DACBench

Release 01.02.2021

Theresa Eimer, Maximilian Reimer

Sep 04, 2022

GETTING STARTED:

1 Dynamic Algorithm Configuration - A Short Overview 1

2 How to Install DACBench 3

3 Using DACBench Containers 5

4 Benchmark Overview 7

5 The Sigmoid Toy Benchmark 11

6 The Luby Toy Benchmark 15

7 The Geometric Toy Benchmark 19

8 The FastDownward Benchmark 23

9 The Theory Benchmark 27

10 The PyCMA CMA-ES Benchmark 31

11 The ModEA Benchmark 35

12 The IOHProfiler ModCMA Benchmark 37

13 The SGD Deep Learning Benchmark 39

14 Saving & Loading Benchmark Configurations 43

15 Modifying Observations & Cost 45

16 Functionality through Wrappers 47

17 Logging Experiments 53

18 Plotting results 59

19 Contributing to DACBench 65

20 Citing DACBench 67

21 API Reference 69

Python Module Index 165

Index 167

CHAPTER

ONE

DYNAMIC ALGORITHM CONFIGURATION - A SHORT OVERVIEW

Dynamic Algorithm Configuration (DAC) [Biedenkapp et al., ECAI 2020, Adriaensen et al., CoRR 2022] is a paradigm for hyperparameter optimization that aims to find the best possible configuration of algorithm hyperparameters for *each step* in the algorithm's execution and for *each algorithm instance*.

That means DAC methods configure hyperparameters dynamically over the runtime of an algorithm because the optimal value could be very different at the start than in the end. An example for this is the learning rate in SGD where at first we want to traverse the loss landscape fairly quickly (= high learning rate), but then need to slow down as to not overshoot the optimum (= gradually decreasing learning rate).

Furthermore, as we configure across a set of instances, DAC methods also need to take into account how these factors change between algorithm instances - a learning rate schedule on a very simple image classification problem like MNIST, for example, will likely look different than on a challenging one like ImageNet.

DAC can be solved by a number of different methods, e.g. classical Algorithm Configuration tools or Reinforcement Learning. As the paradigm is still relatively new, there is a lot of space to experiment with new possibilities or combine existing configuration options. In order to stay up to date on the current DAC literatur, we recommend our *`DAC literature overview <https://www.automl.org/automated-algorithm-design/dac/literature-overview/> `_*.

CHAPTER

TWO

HOW TO INSTALL DACBENCH

This is a guide on how to install DACBench and its benchmarks. Alternatively, you can also use pre-built containers containers.

First clone our GitHub repository:

```
git clone https://github.com/automl/DACBench.git
cd DACBench
git submodule update --init --recursive
```

We recommend installing within a virtual environment:

```
conda create -n dacbench python=3.6
conda activate dacbench
```

Now install DACBench with:

```
pip install -e .
```

To also install all dependecies used in the examples, instead run:

```
pip install -e .[example]
```

You should now have DACBench installed in the base version. This includes on the artificial benchmarks, all others have separate installation dependencies. The full list of options is:

- cma - installs the PyCMA step size control benchmark
- modea - installs the ModEA benchmark
- modcma - installs the IOHProfiler versions of CMA step size and CMA algorithm control
- sgd - installs the SGD benchmark
- theory - installs the theory benchmark
- all - installs all benchmark dependencies
- example - installs example dependencies
- docs - installs documentation dependencies
- dev - installs dev dependencies

Please not that in order to use the FastDownward benchmarks, you don't have to select different dependencies, but you have to build the planner. We recommend using cmake 3.10.2 for this:

```
./dacbench/envs/rl-plan/fast-downward/build.py
```

In the top level directory, you will find folders for tests, examples, code coverage reporting and documentation. The code itself can be found in the 'dacbench' folder. If you want to take advantage of our pre-run static and random baselines (10 runs each with 1000 episodes), you can download them here.

CHAPTER

THREE

USING DACBENCH CONTAINERS

DACBench can run containerized versions of Benchmarks using Singularity containers to isolate their dependencies and make reproducible Singularity images.

To build an existing container, install Singularity and run the following to build the container of your choice. Here is an example for the CMA container:

```
cd dacbench/container/singularity_recipes
sudo singularity build cma cma.def
```

An example on how to use the container can be found in the examples in the repository.

For writing your own recipe to build a Container, you can refer to the recipe template in the repository: dacbench/container/singularity_recipes/recipe_template

CHAPTER

FOUR

BENCHMARK OVERVIEW

DACBench contains a range of benchmarks in different categories and from different domains. There is a range of highly configurable, cheap to run benchmarks that often also include a ground truth solution. We recommend using these as an introduction to DAC, to verify new algorithms and to generate detailed insights. They are both based on artificial functions and real algorithms:

- *Sigmoid* (Artificial Benchmark): Sigmoid function approximation in multiple dimensions.
- *Luby* (Artificial Benchmark): Learning the Luby sequence.
- ToySGD (Artificial Benchmark): Controlling the learning rate in gradient descent.
- *Geometric* (Artificial Benchmark): Approximating several functions at once.
- Toy version of the *FastDownward benchmark*: Heuristic selection for the FastDownward Planner with ground truth.
- *Theory benchmark* with ground truth: RLS algorithm on the LeadingOnes problem.

Beyond these smaller scale problems we know a lot about, DACBench also contains less interpretable algorithms with larger scopes. These are oftentimes noisier, harder to debug and more costly to run and thus present a real challenge for DAC algorithms:

- *FastDownward benchmark*: Heuristic selection for the FastDownward Planner on competition tasks.
- *CMA-ES*: Step-size adpation for CMA-ES.
- *ModEA*: Selection of Algorithm Components for EAs.
- *ModCMA*: Step-size & algorithm component control for EAs backed by IOHProfiler.
- *SGD-DL*: Learning rate adaption for neural networks.

Our benchmarks are based on OpenAI's gym interface for Reinforcement Learning. That means to run a benchmark, you need to create an environment of that benchmark to then interact with it. We include examples of this interaction between environment and DAC methods in our GitHub repository. To instantiate a benchmark environment, run:

```
from dacbench.benchmarks import SigmoidBenchmark
bench = SigmoidBenchmark()
benchmark_env = bench.get_environment()
```

class dacbench.abstract_benchmark.**AbstractBenchmark**(*config_path=None*, *config: Optional[*objdict*] = None*)

Bases: `object`

Abstract template for benchmark classes

get_config()

Return current configuration

Returns
Current config

Return type
dict

get_environment()

Make benchmark environment

Returns
env – Benchmark environment

Return type
gym.Env

process_configspace(*configuration_space*)

This is largely the builting cs.json.write method, but doesn't save the result directly If this is ever implemented in cs, we can replace this method

read_config_file(*path*)

Read configuration from file

Parameters
path (*str*) – Path to config file

serialize_config()

Save configuration to json

Parameters
path (*str*) – File to save config to

set_action_space(*kind*, *args*)

Change action space

Parameters

- **kind** (*str*) – Name of action space class
- **args** (*list*) – List of arguments to pass to action space class

set_observation_space(*kind*, *args*, *data_type*)

Change observation_space

Parameters

- **kind** (*str*) – Name of observation space class
- **args** (*list*) – List of arguments to pass to observation space class
- **data_type** (*type*) – Data type of observation space

set_seed(*seed*)

Set environment seed

Parameters
seed (*int*) – New seed

class dacbench.abstract_benchmark.**objdict**

Bases: dict

Modified dict to make config changes more flexible

copy() → a shallow copy of D

class dacbench.abstract_env.**AbstractEnv**(*config*)

Bases: Env

Abstract template for environments

get_inst_id()

Return instance ID

Returns
ID of current instance

Return type
int

get_instance()

Return current instance

Returns
Currently used instance

Return type
type flexible

get_instance_set()

Return instance set

Returns
List of instances

Return type
list

reset()

Reset environment

Returns
Environment state

Return type
state

reset_(*instance=None*, *instance_id=None*, *scheme=None*)

Pre-reset function for progressing through the instance set Will either use round robin, random or no progression scheme

seed(*seed=None*, *seed_action_space=False*)

Set rng seed

Parameters

- **seed** – seed for rng
- **seed_action_space** (*bool, default False*) – if to seed the action space as well

seed_action_space(*seed=None*)

Seeds the action space. :param seed: if None self.initial_seed is be used :type seed: int, default None

set_inst_id(*inst_id*)

Change current instance ID

Parameters
inst_id (*int*) – New instance index

set_instance(*instance*)

Change currently used instance

Parameters
instance – New instance

set_instance_set(*inst_set*)

Change instance set

Parameters
inst_set (*list*) – New instance set

step(*action*)

Execute environment step

Parameters
action – Action to take

Returns

- *state* – Environment state
- *reward* – Environment reward
- **done** (*bool*) – Run finished flag
- **info** (*dict*) – Additional metainfo

step_()

Pre-step function for step count and cutoff

Returns
End of episode

Return type
bool

use_next_instance(*instance=None*, *instance_id=None*, *scheme=None*)

Changes instance according to chosen instance progession

Parameters

- **instance** – Instance specification for potentional new instances
- **instance_id** – ID of the instance to switch to
- **scheme** – Update scheme for this progression step (either round robin, random or no progression)

use_test_set()

Change to test instance set

use_training_set()

Change to training instance set

CHAPTER

FIVE

THE SIGMOID TOY BENCHMARK

Task: approximate a sigmoid curve at timestep t in each of one or multiple dimensions

Cost: distance between prediction and function

Number of hyperparameters to control: user specified starting at one integer with no fixed upper limit

State Information: Remaining budget, instance descriptions for each dimension (inflection point and slope), last action for each dimension

Noise Level: None

Instance space: one sigmoid curve consisting of inflection point and slope per dimension. Sampling notebook and example datasets in repository.

This benchmark is not built on top of an algorithm, but is simply a function approximation task. In each step until the cutoff, the DAC controller predicts one y-value for a given sigmoid curve per task dimension. The predictions are discrete, that means there is usually some distance between the true function value and the best prediction. This distance is used as a cost function. If multiple task dimensions are used, the total cost is computed by multiplying the costs of all dimensions.

The benchmark is very cheap to run and the instances can be sampled and shaped easily. Therefore it's a good starting point for any new DAC method or to gain specific insights for which fine control over the instance distribution is required.

The Sigmoid benchmark was constructed by Biedenkapp et al. for the paper `"Dynamic Algorithm Configuration: Foundation of a New Meta-Algorithmic Framework" <https://www.tnt.uni-hannover.de/papers/data/1432/20-ECAI-DAC.pdf> `_ at ECAI 2020

class dacbench.benchmarks.sigmoid_benchmark.**SigmoidBenchmark**(*config_path=None*, *config=None*)

Bases: *AbstractBenchmark*

Benchmark with default configuration & relevant functions for Sigmoid

get_benchmark(*dimension=None*, *seed=0*)

Get Benchmark from DAC paper

Parameters

- **dimension** (*int*) – Sigmoid dimension, was 1, 2, 3 or 5 in the paper
- **seed** (*int*) – Environment seed

Returns

env – Sigmoid environment

Return type

SigmoidEnv

get_environment()

Return Sigmoid env with current configuration

Returns
Sigmoid environment

Return type
SigmoidEnv

read_instance_set(*test=False*)

Read instance set from file

set_action_values(*values*)

Adapt action values and update dependencies

Parameters
values (*list*) – A list of possible actions per dimension

Sigmoid environment from "Dynamic Algorithm Configuration:Foundation of a New Meta-Algorithmic Framework" by A. Biedenkapp and H. F. Bozkurt and T. Eimer and F. Hutter and M. Lindauer. Original environment authors: André Biedenkapp, H. Furkan Bozkurt

class dacbench.envs.sigmoid.**ContinuousSigmoidEnv**(*config*)

Bases: *SigmoidEnv*

Environment for tracing sigmoid curves with a continuous state on the x-axis

step(*action: ndarray*)

Execute environment step. !!NOTE!! The action here is a list of floats and not a single number !!NOTE!!

Parameters
action (*list of floats*) – action(s) to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

class dacbench.envs.sigmoid.**ContinuousStateSigmoidEnv**(*config*)

Bases: *SigmoidEnv*

Environment for tracing sigmoid curves with a continuous state on the x-axis

step(*action: int*)

Execute environment step

Parameters
action (*int*) – action to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

class dacbench.envs.sigmoid.**SigmoidEnv**(*config*)

Bases: *AbstractEnv*

Environment for tracing sigmoid curves

`close`() → bool

Close Env

Returns
Closing confirmation

Return type
bool

`render`(*mode: str*) → None

Render env in human mode

Parameters
mode (*str*) – Execution mode

`reset`() → List[int]

Resets env

Returns
Environment state

Return type
numpy.array

`step`(*action: int*)

Execute environment step

Parameters
action (*int*) – action to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

CHAPTER

SIX

THE LUBY TOY BENCHMARK

Task: lning the Luby sequence with variations
Cost: correctness of sequence element prediction
Number of hyperparameters to control: one integer
State Information: Actions and timesteps of the last three iterations
Noise Level: None
Instance space: the Luby sequence with possibilities to modify the starting point of the series (e.g. element 5 instead of 1) as well as the repetition fo each element

This benchmark is not built on top of an algorithm, instead it's a pure sequence learning task. In each step until the cutoff, the DAC controller's task is to predict the next element of the Luby sequence. If the prediction is correct, it is given a reward of 1 and else 0.

The benchmark is very cheap to run, but can be altered to be quite challenging nonetheless. In its basic form, it can serve to validate DAC methods and observe their prowess in learning a series of predictions correctly.

The Luby benchmark was constructed by Biedenkapp et al. for the paper `"Dynamic Algorithm Configuration: Foundation of a New Meta-Algorithmic Framework" <https://www.tnt.uni-hannover.de/papers/data/1432/20-ECAI-DAC.pdf> `_ at ECAI 2020

class dacbench.benchmarks.luby_benchmark.**LubyBenchmark**(*config_path=None*, *config=None*)

Bases: *AbstractBenchmark*

Benchmark with default configuration & relevant functions for Sigmoid

get_benchmark(*L=8*, *fuzziness=1.5*, *seed=0*)

Get Benchmark from DAC paper

Parameters

- **L** (*int*) – Minimum sequence lenght, was 8, 16 or 32 in the paper
- **fuzziness** (*float*) – Amount of noise applied. Was 1.5 for most of the experiments
- **seed** (*int*) – Environment seed

Returns
env – Luby environment

Return type
LubyEnv

get_environment()

Return Luby env with current configuration

Returns
Luby environment

Return type
LubyEnv

`read_instance_set`(*test=False*)

Read instance set from file

`set_cutoff`(*steps*)

Set cutoff and adapt dependencies

Parameters
int – Maximum number of steps

`set_history_length`(*length*)

Set history length and adapt dependencies

Parameters
int – History length

Luby environment from "Dynamic Algorithm Configuration:Foundation of a New Meta-Algorithmic Framework" by A. Biedenkapp and H. F. Bozkurt and T. Eimer and F. Hutter and M. Lindauer. Original environment authors: André Biedenkapp, H. Furkan Bozkurt

`class` dacbench.envs.luby.`LubyEnv`(*config*)

Bases: *AbstractEnv*

Environment to learn Luby Sequence

`close`() → bool

Close Env

Returns
Closing confirmation

Return type
bool

`render`(*mode: str = 'human'*) → None

Render env in human mode

Parameters
mode (*str*) – Execution mode

`reset`() → List[int]

Resets env

Returns
Environment state

Return type
numpy.array

`step`(*action: int*)

Execute environment step

Parameters
action (*int*) – action to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

dacbench.envs.luby.**luby_gen**(*i*)
Generator for the Luby Sequence

CHAPTER

SEVEN

THE GEOMETRIC TOY BENCHMARK

Task: aroximate values of different functions

Cost: normalized distance to actual values

Number of hyperparameters to control: user specified starting at one float

State Information: remaining budget, derivate of each function in the last step, actual value of each function in the last step

Noise Level: None

Instance space: a number of different function types and their instantiations (e.g. sigmoid or linear), correlation between the functions

This is an artifical benchmark using function approximation only. Its goal is to simulate the control of multiple hyperparameters that behave differently with possible correlations between dimensions. In each step, the DAC controller tries to approximate the true value of the function in each dimension. The difference between this prediction and the true value is the cost. There are different ways to accumulate this cost built into the benchmark, by default it is the nmalized sum of costs across all dimensions.

Controlling multiple hyperparameters is a hard problem and thus this fully controllable and cheap to run benchmark aims to provide an easy starting point. Through its flexible instance space and cost functions the difficulty can be scaled up slowly before transitioning to real-world benchmarks with multiple hyperparameters.

class dacbench.benchmarks.geometric_benchmark.**GeometricBenchmark**(*config_path=None*)

Bases: *AbstractBenchmark*

Benchmark with default configuration & relevant functions for Geometric

create_correlation_table()

Create correlation table from Config infos

get_benchmark(*dimension=None*, *seed=0*)

[summary]

Parameters

- **dimension** (*[type], optional*) – [description], by default None
- **seed** (*int, optional*) – [description], by default 0

Returns

[description]

Return type

[type]

get_environment()

Return Geometric env with current configuration

Returns
Geometric environment

Return type
GeometricEnv

read_instance_set()

Read instance set from file Creates a nested List for every Intance. The List contains all functions with their respective values.

set_action_description()

Add Information about Derivative and Coordinate to Description.

set_action_values()

Adapt action values and update dependencies Number of actions can differ between functions if configured in DefaultDict Set observation space args.

Geometric environment. Original environment authors: Rasmus von Glahn

class dacbench.envs.geometric.**GeometricEnv**(*config*)

Bases: *AbstractEnv*

Environment for tracing different curves that are orthogonal to each other Use product approach: f(t,x,y,z) = X(t,x) * Y(t,y) * Z(t,z) Normalize Function Value on a Scale between 0 and 1

- min and max value for normalization over all timesteps

close() → bool

Close Env

Returns
Closing confirmation

Return type
bool

get_default_reward(_) → float

Calculate euclidean distance between action vector and real position of Curve.

Parameters
_ (*self*) – ignore

Returns
Euclidean distance

Return type
float

get_default_state(_) → array

Gather state information.

Parameters
_ – ignore param

Returns
numpy array with state information

Return type
np.array

get_optimal_policy(*instance: Optional[List] = None*, *vector_action: bool = True*) → List[array]

Calculates the optimal policy for an instance

Parameters

- **instance** (*List, optional*) – instance with information about function config.
- **vector_action** (*bool, optional*) – if True return multidim actions else return onedimensional action, by default True

Returns

List with entry for each timestep that holds all optimal values in an array or as int

Return type

List[np.array]

render(*dimensions: List*, *absolute_path: str*)

Multiplot for specific dimensions of benchmark with policy actions.

Parameters

dimensions (*List*) – List of dimensions that get plotted

render_3d_dimensions(*dimensions: List*, *absolute_path: str*)

Plot 2 Dimensions in 3D space

Parameters

dimensions (*List*) – List of dimensions that get plotted. Max 2

reset() → List[int]

Resets env

Returns

Environment state

Return type

numpy.array

step(*action: int*)

Execute environment step

Parameters

action (*int*) – action to execute

Returns

state, reward, done, info

Return type

np.array, float, bool, dict

CHAPTER

EIGHT

THE FASTDOWNWARD BENCHMARK

Task: select heuristics for the FastDownward planner

Cost: number of optimization steps

Number of hyperparameters to control: one categorical

State Information: average value, max value, min value, number of open list entries and variance for each heuristic

Noise Level: fairly large

Instance space: either specifically desigd easy toy instances with ground truth or common planning competition instance sets

This benchmark is an interface to the Fast Downward AI planner, controlling its heuristic hyperparameter. In each step until the algorithm finishes or is terminated via the cutoff, the DAC controller selects one of either two (toy case) or four heuristiccs for the planner to use. The goal is to reduce the runtime of the planner, so every step that is taken in the benchmark incurs a cost of 1.

Out of our real-world benchmarks, FastDownward is likely the fastest running and it has been shown to be suitable to dynamic configuration. Though the noise level is fairly high, most DAC controllers should be able to learn functional policies in a comparatively short time frame.

