) and exploitation subgroup (`g2`) are needed for EPSO. In the original algorithm, `g1` tends to be smaller than `g2`.
- For EPSO, in the first 90% of the generations, both exploration and exploitation subgroups are involved, where `g1` is controlled by CLPSO and `g2` is controlled by all five variants. In the last 10% of the generations, the search focuses on exploitation only, where both `g1 + g2` are controlled by the five variants.
- The value of LP represents the learning period at which the success and fail memories are updated to calculate the success rate for each PSO variant. The success rate represents the probability for each PSO variant to update the position and velocity of the next particle in the group. `LP=3` means the update will occur every 3 generations.
- Look for an optimal balance between `g1`, `g2`, and `ngen`, it is recommended to minimize particle size to allow for more generations.
- Total number of cost evaluations for EPSO is `(g1 + g2) * (ngen + 1)`.

2.3 Hyperparameter Tuning

This section highlights the supported methods to tune the hyperparameters of NEORL algorithms.

2.3.1 Grid Search

A module for grid search of hyperparameters of NEORL algorithms.

Original paper: Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. Journal of machine learning research, 13(2).

Grid Search is an exhaustive search for selecting an optimal set of algorithm hyperparameters. In Grid Search, the analyst sets up a grid of hyperparameter values. A multi-dimensional full grid of all hyperparameters is constructed, which contains all possible combinations of hyperparameters. Afterwards, every combination of hyperparameter values is tested in serial/parallel, where the optimization score (e.g. fitness) is estimated. Grid search can be very expensive for fine grids as well as large number of hyperparameters to tune.

What can you use?

- Multi processing: ✓
- Discrete/Continuous/Mixed spaces: ✓
- Reinforcement Learning Algorithms: ✓
- Evolutionary Algorithms: ✓
- Hybrid Neuroevolution Algorithms: ✓

Parameters

class neorl.tune.gridtune.**GRIDTUNE**(*param_grid*, *fit*)
A module for grid search for hyperparameter tuning

Parameters

- **param_grid** – (dict) the grid (list of possible values) for each hyperparameter provided in a dictionary form. Example: {'x1': [40, 50, 60, 80, 100], 'x2': [0.2, 0.4, 0.8], 'x3': ['blend', 'cx2point']}
- **fit** – (function) the self-defined fitness function that includes the hyperparameters as input and algorithm score as output

tune(*ncores=1*, *csvname=None*, *verbose=True*)
This function starts the tuning process with specified number of processors

Parameters

- **ncores** – (int) number of parallel processors (see the **Notes** section below for an important note about parallel execution)
- **csvname** – (str) the name of the csv file name to save the tuning results (useful for expensive cases as the csv file is updated directly after the case is done)
- **verbose** – (bool) whether to print updates to the screen or not

Example

Example of using grid search to tune three ES hyperparameters for solving the 5-d Sphere function

```
from neorl.tune import GRIDTUNE
from neorl import ES

#************************************************************
# Part I: Original Problem Settings
#************************************************************

#Define the fitness function (for original optimisation)
def sphere(individual):
    y=sum(x**2 for x in individual)
    return y

#************************************************************
# Part II: Define fitness function for hyperparameter tuning
#************************************************************
def tune_fit(cxpb, mutpb, alpha):
```

(continues on next page)

(continued from previous page)

```
    #--setup the parameter space
    nx=5
    BOUNDS={}
    for i in range(1,nx+1):
        BOUNDS['x'+str(i)]=['float', -100, 100]

    #--setup the ES algorithm
    es=ES(mode='min', bounds=BOUNDS, fit=sphere, lambda_=80, mu=40, mutpb=mutpb,␣
↪alpha=alpha,
          cxmode='blend', cxpb=cxpb, ncores=1, seed=1)

    #--Evolute the ES object and obtains y_best
    #--turn off verbose for less algorithm print-out when tuning
    x_best, y_best, es_hist=es.evolute(ngen=100, verbose=0)

    return y_best #returns the best score

#=============================================================
# Part III: Tuning
#=============================================================
param_grid={
#def tune_fit(cxpb, mutpb, alpha):
'cxpb': [0.2, 0.4],  #cxpb is first
'mutpb': [0.05, 0.1],  #mutpb is second
'alpha': [0.1, 0.2, 0.3, 0.4]}  #alpha is third

#setup a grid tune object
gtune=GRIDTUNE(param_grid=param_grid, fit=tune_fit)
#view the generated cases before running them
print(gtune.hyperparameter_cases)
#tune the parameters with method .tune
gridres=gtune.tune(ncores=1, csvname='tune.csv')
print(gridres)
```

Notes

- For `ncores > 1`, the parallel tuning engine starts. **Make sure to run your python script from the terminal NOT from an IDE (e.g. Spyder, Jupyter Notebook)**. IDEs are not robust when running parallel problems with packages like `joblib` or `multiprocessing`. For `ncores = 1`, IDEs seem to work fine.
- If there are large number of hyperparameters to tune (large d), try nested grid search. First, run a grid search on few parameters first, then fix them to their best, and start another grid search for the next group of hyperparameters, and so on.
- Always start with coarse grid for all hyperparameters (small k_i) to obtain an impression about their sensitivity. Then, refine the grids for those hyperparameters with more impact, and execute a more detailed grid search.
- Grid search is ideal to use when the analyst has prior experience on the feasible range of each hyperparameter and the most important hyperparameters to tune.

2.3.2 Random Search

A module for random search of hyperparameters of NEORL algorithms.

Original paper: Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. Journal of machine learning research, 13(2).

Random search is a technique where random combinations of the hyperparameters are used to find the best solution for the algorithm used. Random search tries random combinations of the hyperparameters, where the cost function is evaluated at these random sets in the parameter space. As indicated by the reference above, the chances of finding the optimal hyperparameters are comparatively higher in random search than grid search. This is because of the random search pattern, as the algorithm might end up being used on the optimized hyperparameters without any aliasing or wasting of resources.

What can you use?

- Multi processing: ✓
- Discrete/Continuous/Mixed spaces: ✓
- Reinforcement Learning Algorithms: ✓
- Evolutionary Algorithms: ✓
- Hybrid Neuroevolution Algorithms: ✓

Parameters

class neorl.tune.randtune.**RANDTUNE**(*param_grid*, *fit*, *ncases=50*, *seed=None*)

A module for random search for hyperparameter tuning

Parameters

- **param_grid** – (dict) the type and range of each hyperparameter in a dictionary form (types are `int/discrete` or `float/continuous` or `grid/categorical`). Example: {'x1': [[40, 50, 60, 100], 'grid'], 'x2': [[0.2, 0.8], 'float'], 'x3': [['blend', 'cx2point'], 'grid'], 'x4': [[20, 80], 'int']}
- **fit** – (function) the self-defined fitness function that includes the hyperparameters as input and algorithm score as output
- **ncases** – (int) number of random hyperparameter cases to generate
- **seed** – (int) random seed for sampling reproducibility

tune(*ncores=1*, *csvname=None*, *verbose=True*)

This function starts the tuning process with specified number of processors

Parameters

- **ncores** – (int) number of parallel processors (see the **Notes** section below for an important note about parallel execution)
- **csvname** – (str) the name of the csv file name to save the tuning results (useful for expensive cases as the csv file is updated directly after the case is done)
- **verbose** – (bool) whether to print updates to the screen or not

Example

Example of using random search to tune three ES hyperparameters for solving the 5-d Sphere function

```
from neorl.tune import RANDTUNE
from neorl import ES

#************************************************************
# Part I: Original Problem Settings
#************************************************************

#Define the fitness function (for original optimisation)
def sphere(individual):
    y=sum(x**2 for x in individual)
    return y

#************************************************************
# Part II: Define fitness function for hyperparameter tuning
#************************************************************
def tune_fit(cxpb, mu, alpha, cxmode):

    #--setup the parameter space
    nx=5
    BOUNDS={}
    for i in range(1,nx+1):
        BOUNDS['x'+str(i)]=['float', -100, 100]

    #--setup the ES algorithm
    es=ES(mode='min', bounds=BOUNDS, fit=sphere, lambda_=80, mu=mu, mutpb=0.1,␣
↪alpha=alpha,
          cxmode=cxmode, cxpb=cxpb, ncores=1, seed=1)

    #--Evolute the ES object and obtains y_best
    #--turn off verbose for less algorithm print-out when tuning
    x_best, y_best, es_hist=es.evolute(ngen=100, verbose=0)

    return y_best #returns the best score

#************************************************************
# Part III: Tuning
#************************************************************
#Setup the parameter space
#VERY IMPORTANT: The order of these parameters MUST be similar to their order in tune_
↪fit
#see tune_fit
param_grid={
#def tune_fit(cxpb, mu, alpha, cxmode):

'cxpb': ['float', 0.1, 0.9],             #cxpb is first (low=0.1, high=0.8,␣
↪type=float/continuous)
'mu':   ['int', 30, 60],                 #mu is second (low=30, high=60, type=int/
↪discrete)
'alpha':['grid', (0.1, 0.2, 0.3, 0.4)],    #alpha is third (grid with fixed values,␣
↪type=grid/categorical)
'cxmode':['grid', ('blend', 'cx2point')]}  #cxmode is fourth (grid with fixed values,␣
↪type=grid/categorical)

#setup a random tune object
```

(continues on next page)

(continued from previous page)

```
rtune=RANDTUNE(param_grid=param_grid, fit=tune_fit, ncases=25, seed=1)
#view the generated cases before running them
print(rtune.hyperparameter_cases)
#tune the parameters with method .tune
randres=rtune.tune(ncores=1, csvname='tune.csv')
print(randres)
```

Notes

- For `ncores > 1`, the parallel tuning engine starts. **Make sure to run your python script from the terminal NOT from an IDE (e.g. Spyder, Jupyter Notebook)**. IDEs are not robust when running parallel problems with packages like `joblib` or `multiprocessing`. For `ncores = 1`, IDEs seem to work fine.
- Random search struggles with dimensionality if there are large number of hyperparameters to tune. Therefore, it is always recommended to do a preliminary sensitivity study to exclude or fix the hyperparameters with small impact.
- To determine an optimal `ncases`, try to setup your problem for grid search on paper, calculate the grid search `ncases`, and go for 50% of this number. Achieving similar performance with 50% cost is a promise for random search.
- For difficult problems, the analyst can start with a random search first to narrow the choices of the important hyperparameters. Then, a grid search can be executed on those important parameters with more refined and narrower grids.

2.3.3 Bayesian Search

A module of Bayesian optimisation search for hyperparameter tuning of NEORL algorithms based upon `scikit-optimise`.

Original paper: https://arxiv.org/abs/1012.2599

Bayesian search, in contrast to grid and random searches, keeps track of past evaluation results. Bayesian uses past evaluations to form a probabilistic model mapping hyperparameters to a probability of a score on the objective function (e.g. max/min fitness). Bayesian optimization excels when the objective functions are expensive to evaluate, when we do not have access to derivatives, or when the problem at hand is non-convex.

What can you use?

- Multi processing: ✓ (Multithreading in a single processor)
- Discrete/Continuous/Mixed spaces: ✓
- Reinforcement Learning Algorithms: ✓
- Evolutionary Algorithms: ✓
- Hybrid Neuroevolution Algorithms: ✓

Parameters

class neorl.tune.bayestune.**BAYESTUNE**(*param_grid*, *fit*, *mode='min'*, *ncases=50*, *seed=None*)
A module for Bayesian search for hyperparameter tuning

Parameters

- **param_grid** – (dict) the type and range of each hyperparameter in a dictionary form (types are `int/discrete` or `float/continuous` or `grid/categorical`). Example: {'x1': [[40, 50, 60, 100], 'grid'], 'x2': [[0.2, 0.8], 'float'], 'x3': [['blend', 'cx2point'], 'grid'], 'x4': [[20, 80], 'int']}
- **fit** – (function) the self-defined fitness function that includes the hyperparameters as input and algorithm score as output
- **mode** – (str) problem type, either "min" for minimization problem or "max" for maximization. Default: Bayesian tuner is set to minimize an objective
- **ncases** – (int) number of random hyperparameter cases to generate per core, `ncases >= 11` (see **Notes** for an important remark)
- **seed** – (int) random seed for sampling reproducibility

tune(*ncores=1*, *csvname=None*, *verbose=True*)
This function starts the tuning process with specified number of processors

Parameters

- **nthreads** – (int) number of parallel threads (see the **Notes** section below for an important note about parallel execution)
- **csvname** – (str) the name of the csv file name to save the tuning results (useful for expensive cases as the csv file is updated directly after the case is done)
- **verbose** – (bool) whether to print updates to the screen or not

Example

```
from neorl.tune import BAYESTUNE
from neorl import ES

#************************************************************
# Part I: Original Problem Settings
#************************************************************

#Define the fitness function (for original optimisation)
def sphere(individual):
    y=sum(x**2 for x in individual)
    return y

#************************************************************
# Part II: Define fitness function for hyperparameter tuning
#************************************************************
def tune_fit(cxpb, mu, alpha, cxmode):

    #--setup the parameter space
    nx=5
    BOUNDS={}
    for i in range(1,nx+1):
```

(continues on next page)

(continued from previous page)

```
            BOUNDS['x'+str(i)]=['float', -100, 100]

    #--setup the ES algorithm
    es=ES(mode='min', bounds=BOUNDS, fit=sphere, lambda_=80, mu=mu, mutpb=0.1,␣
↪alpha=alpha,
              cxmode=cxmode, cxpb=cxpb, ncores=1, seed=1)

    #--Evolute the ES object and obtains y_best
    #--turn off verbose for less algorithm print-out when tuning
    x_best, y_best, es_hist=es.evolute(ngen=100, verbose=0)

    return y_best #returns the best score

#************************************************************
# Part III: Tuning
#************************************************************
#Setup the parameter space
#VERY IMPORTANT: The order of these parameters MUST be similar to their order in tune_
↪fit
#see tune_fit
param_grid={
#def tune_fit(cxpb, mu, alpha, cxmode):
'cxpb': ['float', 0.1, 0.9],             #cxpb is first (low=0.1, high=0.8,␣
↪type=float/continuous)
'mu':   ['int', 30, 60],                 #mu is second (low=30, high=60, type=int/
↪discrete)
'alpha':['grid', [0.1, 0.2, 0.3, 0.4]],    #alpha is third (grid with fixed values,␣
↪type=grid/categorical)
'cxmode':['grid', ['blend', 'cx2point']]}  #cxmode is fourth (grid with fixed values,␣
↪type=grid/categorical)

#setup a bayesian tune object
btune=BAYESTUNE(mode='min', param_grid=param_grid, fit=tune_fit, ncases=30)
#tune the parameters with method .tune
bayesres=btune.tune(ncores=1, csvname='bayestune.csv', verbose=True)
print(bayesres)
btune.plot_results(pngname='bayes_conv')
```

Notes

- We allow a weak parallelization of Bayesian search via multithreading. The user can start independent Bayesian search with different seeds by increasing `ncores`. However, all threads will be executed on a single processor, which will slow down every Bayesian sequence. Therefore, this option is recommended when each hyperparameter case is fast-to-evaluate and does not require intensive CPU power.
- If the user sets `ncores=4` and sets `ncases=15`, a total of 60 hyperparameter cases are evaluated, where each thread uses 25% of the CPU power. **The extension to multiprocessing/multi-core capability is on track in future**.
- Keep `ncases >= 11`. If ncases < 11, the optimiser resets `ncases=11`. It is good to start with `ncases=30`, check the optimizer convergence, and increase as needed.
- Relying on `grid/categorical` variables can accelerate the search by a wide margin. Therefore, if the user is aware of certain values of the (`int/discrete`) or the (`float/continuous`) hyperparameters, it is good to convert them to `grid/categorical`.

Acknowledgment

Thanks to our fellows in scikit-optimize, as we used their `gp_minimize` implementation to leverage our Bayesian search module in our framework.

Head, Tim, Gilles Louppe MechCoder, and Iaroslav Shcherbatyi. "scikit-optimize/scikit-optimize: v0.7.1"(2020).

2.3.4 Evolutionary Search

A module of evolutionary search for hyperparameter tuning of NEORL algorithms.

Original paper: E. Bochinski, T. Senst and T. Sikora, "Hyper-parameter optimization for convolutional neural network committees based on evolutionary algorithms," 2017 IEEE International Conference on Image Processing (ICIP), Beijing, China, 2017, pp. 3924-3928, doi: 10.1109/ICIP.2017.8297018.

We have used a compact evolution strategies (ES) module for the purpose of tuning the hyperparameters of NEORL algorithms. See the *ES algorithm* section for more details about the (μ, λ) algorithm. ES tuner leverages a population of individuals, where each individual represents a sample from the hyperparameter space. ES uses recombination, crossover, and mutation operations to improve the individuals from generation to the other. The best of the best individuals in all generations are reported as the top hyperparameter sets to use further with the algorithm.

What can you use?

- Multi processing: ✓
- Discrete/Continuous/Mixed spaces: ✓
- Reinforcement Learning Algorithms: ✓
- Evolutionary Algorithms: ✓
- Hybrid Neuroevolution Algorithms: ✓

Parameters

class `neorl.tune.estune.`**ESTUNE**(*param_grid*, *fit*, *mode='max'*, *ngen=10*, *seed=None*)

A module for Evolutionary search for hyperparameter tuning based on ES algorithm

Parameters

- **param_grid** – (dict) the type and range of each hyperparameter in a dictionary form (types are `int/discrete` or `float/continuous` or `grid/categorical`).
- **fit** – (function) the self-defined fitness function that includes the hyperparameters as input and algorithm score as output
- **mode** – (str) problem type, either `min` for minimization problem or `max` for maximization. Default: Evolutionary tuner is set to maximize an objective
- **ngen** – (int) number of ES generations to run, total number of hyperparameter tests is `ngen * 10` (see **Notes** for an important remark)
- **seed** – (int) random seed for sampling reproducibility

tune(*ncores=1*, *csvname=None*, *verbose=True*)

This function starts the tuning process with specified number of processors

Parameters

- **ncores** – (int) number of parallel processors.

- **csvname** – (str) the name of the csv file name to save the tuning results (useful for expensive cases as the csv file is updated directly after the case is done)
- **verbose** – (bool) whether to print updates to the screen or not

Example

```
from neorl.tune import ESTUNE
from neorl import PSO

#************************************************************
# Part I: Original Problem Settings
#************************************************************

#Define the fitness function (for original optimisation)
def sphere(individual):
    y=sum(x**2 for x in individual)
    return y

#************************************************************
# Part II: Define fitness function for hyperparameter tuning
#************************************************************
def tune_fit(x):

    npar=x[0]
    c1=x[1]
    c2=x[2]
    if x[3] == 1:
        speed_mech='constric'
    elif x[3] == 2:
        speed_mech='timew'
    elif x[3] == 3:
        speed_mech='globw'

    #--setup the parameter space
    nx=5
    BOUNDS={}
    for i in range(1,nx+1):
            BOUNDS['x'+str(i)]=['float', -100, 100]

    #--setup the PSO algorithm
    pso=PSO(mode='min', bounds=BOUNDS, fit=sphere, npar=npar, c1=c1, c2=c2,
             speed_mech=speed_mech, ncores=1, seed=1)

    #--Evolute the PSO object and obtains y_best
    #--turn off verbose for less algorithm print-out when tuning
    x_best, y_best, pso_hist=pso.evolute(ngen=30, verbose=0)

    return y_best #returns the best score

#************************************************************
# Part III: Tuning
#************************************************************
#Setup the parameter space
#VERY IMPORTANT: The order of these parameters MUST be similar to their order in tune_
↪fit
#see tune_fit
```

(continues on next page)

(continued from previous page)

```
param_grid={
#def tune_fit(npar, c1, c2, speed_mech):
'npar': ['int', 40, 60],         #npar is first (low=30, high=60, type=int/discrete)
'c1': ['float', 2.05, 2.15],     #c1 is second (low=2.05, high=2.15, type=float/
↪continuous)
'c2': ['float', 2.05, 2.15],     #c2 is third (low=2.05, high=2.15, type=float/
↪continuous)
'speed_mech': ['int', 1, 3]}     #speed_mech is fourth (categorial variable encoded as␣
↪integer, see tune_fit)

#setup a evolutionary tune object
etune=ESTUNE(mode='min', param_grid=param_grid, fit=tune_fit, ngen=10) #total cases␣
↪is ngen * 10
#tune the parameters with method .tune
evolures=etune.tune(ncores=1, csvname='evolutune.csv', verbose=True)
evolures = evolures.sort_values(['score'], axis='index', ascending=True) #rank the␣
↪scores from min to max
print(evolures)
etune.plot_results(pngname='evolu_conv')
```

Notes

- Evolutionary search uses fixed values for `lambda_=10` and `mu=10`.
- Therefore, the total cost of evolutionary search or the total number of hyperparameter tests is `ngen * 10`.
- For categorical variables, use integers to encode them as integer variables. Then, inside the `tune_fit` function, the integers are converted back to the real categorical value. See how `speed_mech` is handled in the example above.
- The strategy and individual vectors in the ES tuner are updated similarly to the ES algorithm module described *here*.
- For difficult problems, the analyst can start with a random search first to narrow the choices of the important hyperparameters. Then, an evolutionary search can be executed on those important parameters to refine their values.

2.4 Examples

This section highlights various mathematical and engineering examples of NEORL.

2.4.1 Example 1: Traveling Salesman Problem

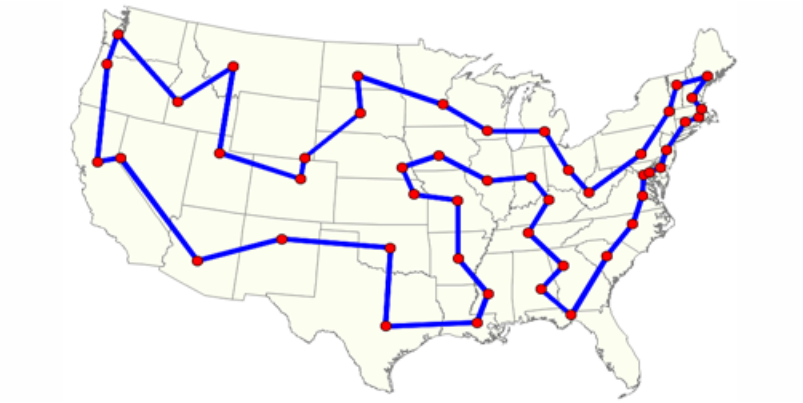
Example of solving the classical discrete optimization problem "Traveling Salesman Problem" (TSP) using NEORL with state-of-the-art reinforcement learning algorithms to demonstrate compatibility with discrete space.

Summary

- Algorithm: PPO, A2C, ACKTR, DQN, ACER
- Type: Discrete/Combinatorial, Single Objective, Constrained
- Field: Computational Mathematics

Problem Description

The Traveling Salesman Problem (TSP) is a well known problem in the discrete optimization community and find applications well beyond computational mathematics: genetics, chemistry,telecommunications, and nuclear reactor optimization. Based on the spatial distribution of cities representing the vertices on a Graph G, the objective is to minimize the length of a tour $\mathcal{T}$ defined as a cycle on the graph that visits all vertices (cities) only once and circle back to the departure state. An example is given in the figure below:



Formally, the graph is described by $G = (V, A)$, where V is the collection of N vertices, $A = \{(i, j) : i, j \in V\}$ is the set of edges with cost $c : A- > \mathbb{R}$, $c : (i, j)- > c_{i,j}$, where c_{ij} is the distance from node i to node j. Lastly, x_{ij} are boolean variables such that $x_{ij} = 1$ if the edge (i, j) is active. One formulation of the problem is (known as the "cutset" formulation):

$$
\begin{aligned}
&\max \quad -\Sigma_i \Sigma_j x_{ij} c_{ij} = f(\mathcal{T}) \\
&s.t \quad \Sigma_i x_{ij} = 1 \quad \forall i : (i, j) \in A \\
&\Sigma_i x_{ji} = 1 \quad \forall i : (j, i) \in A \\
&\Sigma_{i \in S, j \in V \setminus S} \quad x_{ij} \geq 2 \quad \forall S \subset V, S \notin \{\emptyset, V\} \\
&x_{ij} \in \{0, 1\}
\end{aligned}
$$

where S is a proper subset of the vertices V, i.e. it is connected to one of the remaining nodes (translated through the third constraint) and $\mathcal{T} = \{e_1, ..., e_N\}$ is a tour, where $\forall i, e_i \in A$. The first two constraints indicate that every city must be visited at least once and be left to another city. The third constraint indicates that each subset of cities is connected to another subset, which prevents inner cycle to form within the tour.

NEORL script

```
#-------------------------------------------------------
# Import Packages
#-------------------------------------------------------
from neorl.benchmarks import TSP
from neorl import PPO2, DQN, ACER, ACKTR, A2C
from neorl import MlpPolicy, DQNPolicy
from neorl import RLLogger
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import pandas as pd
import sys

#-------------------------------------------------------
# TSP Data
#-------------------------------------------------------
def TSP_Data(n_city):
    """
    Function provides initial data to construct a TSP enviroment

    :param n_city: (int) number of cities, choose either 51 or 100
    :return: city_loc_list (list), optimum_tour_city (list), episode_length (int)
    """
    if n_city == 51:
        #---51 cities
        #locations
        city_loc_list = [[37,52],[49,49],[52,64],[20,26],[40,30],[21,47],[17,63],[31,
↪62],[52,33],[51,21],[42,41],[31,32],[5,25]\
                         ,[12, 42],[36, 16],[52, 41],[27, 23],[17, 33],[13, 13],[57, 58],
↪[62, 42],[42, 57],[16, 57],[8 ,52],[7 ,38],[27, 68],[30, 48]\
                         ,[43, 67],[58, 48],[58, 27],[37, 69],[38, 46],[46, 10],[61, 33],
↪[62, 63],[63, 69],[32, 22],[45, 35],[59, 15],[5 ,6],[10, 17]\
                         ,[21, 10],[5 ,64],[30, 15],[39, 10],[32, 39],[25, 32],[25, 55],
↪[48, 28],[56, 37],[30, 40]]
        #optimal solution for comparison
        optimum_tour_city = [1,22,8,26,31,28,3,36,35,20,2,29,21,16,50,34,30,9,49,10,
↪39,33,45,15,44,42,40,19,41,13,25,14,24,43,7,23,48\
                             ,6,27,51,46,12,47,18,4,17,37,5,38,11,32]
        #episode length
        episode_length = 2

    elif n_city == 100:

        #---100 cities
        city_loc_list = [[-47,2],[49,-21 ],[35,-47 ],[30,-47 ],[-39,-50] ,[-35,-27],[-
↪34,9 ],[-11,-8 ],[32,-44 ],[ 1,35 ],[ 36,37 ]\
                         ,[ 12,37 ],[ 37,36 ],[ -26,-8],[ -21,32],[ -29,13],[ 26,-50],
↪[ -7,-36],[ -34,-2],[ 21,-40],[ -25,46],[ -17,8 ],[ 21,27 ],[ -31,-14]\
                         ,[ -15,-44],[ -33,-34],[ -49,45],[ -40,-1],[ -40,-33],[ -39,-
↪26],[ -17,-16],[ 17,-20],[ 4,-11 ],[ 22,34 ],[ 28,24 ],[ -39,37]\
                         ,[ 25,4 ],[ -35,14],[ 34,-5 ],[ 49,-43],[ 34,-29],[ -4,-50],[␣
↪0,-14 ],[ 48,-25],[ -50,-5],[ -26,0 ],[ -13,21],[ -6,-41],[ 40,-33]\
                         ,[ 12,-48],[ -38,16],[ -26,-38],[ -42,16],[ 13,8 ],[ 4,-8 ],[␣
↪-46,-20],[ -25,36],[ 22,21 ],[ 43,-5 ],[ -24,0 ],[ -12,-32],[ 47, 49 ]\
                         ,[ 31,-35],[ 42,13 ],[ -45,-45],[ -48,-14],[ 28,23 ],[ 23,-
↪43],[ 30,-25],[ 25,34 ],[ -7,32 ],[ -48,42],[ 1,-26 ],[ -45,32],[-20,35]\
```

(continues on next page)

(continued from previous page)

```
                        ,[ -12,21],[ -41,-49],[ -35,32],[ -43,44],[ -43,47],[ 27,20 ],
↪[ -8,-9 ],[ 37,-11],[ -18,16],[ -41,43],[ -30,29],[ -31,-19],[48,22 ]\
                        ,[ -45,-19],[ -15,30],[ 10,-8 ],[ 40,-33],[ 20,20 ],[ -22,33],
↪[ 42,-37],[ 0,-8 ],[ -50,11],[ 37,-27],[ 39,-43],[-7,32]]
        #optimal solution for comparison
        optimum_tour_city = [1,97,53,51,38,16,7,28,19,46,60,22,84,76,47,86,78,36,74,
↪72,27,80,79,85,21,57,94,15,75,90,71,100,10,12,34\
                            ,70,11,13,62,88,64,81,67,35,23,58,93,54,37,39,83,59,2,44,
↪98,41,69,63,49,92,95,40,99,3,9,4,17,68,20,50,42,25,48,18,61,73,32,91,55\
                            ,33,43,96,82,8,31,14,24,87,6,26,52,5,77,65,29,30,89,56,66,
↪45]
        #episode length
        episode_length = 2

    else:
        raise ValueError('--error: n_city is not defined, either choose 51 or 100')

    return city_loc_list, optimum_tour_city, episode_length



#---------------------------------------------------------
# User Parameters for RL Optimisation
#---------------------------------------------------------
try:
    total_steps=int(sys.argv[1])  #get time steps as external argument (for quick␣
↪testing)
except:
    total_steps=500      #or use default total time steps to run all optimizers

n_steps=12        #update frequency for A2C, ACKTR, PPO
n_city=51         #number of cities: choose 51 or 100

#---get some data to initialize the enviroment---
city_locs,optimum_tour,episode_length=TSP_Data(n_city=n_city)
#---------------------------------------------------------
# DQN
#---------------------------------------------------------
#create an enviroment object from the class
env=TSP(city_loc_list=city_locs, optimum_tour_city=optimum_tour,
        episode_length=episode_length, method = 'dqn')
#create a callback function to log data
cb_dqn=RLLogger(check_freq=n_city)
#To activate logger plotter, add following arguments to cb_dqn:
#plot_freq = 51,n_avg_steps=10,pngname='DQN-reward'
#Also applicable to ACER.

#create a RL object based on the env object
dqn = DQN(DQNPolicy, env=env, seed=1)
#optimise the enviroment class
dqn.learn(total_timesteps=total_steps*n_city, callback=cb_dqn)
#---------------------------------------------------------
# ACER
#---------------------------------------------------------
env=TSP(city_loc_list=city_locs, optimum_tour_city=optimum_tour,
        episode_length=episode_length, method = 'acer')
cb_acer=RLLogger(check_freq=n_city)
```

(continues on next page)

(continued from previous page)

```
acer = ACER(MlpPolicy, env=env, seed=1)
acer.learn(total_timesteps=total_steps*n_city, callback=cb_acer)
#----------------------------------------------------------
# PPO
#----------------------------------------------------------
env=TSP(city_loc_list=city_locs, optimum_tour_city=optimum_tour,
        episode_length=episode_length, method = 'ppo')
cb_ppo=RLLogger(check_freq=1)
#To activate logger plotter, add following arguments to cb_ppo:
#plot_freq = 1, n_avg_steps=10, pngname='PPO-reward'
#Also applicable to A2C, ACKTR.
ppo = PPO2(MlpPolicy, env=env, n_steps=n_steps, seed = 1)
ppo.learn(total_timesteps=total_steps, callback=cb_ppo)
#----------------------------------------------------------
# ACKTR
#----------------------------------------------------------
env=TSP(city_loc_list=city_locs, optimum_tour_city=optimum_tour,
        episode_length=episode_length, method = 'acktr')
cb_acktr=RLLogger(check_freq=1)
acktr = ACKTR(MlpPolicy, env=env, n_steps=n_steps, seed = 1)
acktr.learn(total_timesteps=total_steps, callback=cb_acktr)
#----------------------------------------------------------
# A2C
#----------------------------------------------------------
env=TSP(city_loc_list=city_locs, optimum_tour_city=optimum_tour,
        episode_length=episode_length, method = 'a2c')
cb_a2c=RLLogger(check_freq=1)
a2c = A2C(MlpPolicy, env=env, n_steps=n_steps, seed = 1)
a2c.learn(total_timesteps=total_steps, callback=cb_a2c)

#--------------------------------
#Summary Results
#--------------------------------
print('--------------- DQN results ---------------')
print('The best value of x found:', cb_dqn.xbest)
print('The best value of y found:', cb_dqn.rbest)
print('--------------- ACER results ---------------')
print('The best value of x found:', cb_acer.xbest)
print('The best value of y found:', cb_acer.rbest)
print('--------------- PPO results ---------------')
print('The best value of x found:', cb_ppo.xbest)
print('The best value of y found:', cb_ppo.rbest)
print('--------------- ACKTR results ---------------')
print('The best value of x found:', cb_acktr.xbest)
print('The best value of y found:', cb_acktr.rbest)
print('--------------- A2C results ---------------')
print('The best value of x found:', cb_a2c.xbest)
print('The best value of y found:', cb_a2c.rbest)

#--------------------------------
#Summary Plots
#--------------------------------
log_dqn = pd.DataFrame(cb_dqn.r_hist).cummax(axis = 0).values
log_acer = pd.DataFrame(cb_acer.r_hist).cummax(axis = 0).values
log_ppo = pd.DataFrame(cb_ppo.r_hist).cummax(axis = 0).values
log_acktr = pd.DataFrame(cb_acktr.r_hist).cummax(axis = 0).values
log_a2c = pd.DataFrame(cb_a2c.r_hist).cummax(axis = 0).values
```

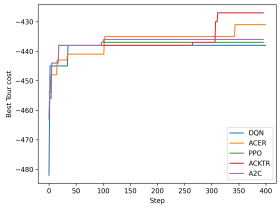
(continues on next page)

(continued from previous page)

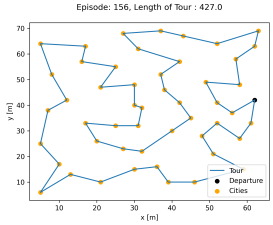
```
plt.figure()
plt.plot(log_dqn, label = "DQN")
plt.plot(log_acer, label = "ACER")
plt.plot(log_ppo, label = "PPO")
plt.plot(log_acktr, label = "ACKTR")
plt.plot(log_a2c, label = "A2C")
plt.xlabel('Step')
plt.ylabel('Best Tour Cost')
plt.legend()
plt.savefig("tsp_history.png",format='png' ,dpi=300, bbox_inches="tight")
plt.show()
```

Results

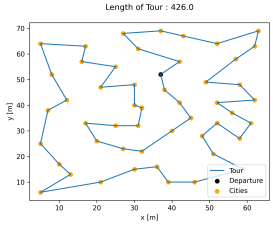
A summary of the results is shown below for the case of **51 cities**. First, all five reinforcement learning algorithms are compared in terms of minimizing the tour length (notice the y-axis is multiplied by -1 to make it a maximization problem). The cost convergence shows that ACKTR is the best algorithm in this case. Therefore, we will limit the reported results to ACKTR.



The best tour cost found by ACKTR is 427, which is really close to the optimal tour of 426. **ACKTR tour** is below



while here is the target **optimal** tour



And here are the final results of all algorithms:

```
--------------- DQN results ---------------
The best value of x found: ['36', '7', '8', '9', '10', '47', '41', '11', '43', '44',
↪'45', '46', '48', '21', '50', '51', '12', '37', '42', '24', '17', '27', '25', '14',
↪'30', '31', '32', '33', '34', '35', '38', '39', '40', '49', '4', '15', '2', '1', '3
↪', '5', '23', '20', '26', '19', '13', '22', '16', '18', '6', '28', '29']
```

(continues on next page)

(continued from previous page)

```
The best value of y found: -438
--------------- ACER results ---------------
The best value of x found: ['35', '3', '12', '44', '23', '49', '19', '26', '18', '32',
↪ '33', '45', '21', '28', '15', '30', '38', '9', '46', '17', '42', '14', '37', '48',
↪'39', '47', '1', '41', '43', '25', '11', '31', '20', '34', '16', '5', '24', '7', '51
↪', '50', '27', '4', '2', '6', '29', '36', '10', '13', '8', '40', '22']
The best value of y found: -431.0
--------------- PPO results ---------------
The best value of x found: ['51', '8', '27', '42', '35', '11', '14', '20', '17', '29',
↪ '40', '38', '49', '50', '41', '34', '5', '36', '21', '13', '45', '37', '26', '1',
↪'19', '46', '22', '28', '2', '43', '30', '31', '3', '47', '15', '24', '4', '7', '9',
↪ '10', '48', '12', '25', '18', '32', '33', '44', '16', '23', '39', '6']
The best value of y found: -437.0
--------------- ACKTR results ---------------
The best value of x found: ['50', '37', '45', '49', '35', '42', '40', '4', '38', '25',
↪ '43', '1', '48', '16', '44', '13', '5', '28', '34', '39', '33', '12', '31', '24',
↪'14', '22', '7', '27', '19', '18', '6', '46', '32', '8', '23', '2', '51', '15', '17
↪', '11', '30', '29', '10', '26', '41', '47', '21', '9', '3', '36', '20']
The best value of y found: -427.0
--------------- A2C results ---------------
The best value of x found: ['47', '5', '14', '39', '34', '13', '35', '41', '28', '33',
↪ '46', '24', '19', '4', '22', '8', '43', '38', '1', '44', '23', '32', '15', '16',
↪'48', '45', '42', '10', '12', '36', '27', '17', '9', '21', '7', '30', '25', '26',
↪'37', '29', '18', '31', '2', '11', '20', '6', '49', '40', '51', '50', '3']
The best value of y found: -436.0
```

2.4.2 Example 2: Ackley with EA

Example of solving the popular continuous optimization function "Ackley" using NEORL evolutionary algorithms.

Summary

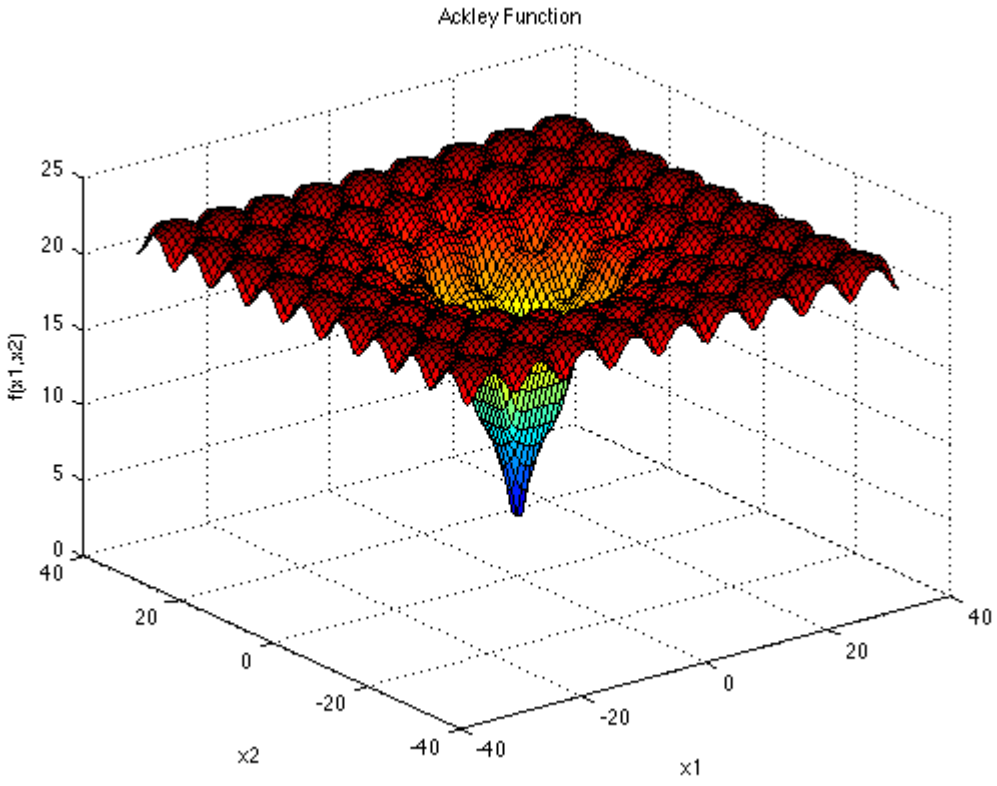
- Algorithms: PSO, XNES, DE
- Type: Continuous, Single-objective, Unconstrained
- Field: Mathematical Optimization

Problem Description

The mathematical definition of Ackley is:

$$f(\vec{x}) = 20 - 20exp\Big(-0.2\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\Big) - exp\Big(\frac{1}{d}\sum_{i=1}^{d} cos(2\pi x_i)\Big) + exp(1)$$

The Ackley function is continuous, non-convex and multimodal. This plot shows Ackley in two-dimensional ($d = 2$) form.