The FastDownward benchmark was constructed by Speck et al. for the paper `"Learning Heuristic Selection with Dynamic Algorithm Configuration" <https://arxiv.org/pdf/2006.08246.pdf> `_ at ICAPS 2021

class dacbench.benchmarks.fast_downward_benchmark.**FastDownwardBenchmark**(*config_path=None*, *config=None*)

Bases: *AbstractBenchmark*

Benchmark with default configuration & relevant functions for Sigmoid

get_benchmark(*seed=0*)

Get published benchmark

Parameters
seed (*int*) – Environment seed

Returns
env – FD environment

Return type
FastDownwardEnv

get_environment()

Return Luby env with current configuration

Returns
Luby environment

Return type
LubyEnv

`read_instance_set`(*test=False*)

Read paths of instances from config into list

Planning environment from "Learning Heuristic Selection with Dynamic Algorithm Configuration" by David Speck, André Biedenkapp, Frank Hutter, Robert Mattmüller und Marius Lindauer. Original environment authors: David Speck, André Biedenkapp

`class` dacbench.envs.fast_downward.`FastDownwardEnv`(*config*)

Bases: *AbstractEnv*

Environment to control Solver Heuristics of FastDownward

`close`()

Close Env

Returns
Closing confirmation

Return type
bool

`kill_connection`()

Kill the connection

`recv_msg`()

Recieve a whole message. The message has to be prepended with its total size Based on comment from SO see [1]

Returns
The message as byte

Return type
bytes

`recvall`(*n: int*)

Given we know the size we want to recieve, we can recieve that amount of bytes. Based on comment from SO see [1]

Parameters
n (*int*) – Number of bytes to expect in the data

Returns
The message as byte

Return type
bytes

`render`(*mode: str = 'human'*) → None

Required by gym.Env but not implemented

Parameters
mode (*str*) – Rendering mode

`reset`()

Reset environment

Returns
State after reset

Return type
np.array

send_msg(*msg: bytes*)

Send message and prepend the message size

Based on comment from SO see [1] [1] https://stackoverflow.com/a/17668009

Parameters
msg (*bytes*) – The message as byte

step(*action: Union[int, List[int]]*)

Environment step

Parameters
action (*Union[int, List[int]]*) – Parameter(s) to apply

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

class dacbench.envs.fast_downward.**StateType**(*value*)

Bases: Enum

Class to define numbers for state types

CHAPTER

NINE

THE THEORY BENCHMARK

Task: controlling number of flips in RLS on LeadingOnes
Cost: number of iterations until solution
Number of hyperparameters to control: one float
State Information: user specified, highly flexible
Noise Level: fairly large
Instance space: different instantiations of LeadingOnes

This benchmark is considered one of our highly controllable ones as there is ground truth available. It is also, however, built on top of the RLS algorithm, so not an artificial benchmark. At each step, the DAC controller chooses how many solution bits to flip. We want to optimize how many algorithm steps are taken, so the number of iterations is the reward.

While this is not an easy to solve benchmark, it is cheap to run and interfaces a real EA. Thus it may be a good entry point for harder EA-based benchmarks and also a good benchmark for analyzing controller behavior.

The Theory benchmark was constructed by Biedenkapp et al. for the paper `"Theory-Inspired Parameter Control Benchmarks for Dynamic Algorithm Configuration" <https://arxiv.org/pdf/2202.03259.pdf> `_ at GECCO 2022

class dacbench.benchmarks.theory_benchmark.**TheoryBenchmark**(*config=None*)

Bases: *AbstractBenchmark*

Benchmark with various settings for (1+(lbd, lbd))-GA and RLS

create_observation_space_from_description(*obs_description*, *env_class=<class 'dacbench.envs.theory.RLSEnvDiscrete'>*)

Create a gym observation space (Box only) based on a string containing observation variable names, e.g. "n, f(x), k, k_{t-1}" Return:

A gym.spaces.Box observation space

get_environment(*test_env=False*)

Return an environment with current configuration

Parameters:

test_env: whether the enviroment is used for train an agent or for testing.

if test_env=False:
cutoff time for an episode is set to 0.8*n^2 (n: problem size) if an action is out of range, stop the episode immediately and return a large negative reward (see envs/theory.py for more details)

otherwise: benchmark's original cutoff time is used, and out-of-range action will be clipped to nearest valid value and the episode will continue.

read_instance_set()

Read instance set from file
we look at the current directory first, if the file doesn't exist, we look in <DACBench>/dacbench/instance_sets/theory/

class dacbench.envs.theory.**BinaryProblem**(*n*, *rng=Generator(PCG64) at 0x7F9822C538B8*)

Bases: object

An abstract class for an individual in binary representation

combine(*xprime*, *locs_xprime*)

combine (crossover) self and xprime by taking xprime's bits at locs_xprime and self's bits at other positions

Parameters

- **xprime** (*1d boolean array*) – the individual to crossover with
- **locs_x** (*1d boolean/integer array*) – positions where we keep current bits of self
- **locs_xprime** (*: 1d boolean/integer array*) – positions where we change to xprime's bits

Returns: the new individual after the crossover

crossover(*xprime*, *p*, *n_childs*, *include_xprime=True*, *count_different_inds_only=True*, *rng=Generator(PCG64) at 0x7F97C2ED16D8*)

Crossover operator:
for each bit, taking value from x with probability p and from self with probability 1-p

Arguments:
x: the individual to crossover with p (float): in [0,1]

flip(*locs*)

flip the bits at position indicated by locs

Parameters
locs (*1d-array*) – positions where bits are flipped

Returns: the new individual after the flip

get_fitness_after_crossover(*xprime*, *locs_x*, *locs_xprime*)

Calculate fitness of the child aftering being crossovered with xprime

Parameters

- **xprime** (*1d boolean array*) – the individual to crossover with
- **locs_x** (*1d boolean/integer array*) – positions where we keep current bits of self
- **locs_xprime** (*: 1d boolean/integer array*) – positions where we change to xprime's bits

get_fitness_after_flipping(*locs*)

Calculate the change in fitness after flipping the bits at positions locs

Parameters
locs (*1d-array*) – positions where bits are flipped

objective after flipping

mutate(*p*, *n_childs*, *rng=Generator(PCG64) at 0x7F97C2ED14F8*)

Draw l ~ binomial(n, p), l>0 Generate n_childs children by flipping exactly l bits Return: the best child (maximum fitness), its fitness and number of evaluations used

mutate_rls(*l*, *rng=Generator(PCG64) at 0x7F97C2ED15E8*)

generate a child by flipping exactly l bits Return: child, its fitness

class dacbench.envs.theory.**LeadingOne**(*n*, *rng=Generator(PCG64) at 0x7F97C2ED17C8*, *initObj=None*)

Bases: *BinaryProblem*

An individual for LeadingOne problem The aim is to maximise the number of leading (and consecutive) 1 bits in the string

get_fitness_after_crossover(*xprime*, *locs_x*, *locs_xprime*)

Calculate fitness of the child aftering being crossovered with xprime

Parameters

- **xprime** (*1d boolean array*) – the individual to crossover with
- **locs_x** (*1d boolean/integer array*) – positions where we keep current bits of self
- **locs_xprime** (*: 1d boolean/integer array*) – positions where we change to xprime's bits

get_fitness_after_flipping(*locs*)

Calculate the change in fitness after flipping the bits at positions locs

Parameters
locs (*1d-array*) – positions where bits are flipped

objective after flipping

class dacbench.envs.theory.**RLSEnv**(*config*, *test_env=False*)

Bases: *AbstractEnv*

Environment for RLS with step size Current assumption: we only consider (1+1)-RLS, so there's only one parameter to tune (r)

close() → bool

Close Env

No additional cleanup necessary

Returns
Closing confirmation

Return type
bool

get_obs_domain_from_name()

Get default lower and upperbound of a observation variable based on its name. The observation space will then be created Return:

Two int values, e.g., 1, np.inf

reset()

Resets env

Returns
Environment state

Return type
numpy.array

step(*action*)

Execute environment step

Parameters
action (*Box*) – action to execute

Returns

- *state, reward, done, info*
- *np.array, float, bool, dict*

class dacbench.envs.theory.**RLSEnvDiscrete**(*config*, *test_env=False*)

Bases: *RLSEnv*

RLS environment where the choices of r is discretised

step(*action*)

Execute environment step

Parameters
action (*Box*) – action to execute

Returns

- *state, reward, done, info*
- *np.array, float, bool, dict*

CHAPTER

TEN

THE PYCMA CMA-ES BENCHMARK

Task: control the step size of CMA-ES on BBOB functions

Cost: negative objective value

Number of hyperparameters to control: one float

State Information: current point, the last 40 objective values, population size, current step size, the deltas between the last 80 objective values, the deltas between the last 40 step sizes

Noise Level: fairly large, depends on target function

Instance space: the BBOB functions with ids, starting point and starting sigma as well as population size

This benchmark uses the PyCMA implementation to control the step size of the CMA-ES algorithm on the BBOB function set. The goal in the optimization is to find the global function minimum before the cutoff, so the cost is defined as the current negativ objective value.

The BBOB functions provide a varied instance space that is well suited for testing generalization capabilites of DAC methods. Due to this large instance space and very different scales of objective values (and thus cost), the CMA-ES benchmark is one of the more difficult to solve ones in DACBench.

The CMA-ES benchmark was constructed by Shala et al. for the paper `"Learning Step-size Adaptation in CMA-ES" <https://ml.informatik.uni-freiburg.de/wp-content/uploads/papers/20-PPSN-LTO-CMA.pdf> `_ at PPSN 2020

class dacbench.benchmarks.cma_benchmark.**CMAESBenchmark**(*config_path=None*, *config=None*)

Bases: *AbstractBenchmark*

Benchmark with default configuration & relevant functions for CMA-ES

get_benchmark(*seed=0*)

Get benchmark from the LTO paper

Parameters
seed (*int*) – Environment seed

Returns
env – CMAES environment

Return type
CMAESEnv

get_environment()

Return CMAESEnv env with current configuration

Returns
CMAES environment

Return type
CMAESEnv

read_instance_set(*test=False*)

Read path of instances from config into list

CMA-ES environment adapted from CMAWorld in "Learning Step-size Adaptation in CMA-ES" by G.Shala and A. Biedenkapp and N.Awad and S. Adriaensen and M.Lindauer and F. Hutter. Original author: Gresa Shala

class dacbench.envs.cma_es.**CMAESEnv**(*config*)

Bases: *AbstractEnv*

Environment to control the step size of CMA-ES

close()

No additional cleanup necessary

Returns
Cleanup flag

Return type
bool

get_default_reward(_)

Compute reward

Returns
Reward

Return type
float

get_default_state(_)

Gather state description

Returns
Environment state

Return type
dict

render(*mode: str = 'human'*)

Render env in human mode

Parameters
mode (*str*) – Execution mode

reset()

Reset environment

Returns
Environment state

Return type
np.array

step(*action*)

Execute environment step

Parameters
action (*list*) – action to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

CHAPTER

ELEVEN

THE MODEA BENCHMARK

Task: control the algorithm components of CMA-ES on BBOB functions
Cost: negative objective value
Number of hyperparameters to control: 11 categorical
State Information: generation size, step size, remaining budget, function ID, instance ID
Noise Level: fairly large, depends on target function
Instance space: the BBOB functions with ids, starting point and starting sigma as well as population size

This benchmark uses the ModEA package to enable dynamic control of several algorithm components of CMA-ES. The components of the algorithm that can be selected or changed are: sequential execution, active update, elitism, orthogonal sampling, convergence threshold enabled, step size adaption scheme, mirrored sampling, the base sampler, weight option, local restarts and bound correction. The goal in the optimization is to find the global function minimum before the cutoff, so the cost is defined as the current negativ objective value.

Just like the ModCMA benchmark (which provides a very similar problem with a different backend), this benchmark is challenging due to the large configuration space. It is an advanced benchmark that should likely not be the starting point for the development of DAC methods.

class dacbench.benchmarks.modea_benchmark.**ModeaBenchmark**(*config_path=None*, *config=None*)

Bases: *AbstractBenchmark*

Benchmark with default configuration & relevant functions for Modea

get_environment()

Return ModeaEnv env with current configuration

Returns
Modea environment

Return type
ModeaEnv

read_instance_set(*test=False*)

Read path of instances from config into list

class dacbench.envs.modea.**ModeaEnv**(*config*)

Bases: *AbstractEnv*

close()

Override close in your subclass to perform any necessary cleanup.

Environments will automatically close() themselves when garbage collected or when the program exits.

`ensureFullLengthRepresentation`(*representation*)

Given a (partial) representation, ensure that it is padded to become a full length customizedES representation, consisting of the required number of structure, population and parameter values. >>> ensureFullLengthRepresentation([]) [0,0,0,0,0,0,0,0,0,0,0, None,None, None,None,None,None,None,None,None,None,None,None,None,None,None] :param representation: List representation of a customizedES instance to check and pad if needed :return: Guaranteed full-length version of the representation

`reset`()

Reset environment

Returns
Environment state

Return type
state

`step`(*action*)

Execute environment step

Parameters
`action` – Action to take

Returns

- *state* – Environment state
- *reward* – Environment reward
- **done** (*bool*) – Run finished flag
- **info** (*dict*) – Additional metainfo

CHAPTER

TWELVE

THE IOHPROFILER MODCMA BENCHMARK

Task: control the step size or algorithm components of CMA-ES on BBOB functions
Cost: negative objective value
Number of hyperparameters to control: either one float or up to 11 categoricals
State Information: generation size, step size, remaining budget, function ID, instance ID
Noise Level: fairly large, depends on target function
Instance space: the BBOB functions with ids, starting point and starting sigma as well as population size

This benchmark is based on the IOHProfiler implementation of CMA-ES and enables both step size cool and algorithm component selection on the BBOB function set. The components of the algorithm that can be selected or changed are: sequential execution, active update, elitism, orthogonal sampling, convergence threshold enabled, step size adaption scheme, mirrored sampling, the base sampler, weight option, local restarts and bound correction. The goal in the optimization is to find the global function minimum before the cutoff, so the cost is defined as the current negativ objective value.

Both versions of this benchmark are challenging due to the large instance space, but the algorithm component control adds another layer of difficulty through its many configuration options. It is an advanced benchmark that should likely not be the starting point for the development of DAC methods.

class dacbench.benchmarks.modcma_benchmark.**ModCMABenchmark**(*config_path: Optional[str] = None*, *step_size=False*, *config=None*)

Bases: *AbstractBenchmark*

get_environment()

Make benchmark environment

Returns
env – Benchmark environment

Return type
gym.Env

class dacbench.envs.modcma.**ModCMAEnv**(*config*)

Bases: *AbstractEnv*

close()

Override close in your subclass to perform any necessary cleanup.

Environments will automatically close() themselves when garbage collected or when the program exits.

reset()

Reset environment

Returns
Environment state

Return type
state

`step`(*action*)

Execute environment step

Parameters
`action` – Action to take

Returns

- *state* – Environment state
- *reward* – Environment reward
- **done** (*bool*) – Run finished flag
- **info** (*dict*) – Additional metainfo

CHAPTER

THIRTEEN

THE SGD DEEP LEARNING BENCHMARK

Task: control the learning rate in deep learning

Cost: log differential validation loss

Number of hyperparameters to control: one float

State Information: predictive change variance, predictive change variance, loss variance, loss variance uncertainty, current learning rate, training loss, validation loss, step, alignment, crashed

Noise Level: fairly large

Instance space: dataset, network architecture, optimizer

Built on top of PyTorch, this benchmark allows for dynamic learning rate control in deep learning. At each step until the cutoff, i.e. after each epoch, the DAC controller provides a new learning rate value to the network. Success is measured by decreasing validation loss.

This is a very flexible benchmark, as in principle all kinds of classification datasets and PyTorch compatible architectures can be included in training. The underlying task is not easy, however, so we recommend starting with small networks and datasets and building up to harder tasks.

class dacbench.benchmarks.sgd_benchmark.**SGDBenchmark**(*config_path=None*, *config=None*)

Bases: *AbstractBenchmark*

Benchmark with default configuration & relevant functions for SGD

get_benchmark(*instance_set_path=None*, *seed=0*)

Get benchmark from the LTO paper

Parameters
seed (*int*) – Environment seed

Returns
env – SGD environment

Return type
SGDEnv

get_environment()

Return SGDEnv env with current configuration

Returns
SGD environment

Return type
SGDEnv

read_instance_set(*test=False*)

Read path of instances from config into list

class dacbench.envs.sgd.**Reward**(*value*)

Bases: IntEnum

An enumeration.

class dacbench.envs.sgd.**SGDEnv**(*config*)

Bases: *AbstractEnv*

Environment to control the learning rate of adam

close()

No additional cleanup necessary

Returns
Cleanup flag

Return type
bool

get_default_state(*_*)

Gather state description

Returns
Environment state

Return type
dict

render(*mode: str = 'human'*)

Render env in human mode

Parameters
mode (*str*) – Execution mode

reset()

Reset environment

Returns
Environment state

Return type
np.array

seed(*seed=None*, *seed_action_space=False*)

Set rng seed

Parameters

- **seed** – seed for rng
- **seed_action_space** (*bool, default False*) – if to seed the action space as well

step(*action*)

Execute environment step

Parameters
action (*list*) – action to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

`val_model`

Samuel Mueller (PhD student in our group) also uses backpack and has ran into a similar memory leak. He solved it calling this custom made RECURSIVE memory_cleanup function: # from backpack import memory_cleanup # def recursive_backpack_memory_cleanup(module: torch.nn.Module): # memory_cleanup(module) # for m in module.modules(): # memory_cleanup(m) (calling this after computing the training loss/gradients and after validation loss should suffice)

Type
TODO

CHAPTER

FOURTEEN

SAVING & LOADING BENCHMARK CONFIGURATIONS

While we encourage the use of the default benchmark settings, we recognize that our benchmarks are not perfect and can be improved upon. Therefore, it is possible to modify benchmarks and save these modifications to share with others.

To load a configuration shared with you, read it using the corresponding benchmark class:

```
from dacbench.benchmarks import SigmoidBenchmark

bench = SigmoidBenchmark()
bench.read_config_file("path/to/your/config.json")
modified_benchmark = bench.get_environment()
```

The get_environment() method overrides wth default configurations with your changes. That way you can directly modify the benchmarks:

```
from dacbench.benchmarks import SigmoidBenchmark

bench = SigmoidBenchmark()

# Increase episode length
bench.config.cutoff = 20
# Decrease slope multiplier
bench.config.slope_multiplier = 1.5

modified_benchmark = bench.get_environment()
```

To then save this configuration:

```
bench.save_config("your/path/config.json")
```

In case you want to modify state information, reward function or other complex benchmark attributes, be sure to adapt all dependencies in the configuration. Benchmarks have methods to do this for common changes like the number of dimensions in Sigmoid.

If any of your changes pass a function to the configuration, please be sure to provide the code for this function along with the configuration itself. If you want to save wrappers to the config (e.g. an instance sampling wrapper), you need to register them beforehand and also provide any functions that may serve as arguments.

CHAPTER

FIFTEEN

MODIFYING OBSERVATIONS & COST

While all benchmarks in DACBench come with a default option for both the reward function and what observations about the algorithm are shown to the DAC controller, both can be configured individually if needed. The standard way of doing this on most benchmark is to use the config to provide a function. A very simple example could look like this:

```
from dacbench.benchmarks import SigmoidBenchmark

def new_reward(env):
    if env.c_step % 2 == 0:
        return 1
    else:
        return 0

bench = SigmoidBenchmark()
bench.config.reward_function = new_reward
modified_benchmark = bench.get_environment()
```

The environment itself is provided as an argument, so all internal information can be used to get the reward. The same goes for the observations:

```
from dacbench.benchmarks import SigmoidBenchmark

def new_obs(env):
    return env.remaining_budget

bench = SigmoidBenchmark()
bench.config.state_method = new_obs
modified_benchmark = bench.get_environment()
```

If the config is logged, information about the updated functions is saved too, but for reusing this config, the code needs to be provided. That means anyone that want to run a setting with altered rewards and observations needs the config plus the code of the new functions. Therefore we advise to provide a file with only these functions in addition to the config - or make a PR to DACBench!