$\vec{x}$ domain: The function is usually evaluated on the hypercube $x_i \in [-32, 32]$, for all $i = 1, \ldots, d$. The global minima for the Ackley function is:

$$f(\vec{x}^*) = 0, \text{ at } \vec{x}^* = [0, 0, ..., 0]$$

NEORL script

The solution below is for a 8-dimensional Ackley function ($d = 8$)

```
#-------------------------------
# Import packages
#-------------------------------
import numpy as np
import matplotlib.pyplot as plt
from neorl import PSO, DE, XNES
from math import exp, sqrt, cos, pi
np.random.seed(50)

#-------------------------------
# Fitness function
#-------------------------------
def ACKLEY(individual):
    #Ackley objective function.
    d = len(individual)
    f=20 - 20 * exp(-0.2*sqrt(1.0/d * sum(x**2 for x in individual))) \
            + exp(1) - exp(1.0/d * sum(cos(2*pi*x) for x in individual))
    return f
```

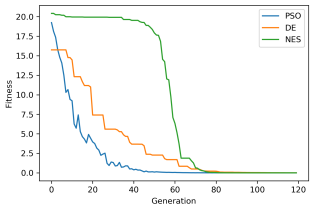
(continues on next page)

(continued from previous page)

```
#----------------------------------
# Parameter Space
#----------------------------------
#Setup the parameter space (d=8)
d=8
lb=-32
ub=32
BOUNDS={}
for i in range(1,d+1):
    BOUNDS['x'+str(i)]=['float', lb, ub]

#----------------------------------
# PSO
#----------------------------------
pso=PSO(mode='min', bounds=BOUNDS, fit=ACKLEY, npar=60,
        c1=2.05, c2=2.1, speed_mech='constric', seed=1)
x_best, y_best, pso_hist=pso.evolute(ngen=120, verbose=1)

#----------------------------------
# DE
#----------------------------------
de=DE(mode='min', bounds=BOUNDS, fit=ACKLEY, npop=60,
      F=0.5, CR=0.7, ncores=1, seed=1)
x_best, y_best, de_hist=de.evolute(ngen=120, verbose=1)

#----------------------------------
# NES
#----------------------------------
amat = np.eye(d)
xnes = XNES(mode='min', fit=ACKLEY, bounds=BOUNDS, A=amat, npop=60,
            eta_Bmat=0.04, eta_sigma=0.1, adapt_sampling=True, ncores=1, seed=1)
x_best, y_best, nes_hist=xnes.evolute(120, verbose=1)

#----------------------------------
# Plot
#----------------------------------
#Plot fitness for both methods
plt.figure()
plt.plot(pso_hist['global_fitness'], label='PSO')
plt.plot(de_hist['global_fitness'], label='DE')
plt.plot(nes_hist['fitness'], label='NES')
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.legend()
plt.savefig('ex2_fitness.png',format='png', dpi=300, bbox_inches="tight")
plt.show()
```

Results

Result summary is below for the three methods in minimizing the Ackley function.




```
------------------------ PSO Summary --------------------------
Best fitness (y) found: 6.384158766614689e-05
Best individual (x) found: [-1.1202021943594622e-05, 1.3222010570577733e-05, -1.
↪0037727362601807e-05, 9.389429054206202e-06, 2.4880207036828872e-05, 1.
↪6872593760849828e-05,    2.076883222303575e-05, 1.458529398292857e-05]
-------------------------------------------------------------
------------------------ DE Summary --------------------------
Best fitness (y) found: 0.0067943767106268815
Best individual (x) found: [-0.0025073247154970765, 0.0020192971595931735, -0.
↪0015127342773181872, -0.0010888556350037238, -0.0015830291353966849, -0.
↪00743962941194097, 0.0002963358699222367, 0.002260054765774109]
-------------------------------------------------------------
------------------------ NES Summary --------------------------
Best fitness (y) found: 1.5121439047582896e-06
Best individual (x) found: [ 5.01688814e-07 -1.12353966e-07  7.64184537e-08  1.
↪37674119e-08
  3.66277722e-07 -5.94627000e-07  3.11206449e-08 -6.19858494e-07]
-------------------------------------------------------------
```

2.4.3 Example 3: Welded-beam design

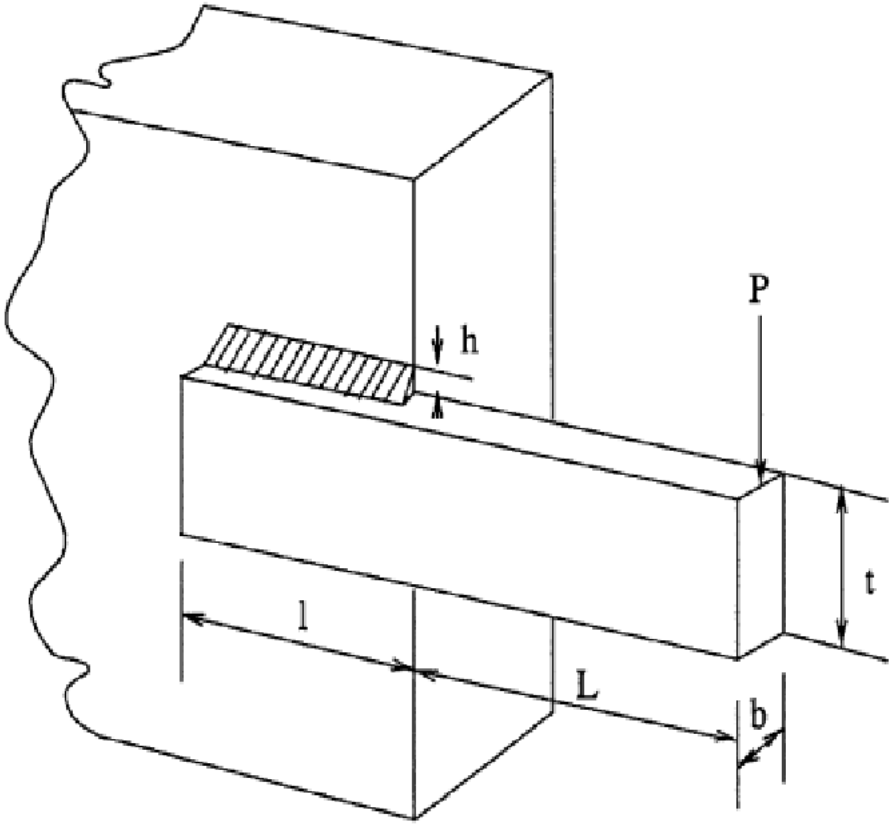
Example of solving the heavily-constrained engineering optimization problem "Welded-beam design" using NEORL with the ES algorithm tuned with Bayesian search.

Summary

- Algorithms: ES, Bayesian search for tuning
- Type: Continuous, Single-objective, Constrained
- Field: Structural Engineering

Problem Description

The welded beam is a common engineering optimisation problem with an objective to find an optimal set of the dimensions $h = x_1$, $l = x_2$, $t = x_3$, and $b = x_4$ such that the fabrication cost of the beam is minimized. This problem is a continuous optimisation problem. See the Figure below for graphical details of the beam dimensions (h, l, t, b) to be optimised.



The cost of the welded beam is formulated as

$$\min_{\vec{x}} f(\vec{x}) = 1.10471 x_1^2 x_2 + 0.04811 x_3 x_4 (14 + x_2),$$

subject to 7 rules/constraints, the first on the shear stress (τ)

$$g_1(\vec{x}) = \tau(\vec{x}) - \tau_{max} \leq 0,$$

the second on the bending stress (σ)

$$g_2(\vec{x}) = \sigma(\vec{x}) - \sigma_{max} \leq 0,$$

three side constraints

$$g_3(\vec{x}) = x_1 - x_4 \leq 0,$$

$$g_4(\vec{x}) = 0.10471 x_1^2 + 0.04811 x_3 x_4 (14 + x_2) - 5 \leq 0,$$

$$g_5(\vec{x}) = 0.125 - x_1 \leq 0,$$

the sixth on the end deflection of the beam (δ)

$$g_6(\vec{x}) = \delta(\vec{x}) - \delta_{max} \leq 0,$$

and the last on the buckling load on the bar (P_c)

$$g_7(\vec{x}) = P - P_c(\vec{x}) \leq 0,$$

while the range of the design variables are:

$$0.1 \leq x_1 \leq 2, \quad 0.1 \leq x_2 \leq 10,$$
$$0.1 \leq x_3 \leq 10, \quad 0.1 \leq x_4 \leq 2.$$

The derived variables and their related constants are expressed as follows:

$$\tau(\vec{x}) = \sqrt{(\tau')^2 + 2\tau'\tau''\frac{x_2}{2R} + (\tau'')^2},$$

$$\tau' = \frac{P}{\sqrt{2}x_1x_2}, \tau'' = \frac{MR}{J}, M = P(L + x_2/2),$$

$$R = \sqrt{\frac{x_2^2}{4} + \frac{(x_1 + x_3)^2}{4}},$$

$$J = 2\left[\sqrt{2}x_1x_2\left(\frac{x_2^2}{12} + \frac{(x_1 + x_3)^2}{4}\right)\right],$$

$$\sigma(\vec{x}) = \frac{6PL}{x_4x_3^2},$$

$$\delta(\vec{x}) = \frac{4PL^3}{Ex_3^3x_4},$$

$$P_c(\vec{x}) = \frac{4.013E\sqrt{\frac{x_3^2x_4^6}{36}}}{L^2}\left(1 - \frac{x_3}{2L}\sqrt{\frac{E}{4G}}\right),$$

$$P = 6000 \text{ lb}, L = 14 \text{ in}, E = 30 \times 10^6 \text{ psi},$$
$$G = 12 \times 10^6 \text{ psi},$$
$$\tau_{max} = 13,600 \text{ psi}, \sigma_{max} = 30,000 \text{ psi}, \delta_{max} = 0.25 \text{ in}$$

NEORL script

```
#----------------------------------
# Import packages
#----------------------------------
import numpy as np
np.random.seed(50)
import matplotlib.pyplot as plt
from math import sqrt
from neorl.tune import BAYESTUNE
from neorl import ES

#************************************************************
# Part I: Original Problem
#************************************************************
#Define the fitness function (for the welded beam)
def BEAM(x):

    y = 1.10471*x[0]**2*x[1]+0.04811*x[2]*x[3]*(14.0+x[1])
```

(continues on next page)

(continued from previous page)

```
    # parameters
    P = 6000; L = 14; E = 30e+6; G = 12e+6;
    t_max = 13600; s_max = 30000; d_max = 0.25;

    M = P*(L+x[1]/2)
    R = sqrt(0.25*(x[1]**2+(x[0]+x[2])**2))
    J = 2*(sqrt(2)*x[0]*x[1]*(x[1]**2/12+0.25*(x[0]+x[2])**2));
    P_c = (4.013*E/(6*L**2))*x[2]*x[3]**3*(1-0.25*x[2]*sqrt(E/G)/L);
    t1 = P/(sqrt(2)*x[0]*x[1]); t2 = M*R/J;
    t = sqrt(t1**2+t1*t2*x[1]/R+t2**2);
    s = 6*P*L/(x[3]*x[2]**2)
    d = 4*P*L**3/(E*x[3]*x[2]**3);
    # Constraints
    g1 = t-t_max; #done
    g2 = s-s_max; #done
    g3 = x[0]-x[3];
    g4 = 0.10471*x[0]**2+0.04811*x[2]*x[3]*(14.0+x[1])-5.0;
    g5 = 0.125-x[0];
    g6 = d-d_max;
    g7 = P-P_c; #done

    g=[g1,g2,g3,g4,g5,g6,g7]
    g_round=np.round(np.array(g),6)
    w1=100
    w2=100

    phi=sum(max(item,0) for item in g_round)
    viol=sum(float(num) > 0 for num in g_round)

    reward = (y + (w1*phi + w2*viol))

    return reward

#************************************************************
# Part II: Setup parameter space
#************************************************************
#--setup the parameter space for the welded beam
lb=[0.1, 0.1, 0.1, 0.1]
ub=[2.0, 10, 10, 2.0]
d2type=['float', 'float', 'float', 'float']
BOUNDS={}
nx=4
for i in range(nx):
    BOUNDS['x'+str(i+1)]=[d2type[i], lb[i], ub[i]]

#************************************************************
# Part III: Define fitness function for hyperparameter tuning
#************************************************************
def tune_fit(cxpb, mu, alpha, cxmode, mutpb):

    #--setup the ES algorithm
    es=ES(mode='min', bounds=BOUNDS, fit=BEAM, lambda_=80, mu=mu, mutpb=mutpb,␣
↪alpha=alpha,
          cxmode=cxmode, cxpb=cxpb, ncores=1, seed=1)

    #--Evolute the ES object and obtains y_best
```

(continues on next page)

(continued from previous page)

```
    #--turn off verbose for less algorithm print-out when tuning
    x_best, y_best, es_hist=es.evolute(ngen=100, verbose=0)

    return y_best #returns the best score

#=======================================================
# Part IV: Tuning
#=======================================================
#Setup the parameter space for Bayesian optimisation
#VERY IMPORTANT: The order of these parameters MUST be similar to their order in tune_
↪fit
#see tune_fit
param_grid={
#def tune_fit(cxpb, mu, alpha, cxmode):
'cxpb': ['float', 0.1, 0.7],             #cxpb is first (low=0.1, high=0.8,␣
↪type=float/continuous)
'mu':   ['int', 30, 60],                 #mu is second (low=30, high=60, type=int/
↪discrete)
'alpha':['grid', [0.1, 0.2, 0.3, 0.4]],    #alpha is third (grid with fixed values,␣
↪type=grid/categorical)
'cxmode':['grid', ['blend', 'cx2point']],
'mutpb': ['float', 0.05, 0.3]}  #cxmode is fourth (grid with fixed values, type=grid/
↪categorical)

#setup a bayesian tune object
btune=BAYESTUNE(param_grid=param_grid, fit=tune_fit, ncases=30)
#tune the parameters with method .tune
bayesres=btune.tune(ncores=1, csvname='bayestune.csv', verbose=True)

print('----Top 10 hyperparameter sets----')
bayesres = bayesres[bayesres['score'] >= 1] #drop the cases with scores < 1 (violates␣
↪the constraints)
bayesres = bayesres.sort_values(['score'], axis='index', ascending=True) #rank the␣
↪scores from best (lowest) to worst (high)
print(bayesres.iloc[0:10,:])   #the results are saved in dataframe and ranked from␣
↪best to worst

#=======================================================
# Part V: Rerun ES with the best hyperparameter set
#=======================================================
es=ES(mode='min', bounds=BOUNDS, fit=BEAM, lambda_=80, mu=bayesres['mu'].iloc[0],
      mutpb=bayesres['mutpb'].iloc[0], alpha=bayesres['alpha'].iloc[0],
      cxmode=bayesres['cxmode'].iloc[0], cxpb=bayesres['cxpb'].iloc[0],
      ncores=1, seed=1)

x_best, y_best, es_hist=es.evolute(ngen=100, verbose=0)

print('Best fitness (y) found:', y_best)
print('Best individual (x) found:', x_best)

#---------------------------------
# Plot
#---------------------------------
#Plot fitness convergence
plt.figure()
plt.plot(es_hist['local_fitness'], label='ES')
plt.xlabel('Generation')
```

(continues on next page)

(continued from previous page)

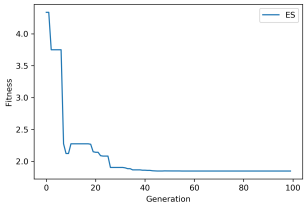
```
plt.ylabel('Fitness')
plt.legend()
plt.savefig('ex3_fitness.png',format='png', dpi=300, bbox_inches="tight")
plt.show()
```

Results

After Bayesian hyperparameter tuning, the top 10 are

```
----Top 10 hyperparameter sets----
        cxpb  mu  alpha     cxmode     mutpb     score
id
13  0.140799  35    0.3      blend  0.110994  1.849573
18  0.139643  37    0.3      blend  0.094496  1.925569
25  0.341248  39    0.1  cx2point  0.197213  2.098090
1   0.177505  32    0.3      blend  0.088050  2.144512
20  0.100000  35    0.3      blend  0.104131  2.198990
22  0.218197  30    0.3      blend  0.114197  2.228448
17  0.364451  34    0.3      blend  0.102634  2.235059
24  0.145365  42    0.3      blend  0.200532  2.292646
19  0.100000  55    0.3      blend  0.104209  2.349494
6   0.573142  38    0.4  cx2point  0.223231  2.349795
```

After re-running the problem with the best hyperparameter set, the convergence of the fitness function is shown below



while the best $\vec{x}(x_1 - x_4)$ and $y = f(x)$ (minimum beam cost) are:

```
Best fitness (y) found: 1.849572817626747
Best individual (x) found: [0.18756483308730693, 4.053366828472939, 8.731994883504612,
↪ 0.2231022567643955]
```

2.4.4 Example 4: Benchmarks

Example of accessing and plotting the built-in benchmark functions in NEORL.

Summary

- Algorithms: No Algorithms
- Type: Continuous, Single-objective
- Field: Benchmarking, Mathematical Optimization

Problem Description

We present an overview of how to access the benchmark functions in NEORL to allow easy testing of different algorithms within the framework. For example, a long list of classical mathematical functions is saved within NEORL such as the Rosenbrock function

$$f(\vec{x}) = \sum_{i=1}^{d-1}(1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2$$

or Ackley function

$$f(\vec{x}) = 20 - 20exp\Big(-0.2\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\Big) - exp\Big(\frac{1}{d}\sum_{i=1}^{d} cos(2\pi x_i)\Big) + exp(1)$$

or Bohachevsky function

$$f(\vec{x}) = \sum_{i=1}^{d-1}(x_i^2 + 2x_{i+1}^2 - 0.3\cos(3\pi x_i) - 0.4\cos(4\pi x_{i+1}) + 0.7)$$

More recent and advanced benchmark functions were also developed such as the CEC'2017 test suite, which includes 30 benchmark functions divided into three groups: Simple (f1-f10), Hybrid (f11-f20), and Composition (f21-f30). The core of the CEC'2017 functions is similar to the classical functions listed above. However, CEC'2017 functions are rotated and shifted to make them more complex to optimise. **CEC'2017 functions are only defined at specific dimensions, namely,** `d=2, 10, 20, 30, 50 or 100`.

NEORL script

```
import numpy as np
import neorl.benchmarks.cec17 as functions    #import all cec17 functions
import neorl.benchmarks.classic as classics   #import all classical functions
from neorl.benchmarks.classic import ackley, levy, bohachevsky  #import specific␣
↪functions
from neorl.benchmarks.cec17 import f3, f10, f21  #import cec17 specific functions
from neorl.benchmarks import bench_2dplot   #import the built-in plotter

d1 = 2 #set dimension for classical functions
d2 = 10 #set dimension for cec functions (choose between 2, 10, 20, 30, 50 or 100)
print('-------------------------------------------------------')
print('Classical Functions')
print('-------------------------------------------------------')
```

(continues on next page)

(continued from previous page)

```
for f in classics.all_functions:
    sample = np.random.uniform(low=0, high=10, size=d1)
    y = f(sample)
    print('Function: {}, x={}, y={}'.format(f.__name__, np.round(sample,2), np.
↪round(y,2)))

print('-------------------------------------------------------')
print('CEC2017 Functions')
print('-------------------------------------------------------')
for f in functions.all_functions:
    sample = np.random.uniform(low=-10, high=10, size=d2)
    y = f(sample)
    print('Function: {}, x={}, y={}'.format(f.__name__, np.round(sample,2), np.
↪round(y,2)))

print('-------------------------------------------------------')
print('Function Plotter')
print('-------------------------------------------------------')
bench_2dplot(f3, domain=(-50,50), points=60)
bench_2dplot(f10, savepng='ex4_f10.png')
bench_2dplot(f21, savepng='ex4_f21.png')

bench_2dplot(ackley, savepng='ex4_ackley.png')
bench_2dplot(levy, domain=(-10,10))
bench_2dplot(bohachevsky, points=50)

#-------------------------------------------------------------------------------
#NOTE: CEC'17 functions: f11-f20, f29, f30 are not defined for d=2 dimensions,
#so the plotter will FAIL for these functions
#-------------------------------------------------------------------------------
```

Results

Selected results from the script output are shown below, which indicates the function evaluation with a random sample

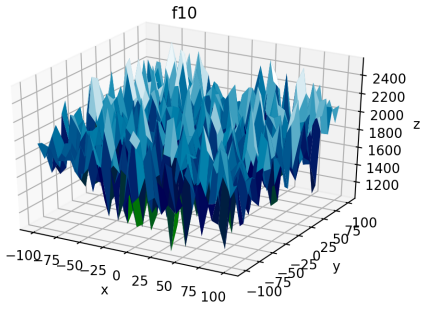
```
-------------------------------------------------------
Classical Functions
-------------------------------------------------------
Function: sphere, x=[4.17 7.2 ], y=69.28
Function: cigar, x=[0.   3.02], y=9140499.76
Function: rosenbrock, x=[1.47 0.92], y=151.59
Function: bohachevsky, x=[1.86 3.46], y=27.63
Function: griewank, x=[3.97 5.39], y=0.48
Function: rastrigin, x=[4.19 6.85], y=74.97
Function: ackley, x=[2.04 8.78], y=15.35
.
.
.

-------------------------------------------------------
CEC2017 Functions
-------------------------------------------------------
Function: f1, x=[ 3.89 -1.72 -9.    0.72  3.28  0.3   8.89  1.73  8.07 -7.25],␣
↪y=276294024527.33
```

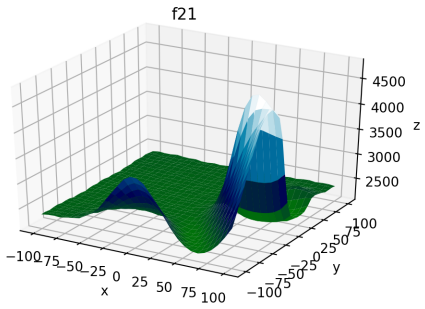
(continues on next page)

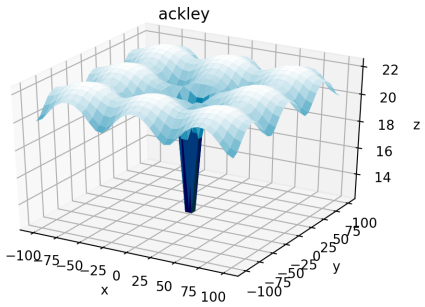
(continued from previous page)

```
Function: f2, x=[-7.21  6.15 -2.05 -6.69  8.55 -3.04  5.02  4.52  7.67  2.47], y=1.
↪39057e+18
Function: f3, x=[ 5.02 -3.02 -4.6   7.92 -1.44  9.3   3.27  2.43 -7.71  8.99],␣
↪y=104499271.6
Function: f4, x=[-1.    1.57 -1.84 -5.26  8.07  1.47 -9.94  2.34 -3.47  0.54], y=6142.
↪43
Function: f5, x=[ 7.72 -2.85  8.17  2.47 -9.68  8.59  3.82  9.95 -6.55 -7.26], y=738.
↪82
Function: f6, x=[ 8.65  3.94 -8.68  5.11  5.08  8.46  4.23 -7.51 -9.6  -9.48], y=817.
↪43
Function: f7, x=[-9.43 -5.08  7.2   0.78  1.06  6.84 -7.52 -4.42  1.72  9.39], y=916.
↪24
.
.
.
-------------------------------------------------------
Function Plotter
-------------------------------------------------------
.
.
.
```

Few samples from the function plots are shown below







2.4.5 Example 5: CEC'2017 Test Suite

Example of solving the first 10 benchmark functions in CEC'2017 test suite with dimensionality `d=2` using Differential Evolution.

Summary

- Algorithms: DE
- Type: Continuous, Single-objective
- Field: Benchmarking, Mathematical Optimization

Problem Description

The following notes are appliable to all test functions in CEC'2017

- All test functions in CEC'2017 are shifted by the vector $\vec{o}$.
- Search range for all functions is $[-100, 100]^d$.
- $\mathbf{M}_i$ is the rotation matrix. Different rotation matrices are assigned to each function in the test suite.
- The shifted and rotated function is defined as $F_i = f_i(\mathbf{M}(\vec{x} - \vec{o})) + F_i^*$

Where $f(.)$ is the base function obtained from the classical functions group (e.g. Zakharov, Cigar, Rosenbrock). The list of the functions in CEC'2017 is shown below based on the reference indicated at the table bottom.

Type	No.	Function	$F^* = F(x^*)$
Unimodal Functions	1	Shifted and Rotated Bent Cigar Function	100
	2	Shifted and Rotated Sum of Different Power Function	200
	3	Shifted and Rotated ZakharovFunction	300
Multimodal Functions	4	Shifted and Rotated Rosenbrock'sFunction	400
	5	Shifted and Rotated Rastrigin'sFunction	500
	6	Shifted and Rotated Expanded Schaffer's Function	600
	7	Shifted and Rotated Lunacek Bi_Rastrigin Function	700
	8	Shifted and Rotated Non-Continuous Rastrigin'sFunction	800
	9	Shifted and Rotated Levy Function	900
	10	Shifted and Rotated Schwefel'sFunction	1000
Hybrid Functions	11	Hybrid Function 1 (N=3)	1100
	12	Hybrid Function 2 (N=3)	1200
	13	Hybrid Function 3 (N=3)	1300
	14	Hybrid Function 4 (N=4)	1400
	15	Hybrid Function 5 (N=4)	1500
	16	Hybrid Function 6 (N=4)	1600
	17	Hybrid Function 7 (N=5)	1700
	18	Hybrid Function 8 (N=5)	1800
	19	Hybrid Function 9 (N=5)	1900
	20	Hybrid Function 10 (N=6)	2000
Composition Functions	21	Composition Function 1 (N=3)	2100
	22	Composition Function 2 (N=3)	2200
	23	Composition Function 3 (N=4)	2300
	24	Composition Function 4 (N=4)	2400
	25	Composition Function 5 (N=5)	2500
	26	Composition Function 6 (N=5)	2600
	27	Composition Function 7 (N=6)	2700
	28	Composition Function 8 (N=6)	2800
	29	Composition Function 9 (N=9)	2900
	30	Composition Function 10 (N=3)	3000

NEORL script

```
import numpy as np
import neorl.benchmarks.cec17 as functions    #import all cec17 functions
from neorl import DE

reduced_func=functions.all_functions[:10] #keep only the first 10 functions

nx = 2 #set dimension
BOUNDS={}
for i in range(1,nx+1):
    BOUNDS['x'+str(i)]=['float', -100, 100]

for FIT in reduced_func:
    #setup and evolute PSO
    de=DE(mode='min', bounds=BOUNDS, fit=FIT, npop=60, F=0.5,
          CR=0.7, ncores=1, seed=1)
    x_best, y_best, de_hist=de.evolute(ngen=100, verbose=0)
    opt=float(FIT.__name__.strip('f'))*100
    print('Function: {}, x-DE={}, y-DE={}, y-Optimal={}'.format(FIT.__name__,
                                                               np.round(x_best,2),
                                                               np.round(y_best,2),
                                                               opt))
```

Results

After running the script above, the output looks like below, which shows that DE was able to converge to the optimal value for all functions, of course, because the problem is simple with `d=2` dimensions.

```
Function: f1, x-DE=[-55.28 -70.43], y-DE=100.0, y-Optimal=100.0
Function: f2, x-DE=[-29.34 -17.05], y-DE=200.0, y-Optimal=200.0
Function: f3, x-DE=[-55.94   4.54], y-DE=300.0, y-Optimal=300.0
Function: f4, x-DE=[32.51  7.76], y-DE=400.0, y-Optimal=400.0
Function: f5, x-DE=[-17.41  56.17], y-DE=500.0, y-Optimal=500.0
Function: f6, x-DE=[ 79.09 -24.57], y-DE=600.0, y-Optimal=600.0
Function: f7, x-DE=[-46.65  42.28], y-DE=700.32, y-Optimal=700.0
Function: f8, x-DE=[ 32.16 -55.97], y-DE=800.0, y-Optimal=800.0
Function: f9, x-DE=[-24.48   2.3 ], y-DE=900.0, y-Optimal=900.0
Function: f10, x-DE=[-15.95 -59.89], y-DE=1000.33, y-Optimal=1000.0
```

2.4.6 Example 6: Three-bar Truss Design

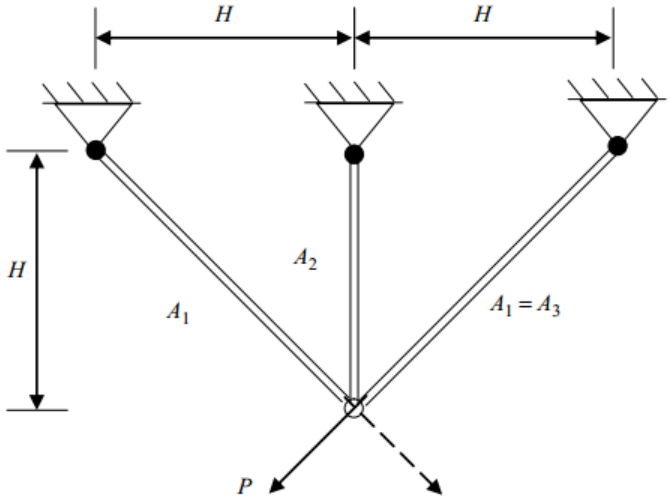
Example of solving the constrained engineering optimization problem "Three-bar truss design" using NEORL with the BAT, GWO, and MFO algorithms.

Summary

- Algorithms: BAT, GWO, MFO
- Type: Continuous, Single-objective, Constrained
- Field: Structural Engineering

Problem Description

The Three-bar truss design is an engineering optimization problem with the objective to evaluate the optimal cross sectional areas $A_1 = A_3 = x_1$ and $A_2 = x_2$ such that the volume of the statically loaded truss structure is minimized accounting for stress (σ) constraints. The figure below shows the dimensions of the three-bar truss structure.



The equation for the volume of the truss structure is

$$\min_{\vec{x}} f(\vec{x}) = (2\sqrt{2}x_1 + x_2) \times H,$$

subject to 3 constraints

$$g_1 = \frac{\sqrt{2}x_1 + x_2}{\sqrt{2}x_1^2 + 2x_1x_2}P - \sigma \leq 0,$$
$$g_2 = \frac{x_2}{\sqrt{2}x_1^2 + 2x_1x_2}P - \sigma \leq 0,$$
$$g_3 = \frac{1}{x_1 + \sqrt{2}x_2}P - \sigma \leq 0,$$

where $0 \leq x_1 \leq 1$, $0 \leq x_2 \leq 1$, $H = 100cm$, $P = 2KN/cm^2$, and $\sigma = 2KN/cm^2$.

NEORL script

```
#--------------------------------
# Import packages
#--------------------------------
import numpy as np
from math import cos, pi, exp, e, sqrt
import matplotlib.pyplot as plt
from neorl import BAT, GWO, MFO

#--------------------------------
# Fitness function
#--------------------------------
def TBT(individual):
    """Three-bar truss Design
    """
    x1 = individual[0]
    x2 = individual[1]

    y = (2*sqrt(2)*x1 + x2) * 100

    #Constraints
    if x1 <= 0:
            g = [1,1,1]
    else:
            g1 = (sqrt(2)*x1+x2)/(sqrt(2)*x1**2 + 2*x1*x2) * 2 - 2
            g2 = x2/(sqrt(2)*x1**2 + 2*x1*x2) * 2 - 2
            g3 = 1/(x1 + sqrt(2)*x2) * 2 - 2
            g = [g1,g2,g3]

    g_round=np.round(np.array(g),6)
    w1=100
    w2=100

    phi=sum(max(item,0) for item in g_round)
    viol=sum(float(num) > 0 for num in g_round)

    return y + w1*phi + w2*viol
#--------------------------------
# Parameter space
#--------------------------------
nx = 2
BOUNDS = {}
for i in range(1, nx+1):
    BOUNDS['x'+str(i)]=['float', 0, 1]


#--------------------------------
# BAT
#--------------------------------
bat=BAT(mode='min', bounds=BOUNDS, fit=TBT, nbats=10, fmin = 0 , fmax = 1, A=0.5,␣
↪r0=0.5, levy = True, seed = 1, ncores=1)
bat_x, bat_y, bat_hist=bat.evolute(ngen=100, verbose=1)

#--------------------------------
# GWO
#--------------------------------
```

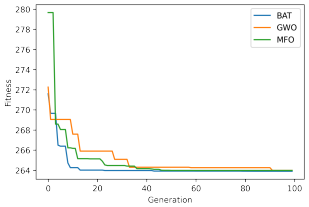
(continues on next page)

(continued from previous page)

```
gwo=GWO(mode='min', fit=TBT, bounds=BOUNDS, nwolves=10, ncores=1, seed=1)
gwo_x, gwo_y, gwo_hist=gwo.evolute(ngen=100, verbose=1)

#-----------------------------------
# MFO
#-----------------------------------
mfo=MFO(mode='min', bounds=BOUNDS, fit=TBT, nmoths=10, b = 0.2, ncores=1, seed=1)
mfo_x, mfo_y, mfo_hist=mfo.evolute(ngen=100, verbose=1)

#-----------------------------------
# Plot
#-----------------------------------
plt.figure()
plt.plot(bat_hist['global_fitness'], label = 'BAT')
plt.plot(gwo_hist['fitness'], label = 'GWO')
plt.plot(mfo_hist['global_fitness'], label = 'MFO')
plt.legend()
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.savefig('TBT_fitness.png',format='png', dpi=300, bbox_inches="tight")
plt.show()
```

Results

A summary of the results for the three differents methods is shown below with the best (x_1, x_2) and $y = f(x)$ (minimum volume).




```
------------------------ BAT Summary --------------------------
Best fitness (y) found: 263.90446934840577
Best individual (x) found: [0.79190302 0.39920471]
---------------------------------------------------------------
------------------------ GWO Summary --------------------------
Best fitness (y) found: 263.99180199625886
Best individual (x) found: [0.78831222 0.41023435]
---------------------------------------------------------------
------------------------ MFO Summary --------------------------
Best fitness (y) found: 263.9847325242824
Best individual (x) found: [0.77788022 0.4396698 ]
---------------------------------------------------------------
```

2.4.7 Example 7: Pressure Vessel Design

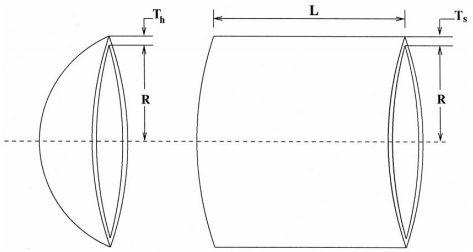
Example of solving the constrained engineering optimization problem "Pressure vessel design" using NEORL with the HHO, ES, PESA, and BAT algorithms to demonstrate compatibility with mixed discrete-continuous space.

Summary

- Algorithms: HHO, ES, PESA, and BAT
- Type: Mixed discrete-continuous, Single-objective, Constrained
- Field: Mechanical Engineering

Problem Description

The pressure vessel design is an engineering optimization problem with the objective to evaluate the optimal thickness of shell ($T_s = x_1$), thickness of head ($T_h = x_2$), inner radius ($R = x_3$), and length of shell ($L = x_4$) such that the total cost of material, forming, and welding is minimized accounting for 4 constraints. T_s and T_h are integer multiples of 0.0625 in., which are the available thicknesses of rolled steel plates, and R and L are continuous. The figure below shows the dimensions of the pressure vessel structure.



The equation for the cost of the pressure vessel is

$$\min_{\vec{x}} f(\vec{x}) = 0.6224x_1x_3x_4 + 1.7781x_2x_3^2 + 3.1661x_1^2x_4 + 19.84x_1^2x_3,$$

subject to 4 constraints

$$\begin{aligned} g_1 &= -x_1 + 0.0193x_3 \leq 0, \\ g_2 &= -x_2 + 0.00954x_3 \leq 0, \\ g_3 &= -\pi x_3^2 x_4 - \frac{4}{3}\pi x_3^3 + 1296000 \leq 0, \\ g_4 &= x_4 - 240 \leq 0, \end{aligned}$$

where $0.0625 \leq x_1 \leq 6.1875$ (with step of 0.0625), $0.0625 \leq x_2 \leq 6.1875$ (with step of 0.0625), $10 \leq x_3 \leq 200$, and $10 \leq x_4 \leq 200$.

NEORL script

```
#########################
# Import Packages
#########################

from neorl import HHO, ES, PESA, BAT
import math
import matplotlib.pyplot as plt

############################
# Define Vessel Function
#(Mixed discrete/continuous)
############################
def Vessel(individual):
    """
    Pressure vesssel design
    x1: thickness (d1)  --> discrete value multiple of 0.0625 in
    x2: thickness of the heads (d1) ---> discrete value multiple of 0.0625 in
    x3: inner radius (r)  ---> cont. value between [10, 200]
    x4: length (L)  ---> cont. value between [10, 200]
    """
    x=individual.copy()
    x[0] *= 0.0625   #convert d1 to "in"
    x[1] *= 0.0625   #convert d2 to "in"

    y = 0.6224*x[0]*x[2]*x[3]+1.7781*x[1]*x[2]**2+3.1661*x[0]**2*x[3]+19.
↪84*x[0]**2*x[2];

    g1 = -x[0]+0.0193*x[2];
    g2 = -x[1]+0.00954*x[2];
    g3 = -math.pi*x[2]**2*x[3]-(4/3)*math.pi*x[2]**3 + 1296000;
    g4 = x[3]-240;
    g=[g1,g2,g3,g4]

    phi=sum(max(item,0) for item in g)
    eps=1e-5 #tolerance to escape the constraint region
    penality=1e7 #large penality to add if constraints are violated

    if phi > eps:
        fitness=phi+penality
    else:
        fitness=y
    return fitness


bounds = {}
bounds['x1'] = ['int', 1, 99]
bounds['x2'] = ['int', 1, 99]
bounds['x3'] = ['float', 10, 200]
bounds['x4'] = ['float', 10, 200]

#########################
# Setup and evolute HHO
#########################

hho = HHO(mode='min', bounds=bounds, fit=Vessel, nhawks=30,
```

(continues on next page)

(continued from previous page)

```
                  int_transform='minmax', ncores=1, seed=1)
x_hho, y_hho, hho_hist=hho.evolute(ngen=200, verbose=False)
assert Vessel(x_hho) == y_hho

########################
# Setup and evolute ES
########################
es = ES(mode='min', fit=Vessel, cxmode='cx2point', bounds=bounds,
                lambda_=60, mu=30, cxpb=0.7, mutpb=0.2, seed=1)
x_es, y_es, es_hist=es.evolute(ngen=200, verbose=False)
assert Vessel(x_es) == y_es

########################
# Setup and evolute PESA
########################
pesa=PESA(mode='min', bounds=bounds, fit=Vessel, npop=60, mu=30, alpha_init=0.1,
          alpha_end=1.0, cxpb=0.7, mutpb=0.2, alpha_backdoor=0.15, seed=1)
x_pesa, y_pesa, pesa_hist=pesa.evolute(ngen=200, verbose=False)
assert Vessel(x_pesa) == y_pesa

########################
# Setup and evolute BAT
########################
bat=BAT(mode='min', bounds=bounds, fit=Vessel, nbats=50, fmin = 0 , fmax = 1,
        A=0.5, r0=0.5, levy = True, seed = 1, ncores=1)
x_bat, y_bat, bat_hist=bat.evolute(ngen=200, verbose=1)
assert Vessel(x_bat) == y_bat

########################
# Plotting
########################

plt.figure()
plt.plot(hho_hist['global_fitness'], label='HHO')
plt.plot(es_hist['global_fitness'], label='ES')
plt.plot(pesa_hist, label='PESA')
plt.plot(bat_hist['global_fitness'], label='BAT')
plt.xlabel('Generation')
plt.ylabel('Fitness')
#plt.ylim([0,10000]) #zoom in
plt.legend()
plt.savefig('ex7_pv_fitness.png',format='png', dpi=300, bbox_inches="tight")
plt.show()

########################
# Comparison
########################

print('---Best HHO Results---')
print(x_hho)
print(y_hho)
print('---Best ES Results---')
print(x_es)
print(y_es)
print('---Best PESA Results---')
print(x_pesa)
print(y_pesa)
```

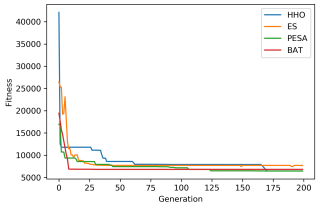
(continues on next page)

(continued from previous page)

```
print('---Best BAT Results---')
print(x_bat)
print(y_bat)
```

Results

A summary of the results is shown below with the best (x_1, x_2, x_3, x_4) and $y = f(x)$ (minimum vessel cost). PESA seems to be the best algorithm in this case.




```
------------------------ HHO Summary --------------------------
Function: Vessel
Best fitness (y) found: 6450.086928941204
Best individual (x) found: [16.          8.         51.38667573 87.7107088 ]
---------------------------------------------------------------
------------------------ ES Summary --------------------------
Best fitness (y) found: 7440.247037114203
Best individual (x) found: [19, 10, 59.20709018618041, 39.15211859223507]
---------------------------------------------------------------
------------------------ PESA Summary --------------------------
Best fitness (y) found: 6446.821261696037
Best individual (x) found: [16, 8, 51.45490215425688, 87.29635265232538]
---------------------------------------------------------------
------------------------ BAT Summary --------------------------
Best fitness (y) found: 6820.372175171242
Best individual (x) found: [18.          9.         58.29066654 43.68984579]
---------------------------------------------------------------
```

2.4.8 Example 8: Pressure Vessel Design with Demonstration of Categorical Parameter

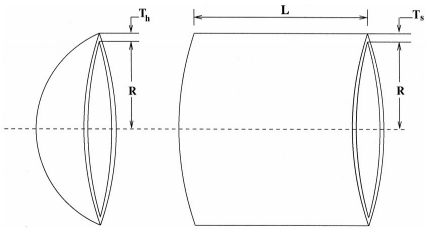
Example of solving the constrained engineering optimization problem "Pressure vessel design" using NEORL with HHO, ES, PESA, and BAT algorithms to demonstrate compatibility with mix of continuous, discrete, and categorical spaces.

Summary

- Algorithms: HHO, ES, PESA, BAT
- Type: Mixed discrete-continuous-categorical, Single-objective, Constrained
- Field: Mechanical Engineering

Problem Description

The pressure vessel design is an engineering optimization problem with the objective to evaluate the optimal thickness of shell ($T_s = x_1$), thickness of head ($T_h = x_2$), inner radius ($R = x_3$), and length of shell ($L = x_4$) such that the total cost of material, forming, and welding is minimized accounting for 4 constraints. T_s and T_h are integer multiples of 0.0625 in., which are the available thicknesses of rolled steel plates, and R and L are continuous. Unlike *Example 7*, T_h will be modeled as a categorical parameter to demonstrate compatibility with a mix of continuous, discrete, and categorical parameters. The figure below shows the dimensions of the pressure vessel structure.