CHAPTER

SIXTEEN

FUNCTIONALITY THROUGH WRAPPERS

In order to comfortably provide additional functionality to environments without changing the interface, we can use so-called wrappers. They execute environment resets and steps internally, but can either alter the environment behavior (e.g. by adding noise) or record information about the environment. To wrap an existing environment is simple:

```
from dacbench.wrappers import PerformanceTrackingWrapper

wrapped_env = PerformanceTrackingWrapper(env)
```

The provided environments for tracking performance, state and action information are designed to be used with DACBench's logging functionality.

class dacbench.wrappers.**ActionFrequencyWrapper**(*env*, *action_interval=None*, *logger=None*)

Wrapper to action frequency. Includes interval mode that returns frequencies in lists of len(interval) instead of one long list.

get_actions()

Get state progression

Returns
all states or all states and interval sorted states

Return type
np.array or np.array, np.array

render_action_tracking()

Render action progression

Returns
RBG data of action tracking

Return type
np.array

step(*action*)

Execute environment step and record state

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

class dacbench.wrappers.**EpisodeTimeWrapper**(*env*, *time_interval=None*, *logger=None*)

Wrapper to track time spent per episode. Includes interval mode that returns times in lists of len(interval) instead of one long list.

get_times()

Get times

Returns
all times or all times and interval sorted times

Return type
np.array or np.array, np.array

render_episode_time()

Render episode times

render_step_time()

Render step times

step(*action*)

Execute environment step and record time

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

class dacbench.wrappers.**InstanceSamplingWrapper**(*env*, *sampling_function=None*, *instances=None*, *reset_interval=0*)

Wrapper to sample a new instance at a given time point. Instances can either be sampled using a given method or a distribution infered from a given list of instances.

fit_dist(*instances*)

Approximate instance distribution in given instance set

Parameters
instances (*List*) – instance set

Returns
sampling method for new instances

Return type
method

reset()

Reset environment and use sampled instance for training

Returns
state

Return type
np.array

class dacbench.wrappers.**ObservationWrapper**(*env*)

Wrapper covert observations spaces to spaces.Box for convenience Currently only supports Dict -> Box

reset()

Execute environment step and record distance

Returns
state

Return type
np.array

step(*action*)

Execute environment step and record distance

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

class dacbench.wrappers.**PerformanceTrackingWrapper**(*env*, *performance_interval=None*, *track_instance_performance=True*, *logger=None*)

Wrapper to track episode performance. Includes interval mode that returns performance in lists of len(interval) instead of one long list.

get_performance()

Get state performance

Returns
all states or all states and interval sorted states

Return type
np.array or np.array, np.array or np.array, dict or np.array, np.arry, dict

render_instance_performance()

Plot mean performance for each instance

render_performance()

Plot performance

step(*action*)

Execute environment step and record performance

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

class dacbench.wrappers.**PolicyProgressWrapper**(*env*, *compute_optimal*)

Wrapper to track progress towards optimal policy. Can only be used if a way to obtain the optimal policy given an instance can be obtained

render_policy_progress()

Plot progress

step(*action*)

Execute environment step and record distance

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

class dacbench.wrappers.**RewardNoiseWrapper**(*env*, *noise_function=None*, *noise_dist='standard_normal'*, *dist_args=None*)

Wrapper to add noise to the reward signal. Noise can be sampled from a custom distribution or any distribution in numpy's random module

add_noise(*dist*, *args*)

Make noise function from distribution name and arguments

Parameters

- **dist** (*str*) – Name of distribution
- **args** (*list*) – List of distribution arguments

Returns
Noise sampling function

Return type
function

step(*action*)

Execute environment step and add noise

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

class dacbench.wrappers.**StateTrackingWrapper**(*env*, *state_interval=None*, *logger=None*)

Wrapper to track state changed over time Includes interval mode that returns states in lists of len(interval) instead of one long list.

get_states()

Get state progression

Returns
all states or all states and interval sorted states

Return type
np.array or np.array, np.array

render_state_tracking()

Render state progression

Returns
RBG data of state tracking

Return type
np.array

`reset()`

Reset environment and record starting state

Returns
state

Return type
np.array

`step(action)`

Execute environment step and record state

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

CHAPTER

SEVENTEEN

LOGGING EXPERIMENTS

As there are many potentially interesting metrics involved in the analysis of DAC methods, DACBench includes the possibility to track and store them.

To log information on an environment, you need a logger object:

```
from dacbench.logger import Logger
from pathlib import Path

logger = Logger(experiment_name="example", output_path=Path("your/path"))
```

If you want to use any of our tracking wrappers, you can then create a logging module for them:

```
from dacbench.wrappers import PerformanceTrackingWrapper

performance_logger = logger.add_module(PerformanceTrackingWrapper)
env = PerformanceTrackingWrapper(env, logger=performance_logger)
logger.add_env()
```

Now the logger will store information in the specified directory in .jsonl files. By adding more wrappers, you will also be provided with more information. The stored data can then be loaded into pandas dataframes:

```
from dacbench.logger import load_logs, log2dataframe

logs = load_logs("your/path/PerformancyTrackingWrapper.jsonl")
df = log2dataframe(logs)
```

class dacbench.logger.**AbstractLogger**(*experiment_name: str*, *output_path: Path*, *step_write_frequency: Optional[int] = None*, *episode_write_frequency: int = 1*)

Logger interface.

The logger classes provide a way of writing structured logs as jsonl files and also help to track information like current episode, step, time ...

In the jsonl log file each row corresponds to a step.

abstract close() → None

Makes sure, that all remaining entries in the are written to file and the file is closed.

abstract log_dict(*data*)

Alternative to log if more the one value should be logged at once.

Parameters

data (*dict*) – a dict with key-value so that each value is a valid value for log

abstract log_space(*key: str*, *value: Union[ndarray, Dict]*, *space_info=None*)

Special for logging gym.spaces.

Currently three types are supported: * Numbers: e.g. samples from Discrete * Fixed length arrays like MultiDiscrete or Box * Dict: assuming each key has fixed length array

Parameters

- **key** – see log
- **value** – see log
- **space_info** – a list of column names. The length of this list must equal the resulting number of columns.

abstract next_episode() → None

Call at the end of episode.

See next_step

abstract next_step() → None

Call at the end of the step. Updates the internal state and dumps the information of the last step into a json

set_env(*env:* AbstractEnv) → None

Needed to infer automatically logged information like the instance id :param env: :type env: AbstractEnv

abstract write() → None

Writes buffered logs to file.

Invoke manually if you want to load logs during a run.

class dacbench.logger.**Logger**(*experiment_name: str*, *output_path: Path*, *step_write_frequency: Optional[int] = None*, *episode_write_frequency: int = 1*)

A logger that manages the creation of the module loggers.

To get a ModuleLogger for you module (e.g. wrapper) call module_logger = Logger(...).add_module("my_wrapper"). From now on module_logger.log(...) or logger.log(..., module="my_wrapper") can be used to log.

The logger module takes care of updating information like episode and step in the subloggers. To indicate to the loggers the end of the episode or the next_step simple call logger.next_episode() or logger.next_step().

add_agent(*agent:* AbstractDACBenchAgent)

Writes information about the agent

Parameters
agent (AbstractDACBenchAgent) –

add_benchmark(*benchmark:* AbstractBenchmark) → None

Writes the config to the experiment path :param benchmark:

add_module(*module: Union[str, type]*) → *ModuleLogger*

Creates a sub-logger. For more details see class level documentation :param module: The module name or Wrapper-Type to create a sub-logger for :type module: str or type

Returns

Return type
ModuleLogger

close()

Makes sure, that all remaining entries (from all sublogger) are written to files and the files are closed.

log_dict(*data*, *module*)

Alternative to log if more the one value should be logged at once.

Parameters
data (*dict*) – a dict with key-value so that each value is a valid value for log

log_space(*key*, *value*, *module*, *space_info=None*)

Special for logging gym.spaces.

Currently three types are supported: * Numbers: e.g. samples from Discrete * Fixed length arrays like MultiDiscrete or Box * Dict: assuming each key has fixed length array

Parameters

- **key** – see log
- **value** – see log
- **space_info** – a list of column names. The length of this list must equal the resulting number of columns.

next_episode()

Call at the end of episode.

See next_step

next_step()

Call at the end of the step. Updates the internal state of all subloggers and dumps the information of the last step into a json

set_env(*env:* AbstractEnv) → None

Needed to infer automatically logged information like the instance id :param env: :type env: AbstractEnv

write()

Writes buffered logs to file.

Invoke manually if you want to load logs during a run.

class dacbench.logger.**ModuleLogger**(*output_path: Path*, *experiment_name: str*, *module: str*, *step_write_frequency: Optional[int] = None*, *episode_write_frequency: int = 1*)

A logger for handling logging of one module. e.g. a wrapper or toplevel general logging.

Don't create manually use Logger to manage ModuleLoggers

close()

Makes sure, that all remaining entries in the are written to file and the file is closed.

get_logfile() → Path

Returns
the path to the log file of this logger

Return type
pathlib.Path

log_dict(*data: Dict*) → None

Alternative to log if more the one value should be logged at once.

Parameters
data (*dict*) – a dict with key-value so that each value is a valid value for log

log_space(*key*, *value*, *space_info=None*)

Special for logging gym.spaces.

Currently three types are supported: * Numbers: e.g. samples from Discrete * Fixed length arrays like MultiDiscrete or Box * Dict: assuming each key has fixed length array

Parameters

- **key** – see log
- **value** – see log
- **space_info** – a list of column names. The length of this list must equal the resulting number of columns.

next_episode()

Writes buffered logs to file.

Invoke manually if you want to load logs during a run.

next_step()

Call at the end of the step. Updates the internal state and dumps the information of the last step into a json

reset_episode() → None

Resets the episode and step.

Be aware that this can lead to ambitious keys if no instance or seed or other identifying additional info is set

Returns

set_additional_info(***kwargs*)

Can be used to log additional information for each step e.g. for seed, and instance id. :param kwargs:

write()

Writes buffered logs to file.

Invoke manually if you want to load logs during a run.

dacbench.logger.**flatten_log_entry**(*log_entry: Dict*) → List[Dict]

Transforms a log entry of format like

{

'step': 0, 'episode': 2, 'some_value': {

'values' : [34, 45], 'times':['28-12-20 16:20:53', '28-12-20 16:21:30'],

}

} into [

{ 'step': 0,'episode': 2, 'value': 34, 'time': '28-12-20 16:20:53'}, { 'step': 0,'episode': 2, 'value': 45, 'time': '28-12-20 16:21:30'}

]

Parameters
log_entry (*Dict*) – A log entry

dacbench.logger.**list_to_tuple**(*list_: List*) → Tuple

Recursively transforms a list of lists into tuples of tuples :param **list_**: (nested) list

Returns

Return type
(nested) tuple

dacbench.logger.**load_logs**(*log_file: Path*) → List[Dict]

Loads the logs from a jsonl written by any logger.

The result is the list of dicts in the format: {

'instance': 0, 'episode': 0, 'step': 1, 'example_log_val': {

'values': [val1, val2, ... valn], 'times: [time1, time2, ..., timen],

} :param log_file: The path to the log file :type log_file: pathlib.Path

Returns

Return type
[Dict, ...]

dacbench.logger.**log2dataframe**(*logs: List[dict]*, *wide: bool = False*, *drop_columns: List[str] = ['time']*) → DataFrame

Converts a list of log entries to a pandas dataframe.

Usually used in combination with load_dataframe.

Parameters

- **logs** (*List*) – List of log entries
- **wide** (*bool*) – wide=False (default) produces a dataframe with columns (episode, step, time, name, value) wide=True returns a dataframe (episode, step, time, name_1, name_2, ...) if the variable name_n has not been logged at (episode, step, time) name_n is NaN.
- **drop_columns** (*List[str]*) – List of column names to be dropped (before reshaping the long dataframe) mostly used in combination with wide=True to reduce NaN values

Returns

Return type
dataframe

dacbench.logger.**split**(*predicate: Callable*, *iterable: Iterable*) → Tuple[List, List]

Splits the iterable into two list depending on the result of predicate.

Parameters

- **predicate** (*Callable*) – A function taking an element of the iterable and return Ture or False
- **iterable** (*Iterable*) –

Returns

Return type
(positives, negatives)

CHAPTER

EIGHTEEN

PLOTTING RESULTS

To immediately plot data stored with DACBench wrappers, you can use the built-in plotting functions. They use seaborn and format loaded dataframes automatically (see examples on GitHub).

dacbench.plotting.**add_multi_level_ticks**(*grid: FacetGrid*, *plot_index: DataFrame*, *x_column: str*, *x_label_columns: str*) → None

Expects a FacedGrid with global_step (x_column) as x-axis and replaces the tick labels to match format episode:step

E.g. Run with 3 episodes, each of 10 steps. This results in 30 global steps. The resulting tick labels could be [‘0’, ‘4’, ‘9’, ‘14’, ‘19’, ‘24’, ‘29’]. After applying this method they will look like [‘0:0’, ‘0:4’, ‘1:0’, ‘1:4’, ‘2:0’, ‘2:4’, ‘3:0’, ‘3:4’]

Parameters

- **grid** (*sns.FacesGrid*) –
- **plot_index** (*pd.DataFrame*) – The mapping between current tick labels (global step values) and new tick labels joined by ‘:’. usually the result from generate_global_step
- **x_column** (*str*) – column label to use for looking up tick values
- **x_label_columns** (*[str, ...]*) – columns labels of columns to use for new labels (joined by ‘:’

dacbench.plotting.**generate_global_step**(*data: DataFrame*, *x_column: str = 'global_step'*, *x_label_columns: str = ['episode', 'step']*) → Tuple[DataFrame, str, List[str]]

Add a global_step column which enumerate all step over all episodes.

Returns the altered data, a data frame containing mapping between global_step, x_column and x_label_columns.

Often used in combination with add_multi_level_ticks.

Parameters

- **data** –
- **x_column** (*str*) – the name of the global_step (default ‘global_step’)
- **x_label_columns** (*[str, ...]*) – the name and hierarchical order of the columns (default [‘episode’, ‘step’]

Returns

Return type
(data, plot_index, x_column, x_label_columns)

dacbench.plotting.**plot**(*plot_function*, *settings: dict*, *title: Optional[str] = None*, *x_label: Optional[str] = None*, *y_label: Optional[str] = None*, ***kwargs*) → FacetGrid

Helper function that: create a FacetGrid 1. Updates settings with kwargs (overwrites values) 2. Plots using plot_function(**settings) 3. Set x and y labels of not provided the columns names will converted to pretty strings using space_sep_upper 4. Sets title (some times has to be readjusted afterwards especially in case of large plots e.g. multiple rows/cols)

Parameters

- **plot_function** – function to generate the FacedGrid. E.g. sns.catplot or sns.catplot
- **settings** (*dict*) – a dicts containing all needed default settings.
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.plotting.**plot_action**(*data*, *show_global_step=False*, *interval=1*, *title=None*, *x_label=None*, *y_label=None*, ***kargs*)

Create a line plot showing actions over time.

Please be aware that action spaces can be quite large and the plots can become quite messy (and take some time) if you try plot all dimensions at once. It is therefore recommended to select a subset of columns before running the plot method.

Per default the mean performance and and one stddev over all instances and seeds is shown if you want to change this specify a property to map those attributes to e.g hue='seed' or/and col='instance'. For more details see: https://seaborn.pydata.org/generated/seaborn.relplot.html

For examples refer to examples/plotting/action_plotting.py

Parameters

- **data** (*pd.DataFrame*) – Dataframe resulting from logging and loading using log2dataframe(logs, wide=True)
- **show_global_step** (*bool*) – If to show the global_step (step enumerated over all episodes) or Episode:Step. (False default)
- **interval** (*int*) – Interval in number of steps to average over. (default = 1)
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.plotting.**plot_episode_time**(*data*, *title=None*, *x_label=None*, *y_label=None*, ***kargs*) → FacetGrid

Create a line plot showing the measured time per episode.

Per default the mean performance and and one stddev over all instances and seeds is shown if you want to change this specify a property to map those attributes to e.g hue='seed' or/and col='instance'. For more details see: https://seaborn.pydata.org/generated/seaborn.relplot.html

For examples refer to examples/plotting/time_plotting.py

Parameters

- **data** (*pd.DataFrame*) – Dataframe resulting from logging and loading using log2dataframe(logs, wide=True)
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.plotting.**plot_performance**(*data*, *title=None*, *x_label=None*, *y_label=None*, ***kwargs*) → FacetGrid

Create a line plot of the performance over episodes.

Per default the mean performance and and one stddev over all instances and seeds is shown if you want to change this specify a property to map those attributes to e.g hue='seed' or/and col='instance'. For more details see: https://seaborn.pydata.org/generated/seaborn.relplot.html

For examples refer to examples/plotting/performance_plotting.py

Parameters

- **data** (*pd.DataFrame*) – Dataframe resulting from logging and loading using log2dataframe(logs, wide=True)
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.plotting.**plot_performance_per_instance**(*data*, *title=None*, *x_label=None*, *y_label=None*, ***args*) → FacetGrid

Create a bar plot of the mean performance per instance ordered by the performance.

Per default the mean performance seeds is shown if you want to change this specify a property to map seed to e.g. col='seed'. For more details see: https://seaborn.pydata.org/generated/seaborn.catplot.html

For examples refer to examples/plotting/performance_plotting.py

Parameters

- **data** (*pd.DataFrame*) – Dataframe resulting from logging and loading using log2dataframe(logs, wide=True)
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.plotting.**plot_space**(*data*, *space_column_name*, *show_global_step*, *interval=1*, *title=None*, *x_label=None*, *y_label=None*, ***args*) → FacetGrid

Create a line plot showing sapce over time.

Please be aware that spaces can be quite large and the plots can become quite messy (and take some time) if you try plot all dimensions at once. It is therefore recommended to select a subset of columns before running the plot method. Especially for dict spaces.

Per default the mean performance and and one stddev over all instances and seeds is shown if you want to change this specify a property to map those attributes to e.g hue='seed' or/and col='instance'. For more details see: https://seaborn.pydata.org/generated/seaborn.relplot.html

For examples refer to
examples/plotting/state_plotting.py or examples/plotting/action_plotting.py

Parameters

- **data** (*pd.DataFrame*) – Dataframe resulting from logging and loading using log2dataframe(logs, wide=True)
- **show_global_step** (*bool*) – If to show the global_step (step enumerated over all episodes) or Episode:Step. (False default)
- **interval** (*int*) – Interval in number of steps to average over. (default = 1)
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.plotting.**plot_state**(*data*, *show_global_step=False*, *interval=1*, *title=None*, *x_label=None*, *y_label=None*, ***kargs*)

Create a line plot showing state over time.

Please be aware that state can be quite large and the plots can become quite messy (and take some time) if you try plot all dimensions at once. It is therefore recommended to select a subset of columns before running the plot method. Especially for dict state spaces.

Per default the mean performance and and one stddev over all instances and seeds is shown if you want to change this specify a property to map those attributes to e.g hue='seed' or/and col='instance'. For more details see: https://seaborn.pydata.org/generated/seaborn.relplot.html

For examples refer to examples/plotting/state_plotting.py

Parameters

- **data** (*pd.DataFrame*) – Dataframe resulting from logging and loading using log2dataframe(logs, wide=True)
- **show_global_step** (*bool*) – If to show the global_step (step enumerated over all episodes) or Episode:Step. (False default)
- **interval** (*int*) – Interval in number of steps to average over. (default = 1)
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.plotting.**plot_step_time**(*data*, *show_global_step=False*, *interval=1*, *title=None*, *x_label=None*, *y_label=None*, ***args*) → FacetGrid

Create a line plot showing the measured time per step.

Per default the mean performance and and one stddev over all instances and seeds is shown if you want to change this specify a property to map those attributes to e.g hue='seed' or/and col='instance'. For more details see: https://seaborn.pydata.org/generated/seaborn.relplot.html

For examples refer to examples/plotting/time_plotting.py

Parameters

- **data** (*pd.DataFrame*) – Dataframe resulting from logging and loading using log2dataframe(logs, wide=True)
- **show_global_step** (*bool*) – If to show the global_step (step enumerated over all episodes) or Episode:Step. (False default)
- **interval** (*int*) – Interval in number of steps to average over. (default = 1)
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.plotting.**space_sep_upper**(*column_name: str*) → str

Separates strings at underscores into headings. Used to generate labels from logging names.