The equation for the cost of the pressure vessel is

$$\min_{\vec{x}} f(\vec{x}) = 0.6224x_1x_3x_4 + 1.7781x_2x_3^2 + 3.1661x_1^2x_4 + 19.84x_1^2x_3,$$

subject to 4 constraints

$$\begin{aligned} g_1 &= -x_1 + 0.0193x_3 \leq 0, \\ g_2 &= -x_2 + 0.00954x_3 \leq 0, \\ g_3 &= -\pi x_3^2 x_4 - \frac{4}{3}\pi x_3^3 + 1296000 \leq 0, \\ g_4 &= x_4 - 240 \leq 0, \end{aligned}$$

where $0.0625 \leq x_1 \leq 6.1875$ (with step of 0.0625), $x_2 \in \{0.0625, 0.125, 0.1875, 0.25, 0.3125, 0.375, 0.4375, 0.5, 0.5625, 0.625\}$, $10 \leq x_3 \leq 200$, and $10 \leq x_4 \leq 200$.

NEORL script

```
#########################
# Import Packages
#########################

from neorl import HHO, ES, PESA, BAT
import math
import matplotlib.pyplot as plt

#################################
# Define Vessel Function
#Mixed discrete/continuous/grid
```

(continues on next page)

(continued from previous page)

```
#################################
def Vessel(individual):
    """
    Pressure vesssel design
    x1: thickness (d1)  --> discrete value multiple of 0.0625 in
    x2: thickness of the heads (d2) ---> categorical value from a pre-defined grid
    x3: inner radius (r)  ---> cont. value between [10, 200]
    x4: length (L)  ---> cont. value between [10, 200]
    """

    x=individual.copy()
    x[0] *= 0.0625   #convert d1 to "in"

    y = 0.6224*x[0]*x[2]*x[3]+1.7781*x[1]*x[2]**2+3.1661*x[0]**2*x[3]+19.
↪84*x[0]**2*x[2];

    g1 = -x[0]+0.0193*x[2];
    g2 = -x[1]+0.00954*x[2];
    g3 = -math.pi*x[2]**2*x[3]-(4/3)*math.pi*x[2]**3 + 1296000;
    g4 = x[3]-240;
    g=[g1,g2,g3,g4]

    phi=sum(max(item,0) for item in g)
    eps=1e-5 #tolerance to escape the constraint region
    penality=1e7 #large penality to add if constraints are violated

    if phi > eps:
        fitness=phi+penality
    else:
        fitness=y
    return fitness


bounds = {}
bounds['x1'] = ['int', 1, 99]
bounds['x2'] = ['grid', (0.0625, 0.125, 0.1875, 0.25, 0.3125, 0.375, 0.4375, 0.5, 0.
↪5625, 0.625)]
bounds['x3'] = ['float', 10, 200]
bounds['x4'] = ['float', 10, 200]

########################
# Setup and evolute HHO
########################
hho = HHO(mode='min', bounds=bounds, fit=Vessel, nhawks=50,
                  int_transform='minmax', ncores=1, seed=1)
x_hho, y_hho, hho_hist=hho.evolute(ngen=200, verbose=False)
assert Vessel(x_hho) == y_hho

########################
# Setup and evolute ES
########################
es = ES(mode='min', fit=Vessel, cxmode='cx2point', bounds=bounds,
                lambda_=60, mu=30, cxpb=0.7, mutpb=0.2, seed=1)
x_es, y_es, es_hist=es.evolute(ngen=200, verbose=False)
assert Vessel(x_es) == y_es

########################
```

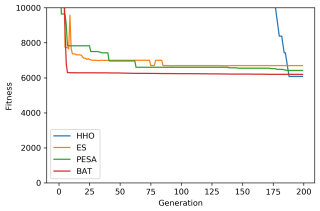
(continues on next page)

(continued from previous page)

```
# Setup and evolute PESA
########################
pesa=PESA(mode='min', bounds=bounds, fit=Vessel, npop=60, mu=30, alpha_init=0.01,
          alpha_end=1.0, cxpb=0.7, mutpb=0.2, alpha_backdoor=0.05)
x_pesa, y_pesa, pesa_hist=pesa.evolute(ngen=200, verbose=False)
assert Vessel(x_pesa) == y_pesa

########################
# Setup and evolute BAT
########################
bat=BAT(mode='min', bounds=bounds, fit=Vessel, nbats=50, fmin = 0 , fmax = 1,
        A=0.5, r0=0.5, levy = True, seed = 1, ncores=1)
x_bat, y_bat, bat_hist=bat.evolute(ngen=200, verbose=1)
assert Vessel(x_bat) == y_bat

########################
# Plotting
########################

plt.figure()
plt.plot(hho_hist['global_fitness'], label='HHO')
plt.plot(es_hist['global_fitness'], label='ES')
plt.plot(pesa_hist, label='PESA')
plt.plot(bat_hist['global_fitness'], label='BAT')
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.ylim([0,10000]) #zoom in
plt.legend()
plt.savefig('ex8_pv_fitness.png',format='png', dpi=300, bbox_inches="tight")
plt.show()

########################
# Comparison
########################

print('---Best HHO Results---')
print(x_hho)
print(y_hho)
print('---Best ES Results---')
print(x_es)
print(y_es)
print('---Best PESA Results---')
print(x_pesa)
print(y_pesa)
print('---Best BAT Results---')
print(x_bat)
print(y_bat)
```

Results

A summary of the results is shown below with the best (x_1, x_2, x_3, x_4) and $y = f(x)$ (minimum vessel cost). The figure is zoomed to a relevant y-scale that shows all methods clearly. HHO is best algorithm for this case.




```
------------------------ HHO Summary --------------------------
Function: Vessel
Best fitness (y) found: 6076.871989481831
Best individual (x) found: [13.0, 0.4375, 41.958165787313035, 178.38267571773872]
---------------------------------------------------------------
------------------------ ES Summary ---------------------------
Best fitness (y) found: 6689.115350860009
Best individual (x) found: [17, 0.5, 52.39036909796362, 80.46789374601103]
---------------------------------------------------------------
------------------------ PESA Summary -------------------------
Best fitness (y) found: 6420.244320020875
Best individual (x) found: [15, 0.5, 48.11672433151982, 114.1606860286298]
---------------------------------------------------------------
------------------------ BAT Summary --------------------------
Best fitness (y) found: 6194.304291280144
Best individual (x) found: [13.0, 0.4375, 41.040436491185176, 190.26719495938994]
---------------------------------------------------------------
```

2.4.9 Example 9: Cantilever Stepped Beam

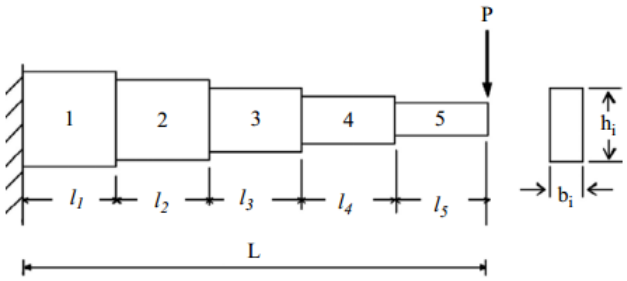
Example of solving the constrained engineering optimization problem "Cantilever Stepped Beam" as well as a simpler square case using NEORL with the PSO, DE, SSA, GWO, MFO, BAT, and PESA2 algorithms.

Summary

- Algorithms: PSO, DE, SSA, GWO, MFO, BAT, PESA2
- Type: Continuous, Single-objective, Constrained
- Field: Structural Engineering

Problem Description

The cantilever stepped beam design is an engineering optimization problem with the objective to evaluate the optimal heights $x_1 - x_5$ and widths $x_6 - x_{10}$ of the five rectangular segments such that the volume of the statically loaded cantilever beam is minimized accounting for stress (σ) constraints. The figure below shows the dimensions of the cantilever structure



The equation for the volume of the cantilever beam is

$$\min_{\vec{x}} f(\vec{x}) = \sum_{i=1}^{5} x_i x_{i+5} l_i,$$

where l_i = 100 cm. The design is subject to 11 constraints:

$$g_1 = \frac{Pl_5}{x_5 x_{10}^2} - 14000 \leq 0,$$
$$g_2 = \frac{P(l_5 + l_4)}{x_4 x_9^2} - 14000 \leq 0,$$
$$g_3 = \frac{P(l_5 + l_4 + l_3)}{x_3 x_8^2} - 14000 \leq 0,$$
$$g_4 = \frac{P(l_5 + l_4 + l_3 + l_2)}{x_2 x_7^2} - 14000 \leq 0,$$
$$g_5 = \frac{P(l_5 + l_4 + l_3 + l_2 + l_1)}{x_1 x_6^2} - 14000 \leq 0,$$
$$g_6 = \frac{x_{10}}{x_5} - 20 \leq 0,$$
$$g_7 = \frac{x_9}{x_4} - 20 \leq 0,$$
$$g_8 = \frac{x_8}{x_3} - 20 \leq 0,$$
$$g_9 = \frac{x_7}{x_2} - 20 \leq 0,$$
$$g_{10} = \frac{x_6}{x_1} - 20 \leq 0,$$
$$g_{11} = \frac{Pl^3}{3E}\left(\frac{1}{I_5} + \frac{7}{I_4} + \frac{19}{I_3} + \frac{37}{I_2} + \frac{61}{I_1}\right) - 2.7 \leq 0,$$

where $1 \leq x_i \leq 5$ $(i = 1, 2, \ldots, 5)$, $30 \leq x_i \leq 65$ $(i = 6, 7, \ldots, 10)$, $P = 50,000N$, and $E = 2 \times 10^7 N/cm^2$.

NEORL script

```
#============================================================
#                Cantilever Stepped Beam
#============================================================

#---------------------------------
# Import packages
#---------------------------------
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
from neorl import PSO, DE, SSA, GWO, MFO, BAT, PESA2

#-----------------------------------
# Fitness function
#-----------------------------------
def CSB(individual):
  """Cantilever Stepped Beam
  individual[i = 0 - 4] are beam widths
  individual[i = 5 - 9] are beam heights
  """

  check=all([item >= BOUNDS['x'+str(i+1)][1] for i,item in enumerate(individual)]) \
               and all([item <= BOUNDS['x'+str(i+1)][2] for i,item in␣
↪enumerate(individual)])
  if not check:
          raise Exception ('--error check fails')

  P = 50000
  E = 2 * 10**7
  l = 100
  g = np.zeros(11)
  g[0] = 600*P/(individual[4] * individual[9]**2) - 14000
  g[1] = 6*P*(2*l)/(individual[3] * individual[8]**2) - 14000
  g[2] = 6*P*(3*l)/(individual[2] * individual[7]**2) - 14000
  g[3] = 6*P*(4*l)/(individual[1] * individual[6]**2) - 14000
  g[4] = 6*P*(5*l)/(individual[0] * individual[5]**2) - 14000
  g[5] = 0
  g[6] = individual[9]/individual[4] - 20
  g[7] = individual[8]/individual[3] - 20
  g[8] = individual[7]/individual[2] - 20
  g[9] = individual[6]/individual[1] - 20
  g[10] = individual[5]/individual[0] - 20

  g_round=np.round(g,6)
  w1=1000
  w2=1000

  phi=sum(max(item,0) for item in g_round)
  viol=sum(float(num) > 0 for num in g_round)

  V = 0
  for i in range(5):
          V += individual[i] * individual[i+5] * l

  return V + w1*phi + w2*viol

#-----------------------------------
# Parameter space
#-----------------------------------
nx=10
BOUNDS={}
for i in range(1, 6):
  BOUNDS['x'+str(i)]=['float', 1, 5]
for i in range(6, 11):
  BOUNDS['x'+str(i)]=['float', 30, 65]
```

(continues on next page)

(continued from previous page)

```
#----------------------------------
# PSO
#----------------------------------
pso=PSO(mode='min', bounds=BOUNDS, fit=CSB, c1=2.05, c2=2.1, npar=50, speed_mech=
↪'constric', ncores=1, seed=1)
pso_x, pso_y, pso_hist=pso.evolute(ngen=300, verbose=0)

#----------------------------------
# DE
#----------------------------------
de=DE(mode='min', bounds=BOUNDS, fit=CSB, npop=50, F=0.5, CR=0.7, ncores=1, seed=1)
de_x, de_y, de_hist=de.evolute(ngen=300, verbose=0)

#----------------------------------
# SSA
#----------------------------------
ssa=SSA(mode='min', bounds=BOUNDS, fit=CSB, nsalps=50, ncores=1, seed=1)
ssa_x, ssa_y, ssa_hist=ssa.evolute(ngen=300, verbose=0)

#----------------------------------
# BAT
#----------------------------------
bat=BAT(mode='min', bounds=BOUNDS, fit=CSB, nbats=50, fmin = 0 , fmax = 1, A=0.5,␣
↪r0=0.5, levy = True, seed = 1, ncores=1)
bat_x, bat_y, bat_hist=bat.evolute(ngen=300, verbose=0)

#----------------------------------
# GWO
#----------------------------------
gwo=GWO(mode='min', fit=CSB, bounds=BOUNDS, nwolves=50, ncores=1, seed=1)
gwo_x, gwo_y, gwo_hist=gwo.evolute(ngen=300, verbose=0)

#----------------------------------
# MFO
#----------------------------------
mfo=MFO(mode='min', bounds=BOUNDS, fit=CSB, nmoths=50, b = 0.2, ncores=1, seed=1)
mfo_x, mfo_y, mfo_hist=mfo.evolute(ngen=300, verbose=0)

#----------------------------------
# PESA2
#----------------------------------
pesa2=PESA2(mode='min', bounds=BOUNDS, fit=CSB, npop=50, nwolves=5, ncores=1, seed=1)
pesa2_x, pesa2_y, pesa2_hist=pesa2.evolute(ngen=600, replay_every=2, verbose=0)

#----------------------------------
# Plot
#----------------------------------
plt.figure()
plt.plot(pso_hist['global_fitness'], label = 'PSO')
plt.plot(de_hist['global_fitness'], label = 'DE')
plt.plot(ssa_hist['global_fitness'], label = 'SSA')
plt.plot(bat_hist['global_fitness'], label = 'BAT')
plt.plot(gwo_hist['fitness'], label = 'GWO')
plt.plot(mfo_hist['global_fitness'], label = 'MFO')
plt.plot(pesa2_hist, label = 'PESA2')
plt.legend()
plt.xlabel('Generation')
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('Fitness')
plt.ylim(0,150000)
plt.savefig('CSB_fitness.png',format='png', dpi=300, bbox_inches="tight")
plt.show()

#************************************************************
#              Square Cantilever Stepped Beam
#************************************************************

#---------------------------------
# Import packages
#---------------------------------
import numpy as np
from math import cos, pi, exp, e, sqrt
import matplotlib.pyplot as plt
from neorl import PSO, DE, SSA, GWO, MFO, BAT, PESA2

#---------------------------------
# Fitness function
#---------------------------------
def CSB_square(individual):
  """Square Cantilever Stepped Beam
  individual[i = 0 - 4] are beam heights and widths
  """

  check=all([item >= BOUNDS['x'+str(i+1)][1] for i,item in enumerate(individual)]) \
               and all([item <= BOUNDS['x'+str(i+1)][2] for i,item in␣
↪enumerate(individual)])
  if not check:
          raise Exception ('--error check fails')

  g = 61/individual[0]**3 + 37/individual[1]**3 + 19/individual[2]**3 + 7/
↪individual[3]**3 + 1/individual[4]**3 - 1

  g_round=np.round(g,6)
  w1=1000
  #phi=max(g_round,0)
  if g_round > 0:
          phi = 1
  else:
          phi = 0

  V = 0.0624*(np.sum(individual))

  return V + w1*phi

#---------------------------------
# Parameter space
#---------------------------------
nx=5
BOUNDS={}
for i in range(1, 6):
  BOUNDS['x'+str(i)]=['float', 0.01, 100]

#---------------------------------
# PSO
#---------------------------------
```

(continues on next page)

(continued from previous page)

```
pso=PSO(mode='min', bounds=BOUNDS, fit=CSB_square, c1=2.05, c2=2.1, npar=50, speed_
↪mech='constric', ncores=1, seed=1)
pso_x, pso_y, pso_hist=pso.evolute(ngen=200, verbose=0)

#----------------------------------
# DE
#----------------------------------
de=DE(mode='min', bounds=BOUNDS, fit=CSB_square, npop=50, F=0.5, CR=0.7, ncores=1,␣
↪seed=1)
de_x, de_y, de_hist=de.evolute(ngen=200, verbose=0)

#----------------------------------
# SSA
#----------------------------------
ssa=SSA(mode='min', bounds=BOUNDS, fit=CSB_square, nsalps=50, ncores=1, seed=1)
ssa_x, ssa_y, ssa_hist=ssa.evolute(ngen=200, verbose=0)

#----------------------------------
# BAT
#----------------------------------
bat=BAT(mode='min', bounds=BOUNDS, fit=CSB_square, nbats=50, fmin = 0 , fmax = 1, A=0.
↪5, r0=0.5, levy = True, seed = 1, ncores=1)
bat_x, bat_y, bat_hist=bat.evolute(ngen=200, verbose=0)

#----------------------------------
# GWO
#----------------------------------
gwo=GWO(mode='min', bounds=BOUNDS, fit=CSB_square, nwolves=50, ncores=1, seed=1)
gwo_x, gwo_y, gwo_hist=gwo.evolute(ngen=200, verbose=0)

#----------------------------------
# MFO
#----------------------------------
mfo=MFO(mode='min', bounds=BOUNDS, fit=CSB_square, nmoths=50, b = 0.2, ncores=1,␣
↪seed=1)
mfo_x, mfo_y, mfo_hist=mfo.evolute(ngen=200, verbose=0)

#----------------------------------
# PESA2
#----------------------------------
pesa2=PESA2(mode='min', bounds=BOUNDS, fit=CSB_square, npop=50, nwolves=5, ncores=1,␣
↪seed=1)
pesa2_x, pesa2_y, pesa2_hist=pesa2.evolute(ngen=400, replay_every=2, verbose=0)

#----------------------------------
# Plot
#----------------------------------
plt.figure()
plt.plot(pso_hist['global_fitness'], label = 'PSO')
plt.plot(de_hist['global_fitness'], label = 'DE')
plt.plot(ssa_hist['global_fitness'], label = 'SSA')
plt.plot(bat_hist['global_fitness'], label = 'BAT')
plt.plot(gwo_hist['fitness'], label = 'GWO')
plt.plot(mfo_hist['global_fitness'], label = 'MFO')
plt.plot(pesa2_hist, label = 'PESA2')
plt.legend()
plt.xlabel('Generation')
```

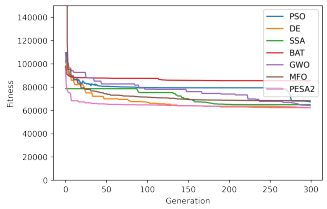
(continues on next page)

(continued from previous page)

```
plt.ylabel('Fitness')
plt.savefig('CSB_square_fitness.png',format='png', dpi=300, bbox_inches="tight")
plt.show()
```

Results

A summary of the results for the different methods is shown below with the best $\vec{x}$ and $y = f(x)$ (minimum volume). DE and PESA2 show the best results




```
      ------------------------ PSO Summary --------------------------
Best fitness (y) found: 67335.91306205148
Best individual (x) found: [3.5688543506518657, 2.7791479019993583, 2.525097748486896,
↪ 2.5931709214818746, 3.0184400437701497, 54.792949608125554, 55.55687861890972, 50.
↪45798494270985, 40.66285990348347, 30]
------------------------------------------------------------
      ------------------------ DE Summary --------------------------
Best fitness (y) found: 62253.08088965673
Best individual (x) found: [3.0024484908957634, 2.805060577916484, 2.5489794459150485,
↪ 2.2104364777790244, 1.7812440232731417, 59.83601779613085, 55.29672939889619, 50.
↪28650239860514, 44.1239666136573, 34.83693680366301]
------------------------------------------------------------
      ------------------------ SSA Summary --------------------------
Best fitness (y) found: 64883.21386055779
Best individual (x) found: [ 3.01217612  3.12691389  2.66584391  2.56789564  2.
↪26413108 59.64053204
 52.35626719 49.10660541 40.85291814 30.76422498]
------------------------------------------------------------
------------------------ BAT Summary --------------------------
Best fitness (y) found: 85509.99706478164
Best individual (x) found: [ 3.24944962  3.15901556  3.71034211  2.88044491  2.
↪72539593 57.42189701
 63.18019888 41.62463495 57.60433819 54.50783013]
------------------------------------------------------------
------------------------ GWO Summary --------------------------
Best fitness (y) found: 64217.56500668205
Best individual (x) found: [ 3.0490241   2.82805632  2.54882906  2.60647295  1.
↪91892183 59.32877115
 55.25800387 50.22171706 40.6816295  36.98217938]
------------------------------------------------------------
------------------------ MFO Summary --------------------------
Best fitness (y) found: 68284.66539072228
Best individual (x) found: [ 3.00200798  3.04903017  2.52470544  4.76307633  2.
↪38149254 59.7430926
 53.04777538 50.46763269 30.00000001 30.00007271]
------------------------------------------------------------
------------------------PESA2 Summary-------------------------
Best fitness (y) found: 62491.80715494685
```

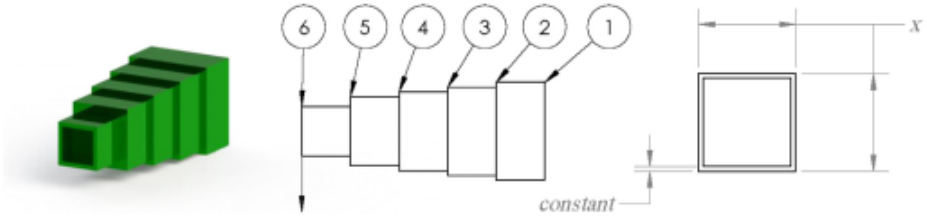
(continues on next page)

(continued from previous page)

```
Best individual (x) found: [3.0160894636962894, 2.7812536032709683, 2.
↪5788148288206765, 2.22105253054037, 1.9267356857359532, 59.60349505892154, 55.
↪515872598414006, 49.94170599870354, 43.93631599428571, 33.408629968533766]
--------------------------------------------------------------
```

Simple Square Case

A simpler case of the cantilever stepped beam design is shown in the figure below where the heights and widths of each section of the beam are equal ($x_i = x_{i+5}$) and the cantilever is hollow with constant thickness.