Parameters
column_name (*str*) –

Returns

Return type
str

CHAPTER

NINETEEN

CONTRIBUTING TO DACBENCH

DACBench is an open-source collaborative project. Since its conception, we have had several valueable contributions and appreciate everyone who wants to make DACBench bigger and better for the community. This document can be a guide on how to get started contributing to DACBench.

In general, there are many ways you can help improve DACBench, including:

- Contributing new benchmarks
- Extending existing benchmarks (by adding e.g. more hyperparameters, extending the state information or providing interesting instances)
- Maining the code & fixing bugs
- Improving the documentation

For most of these, the existing issues should give you an idea where to start. If something is missing or not working for you, consider opening an issue on it, especially if it can't be fixed within a few minutes. Issues are also a good place to request new features and extensions, so don't hesitate to create one.

19.1 Guidelines for Pull-Requests

Code contributions are best made through pull-requests. In order to make the integration as easy as possible, we ask that you follow a few steps:

1. Please describe the changes you made in the PR clearly. This makes reviewing much faster and avoids misunderstandings
2. Run our tests and ideally also test coverage before submitting so your PR doesn't accidentally introduce new errors. You can use pytest for both of these, to only test, run from the top level DACBench dir:

```
pytest tests
```

For tests and test coverage:

```
pytest --cov=dacbench --cov-report html tests
```

3. If you install the 'dev' extras of DACBench, you should have flake8 and the code formatting tool black setup in a pre-commit hook. Both ensure consistent code quality, so ensure that the format is correct.
4. If you make larger changes to the docs, please build them locally using Sphinx. If you're not familiar with the tool, you can find a guide here: https://docs.readthedocs.io/en/stable/intro/getting-started-with-sphinx.html

19.2 Adding a Benchmark

Adding a benchmark can be quite the project depending on the algorithms it's based on. Therefore you can always contact us via e-mail or through the issues to get assistance.

In general, there are several steps to take:

1. Write an environment file This is where the functionality of your benchmark goes, in a subclass of "AbstractEnv". Especially the "reset" and "step" functions are important as they start a run and execute the next step respectively. Additionally, you should create a default reward function and a function to get the default observations. We recommend using one of the simple toy environments as a model, e.g. the Sigmoid environment file. Don't forget to enable seeding of as much of the target algorithm as possible! You should be able to use the numpy generators provided in 'dacbench.abstract_env.AbstractEnv' the instead of 'np.random' for source of randomness and the provided seeding function in most cases. If you need custom source of randomness e.g. for pytorch, please override the seeding function in your environment.

2. Write a benchmark file This is where you specify available options for your environment, in a subclass of "AbstractBenchmark". That includes options for the observation_space, reward range and action space. At least one for each of these is mandatory, if you include multiple options please make sure that you also specify how to switch between them in the environment, e.g. by adding a variable for this. Please also make sure to include a maximum number of steps per episode. To use some specific wrappers, additional information is required. An example is the progress tracking wrapper, for which either an optimal policy for each instance or a way to compute it has to be specified. The current benchmark classes should give you an idea how detailed these options should be. We enourage you to provide as many possibilities to modify the benchmark as possible in order to make use of it in different scenarios, but documenting these options is important to keep the benchmark usable for others. Again we recommend you take a look at e.g. the SigmoidBenchmark file to see how such a structure might look.

3. Provide an instance set (or a way to sample one) Instances are of course important for running the benchmark. The standard way of using them in DACBench is to read an instance set from file. This is not mandatory, however! You can define an instance sampling method to work with our instance sampling wrapper to generate instances every episode or you can sample the instance set once before creating the environment. How exactly you deal with instances should be specified in the benchmark class when creating the environment. An example for the instance sampling wrapper can be found in the SigmoidBenchmark class. Even if you provide an instance set, also making ways of sampling new instance sets possible would of course be helpful to other users. You can furthermore provide more than one instance set, in that case it would be a good idea to label them, e.g. "wide_instance_dist" or "east_instances".

4. Add an example use case & test cases To make the new benchmark accessible to everyone, please provide a small example of training an optimizer on it. It can be short, but it should show any special cases (e.g. CMA uses a dictionary as state representation, therefore the example shows a way to flatten it into an array). Additionally, please provide test cases for your benchmark class to ensure the environment is created properly and methods like reading the instance set work as they should.

5. Submit a pull request Once everything is working, we would be grateful if you want to share the benchmark! Submit a pull request on GitHub in which you briefly describe the benchmark and why it is interesting. The top of the page includes some technical things to pay attention to before submitting. Feel free to include details on how it was modelled and please cite the source if the benchmark uses existing code.

Thank you for your contributions!

CHAPTER

TWENTY

CITING DACBENCH

If you use DACBench your research, please cite us with the following Bibtex file:

```
@inproceedings{eimer-ijcai21,
        author    = {T. Eimer and A. Biedenkapp and M. Reimer and S. Adriaensen and F.␣
↪Hutter and M. Lindauer},
        title     = {DACBench: A Benchmark Library for Dynamic Algorithm Configuration},
        booktitle = {Proceedings of the Thirtieth International Joint Conference on␣
↪Artificial Intelligence ({IJCAI}'21)},
        year      = {2021},
        month     = aug,
        publisher = {ijcai.org}
}
```

CHAPTER

TWENTYONE

API REFERENCE

This page contains auto-generated API reference documentation[1].

21.1 dacbench

DACBench: a benchmark library for Dynamic Algorithm Configuration

21.1.1 Subpackages

dacbench.agents

Submodules

dacbench.agents.dynamic_random_agent

Module Contents

Classes

DynamicRandomAgent	Abstract class to implement for use with the runner function

class dacbench.agents.dynamic_random_agent.**DynamicRandomAgent**(*env*, *switching_interval*)

Bases: *dacbench.abstract_agent.AbstractDACBenchAgent*

Abstract class to implement for use with the runner function

act(*self*, *state*, *reward*)

Compute and return environment action

Parameters

- **state** – Environment state
- **reward** – Environment reward

Returns

Action to take

[1] Created with sphinx-autoapi

Return type
action

train(*self*, *next_state*, *reward*)

Train during episode if needed (pass if not)

Parameters

- **next_state** – Environment state after step
- **reward** – Environment reward

end_episode(*self*, *state*, *reward*)

End of episode training if needed (pass if not)

Parameters

- **state** – Environment state
- **reward** – Environment reward

dacbench.agents.generic_agent

Module Contents

Classes

GenericAgent	Abstract class to implement for use with the runner function

class dacbench.agents.generic_agent.**GenericAgent**(*env*, *policy*)

Bases: *dacbench.abstract_agent.AbstractDACBenchAgent*

Abstract class to implement for use with the runner function

act(*self*, *state*, *reward*)

Compute and return environment action

Parameters

- **state** – Environment state
- **reward** – Environment reward

Returns
Action to take

Return type
action

train(*self*, *next_state*, *reward*)

Train during episode if needed (pass if not)

Parameters

- **next_state** – Environment state after step
- **reward** – Environment reward

end_episode(*self*, *state*, *reward*)

End of episode training if needed (pass if not)

Parameters

- **state** – Environment state
- **reward** – Environment reward

dacbench.agents.simple_agents

Module Contents

Classes

RandomAgent	Abstract class to implement for use with the runner function
StaticAgent	Abstract class to implement for use with the runner function

class dacbench.agents.simple_agents.**RandomAgent**(*env*)

Bases: *dacbench.abstract_agent.AbstractDACBenchAgent*

Abstract class to implement for use with the runner function

act(*self*, *state*, *reward*)

Compute and return environment action

Parameters

- **state** – Environment state
- **reward** – Environment reward

Returns
Action to take

Return type
action

train(*self*, *next_state*, *reward*)

Train during episode if needed (pass if not)

Parameters

- **next_state** – Environment state after step
- **reward** – Environment reward

end_episode(*self*, *state*, *reward*)

End of episode training if needed (pass if not)

Parameters

- **state** – Environment state
- **reward** – Environment reward

class dacbench.agents.simple_agents.**StaticAgent**(*env*, *action*)

Bases: *dacbench.abstract_agent.AbstractDACBenchAgent*

Abstract class to implement for use with the runner function

act(*self*, *state*, *reward*)

Compute and return environment action

Parameters

- **state** – Environment state
- **reward** – Environment reward

Returns
Action to take

Return type
action

train(*self*, *next_state*, *reward*)

Train during episode if needed (pass if not)

Parameters

- **next_state** – Environment state after step
- **reward** – Environment reward

end_episode(*self*, *state*, *reward*)

End of episode training if needed (pass if not)

Parameters

- **state** – Environment state
- **reward** – Environment reward

Package Contents

Classes

GenericAgent	Abstract class to implement for use with the runner function
RandomAgent	Abstract class to implement for use with the runner function
StaticAgent	Abstract class to implement for use with the runner function
DynamicRandomAgent	Abstract class to implement for use with the runner function

class dacbench.agents.**GenericAgent**(*env*, *policy*)

Bases: *dacbench.abstract_agent.AbstractDACBenchAgent*

Abstract class to implement for use with the runner function

act(*self*, *state*, *reward*)

Compute and return environment action

Parameters

- **state** – Environment state
- **reward** – Environment reward

Returns
Action to take

Return type
action

train(*self*, *next_state*, *reward*)

Train during episode if needed (pass if not)

Parameters

- **next_state** – Environment state after step
- **reward** – Environment reward

end_episode(*self*, *state*, *reward*)

End of episode training if needed (pass if not)

Parameters

- **state** – Environment state
- **reward** – Environment reward

class dacbench.agents.**RandomAgent**(*env*)

Bases: *dacbench.abstract_agent.AbstractDACBenchAgent*

Abstract class to implement for use with the runner function

act(*self*, *state*, *reward*)

Compute and return environment action

Parameters

- **state** – Environment state
- **reward** – Environment reward

Returns
Action to take

Return type
action

train(*self*, *next_state*, *reward*)

Train during episode if needed (pass if not)

Parameters

- **next_state** – Environment state after step
- **reward** – Environment reward

end_episode(*self*, *state*, *reward*)

End of episode training if needed (pass if not)

Parameters

- **state** – Environment state
- **reward** – Environment reward

class dacbench.agents.**StaticAgent**(*env*, *action*)

Bases: *dacbench.abstract_agent.AbstractDACBenchAgent*

Abstract class to implement for use with the runner function

act(*self*, *state*, *reward*)

Compute and return environment action

Parameters

- **state** – Environment state
- **reward** – Environment reward

Returns
Action to take

Return type
action

train(*self*, *next_state*, *reward*)

Train during episode if needed (pass if not)

Parameters

- **next_state** – Environment state after step
- **reward** – Environment reward

end_episode(*self*, *state*, *reward*)

End of episode training if needed (pass if not)

Parameters

- **state** – Environment state
- **reward** – Environment reward

class dacbench.agents.**DynamicRandomAgent**(*env*, *switching_interval*)

Bases: *dacbench.abstract_agent.AbstractDACBenchAgent*

Abstract class to implement for use with the runner function

act(*self*, *state*, *reward*)

Compute and return environment action

Parameters

- **state** – Environment state
- **reward** – Environment reward

Returns
Action to take

Return type
action

train(*self*, *next_state*, *reward*)

Train during episode if needed (pass if not)

Parameters

- **next_state** – Environment state after step
- **reward** – Environment reward

end_episode(*self*, *state*, *reward*)

End of episode training if needed (pass if not)

Parameters

- **state** – Environment state
- **reward** – Environment reward

dacbench.benchmarks

Submodules

dacbench.benchmarks.cma_benchmark

Module Contents

Classes

CMAESBenchmark	Benchmark with default configuration & relevant functions for CMA-ES

dacbench.benchmarks.cma_benchmark.**HISTORY_LENGTH = 40**

dacbench.benchmarks.cma_benchmark.**INPUT_DIM = 10**

dacbench.benchmarks.cma_benchmark.**DEFAULT_CFG_SPACE**

dacbench.benchmarks.cma_benchmark.**STEP_SIZE**

dacbench.benchmarks.cma_benchmark.**INFO**

dacbench.benchmarks.cma_benchmark.**CMAES_DEFAULTS**

class dacbench.benchmarks.cma_benchmark.**CMAESBenchmark**(*config_path=None*, *config=None*)

Bases: *dacbench.abstract_benchmark.AbstractBenchmark*

Benchmark with default configuration & relevant functions for CMA-ES

get_environment(*self*)

Return CMAESEnv env with current configuration

Returns
CMAES environment

Return type
CMAESEnv

read_instance_set(*self*, *test=False*)

Read path of instances from config into list

get_benchmark(*self*, *seed=0*)

Get benchmark from the LTO paper

Parameters
seed (*int*) – Environment seed

Returns
env – CMAES environment

Return type
CMAESEnv

dacbench.benchmarks.fast_downward_benchmark

Module Contents

Classes

FastDownwardBenchmark	Benchmark with default configuration & relevant functions for Sigmoid

dacbench.benchmarks.fast_downward_benchmark.**HEURISTICS** = ['tiebreaking([pdb(pattern=manual_pattern([0,1])),weight(g(),-1)])', 'tiebreaking([pdb(pattern=manual_pattern([0,2])),weight(g(),-1)])']

dacbench.benchmarks.fast_downward_benchmark.**DEFAULT_CFG_SPACE**

dacbench.benchmarks.fast_downward_benchmark.**HEURISTIC**

dacbench.benchmarks.fast_downward_benchmark.**INFO**

dacbench.benchmarks.fast_downward_benchmark.**FD_DEFAULTS**

class dacbench.benchmarks.fast_downward_benchmark.**FastDownwardBenchmark**(*config_path=None*, *config=None*)

Bases: *dacbench.abstract_benchmark.AbstractBenchmark*

Benchmark with default configuration & relevant functions for Sigmoid

get_environment(*self*)

Return Luby env with current configuration

Returns
Luby environment

Return type
LubyEnv

read_instance_set(*self*, *test=False*)

Read paths of instances from config into list

set_heuristics(*self*, *heuristics*)

get_benchmark(*self*, *seed=0*)

Get published benchmark

Parameters
seed (*int*) – Environment seed

Returns
env – FD environment

Return type
FastDownwardEnv

dacbench.benchmarks.geometric_benchmark

Module Contents

Classes

GeometricBenchmark	Benchmark with default configuration & relevant functions for Geometric

dacbench.benchmarks.geometric_benchmark.**FILE_PATH**

dacbench.benchmarks.geometric_benchmark.**ACTION_VALUES = [5, 10]**

dacbench.benchmarks.geometric_benchmark.**DEFAULT_CFG_SPACE**

dacbench.benchmarks.geometric_benchmark.**INFO**

dacbench.benchmarks.geometric_benchmark.**GEOMETRIC_DEFAULTS**

class dacbench.benchmarks.geometric_benchmark.**GeometricBenchmark**(*config_path=None*)

Bases: *dacbench.abstract_benchmark.AbstractBenchmark*

Benchmark with default configuration & relevant functions for Geometric

get_environment(*self*)

Return Geometric env with current configuration

Returns
Geometric environment

Return type
GeometricEnv

read_instance_set(*self*)

Read instance set from file Creates a nested List for every Intance. The List contains all functions with their respective values.

get_benchmark(*self*, *dimension=None*, *seed=0*)

[summary]

Parameters

- **dimension** (*[type], optional*) – [description], by default None
- **seed** (*int, optional*) – [description], by default 0

Returns
[description]

Return type
[type]

`set_action_values`(*self*)

Adapt action values and update dependencies Number of actions can differ between functions if configured in DefaultDict Set observation space args.

`set_action_description`(*self*)

Add Information about Derivative and Coordinate to Description.

`create_correlation_table`(*self*)

Create correlation table from Config infos

dacbench.benchmarks.geometric_benchmark.**geo_bench**

dacbench.benchmarks.luby_benchmark

Module Contents

Classes

LubyBenchmark	Benchmark with default configuration & relevant functions for Sigmoid

dacbench.benchmarks.luby_benchmark.**MAX_STEPS**

dacbench.benchmarks.luby_benchmark.**LUBY_SEQUENCE**

dacbench.benchmarks.luby_benchmark.**HISTORY_LENGTH = 5**

dacbench.benchmarks.luby_benchmark.**DEFAULT_CFG_SPACE**

dacbench.benchmarks.luby_benchmark.**SEQ**

dacbench.benchmarks.luby_benchmark.**INFO**

dacbench.benchmarks.luby_benchmark.**LUBY_DEFAULTS**

class dacbench.benchmarks.luby_benchmark.**LubyBenchmark**(*config_path=None*, *config=None*)

Bases: *dacbench.abstract_benchmark.AbstractBenchmark*

Benchmark with default configuration & relevant functions for Sigmoid

`get_environment`(*self*)

Return Luby env with current configuration

Returns
Luby environment

Return type
LubyEnv

set_cutoff(*self*, *steps*)

Set cutoff and adapt dependencies

Parameters
int – Maximum number of steps

set_history_length(*self*, *length*)

Set history length and adapt dependencies

Parameters
int – History length

read_instance_set(*self*, *test=False*)

Read instance set from file

get_benchmark(*self*, *L=8*, *fuzziness=1.5*, *seed=0*)

Get Benchmark from DAC paper

Parameters

- **L** (*int*) – Minimum sequence lenght, was 8, 16 or 32 in the paper
- **fuzziness** (*float*) – Amount of noise applied. Was 1.5 for most of the experiments
- **seed** (*int*) – Environment seed

Returns
env – Luby environment

Return type
LubyEnv

dacbench.benchmarks.modcma_benchmark

Module Contents

Classes

ModCMABenchmark	Abstract template for benchmark classes

dacbench.benchmarks.modcma_benchmark.**DEFAULT_CFG_SPACE**

dacbench.benchmarks.modcma_benchmark.**ACTIVE**

dacbench.benchmarks.modcma_benchmark.**ELITIST**

dacbench.benchmarks.modcma_benchmark.**ORTHOGONAL**

dacbench.benchmarks.modcma_benchmark.**SEQUENTIAL**

dacbench.benchmarks.modcma_benchmark.**THRESHOLD_CONVERGENCE**

dacbench.benchmarks.modcma_benchmark.**STEP_SIZE_ADAPTION**

dacbench.benchmarks.modcma_benchmark.**MIRRORED**

dacbench.benchmarks.modcma_benchmark.**BASE_SAMPLER**

dacbench.benchmarks.modcma_benchmark.**WEIGHTS_OPTION**

dacbench.benchmarks.modcma_benchmark.**LOCAL_RESTART**

dacbench.benchmarks.modcma_benchmark.**BOUND_CORRECTION**

dacbench.benchmarks.modcma_benchmark.**INFO**

dacbench.benchmarks.modcma_benchmark.**MODCMA_DEFAULTS**

class dacbench.benchmarks.modcma_benchmark.**ModCMABenchmark**(*config_path: str = None*, *step_size=False*, *config=None*)

Bases: *dacbench.abstract_benchmark.AbstractBenchmark*

Abstract template for benchmark classes

get_environment(*self*)

Make benchmark environment

Returns
env – Benchmark environment

Return type
gym.Env

read_instance_set(*self*, *test=False*)

get_benchmark(*self*, *seed: int = 0*)

dacbench.benchmarks.modea_benchmark

Module Contents

Classes

ModeaBenchmark	Benchmark with default configuration & relevant functions for Modea

dacbench.benchmarks.modea_benchmark.**INFO**

dacbench.benchmarks.modea_benchmark.**MODEA_DEFAULTS**

class dacbench.benchmarks.modea_benchmark.**ModeaBenchmark**(*config_path=None*, *config=None*)

Bases: *dacbench.abstract_benchmark.AbstractBenchmark*

Benchmark with default configuration & relevant functions for Modea

get_environment(*self*)

Return ModeaEnv env with current configuration

Returns
Modea environment

Return type
ModeaEnv

read_instance_set(*self*, *test=False*)