The equation for the weight of the square cantilever beam to be minimized is:

$$\min_{\vec{x}} f(\vec{x}) = 0.0624 \sum_{i=1}^{5} x_i,$$

with a single constraint

$$g_1 = \frac{61}{x_1^3} + \frac{37}{x_2^3} + \frac{19}{x_3^3} + \frac{7}{x_4^3} + \frac{1}{x_5^3} - 1 \leq 0,$$

where $0.001 \leq x_i \leq 100$

```
#-----------------------------------
# Import packages
#-----------------------------------
import numpy as np
from math import cos, pi, exp, e, sqrt
import matplotlib.pyplot as plt
from neorl import PSO, DE, SSA, GWO, MFO, BAT, PESA2

#-----------------------------------
# Fitness function
#-----------------------------------
def CSB_square(individual):
  """Square Cantilever Stepped Beam
  individual[i = 0 - 4] are beam heights and widths
  """

  check=all([item >= BOUNDS['x'+str(i+1)][1] for i,item in enumerate(individual)]) \
                and all([item <= BOUNDS['x'+str(i+1)][2] for i,item in␣
↪enumerate(individual)])
  if not check:
          raise Exception ('--error check fails')

  g = 61/individual[0]**3 + 37/individual[1]**3 + 19/individual[2]**3 + 7/
↪individual[3]**3 + 1/individual[4]**3 - 1
```

(continues on next page)

(continued from previous page)

```
  g_round=np.round(g,6)
  w1=1000
  #phi=max(g_round,0)
  if g_round > 0:
          phi = 1
  else:
          phi = 0

  V = 0.0624*(np.sum(individual))

  return V + w1*phi

#----------------------------------
# Parameter space
#----------------------------------
nx=5
BOUNDS={}
for i in range(1, 6):
  BOUNDS['x'+str(i)]=['float', 0.01, 100]

#----------------------------------
# PSO
#----------------------------------
pso=PSO(mode='min', bounds=BOUNDS, fit=CSB_square, c1=2.05, c2=2.1, npar=50, speed_
↪mech='constric', ncores=1, seed=1)
pso_x, pso_y, pso_hist=pso.evolute(ngen=200, verbose=0)

#----------------------------------
# DE
#----------------------------------
de=DE(mode='min', bounds=BOUNDS, fit=CSB_square, npop=50, F=0.5, CR=0.7, ncores=1,␣
↪seed=1)
de_x, de_y, de_hist=de.evolute(ngen=200, verbose=0)

#----------------------------------
# SSA
#----------------------------------
ssa=SSA(mode='min', bounds=BOUNDS, fit=CSB_square, nsalps=50, c1=0.05, ncores=1,␣
↪seed=1)
ssa_x, ssa_y, ssa_hist=ssa.evolute(ngen=200, verbose=0)

#----------------------------------
# BAT
#----------------------------------
bat=BAT(mode='min', bounds=BOUNDS, fit=CSB_square, nbats=50, fmin = 0 , fmax = 1, A=0.
↪5, r0=0.5, levy = True, seed = 1, ncores=1)
bat_x, bat_y, bat_hist=bat.evolute(ngen=200, verbose=0)

#----------------------------------
# GWO
#----------------------------------
gwo=GWO(mode='min', bounds=BOUNDS, fit=CSB_square, nwolves=50, ncores=1, seed=1)
gwo_x, gwo_y, gwo_hist=gwo.evolute(ngen=200, verbose=0)

#----------------------------------
# MFO
#----------------------------------
```

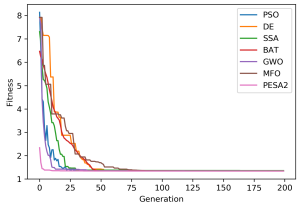
(continues on next page)

(continued from previous page)

```
mfo=MFO(mode='min', bounds=BOUNDS, fit=CSB_square, nmoths=50, b = 0.2, ncores=1,␣
↪seed=1)
mfo_x, mfo_y, mfo_hist=mfo.evolute(ngen=200, verbose=0)

#----------------------------------
# PESA2
#----------------------------------
pesa2=PESA2(mode='min', bounds=BOUNDS, fit=CSB_square, npop=50, nwolves=5, ncores=1,␣
↪seed=1)
pesa2_x, pesa2_y, pesa2_hist=pesa2.evolute(ngen=400, replay_every=2, verbose=0)

#----------------------------------
# Plot
#----------------------------------
plt.figure()
plt.plot(pso_hist, label = 'PSO')
plt.plot(de_hist, label = 'DE')
plt.plot(ssa_hist['global_fitness'], label = 'SSA')
plt.plot(bat_hist['global_fitness'], label = 'BAT')
plt.plot(gwo_hist['fitness'], label = 'GWO')
plt.plot(mfo_hist['global_fitness'], label = 'MFO')
plt.plot(pesa2_hist, label = 'PESA2')
plt.legend()
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.savefig('CSB_square_fitness.png',format='png', dpi=300, bbox_inches="tight")
```

Results

A summary of the results for the different methods is shown below with the best $\vec{x}$ and $y = f(x)$ (minimum volume). All methods seem to provide competitive performances.




```
------------------------ PSO Summary --------------------------
Best fitness (y) found: 1.3399752313729163
Best individual (x) found: [6.009251848152787, 5.30657072479825, 4.487735398237302, 3.
↪5223666630459225, 2.148037406998374]
---------------------------------------------------------------
------------------------ DE Summary --------------------------
Best fitness (y) found: 1.3400333309297923
Best individual (x) found: [6.009134740254158, 5.313241105010421, 4.465292783694765,␣
↪3.529049287512202, 2.1581752074033305]
---------------------------------------------------------------
------------------------ SSA Summary --------------------------
Best fitness (y) found: 1.3588143704307478
Best individual (x) found: [5.78402889 5.66053404 4.28387098 3.33467882 2.71275859]
---------------------------------------------------------------
------------------------ BAT Summary --------------------------
```

(continues on next page)

(continued from previous page)

```
Best fitness (y) found: 1.3399823246990532
Best individual (x) found: [6.00500481 5.29566351 4.51093598 3.49613637 2.16633504]
------------------------------------------------------------
----------------------- GWO Summary --------------------------
Best fitness (y) found: 1.3401094136761165
Best individual (x) found: [6.03840936 5.27194855 4.46957895 3.52048465 2.17569089]
------------------------------------------------------------
----------------------- MFO Summary --------------------------
Best fitness (y) found: 1.340072721543648
Best individual (x) found: [5.98884701 5.35435014 4.45583399 3.51864409 2.15784915]
------------------------------------------------------------
----------------------- PESA2 Summary ------------------------
Best fitness (y) found: 1.3399951046267742
Best individual (x) found: [6.0302855076764095, 5.324816195983351, 4.498012579038754,␣
↪3.474272116432333, 2.1468941237341275]
------------------------------------------------------------
```

2.4.10 Example 10: Knapsack Problem

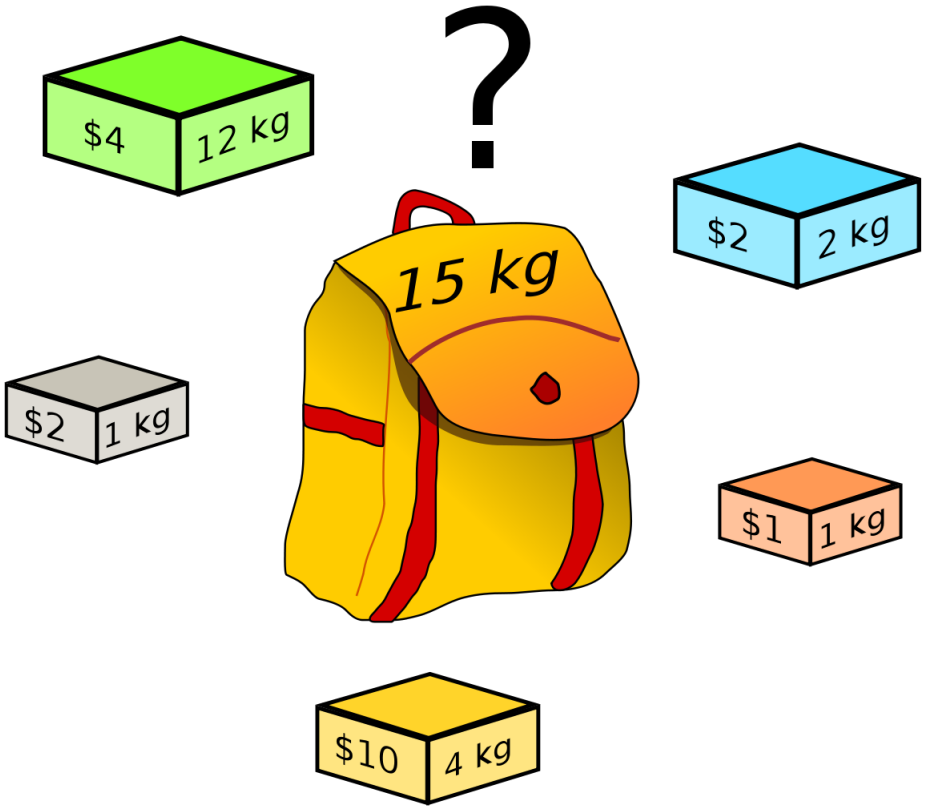
Example of solving the classical discrete optimization problem "Knapsack Problem" (KP) using NEORL with state-of-the-art reinforcement learning algorithms to demonstrate compatibility with discrete space.

Summary

- Algorithm: PPO,A2C, ACKTR, DQN, ACER
- Type: Discrete/Combinatorial, Single Objective, Constrained
- Field: Computational mathematics

Problem Description

The Knapsack Problem (KP) is another combinatorial problem that has been studied for decades. It often arises in resource allocation problem, machine scheduling, and asset optimization for instance. It consists in maximizing the value of a set of items placed in a bag limited by the weight capacity W of the bag. The figure below illustrates the problem. The maximum weight that the bag can contain is 15kg. The optimum set of items is obtained by taking all of them instead of the 12 kg one. A good heuristic is to choose the item with the lowest weight-to-value ratio.



Formally, with $i = 1, ..., n$ items, each characterized by their values and weights (ν_i, w_i), the problem can be formulated as:

$$\max \sum_{i=1}^{n} \nu_i x_i$$
$$s.t. \quad \sum_{i=1}^{n} w_i x_i \leq W$$
$$x_i \in \{0, 1\}$$

where $x_i = 1$ if the item i is in the bag, otherwise $x_i = 0$. This is called the 0-1 Knapsack Problem. The constraint characterizes the fact that the capacity of the bag is limited by W, hence the sum of the weights $\sum_{i=1}^{n} w_i x_i$ of the items should not exceed W.

NEORL script

```
#------------------------------------------------------
# Import Packages
#------------------------------------------------------
from neorl.benchmarks import KP
from neorl import PPO2, DQN, ACER, ACKTR, A2C
from neorl import MlpPolicy, DQNPolicy
from neorl import RLLogger
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import pandas as pd
```

(continues on next page)

(continued from previous page)

```
import numpy as np
import sys

#-----------------------------------------------------------
# KP Data
#-----------------------------------------------------------
def KP_Data(n_objects):
    """
    Function provides initial data to construct a Knapsack problem enviroment

    :param n_objects: (int) number of objects, choose either 50 or 100
    :return: obj_list (list), optimum_knapsack (list), episode_length (int), weight_
↪capacity (int)
    """
    if n_objects == 50:
        #---50 objects
        obj_list = [[3,4],[8,4],[4,2],[9,4],[5,9],[3,6],[3,1],[9,2],[8,3],[6,8],[9,4],
↪[4,2],[4,7],[5,1],[6,4],[5,8],[2,1],[5,7],[2,5],[7,4],\
                    [6,3],[8,2],[7,7],[4,8],[5,8],[2,1],[3,7],[7,4],[9,1],[1,4],[2,2],
↪[6,4],[7,3],[2,6],[7,3],[9,1],[1,1],[1,9],[2,3],[5,8],[5,1],[3,9],\
                    [5,6],[5,7],[4,2],[2,3],[1,4],[8,3],[7,5],[1,6]]
        #optimal solution for comparison
        optimum_knapsack = [1,2,3,4,7,8,9,10,11,12,14,15,16,17,18,20,21,22,23,25,26,
↪28,29,31,32,33,35,36,39,41,43,44,45,48,49]
        #episode length
        episode_length = 2
        weight_capacity = 125

    elif n_objects == 100:

        #---100 objects
        obj_list = [[1,4],[9,5],[9,7],[6,8],[3,7],[8,4],[8,6],[2,1],[2,6],[9,7],[8,2],
↪[6,6],[6,9],[6,7],[4,4],[7,8],[1,9],[1,3],[5,3],[8,1],\
                            [5,7],[8,6],[2,8],[3,5],[3,8],[4,3],[8,2],[6,7],[4,9],[3,
↪5],[9,1],[9,3],[5,6],[2,2],[2,1],[5,9],[6,2],[1,3],[8,3],[8,8],[3,8],[4,6],\
                            [4,7],[9,7],[9,4],[8,8],[2,7],[4,4],[1,2],[3,4],[8,8],[6,
↪9],[4,7],[6,8],[8,7],[4,8],[7,9],[5,9],[8,8],[5,4],[2,2],[4,9],[1,4],[1,8],\
                            [8,6],[4,5],[9,1],[3,1],[6,2],[7,1],[1,6],[1,7],[9,7],[7,
↪5],[7,1],[5,6],[3,5],[8,8],[8,9],[2,9],[3,1],[5,9],[7,8],[4,3],[2,8],[8,4],\
                            [9,5],[6,7],[8,2],[3,5],[2,6],[3,2],[9,7],[1,1],[6,7],[7,
↪4],[6,4],[7,6],[6,4],[3,2]]
        #optimal solution for comparison
        optimum_knapsack = [2,3,6,7,8,10,11,12,14,15,16,19,20,22,26,27,28,31,32,34,35,
↪37,39,40,44,45,46,48,51,55,59,60,61,65,66,67,68,69,\
                                70,73,74,75,78,79,81,83,84,86,87,89,92,93,94,96,
↪97,98,99,100]

        #episode length
        episode_length = 2
        weight_capacity= 250
    else:
        raise ValueError('--error: n_objects is not defined, either choose 50 or 100')

    return obj_list, optimum_knapsack, episode_length, weight_capacity

#-----------------------------------------------------------
# User Parameters for RL Optimisation
```

(continues on next page)

(continued from previous page)

```
#-------------------------------------------------------
try:
    total_steps=int(sys.argv[1])  #get time steps as external argument (for quick␣
↪testing)
except:
    total_steps=8000      #or use default total time steps to run all optimizers

n_steps=12          #update frequency for A2C, ACKTR, PPO
n_objects=50        #number of objects: choose 50 or 100
n_sum_steps=10      #this is for logging and averaging purposes

#---get some data to initialize the enviroment---
obj_list, optimum_knapsack, episode_length, weight_capacity=KP_Data(n_objects=n_
↪objects)
#-------------------------------------------------------
# DQN
#-------------------------------------------------------
#create an enviroment object from the class
env=KP(obj_list=obj_list, optimum_knapsack=optimum_knapsack,
        episode_length=episode_length, weight_capacity=weight_capacity, method = 'dqn
↪')
#create a callback function to log data
cb_dqn=RLLogger(check_freq=1)
#To activate logger plotter, add following arguments to cb_dqn:
#plot_freq = 50,n_avg_steps=10,pngname='DQN-reward'
#Also applicable to ACER.

#create a RL object based on the env object
dqn = DQN(DQNPolicy, env=env, seed=1)
#optimise the enviroment class
dqn.learn(total_timesteps=total_steps*n_sum_steps, callback=cb_dqn)
#-------------------------------------------------------
# ACER
#-------------------------------------------------------
env=KP(obj_list=obj_list, optimum_knapsack=optimum_knapsack,
        episode_length=episode_length, weight_capacity=weight_capacity, method = 'acer
↪')
cb_acer=RLLogger(check_freq=1)
acer = ACER(MlpPolicy, env=env, seed=1)
acer.learn(total_timesteps=total_steps*n_sum_steps, callback=cb_acer)
#-------------------------------------------------------
# PPO
#-------------------------------------------------------
env=KP(obj_list=obj_list, optimum_knapsack=optimum_knapsack,
        episode_length=episode_length, weight_capacity=weight_capacity, method = 'ppo
↪')
cb_ppo=RLLogger(check_freq=1)
#To activate logger plotter, add following arguments to cb_ppo:
#plot_freq = 1, n_avg_steps=10, pngname='PPO-reward'
#Also applicable to A2C, ACKTR.
ppo = PPO2(MlpPolicy, env=env, n_steps=n_steps, seed = 1)
ppo.learn(total_timesteps=total_steps, callback=cb_ppo)
#-------------------------------------------------------
# ACKTR
#-------------------------------------------------------
env=KP(obj_list=obj_list, optimum_knapsack=optimum_knapsack,
        episode_length=episode_length, weight_capacity=weight_capacity, method =
↪'acktr')
```

(continues on next page)

(continued from previous page)

```
cb_acktr=RLLogger(check_freq=1)
acktr = ACKTR(MlpPolicy, env=env, n_steps=n_steps, seed = 1)
acktr.learn(total_timesteps=total_steps, callback=cb_acktr)
#-----------------------------------------------------
# A2C
#-----------------------------------------------------
env=KP(obj_list=obj_list, optimum_knapsack=optimum_knapsack,
       episode_length=episode_length, weight_capacity=weight_capacity, method = 'a2c
↪')
cb_a2c=RLLogger(check_freq=1)
a2c = A2C(MlpPolicy, env=env, n_steps=n_steps, seed = 1)
a2c.learn(total_timesteps=total_steps, callback=cb_a2c)

#---------------------------------
#Summary Results
#---------------------------------
print('--------------- DQN results ---------------')
print('The best value of x found:', cb_dqn.xbest)
print('The best value of y found:', cb_dqn.rbest)
print('--------------- ACER results ---------------')
print('The best value of x found:', cb_acer.xbest)
print('The best value of y found:', cb_acer.rbest)
print('--------------- PPO results ---------------')
print('The best value of x found:', cb_ppo.xbest)
print('The best value of y found:', cb_ppo.rbest)
print('--------------- ACKTR results ---------------')
print('The best value of x found:', cb_acktr.xbest)
print('The best value of y found:', cb_acktr.rbest)
print('--------------- A2C results ---------------')
print('The best value of x found:', cb_a2c.xbest)
print('The best value of y found:', cb_a2c.rbest)

#---------------------------------
#Summary Plots
#---------------------------------

log_dqn = pd.DataFrame(cb_dqn.r_hist).cummax(axis = 0).values
log_acer = pd.DataFrame(cb_acer.r_hist).cummax(axis = 0).values
log_ppo = pd.DataFrame(cb_ppo.r_hist).cummax(axis = 0).values
log_acktr = pd.DataFrame(cb_acktr.r_hist).cummax(axis = 0).values
log_a2c = pd.DataFrame(cb_a2c.r_hist).cummax(axis = 0).values

def update_log_dqn_acer(log, n_sum_steps):
    # This is a helper function to convert right logger for DQN/ACER to
    # to be equivalent to other algs.
    #Inputs:
        #log: orginal DQN/ACER logger
        #n_sum_steps: number of steps to group and sum.
    #Outputs:
        #updated_log: the converted log for DQN/ACER
    data=np.transpose(log.tolist())[0]
    size=len(data)
    updated_log = [sum(data[i:i+n_sum_steps])/n_sum_steps for i in range(0,size,n_sum_
↪steps)]
    updated_log.pop(0); updated_log.pop(0); updated_log.pop(-1)  #remove extraneous␣
↪entries
    return updated_log
```

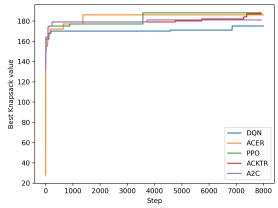
(continues on next page)

(continued from previous page)

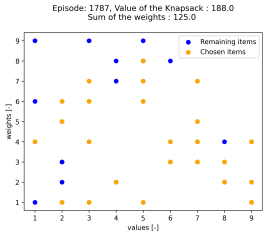
```
#update the DQN/ACER loggers
log_dqn=update_log_dqn_acer(log_dqn, n_sum_steps)
log_acer=update_log_dqn_acer(log_acer, n_sum_steps)

#plots
plt.figure()
plt.plot(log_dqn, label = "DQN")
plt.plot(log_acer, label = "ACER")
plt.plot(log_ppo, label = "PPO")
plt.plot(log_acktr, label = "ACKTR")
plt.plot(log_a2c, label = "A2C")
plt.xlabel('Step')
plt.ylabel('Best Knapsack value')
plt.legend()
plt.savefig("kp_history.png",format='png' ,dpi=300, bbox_inches="tight")
plt.show()
```

Results

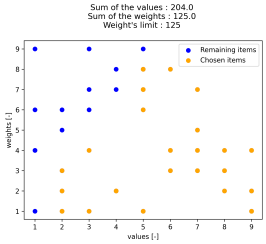
A summary of the results is shown below for the case of **50 items/objects**. First, all five reinforcement algorithms are compared in terms of maximizing the sum of item weights. The fitness convergence shows that PPO and ACKTR are the best algorithms in this case, with PPO slightly achieved a better fitness. Therefore, we will limit the reported results to PPO.