Read path of instances from config into list

dacbench.benchmarks.sgd_benchmark

Module Contents

Classes

SGDBenchmark	Benchmark with default configuration & relevant functions for SGD

Functions

__default_loss_function(**kwargs)	

dacbench.benchmarks.sgd_benchmark.**DEFAULT_CFG_SPACE**

dacbench.benchmarks.sgd_benchmark.**LR**

dacbench.benchmarks.sgd_benchmark.**__default_loss_function**(***kwargs*)

dacbench.benchmarks.sgd_benchmark.**INFO**

dacbench.benchmarks.sgd_benchmark.**SGD_DEFAULTS**

dacbench.benchmarks.sgd_benchmark.**reward_range**

class dacbench.benchmarks.sgd_benchmark.**SGDBenchmark**(*config_path=None*, *config=None*)

Bases: *dacbench.abstract_benchmark.AbstractBenchmark*

Benchmark with default configuration & relevant functions for SGD

get_environment(*self*)

Return SGDEnv env with current configuration

Returns
SGD environment

Return type
SGDEnv

read_instance_set(*self*, *test=False*)

Read path of instances from config into list

get_benchmark(*self*, *instance_set_path=None*, *seed=0*)

Get benchmark from the LTO paper

Parameters
seed (*int*) – Environment seed

Returns
env – SGD environment

Return type
SGDEnv

dacbench.benchmarks.sigmoid_benchmark

Module Contents

Classes

SigmoidBenchmark	Benchmark with default configuration & relevant functions for Sigmoid

dacbench.benchmarks.sigmoid_benchmark.**ACTION_VALUES = [5, 10]**

dacbench.benchmarks.sigmoid_benchmark.**DEFAULT_CFG_SPACE**

dacbench.benchmarks.sigmoid_benchmark.**X**

dacbench.benchmarks.sigmoid_benchmark.**INFO**

dacbench.benchmarks.sigmoid_benchmark.**SIGMOID_DEFAULTS**

class dacbench.benchmarks.sigmoid_benchmark.**SigmoidBenchmark**(*config_path=None*, *config=None*)

Bases: *dacbench.abstract_benchmark.AbstractBenchmark*

Benchmark with default configuration & relevant functions for Sigmoid

get_environment(*self*)

Return Sigmoid env with current configuration

Returns
Sigmoid environment

Return type
SigmoidEnv

set_action_values(*self*, *values*)

Adapt action values and update dependencies

Parameters
values (*list*) – A list of possible actions per dimension

read_instance_set(*self*, *test=False*)

Read instance set from file

get_benchmark(*self*, *dimension=None*, *seed=0*)

Get Benchmark from DAC paper

Parameters

- **dimension** (*int*) – Sigmoid dimension, was 1, 2, 3 or 5 in the paper
- **seed** (*int*) – Environment seed

Returns
env – Sigmoid environment

Return type
SigmoidEnv

dacbench.benchmarks.theory_benchmark

Module Contents

Classes

TheoryBenchmark	Benchmark with various settings for (1+(lbd, lbd))-GA and RLS

dacbench.benchmarks.theory_benchmark.**INFO**

dacbench.benchmarks.theory_benchmark.**THEORY_DEFAULTS**

class dacbench.benchmarks.theory_benchmark.**TheoryBenchmark**(*config=None*)

Bases: *dacbench.abstract_benchmark.AbstractBenchmark*

Benchmark with various settings for (1+(lbd, lbd))-GA and RLS

create_observation_space_from_description(*self*, *obs_description*, *env_class=RLSEnvDiscrete*)

Create a gym observation space (Box only) based on a string containing observation variable names, e.g. "n, f(x), k, k_{t-1}" Return:

A gym.spaces.Box observation space

get_environment(*self*, *test_env=False*)

Return an environment with current configuration

Parameters:

test_env: whether the enviroment is used for train an agent or for testing.

if test_env=False:
cutoff time for an episode is set to 0.8*n^2 (n: problem size) if an action is out of range, stop the episode immediately and return a large negative reward (see envs/theory.py for more details)

otherwise: benchmark's original cutoff time is used, and out-of-range action will be clipped to nearest valid value and the episode will continue.

read_instance_set(*self*)

Read instance set from file we look at the current directory first, if the file doesn't exist, we look in <DACBench>/dacbench/instance_sets/theory/

dacbench.benchmarks.toysgd_benchmark

Module Contents

Classes

ToySGDBenchmark	Abstract template for benchmark classes

dacbench.benchmarks.toysgd_benchmark.**DEFAULT_CFG_SPACE**

dacbench.benchmarks.toysgd_benchmark.**LR**

dacbench.benchmarks.toysgd_benchmark.**MOMENTUM**

dacbench.benchmarks.toysgd_benchmark.**INFO**

dacbench.benchmarks.toysgd_benchmark.**DEFAULTS**

class dacbench.benchmarks.toysgd_benchmark.**ToySGDBenchmark**(*config_path=None*, *config=None*)

Bases: *dacbench.abstract_benchmark.AbstractBenchmark*

Abstract template for benchmark classes

get_environment(*self*)

Return SGDEnv env with current configuration

Returns
SGD environment

Return type
SGDEnv

read_instance_set(*self*, *test=False*)

Read path of instances from config into list

Package Contents

Classes

LubyBenchmark	Benchmark with default configuration & relevant functions for Sigmoid
SigmoidBenchmark	Benchmark with default configuration & relevant functions for Sigmoid
ToySGDBenchmark	Abstract template for benchmark classes
GeometricBenchmark	Benchmark with default configuration & relevant functions for Geometric
FastDownwardBenchmark	Benchmark with default configuration & relevant functions for Sigmoid

class dacbench.benchmarks.**LubyBenchmark**(*config_path=None*, *config=None*)

Bases: *dacbench.abstract_benchmark.AbstractBenchmark*

Benchmark with default configuration & relevant functions for Sigmoid

get_environment(*self*)

Return Luby env with current configuration

Returns
Luby environment

Return type
LubyEnv

set_cutoff(*self*, *steps*)

Set cutoff and adapt dependencies

Parameters
int – Maximum number of steps

set_history_length(*self*, *length*)

Set history length and adapt dependencies

Parameters
int – History length

read_instance_set(*self*, *test=False*)

Read instance set from file

get_benchmark(*self*, *L=8*, *fuzziness=1.5*, *seed=0*)

Get Benchmark from DAC paper

Parameters

- **L** (*int*) – Minimum sequence lenght, was 8, 16 or 32 in the paper
- **fuzziness** (*float*) – Amount of noise applied. Was 1.5 for most of the experiments
- **seed** (*int*) – Environment seed

Returns
env – Luby environment

Return type
LubyEnv

class dacbench.benchmarks.**SigmoidBenchmark**(*config_path=None*, *config=None*)

Bases: *dacbench.abstract_benchmark.AbstractBenchmark*

Benchmark with default configuration & relevant functions for Sigmoid

get_environment(*self*)

Return Sigmoid env with current configuration

Returns
Sigmoid environment

Return type
SigmoidEnv

set_action_values(*self*, *values*)

Adapt action values and update dependencies

Parameters
values (*list*) – A list of possible actions per dimension

read_instance_set(*self*, *test=False*)

Read instance set from file

get_benchmark(*self*, *dimension=None*, *seed=0*)

Get Benchmark from DAC paper

Parameters

- **dimension** (*int*) – Sigmoid dimension, was 1, 2, 3 or 5 in the paper
- **seed** (*int*) – Environment seed

Returns
env – Sigmoid environment

Return type
SigmoidEnv

class dacbench.benchmarks.**ToySGDBenchmark**(*config_path=None*, *config=None*)

Bases: *dacbench.abstract_benchmark.AbstractBenchmark*

Abstract template for benchmark classes

get_environment(*self*)

Return SGDEnv env with current configuration

Returns
SGD environment

Return type
SGDEnv

read_instance_set(*self*, *test=False*)

Read path of instances from config into list

class dacbench.benchmarks.**GeometricBenchmark**(*config_path=None*)

Bases: *dacbench.abstract_benchmark.AbstractBenchmark*

Benchmark with default configuration & relevant functions for Geometric

get_environment(*self*)

Return Geometric env with current configuration

Returns
Geometric environment

Return type
GeometricEnv

read_instance_set(*self*)

Read instance set from file Creates a nested List for every Intance. The List contains all functions with their respective values.

get_benchmark(*self*, *dimension=None*, *seed=0*)

[summary]

Parameters

- **dimension** (*[type], optional*) – [description], by default None
- **seed** (*int, optional*) – [description], by default 0

Returns
[description]

Return type
[type]

set_action_values(*self*)

Adapt action values and update dependencies Number of actions can differ between functions if configured in DefaultDict Set observation space args.

set_action_description(*self*)

Add Information about Derivative and Coordinate to Description.

create_correlation_table(*self*)

Create correlation table from Config infos

class dacbench.benchmarks.**FastDownwardBenchmark**(*config_path=None*, *config=None*)

Bases: *dacbench.abstract_benchmark.AbstractBenchmark*

Benchmark with default configuration & relevant functions for Sigmoid

get_environment(*self*)

Return Luby env with current configuration

Returns
Luby environment

Return type
LubyEnv

read_instance_set(*self*, *test=False*)

Read paths of instances from config into list

set_heuristics(*self*, *heuristics*)

get_benchmark(*self*, *seed=0*)

Get published benchmark

Parameters
seed (*int*) – Environment seed

Returns
env – FD environment

Return type
FastDownwardEnv

dacbench.container

Submodules

dacbench.container.container_utils

Module Contents

Classes

Encoder	Json Encoder to save tuple and or numpy arrays | numpy floats / integer.
Decoder	Adapted from: https://github.com/automl/HPOBench/blob/master/hpobench/util/container_utils.py

Functions

deserialize_random_state(random_state_dict: Dict) → numpy.random.RandomState	
serialize_random_state(random_state: numpy.random.RandomState) → Tuple[(int, List, int, int, int)]	
wait_for_unixsocket(path: str, timeout: float = 10.0) → None	Wait for a UNIX socket to be created.
wait_for_port(port, host='localhost', timeout=5.0)	Taken from https://gist.github.com/butla/2d9a4c0f35ea47b7452156c96a4e7b12

class dacbench.container.container_utils.**Encoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: json.JSONEncoder

Json Encoder to save tuple and or numpy arrays | numpy floats / integer. Adapted from: https://github.com/automl/HPOBench/blob/master/hpobench/util/container_utils.py Serializing tuple/numpy array may not work. We need to annotate those types, to reconstruct them correctly.

static hint(*item*)

encode(*self*, *obj*)