The maximum value of the Knapsack tour cost found by PPO is 188, which is fairly close to the optimal sum of item values of 204. The **PPO Knapsack** is below



while here is the target **optimal** Knapsack



And here are the final results of all algorithms:

```
--------------- DQN results ---------------
The best value of x found: ['7', '8', '9', '10', '11', '12', '13', '14', '15', '16',
↪'17', '18', '19', '20', '21', '22', '23', '24', '25', '26', '27', '28', '29', '33',
↪'45', '36', '48', '35', '6', '37', '4', '30', '41']
The best value of y found: 175.0
--------------- ACER results ---------------
The best value of x found: ['41', '36', '10', '2', '6', '22', '13', '7', '15', '48',
↪'19', '8', '21', '17', '49', '3', '11', '4', '23', '26', '25', '1', '28', '50', '9',
↪ '20', '24', '33', '29', '32', '31', '14', '43', '27']
The best value of y found: 186.0
--------------- PPO results ---------------
The best value of x found: ['11', '28', '12', '4', '30', '47', '33', '9', '17', '29',
↪'49', '26', '41', '43', '6', '3', '45', '22', '35', '8', '48', '15', '36', '44', '23
↪', '7', '32', '21', '34', '1', '25', '27', '19', '20', '14', '40']
The best value of y found: 188.0
--------------- ACKTR results ---------------
The best value of x found: ['29', '12', '31', '4', '7', '36', '39', '24', '10', '40',
↪'15', '22', '33', '48', '9', '19', '32', '45', '34', '14', '8', '25', '2', '11', '46
↪', '28', '23', '3', '49',
'26', '21', '35', '16', '38']
The best value of y found: 187.0
--------------- A2C results ---------------
The best value of x found: ['16', '28', '15', '4', '32', '23', '20', '1', '43', '8',
↪'36', '3', '35', '46', '2', '22', '21', '41', '18', '7', '24', '38', '40', '11', '12
↪', '29', '44', '49', '48', '37', '33', '9']
The best value of y found: 181.0
```

2.4.11 Example 11: Microreactor Control with Malfunction

Example demonstrating NEORL used to find optimal control mechanism positions to acheive favorable operation of a nuclear microreactor.

Original paper: Price, D., Radaideh, M. I., Kochunas, B. (2022). Multiobjective optimization of nuclear microreactor reactivity control system operation with swarm and evolutionary algorithms. Nuclear Engineering and Design, 393, 111776.

https://doi.org/10.1016/j.nucengdes.2022.111776

Summary

- Algorithm: DE, ES, MFO
- Type: Multi Objective, Unconstrained
- Field: Nuclear Engineering

Problem Description

The HOLOS-Quad reactor design is a high-temperature gas-cooled microreactor which has 8 cylindrical control drums for reactivity control. Each of the drums has a portion of their outer surface covered by absorbing material which, when rotated towards the body of the core, can absorb neutrons. The position of a drum will refer to its rotation angle from fully inserted. A view of the core design is given below:

<<awaiting approval to include picture>>

In this example, one of the 8 drums is immobilized and the positions of the remaining 7 drums need to be selected to satisfy three criteria:

1. Insert a target reactivity: In order for a nuclear reactor to effectively operate, the control system needs to adjust such that the core can have a criticality of 1.
2. Minimize the differences in quadrant powers: Nuclear reactors must maintain relatively even power distributions to effectively operate because the maximum reactor power is limited by the highest power region in the core. Even power distributions allow for higher power output while maintaining safe operation.
3. Minimize the travel distance of the farthest traveling drum: Control drums have a maximum rotation speed which dictates the amount of time required to a drum to a certain position. Therefore, minimizing the travel distance of the farthest traveling drum will minimize the time required to position all control drums.

These criteria must be expressed mathematically to create the objective functions. Due to the large computational cost associated with conventional calculation methods for core criticality and quadrant powers, surrogate models must be used. For the target reactivity objective, a method which uses a physics-based model augmented with a statistical model is used to predict the reactivity inserted from a particular control drum configuration. This model is described in detail in the paper below:

Price, D., Kinast, S., Barr, K., Kochunas, B., & Filippone, C. (2021), A perturbation-based hybrid methodology for control drum worth prediction applied to the HOLOS-Quad microreactor concept. Annals of Nuclear Energy. (In Press)

This model will be refered to as $\rho(\vec{x})$ where $\vec{x}$ is a vector of 7 control drum positions. Each component of $\vec{x}$ is bounded by -π and π radians. The objective function can then be written as:

$$\hat{f}_c(\vec{x}) = |\rho_{tgt} - \rho(\vec{x})|$$

where ρ_{tgt} is the target reactivity. The c subscript on $\hat{f}$ is used to show that this objective function corresponds to the criticality conditions of the core. The circumflex indicates that this objective is currently unscaled.

For the next objective, that is to minimize the differences in quadrant powers, a neural network is used to predict fractional quadrant powers for a particlar control drum configuration. More information on this neural network is given in the paper given at the top of this page as "Original Paper". If the power in each quadrant of the core can be represented by P with some subscript indicating which of the four quadrants P represents, the objective function can be given as:

$$\hat{f}_p(\vec{x}) = \sum_{i=1}^{4} \left| \frac{P_i}{P_1 + P_2 + P_3 + P_4} - \frac{1}{4} \right|$$

Here, the p subscript is included to indicate the objective function corresponding to the core quadrant powers.

The final objective is the simplest of the three, it is simply the maximum drum rotation angle present in $\vec{x}$.

$$\hat{f}_d(\vec{x}) = \max \vec{x}$$

Now that the three objective functions have been defined, they can be scaled such that their outputs occupy similar scales. Although not strictly necessary, it makes the weight selection with the scalarization method considerably more

straightforward. Separately for each of the three objectives, $\hat{f}(\vec{x})$ is transformed into $f(\vec{x})$ using the equation given below:

$$f(\vec{x}) = \frac{\hat{f}(\vec{x}) - \hat{f}_{min}}{\hat{f}_{max} - \hat{f}_{min}}$$

where $\hat{f}_{max}$ and $\hat{f}_{min}$ denote the maxima and minima of the objective being scaled. This can be obtained with a simple single objective optimization problem or it can be obtained using knowledge of the problem. Nevertheless, in this application, these extrema are given in the original paper.

Next, scalarization is performed. Scalarization is a method used to reduce a multi objective optimization problem into a single objective optimization problem by assigning weights to each objective and summing them together. Mathematically, this can be written as:

$$F(\vec{x}) = w_c f_c(\vec{x}) + w_p f_p(\vec{x}) + w_d f_d(\vec{x})$$

Here, $F(\vec{x})$ is the function that will be plugged into an optimizer and w is used to indicate the weight assigned to each of the objective functions. Moreover, the selection of these weights is nontrivial and an important part of complex optimization analyses. For this application, $w_c = 0.50, w_p = 0.40$ and $w_d = 0.10$. More discussion is given in the original paper on the selection of these weights.

NEORL script

```
import numpy as np
import matplotlib.pyplot as plt
import sys
from neorl import DE, ES, MFO
from neorl.benchmarks.reactivity_model import ReactivityModel
from neorl.benchmarks.qpower_model import QPowerModel

#import models from other files in repo
rm = ReactivityModel()
pm = QPowerModel()

#define unscaled objectives
def hatfc(x):
    thetas = np.zeros(8)
    thetas[0] = x[0]
    thetas[2:] = x[1:]
    react = rm.eval(thetas)
    return np.abs(react - 0.03308)

def hatfp(x):
    thetas = np.zeros(8)
    thetas[0] = x[0]
    thetas[2:] = x[1:]
    powers = pm.eval(thetas)
    targets = np.zeros(4)+0.25
    return np.abs(powers - targets).sum()

def hatfd(x):
    return np.max(np.abs(x))

#define objective scaling parameters
fc_max = 0.03308
fc_min = 0
```

(continues on next page)

(continued from previous page)

```
fp_max = 0.0345
fp_min = 0

fd_max = np.pi
fd_min = 0

#define scaling objectives
fc = lambda x : (hatfc(x) - fc_min)/(fc_max - fc_min)
fp = lambda x : (hatfp(x) - fp_min)/(fp_max - fp_min)
fd = lambda x : (hatfd(x) - fd_min)/(fd_max - fd_min)

#define function weights
wc = 0.5
wp = 0.4
wd = 0.1

#define single objective function
F = lambda x : wc*fc(x) + wp*fp(x) + wd*fd(x)

#define drum rotation bounds
BOUNDS = {"x%i"%i : ["float", -1.*np.pi, 1.*np.pi] for i in range(1, 8)}

#run de optimization
npop = 20
F_de = 0.4
CR = 0.3
de = DE(mode = "min", bounds = BOUNDS, fit = F, npop = npop, F = F_de, CR = CR, seed␣
↪= 1)
de_x, de_y, de_hist = de.evolute(100, verbose = True)

#run es optimization
mu = 25
cxpb = 0.6
mutpb = 0.3
es = ES(mode = "min", bounds = BOUNDS, fit = F, lambda_ = 50, mu = mu, cxpb = 0.6,
        mutpb = 0.3, seed = 1)
es_x, es_y, es_hist = es.evolute(100, verbose = True)

#run mfo optimization
nmoths = 55
mfo = MFO(mode = "min", bounds = BOUNDS, fit = F, nmoths = nmoths, b = 1, seed = 1)
mfo_x, mfo_y, mfo_hist = mfo.evolute(100, verbose = True)

plt.plot(de_hist["global_fitness"], label = "DE")
plt.plot(es_hist["global_fitness"], label = "ES")
plt.plot(mfo_hist["global_fitness"], label = "MFO")

plt.xlabel("Generation")
plt.ylabel("Fitness")
plt.legend()
plt.show()

print("MFO fc hat")
print(hatfc(mfo_x))
print("MFO fp hat")
print(hatfp(mfo_x))
```

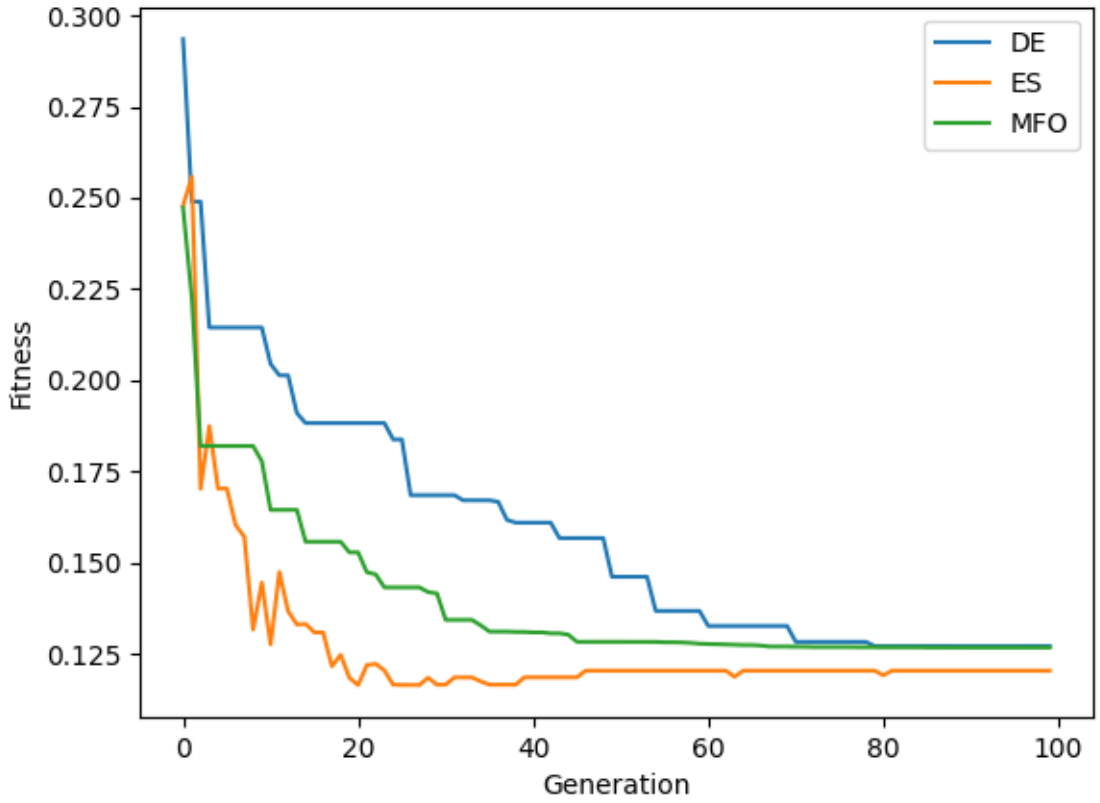
(continues on next page)

(continued from previous page)

```
print("MFO fd hat")
print(hatfd(mfo_x))
```

Results

A summary of the results for the different methods is shown below with the best $\vec{x}$ and $F(\vec{x})$. All methods seem to reasonably seek minima.




```
------------------------ DE Summary --------------------------
Best fitness (y) found: 0.12723682745792148
Best individual (x) found: [3.113469132155524, 2.52205504536713, -1.940784552123703,␣
↪2.3264933610351117, -2.0539691214048084, 3.089626887713435, 1.4072560227038484]
--------------------------------------------------------------
------------------------ ES Summary --------------------------
Best fitness (y) found: 0.11653471587218096
Best individual (x) found: [3.141592653589793, 2.0878715421838763, 2.0334381504862433,
↪ 2.2178488588636247, -2.2914574224308626, 2.4524812539265213, 1.7243458084183882]
--------------------------------------------------------------
------------------------ MFO Summary --------------------------
Best fitness (y) found: 0.12684771880454485
Best individual (x) found: [3.14159265 3.14159265 1.29847427 1.85712596 3.14159265 2.
↪77812329 1.89814577]
--------------------------------------------------------------
```

The errors in the unscaled objectives can also be obtained:

```
print("MFO fc hat")
print(hatfc(mfo_x))
print("MFO fp hat")
print(hatfp(mfo_x))
print("MFO fd hat")
print(hatfd(mfo_x))
```

```
MFO fc hat
1.9822642943062574e-07
MFO fp hat
0.0023153573274612427
MFO fd hat
3.141592653589793
```

2.5 Change Log

2.5.1 Stable Releases (for users)

Release 1.8 (2022-6-5)

```
pip install neorl==1.8
```

- Improvements in mixed-discrete optimization for various algorithms.
- Introducing new ensemble optimization algorithms: AEO, EPSO, EDEV, HCLPSO.
- New features for simulated annealing.
- More organized documentation with subsections.
- Application of NEORL to nuclear microreactors, recently published here: https://doi.org/10.1016/j.nucengdes.2022.111776

Release 1.7 (2021-11-24)

```
pip install neorl==1.7
```

- Fixed a major bug following scikit-learn update to 1.0.0. Now, NEORL supports scikit-learn <= 0.24.0
- Misc. minor updates for the documentation and the source code

Release 1.6 (2021-09-10)

```
pip install neorl==1.6
```

- The first NEORL stable release.
- Includes all changes in all previous beta releases: 1.2.0b-1.5.7b.
- Summary: 28 algorithms, 4 tuning methods, 10 real-world examples, and 39 unit tests.
- Documentation and Github repo are up-to-date.

2.5.2 Beta Releases (for developers)

Release 1.7.8b (2022-6-4)

```
pip install neorl==1.7.8b --extra-index-url https://test.pypi.org/simple
```

- Fixed bug related to `protobuf <= 3.20` version to avoid tensorflow error when installation.

Release 1.7.7b (2022-6-4)

```
pip install neorl==1.7.7b --extra-index-url https://test.pypi.org/simple
```

- Fixed bugs when initializing methods (ES, PSO, GWO, etc.) for discrete/mixed optimization with the argument `x0`.
- Fixed bugs in TS, CS, SA for discrete/mixed optimization.
- Enabled discrete/mixed optimization for AEO.

Release 1.7.6b (2022-4-25)

```
pip install neorl==1.7.6b --extra-index-url https://test.pypi.org/simple
```

- Added a new features for parallel simulated annealing (PSA).
- PSA features include new equilibrium schedule and new solution enforcement.

Release 1.7.5b (2022-3-29)

```
pip install neorl==1.7.5b --extra-index-url https://test.pypi.org/simple
```

- Added a full module for EPSO with parallel support and all space types.
- Fixed a minor bug in HCLPSO.

Release 1.7.4b (2022-3-26)

```
pip install neorl==1.7.4b --extra-index-url https://test.pypi.org/simple
```

- Added a full module for HCLPSO with parallel support and all space types.
- Added a full module for EDEV with parallel support and all space types.

Release 1.7.3b (2022-3-6)

```
pip install neorl==1.7.3b --extra-index-url https://test.pypi.org/simple
```

- Fixed a division by zero bug for PSO mode `globw`.
- Added a first test module for a new hybrid algorithm AEO (Aniormphic Ensemble Optimization).
- Added documentation for AEO on the website.

Release 1.7.2b (2022-1-17)

```
pip install neorl==1.7.2b --extra-index-url https://test.pypi.org/simple
```

- Fixed a plotting bug in NEORL benchmarks.
- Added `last_pop` results to the returned history dictionary of ACO, SSA, JAYA, BAT, CS, XNES.
- Allowed passing annealing parameters via `**kwargs` to WOA, GWO, PSO, MFO, HHO (this is for AEO ensemble research).

Release 1.7.1b (2022-1-03)

```
pip install neorl==1.7.1b --extra-index-url https://test.pypi.org/simple
```

- Added new NEORL example 11 on nuclear microreactor application.
- Documentation structure updates. Now subsections are part of the documentation structure.

Release 1.6.2b (2021-10-07)

```
pip install neorl==1.6.2b --extra-index-url https://test.pypi.org/simple
```

- Removed summary files from RL runners.
- Added a capability to save current model for RL runners. Currently best model and last model are saved.

Release 1.6.1b (2021-09-20)

```
pip install neorl==1.6.1b --extra-index-url https://test.pypi.org/simple
```

- Fixed a bounding check bug in FNEAT and RNEAT.
- Fixed different typos in the documentation.
- Increased the width of the online documentation page to fit more code/words.

Releases 1.5.3b-1.5.7b (2021-09-10)

```
pip install neorl==1.5.7b --extra-index-url https://test.pypi.org/simple
```

- Added hybrid neuroevolution algorithm: Neural genetic algorithm (NGA)
- Added hybrid neuroevolution algorithm: Neural Harris hawks optimization (NHHO)
- Added Cuckoo Search with all spaces handled.
- Added Ant Colony optimization for continuous domains.
- Added Tabu Search for discrete domains.
- Fixed a critical bug in the terminal API in the followup 1.5.4b
- Fixed a bug in the terminal API continue mode in the followups 1.5.5b-1.5.6b.
- Fixed hyperthreading issue for RL algorithms in the followup 1.5.7b.

Release 1.5.2b (2021-08-10)

```
pip install neorl==1.5.2b --extra-index-url https://test.pypi.org/simple
```

- Added hybrid neuroevolution algorithm PPO-ES.
- Added hybrid neuroevolution algorithm ACKTR-DE.
- Updated documentation for RL algorithms.

Release 1.5.1b (2021-08-01)

```
pip install neorl==1.5.1b --extra-index-url https://test.pypi.org/simple
```

- Added RNEAT and FNEAT with full documentation.
- Added mixed discrete optimization to WOA, GWO, SSA, DE, MFO, JAYA, PESA2
- Added friendly implementation to construct parallel environments for RL: DQN, ACKTR, A2C, PPO

Release 1.5.0b (2021-07-28)

```
pip install neorl==1.5.0b --extra-index-url https://test.pypi.org/simple
```

- Updated Example 1 on using RL to solve Travel Salesman problem
- Added Example 10 on using RL to solve Knapsack problem
- Added CEC-2008 benchmark functions for large-scale optimization

Release 1.4.8b (2021-07-14)

```
pip install neorl==1.4.8b --extra-index-url https://test.pypi.org/simple
```

- Added environment class constructor for DQN, ACER, PPO, ACKTR, A2C
- Added mixed discrete/continuous optimization for PPO, ACKTR, A2C
- Added categorical/discrete optimization for ACER, DQN.

Releases 1.4.6b-1.4.7b (2021-07-09)

```
pip install neorl==1.4.7b --extra-index-url https://test.pypi.org/simple
```

- Modifying Bat algorithm to handle mixed spaces.
- Added Example 6 on three-bar truss design.
- Added Examples 7 and 8 on pressure vessel design.
- Added Example 9 on cantilever stepped beam.
- Fixing bugs after 1.4.6b.

Releases 1.4.1b-1.4.5b (2021-07-05)

```
pip install neorl==1.4.5b --extra-index-url https://test.pypi.org/simple
```

- Fixing bounding issues in most evolutionary algorithms.
- Fixing PESA/PESA2 parallel mode.
- Replacing XNES with WOA in modern PESA2.
- Added a module for Harris Hawks Optimization.
- Added the BAT algorithm.
- Removed deprecation warnings of Tensorflow from NEORL.
- Added a module for JAYA.
- Added a module for MFO.

2.5.3 Old Releases (outdated)

Release 1.4.0b (2021-05-15)

- Added a module for Simulated Annealing (SA).
- Added a Genetic/Evolutionary hyperparameter tuning module.
- Added ACER module for RL optimization.
- Added ACKTR module for RL optimization.
- Added a WOA module for evolutionary optimization.
- Added a SSA module for evolutionary optimization.

Release 1.3.5b (2021-05-10)

- Added CEC'2017 Test Suite benchmarks
- Added a set of classical mathematical functions
- Added new Example (4) on the website on how to access the benchmarks
- Added new Example (5) on the website on how to optimize the benchmarks

Releases 1.3.1b-1.3.2b (2021-05-4)

- Fixing miscellaneous bugs

Release 1.3.0b (2021-05-1)

- Added a module for the hybrid algorithm PESA.
- Added a module for the modern hybrid algorithm PESA2.
- Added a GWO module.
- Adding min/max modes for all algorithms.

Release 1.2.0b (2021-04-15)

- **The first public open-source version of NEORL**
- Added DE with serial implementation.
- Added XNES with parallel implementation.
- Restructuring the input parameter space.
- Detailed README file in the Github page.
- Added unit tests to NEORL.
- Automatic documentation via Sphinx

Release 1.1.0-Private (2020-12-15)

- Added Bayesian hyperparameter tuning from `scikit-optimise`.
- Added parallel evolutionary strategies(ES).
- Updated documentation.

Release 1.0.0-Private (2020-09-15)

- Added evolutionary strategies ES.
- Added a local PDF documentation.
- Added parallel PSO.
- Added Random search hyperparameter tuning.

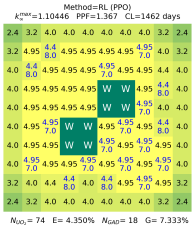
Release 0.1.1-Private (2020-03-15)

- A support for both classical (evolutionary) and modern (machine learning) optimization in the same package. Currently, DQN (serial), PPO (parallel), A2C (parallel), GA (serial), SA (serial) are supported. All RL algorithms are based upon `stable-baselines`.
- Easy-to-use syntax and friendly interaction with the package.
- A support for parallel computing.
- Added grid search hyperparameter tuning.
- For developers: an organized implementation and source code structure to facilitate the job of future external contributors.
- NEORL examples are provided in the "examples" directory.

2.6 Projects

This is a list of projects using NEORL. Please contact us if you want your project to appear on this page.

2.6.1 Physics-informed Reinforcement Learning Optimisation with NEORL

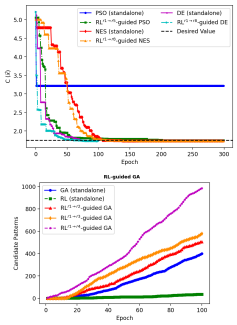


Optimization of nuclear fuel assemblies if performed effectively, will lead to fuel efficiency improvement, cost reduction, and safety assurance. However, assembly optimization involves solving high-dimensional and computationally expensive combinatorial problems. As such, fuel designers' expert judgement has commonly prevailed over the use of stochastic optimization (SO) algorithms such as genetic algorithms and simulated annealing. To improve the state-of-art, we explore a class of artificial intelligence (AI) algorithms, namely, reinforcement learning (RL) in this work. We propose a physics-informed AI optimization methodology by establishing a connection through reward shaping between RL and the tactics fuel designers follow in practice by moving fuel rods in the assembly to meet specific constraints and objectives. The methodology utilizes RL algorithms, deep Q learning and proximal policy optimization, and compares their performance to SO algorithms. The methodology is applied on two boiling water reactor assemblies of low-dimensional (combinations) and high-dimensional (combinations) natures. The results demonstrate that RL is more effective than SO in solving high dimensional problems, i.e., 10×10 assembly, through embedding expert knowledge in form of game rules and effectively exploring the search space. For a given computational resources and timeframe relevant to fuel designers, RL algorithms outperformed SO through finding more feasible patterns, 4–5

times more than SO, and through increasing search speed, as indicated by the RL outstanding computational efficiency. The results of this work clearly demonstrate RL effectiveness as another decision support tool for nuclear fuel assembly optimization.

Authors: Majdi I. Radaideh et al., 2021.
Reference: https://doi.org/10.1016/j.nucengdes.2020.110966

2.6.2 Reinforcement Learning Guiding Evolutionary Algorithms in Constrained Optimization



For practical engineering optimization problems, the design space is typically narrow, given all the real-world constraints. Reinforcement Learning (RL) has commonly been guided by stochastic algorithms to tune hyperparameters and leverage exploration. Conversely in this work, we propose a rule-based RL methodology to guide evolutionary algorithms (EA) in constrained optimization. First, RL proximal policy optimization agents are trained to master matching some of the problem rules/constraints, then RL is used to inject experiences to guide various evolutionary/stochastic algorithms such as genetic algorithms, simulated annealing, particle swarm optimization, differential evolution, and natural evolution strategies. Accordingly, we develop RL-guided EAs, which are benchmarked against their standalone counterparts. RL-guided EA in continuous optimization demonstrates significant improvement over standalone EA for two engineering benchmarks. The main problem analyzed is nuclear fuel assembly combinatorial optimization with high-dimensional and computationally expensive physics. The results demonstrate the ability of RL to efficiently learn the rules that nuclear fuel engineers follow to realize candidate solutions. Without these rules, the design space is large for RL/EA to find many candidates. With imposing the rule-based RL methodology, we found that RL-guided EA outperforms standalone algorithms by a wide margin, with times improvement in exploration capabilities and computational efficiency. These insights imply that when facing a constrained problem with numerous local optima, RL can be useful in focusing the search space in the areas where expert knowledge has demonstrated merit, while evolutionary/stochastic algorithms utilize their exploratory features to improve the number of feasible solutions.

Authors: Majdi I. Radaideh and Koroush Shirvan
Reference: https://doi.org/10.1016/j.knosys.2021.106836

2.6.3 Application of GA and DQN for in-core fuel management

The nuclear reactor core is composed of few hundred assemblies. The loading of these assemblies is done with the goal of reducing its overall cost while maintaining safety limits. Typically, the core designers choose a unique position and fuel enrichment for each assembly through use of expert judgement. In this thesis, alternatives to the current core reload design process are explored. Genetic algorithm and deep Q-learning are applied in an attempt to reduce core design time and improve the final core layout. The reference core represents a 4-loop pressurized water reactor where fixed number of fuel enrichments and burnable poison distributions are assumed. The algorithms automatically shuffles the assembly positions to find the optimum loading pattern. It is determined that both algorithms are able to successfully start with a poorly performing core loading pattern and discover a well performing one, by the metrics of boron concentration, cycle exposure, enthalpy-rise factor, and pin power peaking. This shows potential for further applications of these algorithms for core design with a more expanded search space.

Author: Jane Reed
Reference: http://34.201.211.163/handle/1721.1/127308

2.7 Contributors

These bios and affiliations are not continuously updated, and were added during the contribution period

Majdi I. Radaideh is a Research Scientist at MIT Nuclear Science and Engineering. He obtained his MS and PhD degrees from the University of Illinois at Urbana Champaign (UIUC) between August 2015 - August 2019. Radaideh's research focuses on algorithms, data engineering, deep learning, physics-informed machine learning, reinforcement learning, uncertainty quantification, and large-scale optimization with applications related to nuclear reactor safety and multiphysics simulations. Radaideh is the founder of NEORL with contributions include establishing the framework structure, algorithm development, open-source and documentation handling, benchmarking and testing, applications to nuclear fuel assembly optimisation, user interface handling, parallel computing, and many other minor contributions.

Github: https://github.com/mradaideh

Koroush Shirvan is the John Clark Hardwick (1986) Career Development Professor in the Department of Nuclear Science and Engineering. Previously, he was a principal research scientist at Center for Advanced Nuclear Energy Systems (CANES). He specializes in development and assessment of advanced nuclear reactor technology. His research combines multiple scales, physics and disciplines, and machine learning to realize innovative solutions in the highly regulated nuclear energy sector. His contribution to NEORL includes securing the funding for NEORL, significant contributions to the development of the physics-based applications of NEORL in the nuclear engineering area,

and direct administration of the project.

Webpage: https://web.mit.edu/nse/people/faculty/shirvan.html

Paul RM. Seurin is a Ph.D student in the joint program between the Nuclear Science and Engineering department and the Center for Computational Science and Engineering at MIT. He joined MIT in September 2019 after receiving an Engineering Diploma and a Master in Nuclear Engineering from France. His research involves algorithm design for optimization with focuses on heuristics methods, deep, and Reinforcement Learning with applications to nuclear reactor economics and licensing. For NEORL, he is currently working on validation and benchmarking, applying the framework to Pressurized Water Reactor optimization, and to solve combinatorial problems like Travel Salesman, Knapsack, and Job Scheduling problems. In algorithms, he developed the base classes for Cuckoo search and Tabu search, while he is also planning to develop a combinatorial version of the Ant Colony optimizer.

Haijia Wang is an undergraduate student at MIT in the MIT Electrical Engineering and Computer Science Department. She is in the class of 2023, studying Computer Science and Engineering. She is passionate about algorithm design and optimization, artificial intelligence and machine learning, and advanced computing technologies. Haijia is interested in exploring these fields in her studies and future career. She worked as a summer research intern with NEORL in 2020, and her contribution was adding different hyperparameter tuning methods to NEORL, including evolutioanry search and Bayesian search. She also validated the classes for grid and random search.

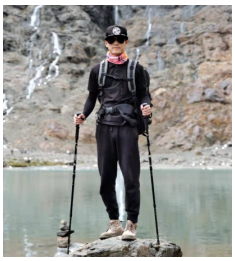
Linkedin: https://www.linkedin.com/in/haijia-wang

Katelin Du is an undergraduate student at MIT in the class of 2023. She is studying nuclear science and engineering with a focus in computational modeling and simulation methods. She is interested in the use of computational tools such as simulation methods and machine learning in engineering applications. In the summer of 2021, she worked as an undergraduate researcher and contributed to NEORL by adding to its existing set of optimization algorithms and performing engineering benchmarking. Algorithms she worked on include Harris Hawks Optimization and hybrid neuroevolution Harris Hawks Optimization, and genetic algorithms.

Devin Seyler is an undergraduate at MIT in the class of 2022 studying physics and mathematics. He is interested in nuclear physics and the use of algorithms in engineering applications, particularly to solve problems related to energy design and optimization. He worked as an undergraduate researcher with NEORL during the summer of 2021, contributing to its set of optimization algorithms. Specifically, he worked on Bat Algorithm and a hybrid neuroevolutionary Grey Wolf Optimizer. He also performed benchmarking with various engineering problems along with application of NEORL on fuel cell energy optimization.

John Mobley IV is a Ph.D. student within the Department of Nuclear Science and Engineering at MIT. He joined MIT in June 2021 following the completion of his undergraduate career at Iowa State University where he earned a B.S. in Physics and a B.S. in Mathematics. Passionate about the development and deployment of advanced nuclear reactor technologies from Generation IV and beyond, John is interested in exploring the burgeoning domain of small-scale, reduced capital, modular, self-contained concepts. Contributing to NEORL, he has worked to identify the implications of hyperparameters in an effort to tune and isolate optimal settings. He also has worked on application of NEORL to Pressurized Water Reactor core optimization.

Linkedin: https://www.linkedin.com/in/johnmobleyiv/



Xubo GU obtained his MS degree from Shanghai Jiao Tong University in 2020. His research involves machine learning, reinforcement learning, data analytics, large-scale optimization with heuristic methods, and their applications in the industry. He has one year of working experience in an industrial-intelligence company, where he applied optimization techniques and machine learning to solve practical industrial problems. For NEORL, he developed three evolutionary algorithms, including MFO, JAYA, and ACO. He also developed two hybrid neuroevolutionary algorithms for NEORL - FNEAT and RNEAT. Besides, he is the main developer of OpenNeoMC, which combines NEORL with OpenMC to empower particle transport code with state-of-the-art optimization techniques of NEORL.

Dean Price is a graduate student at the University of Michigan studying for a PhD in Nuclear Engineering. Dean has

experience in various areas of nuclear reactor modeling and simulation including uncertainty quantification, sensitivity analysis, burnup credit, spent fuel criticality safety, surrogate modeling and multiphysics coupling. For NEORL, he helps maintain the code base and has demonstrated the capabilities of NEORL on microreactor control applications. Currently, he is working to develop new optimization algorithms to be included in future versions of NEORL.

ORCID: https://orcid.org/0000-0003-0999-0111

2.8 Contribution Guide

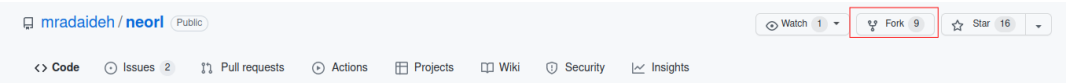
2.8.1 Bug Reporting

All bugs or general issues can be reported on the Github issues page: https://github.com/mradaideh/neorl/issues

2.8.2 Getting Started

In order to contribute code that may be included in the main NEORL distribution, the following steps can be taken. In general, all contributions should be submitted as a pull request to the main NEORL repository. Specifically, the steps provided below can be used to incorporate code into the NEORL repository.

1. Create a fork of the main NEORL repository. Through Github, this can be done by selecting the "fork" button on the NEORL page found at: https://github.com/mradaideh/neorl



2. Once this is done, a new repository will be created under your ownership. It will exist under *github_username/neorl*. From here, the forked repository can be cloned like any other repository. Navigate to a directory you plan on working in and enter the command: `git clone git@github.com:github-un/neorl.git`

   Replace the **github-un** with your Github username

3. From here, the forked repository can be committed to using typical *git* practices.

2.8.3 Incorporating Changes From the Main Repository

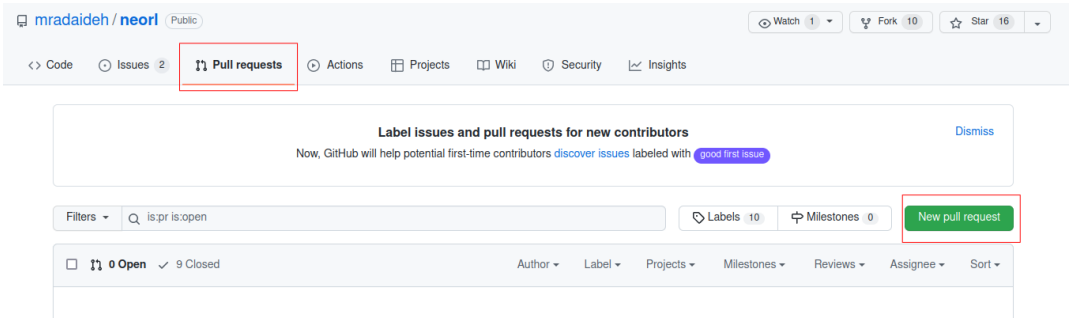
Often when making large contributions, it may be necessary to work on the fork for an extended period of time where updates have been pushed to the upstream (*mradaideh/neorl*) repository after you originally created the fork. In order to incorporate these new updates into your local fork:

1. Commit and push all work in your local repository with regular `git commit` and `git push` commands.
2. First add the upstream repository into known remote repositories: `git remote add upstream https://github.com/mradaideh/neorl`
3. Fetch changes that have been made to upstream repository: `git fetch upstream`
4. Merge changes made to upstream repository into your local repository: `git merge upstream/master master`
5. Push from your local fork (on your PC) to your remote fork (on Github): `git push`

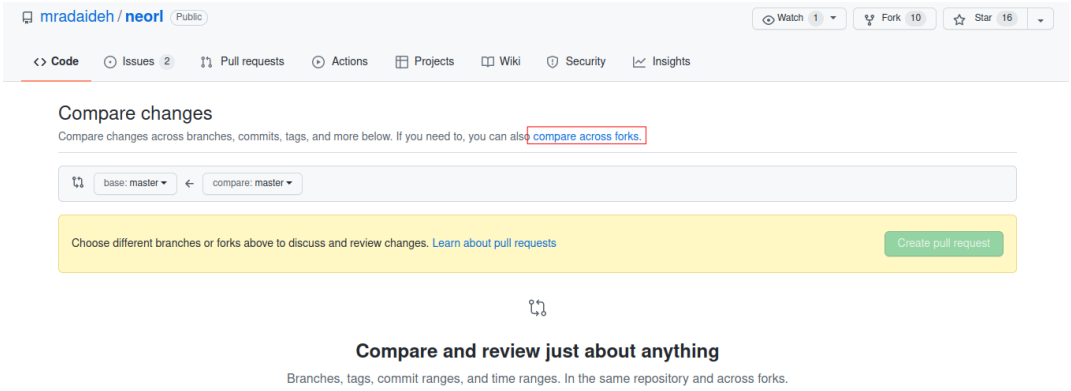
2.8.4 Submitting a Pull Request

Once a major contribution to the NEORL code base has been developed and is ready to be incorporated into the upstream repository (*mradaideh/neorl*), a pull request can be submitted:

1. While logged into the account with the forked repository containing the changes, navigate to the Github page for the upstream NEORL repository: https://github.com/mradaideh/neorl
2. Select the "Pull requests" tab near the top of the page and select "new pull request".



3. Select "compare across forks" link.



4. From here, the "base repository" should be set to *mradaideh/neorl/master* and the "head repository" should point to the repository that is under your name.
5. Press the "Create pull request" button and fill out the submission fields and submit it for review!

CHAPTER

THREE

PROJECTS/PAPERS USING NEORL

1- Radaideh, M. I., Wolverton, I., Joseph, J., Tusar, J. J., Otgonbaatar, U., Roy, N., Forget, B., Shirvan, K. (2021). Physics-informed reinforcement learning optimization of nuclear assembly design. *Nuclear Engineering and Design*, **372**, p. 110966 [LINK1].

2- Radaideh, M. I., Shirvan, K. (2021). Rule-based reinforcement learning methodology to inform evolutionary algorithms for constrained optimization of engineering applications. *Knowledge-Based Systems*, **217**, p. 106836 [LINK2].

3- Radaideh, M. I., Forget, B., Shirvan, K. (2021). Large-scale design optimisation of boiling water reactor bundles with neuroevolution. Annals of Nuclear Energy, 160, 108355 [LINK3].

CHAPTER

FOUR

CITING THE PROJECT

To cite this repository in publications:

```
@article{radaideh2021neorl,
  title={NEORL: NeuroEvolution Optimization with Reinforcement Learning},
  author={Radaideh, Majdi I and Du, Katelin and Seurin, Paul and Seyler, Devin and Gu,
↪ Xubo and Wang, Haijia and Shirvan, Koroush},
  journal={arXiv preprint arXiv:2112.07057},
  year={2021}
}
```

CHAPTER

FIVE

ACKNOWLEDGMENTS

NEORL was established in MIT back to 2020 with feedback, validation, and usage of different colleagues: Issac Wolverton (MIT Quest for Intelligence), Joshua Joseph (MIT Quest for Intelligence), Benoit Forget (MIT Nuclear Science and Engineering), Ugi Otgonbaatar (Exelon Corporation).

PYTHON MODULE INDEX

n

INDEX

M

N