Return a JSON string representation of a Python data structure.

```
>>> from json.encoder import JSONEncoder
>>> JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

static encode_space(*space_obj: gym.Space*)

class dacbench.container.container_utils.**Decoder**(**args*, ***kwargs*)

Bases: json.JSONDecoder

Adapted from: https://github.com/automl/HPOBench/blob/master/hpobench/util/container_utils.py

object_hook(*self*, *obj: Any*) → Union[Union[tuple, numpy.ndarray, float, float, int], Any]

decode_space(*self*, *space_dict: Dict*) → gym.Space

dacbench.container.container_utils.**deserialize_random_state**(*random_state_dict: Dict*) → numpy.random.RandomState

dacbench.container.container_utils.**serialize_random_state**(*random_state: numpy.random.RandomState*) → Tuple[int, List, int, int, int]

dacbench.container.container_utils.**wait_for_unixsocket**(*path: str*, *timeout: float = 10.0*) → None

Wait for a UNIX socket to be created.

Parameters

- **path** – path to the socket
- **timeout** – timeout in seconds

Returns

dacbench.container.container_utils.**wait_for_port**(*port*, *host='localhost'*, *timeout=5.0*)

Taken from https://gist.github.com/butla/2d9a4c0f35ea47b7452156c96a4e7b12 Wait until a port starts accepting TCP connections.

Parameters

- **port** (*int*) – Port number to check.
- **host** (*str*) – Host to check.
- **timeout** (*float*) – Timeout in seconds.
- **Raises** –
- **------** –
- **TimeoutError** (The port isn't accepting connection after time specified in *timeout*.) –

dacbench.container.remote_env

Module Contents

Classes

RemoteEnvironmentServer
RemoteEnvironmentClient

Functions

json_encode(obj: Jsonable) → str
json_decode(json_str: str) → Jsonable

dacbench.container.remote_env.**NumpyTypes**

dacbench.container.remote_env.**DefaultJsonable**

dacbench.container.remote_env.**Jsonable**

dacbench.container.remote_env.**json_encode**(*obj: Jsonable*) → str

dacbench.container.remote_env.**json_decode**(*json_str: str*) → Jsonable

class dacbench.container.remote_env.**RemoteEnvironmentServer**(*env*)

step(*self*, *action: Union[Dict[str, List[numbers.Number]], List[numbers.Number]]*)

reset(*self*)

render(*self*, *mode='human'*)

close(*self*)

property action_space(*self*)

class dacbench.container.remote_env.**RemoteEnvironmentClient**(*env:* dacbench.container.remote_env.RemoteEnvironmentServ

step(*self*, *action: Union[Dict[str, numpy.ndarray], numpy.ndarray]*) → Tuple[Union[Dict[str, numpy.ndarray], numpy.ndarray], numbers.Number, bool, dict]

reset(*self*) → Union[Dict[str, numpy.ndarray], numpy.ndarray]

close(*self*)

property action_space(*self*)

dacbench.container.remote_runner

This is strongly guided and partially copy from https://github.com/automl/HPOBench/blob/master/hpobench/container/client_abstract_benchmark.py

Module Contents

Classes

RemoteRunnerServer

RemoteRunner

RemoteRunnerServerFactory

dacbench.container.remote_runner.**SERVERTYPE = multiplex**

dacbench.container.remote_runner.**log_level_str**

dacbench.container.remote_runner.**LOG_LEVEL**

dacbench.container.remote_runner.**LOG_LEVEL**

dacbench.container.remote_runner.**root**

dacbench.container.remote_runner.**logger**

dacbench.container.remote_runner.**excepthook**

dacbench.container.remote_runner.**MAX_TRIES = 5**

dacbench.container.remote_runner.**SOCKET_PATH**

class dacbench.container.remote_runner.**RemoteRunnerServer**(*pyro_demon*)

start(*self*, *config: str*, *benchmark: Tuple[str, str]*)

get_environment(*self*) → str

class dacbench.container.remote_runner.**RemoteRunner**(*benchmark:* dacbench.abstract_benchmark.AbstractBenchmark, *container_name: str = None*, *container_source: Optional[str] = None*, *container_tag: str = 'latest'*, *env_str: Optional[str] = ''*, *bind_str: Optional[str] = ''*, *gpu: Optional[bool] = False*, *socket_id=None*)

FACTORY_NAME :str = RemoteRunnerServerFactory

property socket(*self*) → pathlib.Path

static id_generator() → str

Helper function: Creates unique socket ids for the benchmark server

static socket_from_id(*socket_id: str*) → pathlib.Path

__start_server(*self*, *env_str*, *bind_str*, *gpu*)

Starts container and the pyro server

Parameters

- **env_str** (*str*) – Environment string for the container
- **bind_str** (*str*) – Bind string for the container
- **gpu** (*bool*) – True if the container should use gpu, False otherwise

__connect_to_server(*self*, *benchmark:* dacbench.abstract_benchmark.AbstractBenchmark)

Connects to the server and initializes the benchmark

get_environment(*self*)

run(*self*, *agent:* dacbench.abstract_agent.AbstractDACBenchAgent, *number_of_episodes: int*)

close(*self*)

__del__(*self*)

load_benchmark(*self*, *benchmark:* dacbench.abstract_benchmark.AbstractBenchmark, *container_name: str*, *container_source: Union[str, pathlib.Path]*, *container_tag: str*)

class dacbench.container.remote_runner.**RemoteRunnerServerFactory**(*pyro_demon*)

create(*self*)

__call__(*self*)

dacbench.container.remote_runner.**parser**

dacbench.envs

Subpackages

dacbench.envs.policies

Submodules

dacbench.envs.policies.csa_cma

Module Contents

Functions

csa(env, state)	

dacbench.envs.policies.csa_cma.**csa**(*env*, *state*)

dacbench.envs.policies.optimal_fd

Module Contents

Functions

get_optimum(env, state)	

dacbench.envs.policies.optimal_fd.**get_optimum**(*env*, *state*)

dacbench.envs.policies.optimal_luby

Module Contents

Functions

luby_gen(i)	Generator for the Luby Sequence
get_optimum(env, state)	

dacbench.envs.policies.optimal_luby.**luby_gen**(*i*)

Generator for the Luby Sequence

dacbench.envs.policies.optimal_luby.**get_optimum**(*env*, *state*)

dacbench.envs.policies.optimal_sigmoid

Module Contents

Functions

sig(x, scaling, inflection)	Simple sigmoid function
get_optimum(env, state)	

dacbench.envs.policies.optimal_sigmoid.**sig**(*x*, *scaling*, *inflection*)

Simple sigmoid function

dacbench.envs.policies.optimal_sigmoid.**get_optimum**(*env*, *state*)

dacbench.envs.policies.sgd_ca

Module Contents

Classes

CosineAnnealingAgent	Abstract class to implement for use with the runner function

class dacbench.envs.policies.sgd_ca.**CosineAnnealingAgent**(*env*, *base_lr=0.1*, *t_max=1000*, *eta_min=0*)

Bases: *dacbench.abstract_agent.AbstractDACBenchAgent*

Abstract class to implement for use with the runner function

act(*self*, *state*, *reward*)

Compute and return environment action

Parameters

- **state** – Environment state
- **reward** – Environment reward

Returns

Action to take

Return type

action

train(*self*, *state*, *reward*)

Train during episode if needed (pass if not)

Parameters

- **next_state** – Environment state after step
- **reward** – Environment reward

end_episode(*self*, *state*, *reward*)

End of episode training if needed (pass if not)

Parameters

- **state** – Environment state
- **reward** – Environment reward

Package Contents

Functions

csa(env, state)	
optimal_fd	
optimal_luby	
optimal_sigmoid(env, state)	

dacbench.envs.policies.**csa**(*env*, *state*)

dacbench.envs.policies.**optimal_fd**(*env*, *state*)

dacbench.envs.policies.**optimal_luby**(*env*, *state*)

dacbench.envs.policies.**optimal_sigmoid**(*env*, *state*)

dacbench.envs.policies.**OPTIMAL_POLICIES**

dacbench.envs.policies.**NON_OPTIMAL_POLICIES**

dacbench.envs.policies.**ALL_POLICIES**

Submodules

dacbench.envs.cma_es

CMA-ES environment adapted from CMAWorld in "Learning Step-size Adaptation in CMA-ES" by G.Shala and A. Biedenkapp and N.Awad and S. Adriaensen and M.Lindauer and F. Hutter. Original author: Gresa Shala

Module Contents

Classes

CMAESEnv	Environment to control the step size of CMA-ES

Functions

_norm(x)

dacbench.envs.cma_es.**_norm**(*x*)

class dacbench.envs.cma_es.**CMAESEnv**(*config*)

Bases: *dacbench.AbstractEnv*

Environment to control the step size of CMA-ES

step(*self*, *action*)

Execute environment step

Parameters
action (*list*) – action to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

reset(*self*)

Reset environment

Returns
Environment state

Return type
np.array

close(*self*)

No additional cleanup necessary

Returns
Cleanup flag

Return type
bool

render(*self*, *mode: str = 'human'*)

Render env in human mode

Parameters
mode (*str*) – Execution mode

get_default_reward(*self*, _)

Compute reward

Returns
Reward

Return type
float

get_default_state(*self*, _)

Gather state description

Returns
Environment state

Return type
dict

dacbench.envs.cma_step_size

Module Contents

Classes

CMAStepSizeEnv	Abstract template for environments

class dacbench.envs.cma_step_size.**CMAStepSizeEnv**(*config*)

Bases: *dacbench.AbstractEnv*

Abstract template for environments

reset(*self*)

Reset environment

Returns
Environment state

Return type
state

step(*self*, *action*)

Execute environment step

Parameters
action – Action to take

Returns

- *state* – Environment state
- *reward* – Environment reward
- **done** (*bool*) – Run finished flag
- **info** (*dict*) – Additional metainfo

close(*self*)

Override close in your subclass to perform any necessary cleanup.

Environments will automatically close() themselves when garbage collected or when the program exits.

get_default_reward(*self*, *_)

get_default_state(*self*, *_)

dacbench.envs.fast_downward

Planning environment from "Learning Heuristic Selection with Dynamic Algorithm Configuration" by David Speck, André Biedenkapp, Frank Hutter, Robert Mattmüller und Marius Lindauer. Original environment authors: David Speck, André Biedenkapp

Module Contents

Classes

StateType	Class to define numbers for state types
FastDownwardEnv	Environment to control Solver Heuristics of FastDownward

class dacbench.envs.fast_downward.**StateType**

Bases: enum.Enum

Class to define numbers for state types

RAW = 1

DIFF = 2

ABSDIFF = 3

NORMAL = 4

NORMDIFF = 5

NORMABSDIFF = 6

class dacbench.envs.fast_downward.**FastDownwardEnv**(*config*)

Bases: *dacbench.AbstractEnv*

Environment to control Solver Heuristics of FastDownward

property port(*self*)

property argstring(*self*)

static _save_div(*a*, *b*)

Helper method for safe division

Parameters

- **a** (*list or np.array*) – values to be divided

- **b** (*list or np.array*) – values to divide by

Returns
Division result

Return type
np.array

send_msg(*self*, *msg: bytes*)

Send message and prepend the message size

Based on comment from SO see [1] [1] https://stackoverflow.com/a/17668009

Parameters
msg (*bytes*) – The message as byte

recv_msg(*self*)

Recieve a whole message. The message has to be prepended with its total size Based on comment from SO see [1]

Returns
The message as byte

Return type
bytes

recvall(*self*, *n: int*)

Given we know the size we want to recieve, we can recieve that amount of bytes. Based on comment from SO see [1]

Parameters
n (*int*) – Number of bytes to expect in the data

Returns
The message as byte

Return type
bytes

_process_data(*self*)

Split received json into state reward and done

Returns
state, reward, done

Return type
np.array, float, bool

step(*self*, *action: Union[int, List[int]]*)

Environment step

Parameters
action (*Union[int, List[int]]*) – Parameter(s) to apply

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

reset(*self*)

Reset environment

Returns
State after reset

Return type
np.array

kill_connection(*self*)

Kill the connection

close(*self*)

Close Env

Returns
Closing confirmation

Return type
bool

render(*self*, *mode: str = 'human'*) → None

Required by gym.Env but not implemented

Parameters
mode (*str*) – Rendering mode

dacbench.envs.geometric

Geometric environment. Original environment authors: Rasmus von Glahn

Module Contents

Classes

GeometricEnv	Environment for tracing different curves that are orthogonal to each other
Functions	

class dacbench.envs.geometric.**GeometricEnv**(*config*)

Bases: *dacbench.AbstractEnv*

Environment for tracing different curves that are orthogonal to each other Use product approach: f(t,x,y,z) = X(t,x) * Y(t,y) * Z(t,z) Normalize Function Value on a Scale between 0 and 1

- min and max value for normalization over all timesteps

get_optimal_policy(*self*, *instance: List = None*, *vector_action: bool = True*) → List[numpy.array]

Calculates the optimal policy for an instance

Parameters

- **instance** (*List, optional*) – instance with information about function config.

- **vector_action** (*bool, optional*) – if True return multidim actions else return onedimensional action, by default True

Returns
List with entry for each timestep that holds all optimal values in an array or as int

Return type
List[np.array]

step(*self*, *action: int*)

Execute environment step

Parameters
action (*int*) – action to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

reset(*self*) → List[int]

Resets env

Returns
Environment state

Return type
numpy.array

get_default_reward(*self*, *_*) → float

Calculate euclidean distance between action vector and real position of Curve.

Parameters
_ (*self*) – ignore

Returns
Euclidean distance

Return type
float

get_default_state(*self*, *_*) → numpy.array

Gather state information.

Parameters
_ – ignore param

Returns
numpy array with state information

Return type
np.array

close(*self*) → bool

Close Env

Returns
Closing confirmation

Return type
bool

render(*self*, *dimensions: List*, *absolute_path: str*)

Multiplot for specific dimensions of benchmark with policy actions.

Parameters
dimensions (*List*) – List of dimensions that get plotted

render_3d_dimensions(*self*, *dimensions: List*, *absolute_path: str*)

Plot 2 Dimensions in 3D space

Parameters
dimensions (*List*) – List of dimensions that get plotted. Max 2

_pre_reward(*self*) → Tuple[numpy.ndarray, List]

Prepare actions and coordinates for reward calculation.

Returns
[description]

Return type
Tuple[np.ndarray, List]

class dacbench.envs.geometric.**Functions**(*n_steps: int*, *n_actions: int*, *n_instances: int*, *correlation: bool*, *correlation_table: numpy.ndarray*, *correlation_depth: int*, *derivative_interval: int*)

set_instance(*self*, *instance: List*, *instance_index*)

update instance

get_coordinates(*self*, *instance: List = None*) → List[numpy.array]

Calculates coordinates for instance over all time_steps. The values will change if correlation is applied and not optimal actions are taken.

Parameters
instance (*List, optional*) – Instance that holds information about functions, by default None

Returns
Index of List refers to time step

Return type
List[np.array]

get_coordinates_at_time_step(*self*, *time_step: int*) → numpy.array

Calculate coordiantes at time_step. Apply correlation.

Parameters

- **instance** (*List*) – Instance that holds information about functions
- **time_step** (*int*) – Time step of functions

Returns
array of function values at timestep

Return type
np.array

calculate_derivative(*self*, *trajectory: List*, *c_step: int*) → numpy.array

Calculate derivatives of each dimension, based on trajectories.

Parameters

- **trajectory** (*List*) – List of actions or coordinates already taken

- **c_step** (*int*) – current timestep

Returns
derivatives for each dimension

Return type
np.array

calculate_norm_values(*self*, *instance_set: Dict*)

Norm Functions to Intervall between -1 and 1

_calculate_function_value(*self*, *time_step: int*, *function_infos: List*, *func_idx: int*) → float

Call different functions with their speicifc parameters and norm them.

Parameters

- **function_infos** (*List*) – Consists of function name and the coefficients
- **time_step** (*int*) – time step for each function
- **calculate_norm** (*bool, optional*) – True if norm gets calculated, by default False

Returns
coordinate in dimension of function

Return type
float

_add_correlation(*self*, *value_array: numpy.ndarray*, *time_step: int*)

Adds correlation between dimensions but clips at -1 and 1. Correlation table holds numbers between -1 and 1. e.g. correlation_table[0][2] = 0.5 if dimension 1 changes dimension 3 changes about 50% of dimension one

Parameters
correlation_table (*np.array*) – table that holds all values of correlation between dimensions [n,n]

_apply_correlation_update(*self*, *idx: int*, *diff: float*, *depth*)

Recursive function for correlation updates Call function recursively till depth is 0 or diff is too small.

_sigmoid(*self*, *t: float*, *scaling: float*, *inflection: float*)

Simple sigmoid function

_linear(*self*, *t: float*, *a: float*, *b: float*)

Linear function

_parabel(*self*, *t: float*, *sig: int*, *x_int: int*, *y_int: int*)

Parabel function

_cubic(*self*, *t: float*, *sig: int*, *x_int: int*, *y_int: int*)

cubic function

_logarithmic(*self*, *t: float*, *a: float*)

Logarithmic function

_constant(*self*, *c: float*)

Constant function

_sinus(*self*, *t: float*, *scale: float*)

Sinus function

dacbench.envs.luby

Luby environment from "Dynamic Algorithm Configuration:Foundation of a New Meta-Algorithmic Framework" by A. Biedenkapp and H. F. Bozkurt and T. Eimer and F. Hutter and M. Lindauer. Original environment authors: André Biedenkapp, H. Furkan Bozkurt

Module Contents

Classes

LubyEnv	Environment to learn Luby Sequence

Functions

luby_gen(i)	Generator for the Luby Sequence

class dacbench.envs.luby.**LubyEnv**(*config*)

Bases: *dacbench.AbstractEnv*

Environment to learn Luby Sequence

step(*self*, *action: int*)

Execute environment step

Parameters
action (*int*) – action to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

reset(*self*) → List[int]

Resets env

Returns
Environment state

Return type
numpy.array

get_default_reward(*self*, _)

get_default_state(*self*, _)

close(*self*) → bool

Close Env

Returns
Closing confirmation

Return type
bool

render(*self*, *mode: str = 'human'*) → None

Render env in human mode

Parameters
mode (*str*) – Execution mode

dacbench.envs.luby.**luby_gen**(*i*)

Generator for the Luby Sequence

dacbench.envs.modcma

Module Contents

Classes

ModCMAEnv	Abstract template for environments

class dacbench.envs.modcma.**ModCMAEnv**(*config*)

Bases: *dacbench.AbstractEnv*

Abstract template for environments

reset(*self*)

Reset environment

Returns
Environment state

Return type
state

step(*self*, *action*)

Execute environment step

Parameters
action – Action to take

Returns

- *state* – Environment state
- *reward* – Environment reward
- **done** (*bool*) – Run finished flag
- **info** (*dict*) – Additional metainfo

close(*self*)

Override close in your subclass to perform any necessary cleanup.

Environments will automatically close() themselves when garbage collected or when the program exits.

get_default_reward(*self*, **_*)

get_default_state(*self*, **_*)

dacbench.envs.modea

Module Contents

Classes

ModeaEnv	Abstract template for environments

class dacbench.envs.modea.**ModeaEnv**(*config*)

Bases: *dacbench.AbstractEnv*

Abstract template for environments

reset(*self*)

Reset environment

Returns
Environment state

Return type
state

step(*self*, *action*)

Execute environment step

Parameters
action – Action to take

Returns

- *state* – Environment state
- *reward* – Environment reward
- **done** (*bool*) – Run finished flag
- **info** (*dict*) – Additional metainfo

update_parameters(*self*)

restart(*self*)

determineRegime(*self*)

get_default_state(*self*, _)

get_default_reward(*self*, _)

close(*self*)

Override close in your subclass to perform any necessary cleanup.

Environments will automatically close() themselves when garbage collected or when the program exits.

switchConfiguration(*self*, *opts*)

setConfigurationParameters(*self*, *functions*, *parameters*)

ensureFullLengthRepresentation(*self*, *representation*)

Given a (partial) representation, ensure that it is padded to become a full length customizedES representation, consisting of the required number of structure, population and parameter values. >>> ensureFullLengthRepresentation([]) [0,0,0,0,0,0,0,0,0,0,0, None,None, None,None,None,None,None,None,None,None,None,None,None,None] :param representation: List representation of a customizedES instance to check and pad if needed :return: Guaranteed full-length version of the representation

dacbench.envs.sgd

Module Contents

Classes

Reward	Enum where members are also (and must be) ints
SGDEnv	Environment to control the learning rate of adam

Functions

reward_range(frange)

dacbench.envs.sgd.**reward_range**(*frange*)

class dacbench.envs.sgd.**Reward**

Bases: enum.IntEnum

Enum where members are also (and must be) ints

TrainingLoss

ValidationLoss

LogTrainingLoss

LogValidationLoss

DiffTraining

DiffValidation

LogDiffTraining

LogDiffValidation

FullTraining

__call__(*self*, *f*)

class dacbench.envs.sgd.**SGDEnv**(*config*)

Bases: *dacbench.AbstractEnv*

Environment to control the learning rate of adam

val_model

Samuel Mueller (PhD student in our group) also uses backpack and has ran into a similar memory leak. He solved it calling this custom made RECURSIVE memory_cleanup function: # from backpack import memory_cleanup # def recursive_backpack_memory_cleanup(module: torch.nn.Module): # memory_cleanup(module) # for m in module.modules(): # memory_cleanup(m) (calling this after computing the training loss/gradients and after validation loss should suffice)

Type
TODO

get_reward(*self*)

get_training_reward(*self*)

get_validation_reward(*self*)

get_log_training_reward(*self*)

get_log_validation_reward(*self*)

get_log_diff_training_reward(*self*)

get_log_diff_validation_reward(*self*)

get_diff_training_reward(*self*)

get_diff_validation_reward(*self*)

get_full_training_reward(*self*)

get_full_training_loss(*self*)

property crash(*self*)

seed(*self*, *seed=None*, *seed_action_space=False*)

Set rng seed

Parameters

- **seed** – seed for rng
- **seed_action_space** (*bool, default False*) – if to seed the action space as well

step(*self*, *action*)

Execute environment step

Parameters
action (*list*) – action to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

_architecture_constructor(*self*, *arch_str*)

reset(*self*)

Reset environment

Returns
Environment state

Return type
np.array

`set_writer`(*self*, *writer*)

`close`(*self*)

No additional cleanup necessary

Returns
Cleanup flag

Return type
bool

`render`(*self*, *mode: str = 'human'*)

Render env in human mode

Parameters
mode (*str*) – Execution mode

`get_default_state`(*self*, _)

Gather state description

Returns
Environment state

Return type
dict

`_train_batch_`(*self*)

`train_network`(*self*)

`_get_full_training_loss`(*self*, *loader*)

`property current_validation_loss`(*self*)

`_get_validation_loss_`(*self*)

`_get_validation_loss`(*self*)

`_get_gradients`(*self*)

`_get_momentum`(*self*, *gradients*)

`get_adam_direction`(*self*)

`get_rmsprop_direction`(*self*)

`get_momentum_direction`(*self*)

`_get_loss_features`(*self*)

`_get_predictive_change_features`(*self*, *lr*)

`_get_alignment`(*self*)

`generate_instance_file`(*self*, *file_name*, *mode='test'*, *n=100*)

dacbench.envs.sigmoid

Sigmoid environment from "Dynamic Algorithm Configuration:Foundation of a New Meta-Algorithmic Framework" by A. Biedenkapp and H. F. Bozkurt and T. Eimer and F. Hutter and M. Lindauer. Original environment authors: André Biedenkapp, H. Furkan Bozkurt

Module Contents

Classes

SigmoidEnv	Environment for tracing sigmoid curves
ContinuousStateSigmoidEnv	Environment for tracing sigmoid curves with a continuous state on the x-axis
ContinuousSigmoidEnv	Environment for tracing sigmoid curves with a continuous state on the x-axis

class dacbench.envs.sigmoid.**SigmoidEnv**(*config*)

Bases: *dacbench.AbstractEnv*

Environment for tracing sigmoid curves

_sig(*self*, *x*, *scaling*, *inflection*)

Simple sigmoid function

step(*self*, *action: int*)

Execute environment step

Parameters
action (*int*) – action to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

reset(*self*) → List[int]

Resets env

Returns
Environment state

Return type
numpy.array

get_default_reward(*self*, _)

get_default_state(*self*, _)

close(*self*) → bool

Close Env

Returns
Closing confirmation

Return type
bool

render(*self*, *mode: str*) → None

Render env in human mode

Parameters
mode (*str*) – Execution mode

class dacbench.envs.sigmoid.**ContinuousStateSigmoidEnv**(*config*)

Bases: *dacbench.envs.sigmoid.SigmoidEnv*

Environment for tracing sigmoid curves with a continuous state on the x-axis

step(*self*, *action: int*)

Execute environment step

Parameters
action (*int*) – action to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

class dacbench.envs.sigmoid.**ContinuousSigmoidEnv**(*config*)

Bases: *dacbench.envs.sigmoid.SigmoidEnv*

Environment for tracing sigmoid curves with a continuous state on the x-axis

step(*self*, *action: numpy.ndarray*)

Execute environment step. !!NOTE!! The action here is a list of floats and not a single number !!NOTE!!

Parameters
action (*list of floats*) – action(s) to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

dacbench.envs.sigmoid.**config**

dacbench.envs.theory

Module Contents

Classes

BinaryProblem	An abstract class for an individual in binary representation
LeadingOne	An individual for LeadingOne problem
RLSEnv	Environment for RLS with step size
RLSEnvDiscrete	RLS environment where the choices of r is discretised

class dacbench.envs.theory.**BinaryProblem**(*n*, *rng=np.random.default_rng()*)

An abstract class for an individual in binary representation

initialise_with_fixed_number_of_bits(*self*, *k*, *rng=np.random.default_rng()*)

is_optimal(*self*)

get_optimal(*self*)

eval(*self*)

abstract get_fitness_after_flipping(*self*, *locs*)

Calculate the change in fitness after flipping the bits at positions locs

Parameters
locs (*1d-array*) – positions where bits are flipped

objective after flipping

abstract get_fitness_after_crossover(*self*, *xprime*, *locs_x*, *locs_xprime*)

Calculate fitness of the child aftering being crossovered with xprime

Parameters

- **xprime** (*1d boolean array*) – the individual to crossover with
- **locs_x** (*1d boolean/integer array*) – positions where we keep current bits of self
- **locs_xprime** (*: 1d boolean/integer array*) – positions where we change to xprime's bits

flip(*self*, *locs*)

flip the bits at position indicated by locs

Parameters
locs (*1d-array*) – positions where bits are flipped

Returns: the new individual after the flip

combine(*self*, *xprime*, *locs_xprime*)

combine (crossover) self and xprime by taking xprime's bits at locs_xprime and self's bits at other positions

Parameters

- **xprime** (*1d boolean array*) – the individual to crossover with
- **locs_x** (*1d boolean/integer array*) – positions where we keep current bits of self
- **locs_xprime** (*: 1d boolean/integer array*) – positions where we change to xprime's bits

Returns: the new individual after the crossover

mutate(*self*, *p*, *n_childs*, *rng=np.random.default_rng()*)

Draw l ~ binomial(n, p), l>0 Generate n_childs children by flipping exactly l bits Return: the best child (maximum fitness), its fitness and number of evaluations used

mutate_rls(*self*, *l*, *rng=np.random.default_rng()*)

generate a child by flipping exactly l bits Return: child, its fitness

crossover(*self*, *xprime*, *p*, *n_childs*, *include_xprime=True*, *count_different_inds_only=True*, *rng=np.random.default_rng()*)

Crossover operator:
for each bit, taking value from x with probability p and from self with probability 1-p

Arguments:
x: the individual to crossover with p (float): in [0,1]

class dacbench.envs.theory.**LeadingOne**(*n*, *rng=np.random.default_rng()*, *initObj=None*)

Bases: *dacbench.envs.theory.BinaryProblem*

An individual for LeadingOne problem The aim is to maximise the number of leading (and consecutive) 1 bits in the string

eval(*self*)

is_optimal(*self*)

get_optimal(*self*)

get_fitness_after_flipping(*self*, *locs*)

Calculate the change in fitness after flipping the bits at positions locs

Parameters
locs (*1d-array*) – positions where bits are flipped

objective after flipping

get_fitness_after_crossover(*self*, *xprime*, *locs_x*, *locs_xprime*)

Calculate fitness of the child aftering being crossovered with xprime

Parameters

- **xprime** (*1d boolean array*) – the individual to crossover with
- **locs_x** (*1d boolean/integer array*) – positions where we keep current bits of self
- **locs_xprime** (*: 1d boolean/integer array*) – positions where we change to xprime's bits

dacbench.envs.theory.**MAX_INT = 100000000.0**

dacbench.envs.theory.**HISTORY_LENGTH = 5**

class dacbench.envs.theory.**RLSEnv**(*config*, *test_env=False*)

Bases: *dacbench.AbstractEnv*

Environment for RLS with step size Current assumption: we only consider (1+1)-RLS, so there's only one parameter to tune (r)

get_obs_domain_from_name(*var_name*)

Get default lower and upperbound of a observation variable based on its name. The observation space will then be created Return:

Two int values, e.g., 1, np.inf

reset(*self*)

Resets env

Returns
Environment state

Return type
numpy.array

get_state(*self*)

step(*self*, *action*)

Execute environment step

Parameters
action (*Box*) – action to execute

Returns

- *state, reward, done, info*
- *np.array, float, bool, dict*

close(*self*) → bool

Close Env

No additional cleanup necessary

Returns
Closing confirmation

Return type
bool

class dacbench.envs.theory.**RLSEnvDiscrete**(*config*, *test_env=False*)

Bases: *dacbench.envs.theory.RLSEnv*

RLS environment where the choices of r is discretised

step(*self*, *action*)

Execute environment step

Parameters
action (*Box*) – action to execute

Returns

- *state, reward, done, info*
- *np.array, float, bool, dict*

dacbench.envs.toysgd

Module Contents

Classes

ToySGDEnv	Optimize toy functions with SGD + Momentum.

Functions

create_polynomial_instance_set(out_fname: str, n_samples: int = 100, order: int = 2, low: float = -10, high: float = 10)	
sample_coefficients(order: int = 2, low: float = -10, high: float = 10)	

dacbench.envs.toysgd.**create_polynomial_instance_set**(*out_fname: str*, *n_samples: int = 100*, *order: int = 2*, *low: float = -10*, *high: float = 10*)

dacbench.envs.toysgd.**sample_coefficients**(*order: int = 2*, *low: float = -10*, *high: float = 10*)

class dacbench.envs.toysgd.**ToySGDEnv**(*config*)

Bases: *dacbench.AbstractEnv*

Optimize toy functions with SGD + Momentum.

Action: [log_learning_rate, log_momentum] (log base 10) State: Dict with entries remaining_budget, gradient, learning_rate, momentum Reward: negative log regret of current and true function value

An instance can look as follows: ID 0 family polynomial order 2 low -2 high 2 coefficients [1.40501053 -0.59899755 1.43337392]

build_objective_function(*self*)

get_initial_position(*self*)

step(*self*, *action: Union[float, Tuple[float, float]]*) → Tuple[Dict[str, float], float, bool, Dict]

Take one step with SGD

Parameters

action (*Tuple[float, Tuple[float, float]]*) – If scalar, action = (log_learning_rate) If tuple, action = (log_learning_rate, log_momentum)

Returns

- **state**
  [Dict[str, float]] State with entries "remaining_budget", "gradient", "learning_rate", "momentum"
- reward : float
- done : bool
- info : Dict

Return type

Tuple[Dict[str, float], float, bool, Dict]

reset(*self*)

Reset environment

Returns

Environment state

Return type

np.array

`render`(*self*, ***kwargs*)

Renders the environment.

The set of supported modes varies per environment. (And some environments do not support rendering at all.) By convention, if mode is:

- human: render to the current display or terminal and return nothing. Usually for human consumption.
- rgb_array: Return an numpy.ndarray with shape (x, y, 3), representing RGB values for an x-by-y pixel image, suitable for turning into a video.
- ansi: Return a string (str) or StringIO.StringIO containing a terminal-style text representation. The text can include newlines and ANSI escape sequences (e.g. for colors).

Note:

Make sure that your class's metadata 'render.modes' key includes
the list of supported modes. It's recommended to call super() in implementations to use the functionality of this method.

Args:
mode (str): the mode to render with

Example:

class MyEnv(Env):
metadata = {'render.modes': ['human', 'rgb_array']}

def render(self, mode='human'):

if mode == 'rgb_array':
return np.array(...) # return RGB frame suitable for video

elif mode == 'human':
... # pop up a window and render

else:
super(MyEnv, self).render(mode=mode) # just raise an exception

`close`(*self*)

Override close in your subclass to perform any necessary cleanup.

Environments will automatically close() themselves when garbage collected or when the program exits.

Package Contents

Classes

LubyEnv	Environment to learn Luby Sequence
SigmoidEnv	Environment for tracing sigmoid curves
FastDownwardEnv	Environment to control Solver Heuristics of FastDownward
ToySGDEnv	Optimize toy functions with SGD + Momentum.
GeometricEnv	Environment for tracing different curves that are orthogonal to each other

Functions

luby_gen(i)	Generator for the Luby Sequence

class dacbench.envs.**LubyEnv**(*config*)

Bases: *dacbench.AbstractEnv*

Environment to learn Luby Sequence

step(*self*, *action: int*)

Execute environment step

Parameters
action (*int*) – action to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

reset(*self*) → List[int]

Resets env

Returns
Environment state

Return type
numpy.array

get_default_reward(*self*, *_*)

get_default_state(*self*, *_*)

close(*self*) → bool

Close Env

Returns
Closing confirmation

Return type
bool

render(*self*, *mode: str = 'human'*) → None

Render env in human mode

Parameters
mode (*str*) – Execution mode

dacbench.envs.**luby_gen**(*i*)

Generator for the Luby Sequence

class dacbench.envs.**SigmoidEnv**(*config*)

Bases: *dacbench.AbstractEnv*

Environment for tracing sigmoid curves

_sig(*self*, *x*, *scaling*, *inflection*)

Simple sigmoid function

step(*self*, *action: int*)

Execute environment step

Parameters
action (*int*) – action to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

reset(*self*) → List[int]

Resets env

Returns
Environment state

Return type
numpy.array

get_default_reward(*self*, *_*)

get_default_state(*self*, *_*)

close(*self*) → bool

Close Env

Returns
Closing confirmation

Return type
bool

render(*self*, *mode: str*) → None

Render env in human mode

Parameters
mode (*str*) – Execution mode

class dacbench.envs.**FastDownwardEnv**(*config*)

Bases: *dacbench.AbstractEnv*

Environment to control Solver Heuristics of FastDownward

property port(*self*)

property argstring(*self*)

static _save_div(*a*, *b*)

Helper method for safe division

Parameters

- **a** (*list or np.array*) – values to be divided
- **b** (*list or np.array*) – values to divide by

Returns
Division result

Return type
np.array

send_msg(*self*, *msg: bytes*)

Send message and prepend the message size

Based on comment from SO see [1] [1] https://stackoverflow.com/a/17668009

Parameters
msg (*bytes*) – The message as byte

recv_msg(*self*)

Recieve a whole message. The message has to be prepended with its total size Based on comment from SO see [1]

Returns
The message as byte

Return type
bytes

recvall(*self*, *n: int*)

Given we know the size we want to recieve, we can recieve that amount of bytes. Based on comment from SO see [1]

Parameters
n (*int*) – Number of bytes to expect in the data

Returns
The message as byte

Return type
bytes

_process_data(*self*)

Split received json into state reward and done

Returns
state, reward, done

Return type
np.array, float, bool

step(*self*, *action: Union[int, List[int]]*)

Environment step

Parameters
action (*Union[int, List[int]]*) – Parameter(s) to apply

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

reset(*self*)

Reset environment

Returns
State after reset

Return type
np.array

kill_connection(*self*)

Kill the connection

close(*self*)

Close Env

Returns
Closing confirmation

Return type
bool

render(*self*, *mode: str = 'human'*) → None

Required by gym.Env but not implemented

Parameters
mode (*str*) – Rendering mode

class dacbench.envs.**ToySGDEnv**(*config*)

Bases: *dacbench.AbstractEnv*

Optimize toy functions with SGD + Momentum.

Action: [log_learning_rate, log_momentum] (log base 10) State: Dict with entries remaining_budget, gradient, learning_rate, momentum Reward: negative log regret of current and true function value

An instance can look as follows: ID 0 family polynomial order 2 low -2 high 2 coefficients [1.40501053 -0.59899755 1.43337392]

build_objective_function(*self*)

get_initial_position(*self*)

step(*self*, *action: Union[float, Tuple[float, float]]*) → Tuple[Dict[str, float], float, bool, Dict]

Take one step with SGD

Parameters
action (*Tuple[float, Tuple[float, float]]*) – If scalar, action = (log_learning_rate) If tuple, action = (log_learning_rate, log_momentum)

Returns

- **state**
  [Dict[str, float]] State with entries "remaining_budget", "gradient", "learning_rate", "momentum"
- reward : float
- done : bool
- info : Dict

Return type
Tuple[Dict[str, float], float, bool, Dict]

reset(*self*)

Reset environment

Returns
Environment state

Return type
np.array

render(*self*, ***kwargs*)

Renders the environment.

The set of supported modes varies per environment. (And some environments do not support rendering at all.) By convention, if mode is:

- human: render to the current display or terminal and return nothing. Usually for human consumption.
- rgb_array: Return an numpy.ndarray with shape (x, y, 3), representing RGB values for an x-by-y pixel image, suitable for turning into a video.
- ansi: Return a string (str) or StringIO.StringIO containing a terminal-style text representation. The text can include newlines and ANSI escape sequences (e.g. for colors).

Note:

Make sure that your class's metadata 'render.modes' key includes
the list of supported modes. It's recommended to call super() in implementations to use the functionality of this method.

Args:
mode (str): the mode to render with

Example:

class MyEnv(Env):
metadata = {'render.modes': ['human', 'rgb_array']}

def render(self, mode='human'):

if mode == 'rgb_array':
return np.array(...) # return RGB frame suitable for video

elif mode == 'human':
... # pop up a window and render

else:
super(MyEnv, self).render(mode=mode) # just raise an exception

close(*self*)

Override close in your subclass to perform any necessary cleanup.

Environments will automatically close() themselves when garbage collected or when the program exits.

class dacbench.envs.GeometricEnv(*config*)

Bases: *dacbench.AbstractEnv*

Environment for tracing different curves that are orthogonal to each other Use product approach: f(t,x,y,z) = X(t,x) * Y(t,y) * Z(t,z) Normalize Function Value on a Scale between 0 and 1

- min and max value for normalization over all timesteps

get_optimal_policy(*self*, *instance: List = None*, *vector_action: bool = True*) → List[numpy.array]

Calculates the optimal policy for an instance

Parameters

- **instance** (*List, optional*) – instance with information about function config.
- **vector_action** (*bool, optional*) – if True return multidim actions else return onedimensional action, by default True

Returns
List with entry for each timestep that holds all optimal values in an array or as int

Return type
List[np.array]

step(*self*, *action: int*)

Execute environment step

Parameters
action (*int*) – action to execute

Returns
state, reward, done, info

Return type
np.array, float, bool, dict

reset(*self*) → List[int]

Resets env

Returns
Environment state

Return type
numpy.array

get_default_reward(*self*, *_*) → float

Calculate euclidean distance between action vector and real position of Curve.

Parameters
_ (*self*) – ignore

Returns
Euclidean distance

Return type
float

get_default_state(*self*, *_*) → numpy.array

Gather state information.

Parameters
_ – ignore param

Returns
numpy array with state information

Return type
np.array

close(*self*) → bool

Close Env

Returns
Closing confirmation

Return type
bool

render(*self*, *dimensions: List*, *absolute_path: str*)

Multiplot for specific dimensions of benchmark with policy actions.

Parameters
dimensions (*List*) – List of dimensions that get plotted

render_3d_dimensions(*self*, *dimensions: List*, *absolute_path: str*)

Plot 2 Dimensions in 3D space

Parameters
dimensions (*List*) – List of dimensions that get plotted. Max 2

_pre_reward(*self*) → Tuple[numpy.ndarray, List]

Prepare actions and coordinates for reward calculation.

Returns
[description]

Return type
Tuple[np.ndarray, List]

dacbench.wrappers

Submodules

dacbench.wrappers.action_tracking_wrapper

Module Contents

Classes

ActionFrequencyWrapper	Wrapper to action frequency.

dacbench.wrappers.action_tracking_wrapper.**current_palette**

class dacbench.wrappers.action_tracking_wrapper.**ActionFrequencyWrapper**(*env*, *action_interval=None*, *logger=None*)

Bases: gym.Wrapper

Wrapper to action frequency. Includes interval mode that returns frequencies in lists of len(interval) instead of one long list.

__setattr__(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

__getattribute__(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

step(*self*, *action*)

Execute environment step and record state

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

get_actions(*self*)

Get state progression

Returns
all states or all states and interval sorted states

Return type
np.array or np.array, np.array

render_action_tracking(*self*)

Render action progression

Returns
RBG data of action tracking

Return type
np.array

dacbench.wrappers.episode_time_tracker

Module Contents

Classes

EpisodeTimeWrapper	Wrapper to track time spent per episode.

dacbench.wrappers.episode_time_tracker.**current_palette**

class dacbench.wrappers.episode_time_tracker.**EpisodeTimeWrapper**(*env*, *time_interval=None*, *logger=None*)

Bases: gym.Wrapper

Wrapper to track time spent per episode. Includes interval mode that returns times in lists of len(interval) instead of one long list.

__setattr__(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

__getattribute__(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

step(*self*, *action*)

Execute environment step and record time

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

get_times(*self*)

Get times

Returns
all times or all times and interval sorted times

Return type
np.array or np.array, np.array

render_step_time(*self*)

Render step times

render_episode_time(*self*)

Render episode times

dacbench.wrappers.instance_sampling_wrapper

Module Contents

Classes

InstanceSamplingWrapper	Wrapper to sample a new instance at a given time point.

class dacbench.wrappers.instance_sampling_wrapper.**InstanceSamplingWrapper**(*env*, *sampling_function=None*, *instances=None*, *reset_interval=0*)

Bases: gym.Wrapper

Wrapper to sample a new instance at a given time point. Instances can either be sampled using a given method or a distribution infered from a given list of instances.

__setattr__(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

__getattribute__(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

reset(*self*)

Reset environment and use sampled instance for training

Returns
state

Return type
np.array

fit_dist(*self*, *instances*)

Approximate instance distribution in given instance set

Parameters
instances (*List*) – instance set

Returns
sampling method for new instances

Return type
method

dacbench.wrappers.observation_wrapper

Module Contents

Classes

ObservationWrapper	Wrapper covert observations spaces to spaces.Box for convenience

class dacbench.wrappers.observation_wrapper.**ObservationWrapper**(*env*)

Bases: gym.Wrapper

Wrapper covert observations spaces to spaces.Box for convenience Currently only supports Dict -> Box

__setattr__(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

__getattribute__(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

step(*self*, *action*)

Execute environment step and record distance

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

reset(*self*)

Execute environment step and record distance

Returns
state

Return type
np.array

flatten(*self*, *state_dict*)

dacbench.wrappers.performance_tracking_wrapper

Module Contents

Classes

PerformanceTrackingWrapper	Wrapper to track episode performance.

dacbench.wrappers.performance_tracking_wrapper.**current_palette**

class dacbench.wrappers.performance_tracking_wrapper.**PerformanceTrackingWrapper**(*env*, *performance_interval=None*, *track_instance_performance=*
log-
ger=None)

Bases: gym.Wrapper

Wrapper to track episode performance. Includes interval mode that returns performance in lists of len(interval) instead of one long list.

__setattr__(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

__getattribute__(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

step(*self*, *action*)

Execute environment step and record performance

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

get_performance(*self*)

Get state performance

Returns
all states or all states and interval sorted states

Return type
np.array or np.array, np.array or np.array, dict or np.array, np.arry, dict

render_performance(*self*)

Plot performance

render_instance_performance(*self*)

Plot mean performance for each instance

dacbench.wrappers.policy_progress_wrapper

Module Contents

Classes

PolicyProgressWrapper	Wrapper to track progress towards optimal policy.

class dacbench.wrappers.policy_progress_wrapper.**PolicyProgressWrapper**(*env*, *compute_optimal*)

Bases: gym.Wrapper

Wrapper to track progress towards optimal policy. Can only be used if a way to obtain the optimal policy given an instance can be obtained

__setattr__(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

__getattribute__(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

step(*self*, *action*)

Execute environment step and record distance

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

render_policy_progress(*self*)

Plot progress

dacbench.wrappers.reward_noise_wrapper

Module Contents

Classes

RewardNoiseWrapper	Wrapper to add noise to the reward signal.

class dacbench.wrappers.reward_noise_wrapper.**RewardNoiseWrapper**(*env*, *noise_function=None*, *noise_dist='standard_normal'*, *dist_args=None*)

Bases: gym.Wrapper

Wrapper to add noise to the reward signal. Noise can be sampled from a custom distribution or any distribution in numpy's random module

__setattr__(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

__getattribute__(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

step(*self*, *action*)

Execute environment step and add noise

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

add_noise(*self*, *dist*, *args*)

Make noise function from distribution name and arguments

Parameters

- **dist** (*str*) – Name of distribution
- **args** (*list*) – List of distribution arguments

Returns
Noise sampling function

Return type
function

dacbench.wrappers.state_tracking_wrapper

Module Contents

Classes

StateTrackingWrapper	Wrapper to track state changed over time

dacbench.wrappers.state_tracking_wrapper.**current_palette**

class dacbench.wrappers.state_tracking_wrapper.**StateTrackingWrapper**(*env*, *state_interval=None*, *logger=None*)

Bases: gym.Wrapper

Wrapper to track state changed over time Includes interval mode that returns states in lists of len(interval) instead of one long list.

__setattr__(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

__getattribute__(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

reset(*self*)

Reset environment and record starting state

Returns
state

Return type
np.array

step(*self*, *action*)

Execute environment step and record state

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

get_states(*self*)

Get state progression

Returns
all states or all states and interval sorted states

Return type
np.array or np.array, np.array

render_state_tracking(*self*)

Render state progression

Returns
RBG data of state tracking

Return type
np.array

Package Contents

Classes

ActionFrequencyWrapper	Wrapper to action frequency.
EpisodeTimeWrapper	Wrapper to track time spent per episode.
InstanceSamplingWrapper	Wrapper to sample a new instance at a given time point.
PolicyProgressWrapper	Wrapper to track progress towards optimal policy.
RewardNoiseWrapper	Wrapper to add noise to the reward signal.
StateTrackingWrapper	Wrapper to track state changed over time
PerformanceTrackingWrapper	Wrapper to track episode performance.
ObservationWrapper	Wrapper covert observations spaces to spaces.Box for convenience

class dacbench.wrappers.**ActionFrequencyWrapper**(*env*, *action_interval=None*, *logger=None*)

Bases: gym.Wrapper

Wrapper to action frequency. Includes interval mode that returns frequencies in lists of len(interval) instead of one long list.

__setattr__(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

__getattribute__(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

`step`(*self*, *action*)

Execute environment step and record state

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

`get_actions`(*self*)

Get state progression

Returns
all states or all states and interval sorted states

Return type
np.array or np.array, np.array

`render_action_tracking`(*self*)

Render action progression

Returns
RBG data of action tracking

Return type
np.array

`class` dacbench.wrappers.**EpisodeTimeWrapper**(*env*, *time_interval=None*, *logger=None*)

Bases: gym.Wrapper

Wrapper to track time spent per episode. Includes interval mode that returns times in lists of len(interval) instead of one long list.

`__setattr__`(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

`__getattribute__`(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

step(*self*, *action*)

Execute environment step and record time

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

get_times(*self*)

Get times

Returns
all times or all times and interval sorted times

Return type
np.array or np.array, np.array

render_step_time(*self*)

Render step times

render_episode_time(*self*)

Render episode times

class dacbench.wrappers.**InstanceSamplingWrapper**(*env*, *sampling_function=None*, *instances=None*, *reset_interval=0*)

Bases: gym.Wrapper

Wrapper to sample a new instance at a given time point. Instances can either be sampled using a given method or a distribution infered from a given list of instances.

__setattr__(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

__getattribute__(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

reset(*self*)

Reset environment and use sampled instance for training

Returns
state

Return type
np.array

fit_dist(*self*, *instances*)

Approximate instance distribution in given instance set

Parameters
instances (*List*) – instance set

Returns
sampling method for new instances

Return type
method

class dacbench.wrappers.**PolicyProgressWrapper**(*env*, *compute_optimal*)

Bases: gym.Wrapper

Wrapper to track progress towards optimal policy. Can only be used if a way to obtain the optimal policy given an instance can be obtained

__setattr__(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

__getattribute__(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

step(*self*, *action*)

Execute environment step and record distance

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

render_policy_progress(*self*)

Plot progress

class dacbench.wrappers.**RewardNoiseWrapper**(*env*, *noise_function=None*, *noise_dist='standard_normal'*, *dist_args=None*)

Bases: gym.Wrapper

Wrapper to add noise to the reward signal. Noise can be sampled from a custom distribution or any distribution in numpy's random module

__setattr__(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

__getattribute__(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

step(*self*, *action*)

Execute environment step and add noise

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

add_noise(*self*, *dist*, *args*)

Make noise function from distribution name and arguments

Parameters

- **dist** (*str*) – Name of distribution
- **args** (*list*) – List of distribution arguments

Returns
Noise sampling function

Return type
function

class dacbench.wrappers.**StateTrackingWrapper**(*env*, *state_interval=None*, *logger=None*)

Bases: gym.Wrapper

Wrapper to track state changed over time Includes interval mode that returns states in lists of len(interval) instead of one long list.

__setattr__(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

__getattribute__(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

reset(*self*)

Reset environment and record starting state

Returns
state

Return type
np.array

step(*self*, *action*)

Execute environment step and record state

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

get_states(*self*)

Get state progression

Returns
all states or all states and interval sorted states

Return type
np.array or np.array, np.array

render_state_tracking(*self*)

Render state progression

Returns
RBG data of state tracking

Return type
np.array

class dacbench.wrappers.**PerformanceTrackingWrapper**(*env*, *performance_interval=None*, *track_instance_performance=True*, *logger=None*)

Bases: gym.Wrapper

Wrapper to track episode performance. Includes interval mode that returns performance in lists of len(interval) instead of one long list.

__setattr__(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

__getattribute__(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

step(*self*, *action*)

Execute environment step and record performance

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

get_performance(*self*)

Get state performance

Returns
all states or all states and interval sorted states

Return type
np.array or np.array, np.array or np.array, dict or np.array, np.arry, dict

render_performance(*self*)

Plot performance

render_instance_performance(*self*)

Plot mean performance for each instance

class dacbench.wrappers.ObservationWrapper(*env*)

Bases: gym.Wrapper

Wrapper covert observations spaces to spaces.Box for convenience Currently only supports Dict -> Box

__setattr__(*self*, *name*, *value*)

Set attribute in wrapper if available and in env if not

Parameters

- **name** (*str*) – Attribute to set
- **value** – Value to set attribute to

__getattribute__(*self*, *name*)

Get attribute value of wrapper if available and of env if not

Parameters
name (*str*) – Attribute to get

Returns
Value of given name

Return type
value

step(*self*, *action*)

Execute environment step and record distance

Parameters
action (*int*) – action to execute

Returns
state, reward, done, metainfo

Return type
np.array, float, bool, dict

reset(*self*)

Execute environment step and record distance

Returns
state

Return type
np.array

flatten(*self*, *state_dict*)

21.1.2 Submodules

dacbench.abstract_agent

Module Contents

Classes

AbstractDACBenchAgent	Abstract class to implement for use with the runner function

class dacbench.abstract_agent.**AbstractDACBenchAgent**(*env*)

Abstract class to implement for use with the runner function

abstract act(*self*, *state*, *reward*)

Compute and return environment action

Parameters

- **state** – Environment state
- **reward** – Environment reward

Returns
Action to take

Return type
action

abstract train(*self*, *next_state*, *reward*)

Train during episode if needed (pass if not)

Parameters

- **next_state** – Environment state after step
- **reward** – Environment reward

abstract end_episode(*self*, *state*, *reward*)

End of episode training if needed (pass if not)

Parameters

- **state** – Environment state
- **reward** – Environment reward

dacbench.abstract_benchmark

Module Contents

Classes

AbstractBenchmark	Abstract template for benchmark classes
objdict	Modified dict to make config changes more flexible

class dacbench.abstract_benchmark.**AbstractBenchmark**(*config_path=None*, *config:* dacbench.abstract_benchmark.objdict = *None*)

Abstract template for benchmark classes

get_config(*self*)

Return current configuration

Returns
Current config

Return type
dict

serialize_config(*self*)

Save configuration to json

Parameters
path (*str*) – File to save config to

process_configspace(*self*, *configuration_space*)

This is largely the builting cs.json.write method, but doesn't save the result directly If this is ever implemented in cs, we can replace this method

classmethod from_json(*cls*, *json_config*)

to_json(*self*)

save_config(*self*, *path*)

jsonify_wrappers(*self*)

dejson_wrappers(*self*, *wrapper_list*)

static __import_from(*module: str*, *name: str*)

Imports the class / function / … with name from module :param module: :param name:

Returns

Return type
the imported object

classmethod class_to_str(*cls*)

static __decorate_config_with_functions(*conf: dict*)

Replaced the stringified functions with the callable objects :param config:

static __stringify_functions(*conf: dict*) → dict

Replaced all callables in the config with a triple ('function', module_name, function_name)

Parameters
config –

Returns

Return type
modified dict

space_to_list(*self*, *space*)

list_to_space(*self*, *space_list*)

jsonify_dict_space(*self*, *dict_space*)

dictify_json(*self*, *dict_list*)

load_config(*self*, *config:* dacbench.abstract_benchmark.objdict)

read_config_file(*self*, *path*)

Read configuration from file

Parameters
path (*str*) – Path to config file

abstract get_environment(*self*)

Make benchmark environment

Returns
env – Benchmark environment

Return type
gym.Env

set_seed(*self*, *seed*)

Set environment seed

Parameters
seed (*int*) – New seed

set_action_space(*self*, *kind*, *args*)

Change action space

Parameters

- **kind** (*str*) – Name of action space class

- **args** (*list*) – List of arguments to pass to action space class

set_observation_space(*self*, *kind*, *args*, *data_type*)

Change observation_space

Parameters

- **kind** (*str*) – Name of observation space class
- **args** (*list*) – List of arguments to pass to observation space class
- **data_type** (*type*) – Data type of observation space

register_wrapper(*self*, *wrap_func*)

__eq__(*self*, *other*)

Return self==value.

class dacbench.abstract_benchmark.**objdict**

Bases: dict

Modified dict to make config changes more flexible

__getattr__(*self*, *name*)

__setattr__(*self*, *name*, *value*)

Implement setattr(self, name, value).

__delattr__(*self*, *name*)

Implement delattr(self, name).

copy(*self*)

D.copy() -> a shallow copy of D

__eq__(*self*, *other*)

return isinstance(other, dict) and set(other.keys()) == set(self.keys()) and all(

np.array_equal(self[key], other[key]) if any(isinstance(obj[key], np.ndarray) for obj in (self, other)) else other[key] == self[key]

for key in self.keys()

)

__ne__(*self*, *other*)

Return self!=value.

dacbench.abstract_env

Module Contents

Classes

AbstractEnv	Abstract template for environments

class dacbench.abstract_env.**AbstractEnv**(*config*)

Bases: gym.Env

Abstract template for environments

step_(*self*)

Pre-step function for step count and cutoff

Returns
End of episode

Return type
bool

reset_(*self*, *instance=None*, *instance_id=None*, *scheme=None*)

Pre-reset function for progressing through the instance set Will either use round robin, random or no progression scheme

use_next_instance(*self*, *instance=None*, *instance_id=None*, *scheme=None*)

Changes instance according to chosen instance progession

Parameters

- **instance** – Instance specification for potentional new instances
- **instance_id** – ID of the instance to switch to
- **scheme** – Update scheme for this progression step (either round robin, random or no progression)

abstract step(*self*, *action*)

Execute environment step

Parameters
action – Action to take

Returns

- *state* – Environment state
- *reward* – Environment reward
- **done** (*bool*) – Run finished flag
- **info** (*dict*) – Additional metainfo

abstract reset(*self*)

Reset environment

Returns
Environment state

Return type
state

get_inst_id(*self*)

Return instance ID

Returns
ID of current instance

Return type
int

get_instance_set(*self*)

Return instance set

Returns
List of instances

Return type
list

get_instance(*self*)

Return current instance

Returns
Currently used instance

Return type
type flexible

set_inst_id(*self*, *inst_id*)

Change current instance ID

Parameters
inst_id (*int*) – New instance index

set_instance_set(*self*, *inst_set*)

Change instance set

Parameters
inst_set (*list*) – New instance set

set_instance(*self*, *instance*)

Change currently used instance

Parameters
instance – New instance

seed_action_space(*self*, *seed=None*)

Seeds the action space. :param seed: if None self.initial_seed is be used :type seed: int, default None

seed(*self*, *seed=None*, *seed_action_space=False*)

Set rng seed

Parameters

- **seed** – seed for rng
- **seed_action_space** (*bool, default False*) – if to seed the action space as well

use_test_set(*self*)

Change to test instance set

use_training_set(*self*)

Change to training instance set

dacbench.argument_parsing

Module Contents

Classes

PathType	Custom argument type for path validation.

class dacbench.argument_parsing.**PathType**(*exists=True*, *type='file'*, *dash_ok=True*)

Bases: object

Custom argument type for path validation.

Adapted from: https://stackoverflow.com/questions/11415570/directory-path-types-with-argparse

__call__(*self*, *string: str*)

dacbench.logger

Module Contents

Classes

AbstractLogger	Logger interface.
ModuleLogger	A logger for handling logging of one module. e.g. a wrapper or toplevel general logging.
Logger	A logger that manages the creation of the module loggers.

Functions

load_logs(log_file: pathlib.Path) → List[Dict]	Loads the logs from a jsonl written by any logger.
split(predicate: Callable, iterable: Iterable) → Tuple[(List, List)]	Splits the iterable into two list depending on the result of predicate.
flatten_log_entry(log_entry: Dict) → List[Dict]	Transforms a log entry of format like
list_to_tuple(list_: List) → Tuple	Recursively transforms a list of lists into tuples of tuples
log2dataframe(logs: List[dict], wide: bool = False, drop_columns: List[str] = ['time']) → pandas.DataFrame	Converts a list of log entries to a pandas dataframe.
seed_mapper(self)	
instance_mapper(self)	

dacbench.logger.**load_logs**(*log_file: pathlib.Path*) → List[Dict]

Loads the logs from a jsonl written by any logger.

The result is the list of dicts in the format: {

'instance': 0, 'episode': 0, 'step': 1, 'example_log_val': {

'values': [val1, val2, … valn], 'times: [time1, time2, …, timen],

} :param log_file: The path to the log file :type log_file: pathlib.Path

Returns

Return type
[Dict, …]

dacbench.logger.**split**(*predicate: Callable*, *iterable: Iterable*) → Tuple[List, List]

Splits the iterable into two list depending on the result of predicate.

Parameters

- **predicate** (*Callable*) – A function taking an element of the iterable and return Ture or False
- **iterable** (*Iterable*) –

Returns

Return type
(positives, negatives)

dacbench.logger.**flatten_log_entry**(*log_entry: Dict*) → List[Dict]

Transforms a log entry of format like

{

'step': 0, 'episode': 2, 'some_value': {

'values' : [34, 45], 'times':['28-12-20 16:20:53', '28-12-20 16:21:30'],

}

} into [

{ 'step': 0,'episode': 2, 'value': 34, 'time': '28-12-20 16:20:53'}, { 'step': 0,'episode': 2, 'value': 45, 'time': '28-12-20 16:21:30'}

]

Parameters
log_entry (*Dict*) – A log entry

dacbench.logger.**list_to_tuple**(*list_: List*) → Tuple

Recursively transforms a list of lists into tuples of tuples :param list_: (nested) list

Returns

Return type
(nested) tuple

dacbench.logger.**log2dataframe**(*logs: List[dict]*, *wide: bool = False*, *drop_columns: List[str] = ['time']*) → pandas.DataFrame

Converts a list of log entries to a pandas dataframe.

Usually used in combination with load_dataframe.

Parameters

- **logs** (*List*) – List of log entries

- **wide** (*bool*) – wide=False (default) produces a dataframe with columns (episode, step, time, name, value) wide=True returns a dataframe (episode, step, time, name_1, name_2, ...) if the variable name_n has not been logged at (episode, step, time) name_n is NaN.
- **drop_columns** (*List[str]*) – List of column names to be dropped (before reshaping the long dataframe) mostly used in combination with wide=True to reduce NaN values

Returns

Return type
dataframe

dacbench.logger.**seed_mapper**(*self*)

dacbench.logger.**instance_mapper**(*self*)

class dacbench.logger.**AbstractLogger**(*experiment_name: str*, *output_path: pathlib.Path*, *step_write_frequency: int = None*, *episode_write_frequency: int = 1*)

Logger interface.

The logger classes provide a way of writing structured logs as jsonl files and also help to track information like current episode, step, time ...

In the jsonl log file each row corresponds to a step.

valid_types

property additional_info(*self*)

set_env(*self*, *env:* dacbench.AbstractEnv) → None

Needed to infer automatically logged information like the instance id :param env: :type env: AbstractEnv

static _pretty_valid_types() → str

Returns a string pretty string representation of the types that can be logged as values

static _init_logging_dir(*log_dir: pathlib.Path*) → None

Prepares the logging directory :param log_dir: :type log_dir: pathlib.Path

Returns

Return type
None

is_of_valid_type(*self*, *value: Any*) → bool

abstract close(*self*) → None

Makes sure, that all remaining entries in the are written to file and the file is closed.

abstract next_step(*self*) → None

Call at the end of the step. Updates the internal state and dumps the information of the last step into a json

abstract next_episode(*self*) → None

Call at the end of episode.

See next_step

abstract write(*self*) → None

Writes buffered logs to file.

Invoke manually if you want to load logs during a run.

abstract log(*self*, *key: str*, *value*) → None

abstract log_dict(*self*, *data*)

Alternative to log if more the one value should be logged at once.

Parameters
data (*dict*) – a dict with key-value so that each value is a valid value for log

abstract log_space(*self*, *key: str*, *value: Union[numpy.ndarray, Dict]*, *space_info=None*)

Special for logging gym.spaces.

Currently three types are supported: * Numbers: e.g. samples from Discrete * Fixed length arrays like MultiDiscrete or Box * Dict: assuming each key has fixed length array

Parameters

- **key** – see log
- **value** – see log
- **space_info** – a list of column names. The length of this list must equal the resulting number of columns.

class dacbench.logger.**ModuleLogger**(*output_path: pathlib.Path*, *experiment_name: str*, *module: str*, *step_write_frequency: int = None*, *episode_write_frequency: int = 1*)

Bases: *dacbench.logger.AbstractLogger*

A logger for handling logging of one module. e.g. a wrapper or toplevel general logging.

Don't create manually use Logger to manage ModuleLoggers

get_logfile(*self*) → pathlib.Path

Returns
the path to the log file of this logger

Return type
pathlib.Path

close(*self*)

Makes sure, that all remaining entries in the are written to file and the file is closed.

__del__(*self*)

static __json_default(*object*)

Add supoort for dumping numpy arrays and numbers to json :param object:

__end_step(*self*)

static __init_dict()

reset_episode(*self*) → None

Resets the episode and step.

Be aware that this can lead to ambitious keys if no instance or seed or other identifying additional info is set

Returns

__reset_step(*self*)

next_step(*self*)

Call at the end of the step. Updates the internal state and dumps the information of the last step into a json

next_episode(*self*)

Writes buffered logs to file.

Invoke manually if you want to load logs during a run.

write(*self*)

Writes buffered logs to file.

Invoke manually if you want to load logs during a run.

__buffer_to_file(*self*)

set_additional_info(*self*, ***kwargs*)

Can be used to log additional information for each step e.g. for seed, and instance id. :param kwargs:

log(*self*, *key: str*, *value: Union[Dict, List, Tuple, str, int, float, bool]*) → None

__log(*self*, *key*, *value*, *time*)

log_dict(*self*, *data: Dict*) → None

Alternative to log if more the one value should be logged at once.

Parameters

data (*dict*) – a dict with key-value so that each value is a valid value for log

static __space_dict(*key: str*, *value*, *space_info*)

log_space(*self*, *key*, *value*, *space_info=None*)

Special for logging gym.spaces.

Currently three types are supported: * Numbers: e.g. samples from Discrete * Fixed length arrays like MultiDiscrete or Box * Dict: assuming each key has fixed length array

Parameters

- **key** – see log
- **value** – see log
- **space_info** – a list of column names. The length of this list must equal the resulting number of columns.

class dacbench.logger.**Logger**(*experiment_name: str*, *output_path: pathlib.Path*, *step_write_frequency: int = None*, *episode_write_frequency: int = 1*)

Bases: *dacbench.logger.AbstractLogger*

A logger that manages the creation of the module loggers.

To get a ModuleLogger for you module (e.g. wrapper) call module_logger = Logger(...).add_module("my_wrapper"). From now on module_logger.log(...) or logger.log(..., module="my_wrapper") can be used to log.

The logger module takes care of updating information like episode and step in the subloggers. To indicate to the loggers the end of the episode or the next_step simple call logger.next_episode() or logger.next_step().

set_env(*self*, *env:* dacbench.AbstractEnv) → None

Needed to infer automatically logged information like the instance id :param env: :type env: AbstractEnv

close(*self*)

Makes sure, that all remaining entries (from all sublogger) are written to files and the files are closed.

__del__(*self*)

next_step(*self*)

Call at the end of the step. Updates the internal state of all subloggers and dumps the information of the last step into a json

next_episode(*self*)

Call at the end of episode.

See next_step

reset_episode(*self*)

write(*self*)

Writes buffered logs to file.

Invoke manually if you want to load logs during a run.

add_module(*self*, *module: Union[str, type]*) → *dacbench.logger.ModuleLogger*

Creates a sub-logger. For more details see class level documentation :param module: The module name or Wrapper-Type to create a sub-logger for :type module: str or type

Returns

Return type
ModuleLogger

add_agent(*self*, *agent:* dacbench.abstract_agent.AbstractDACBenchAgent)

Writes information about the agent :param agent: :type agent: AbstractDACBenchAgent

add_benchmark(*self*, *benchmark:* dacbench.AbstractBenchmark) → None

Writes the config to the experiment path :param benchmark:

set_additional_info(*self*, ***kwargs*)

log(*self*, *key*, *value*, *module*)

log_space(*self*, *key*, *value*, *module*, *space_info=None*)

Special for logging gym.spaces.

Currently three types are supported: * Numbers: e.g. samples from Discrete * Fixed length arrays like MultiDiscrete or Box * Dict: assuming each key has fixed length array

Parameters

- **key** – see log
- **value** – see log
- **space_info** – a list of column names. The length of this list must equal the resulting number of columns.

log_dict(*self*, *data*, *module*)

Alternative to log if more the one value should be logged at once.

Parameters
data (*dict*) – a dict with key-value so that each value is a valid value for log

dacbench.plotting

Module Contents

Functions

space_sep_upper(column_name: str) → str	Separates strings at underscores into headings.
generate_global_step(data: pandas.DataFrame, x_column: str = 'global_step', x_label_columns: str = ['episode', 'step']) → Tuple[(pandas.DataFrame, str, List[str])]	Add a global_step column which enumerate all step over all episodes.
add_multi_level_ticks(grid: seaborn.FacetGrid, plot_index: pandas.DataFrame, x_column: str, x_label_columns: str) → None	Expects a FacedGrid with global_step (x_column) as x-axis and replaces the tick labels to match format episode:step
plot(plot_function, settings: dict, title: str = None, x_label: str = None, y_label: str = None, **kwargs) → seaborn.FacetGrid	Helper function that: create a FacetGrid
plot_performance(data, title=None, x_label=None, y_label=None, **kwargs) → seaborn.FacetGrid	Create a line plot of the performance over episodes.
plot_performance_per_instance(data, title=None, x_label=None, y_label=None, **args) → seaborn.FacetGrid	Create a bar plot of the mean performance per instance ordered by the performance.
plot_step_time(data, show_global_step=False, interval=1, title=None, x_label=None, y_label=None, **args) → seaborn.FacetGrid	Create a line plot showing the measured time per step.
plot_episode_time(data, title=None, x_label=None, y_label=None, **kargs) → seaborn.FacetGrid	Create a line plot showing the measured time per episode.
plot_action(data, show_global_step=False, interval=1, title=None, x_label=None, y_label=None, **kargs)	Create a line plot showing actions over time.
plot_state(data, show_global_step=False, interval=1, title=None, x_label=None, y_label=None, **kargs)	Create a line plot showing state over time.
plot_space(data, space_column_name, show_global_step, interval=1, title=None, x_label=None, y_label=None, **args) → seaborn.FacetGrid	Create a line plot showing sapce over time.

dacbench.plotting.**space_sep_upper**(*column_name: str*) → str

Separates strings at underscores into headings. Used to generate labels from logging names.

Parameters

column_name (*str*) –

Returns

Return type

str

dacbench.plotting.**generate_global_step**(*data: pandas.DataFrame*, *x_column: str = 'global_step'*, *x_label_columns: str = ['episode', 'step']*) → Tuple[pandas.DataFrame, str, List[str]]

Add a global_step column which enumerate all step over all episodes.

Returns the altered data, a data frame containing mapping between global_step, x_column and x_label_columns.

Often used in combination with add_multi_level_ticks.

Parameters

- **data** –
- **x_column** (*str*) – the name of the global_step (default 'global_step')
- **x_label_columns** (*[str, ...]*) – the name and hierarchical order of the columns (default ['episode', 'step']

Returns

Return type
(data, plot_index, x_column, x_label_columns)

dacbench.plotting.**add_multi_level_ticks**(*grid: seaborn.FacetGrid*, *plot_index: pandas.DataFrame*, *x_column: str*, *x_label_columns: str*) → None

Expects a FacedGrid with global_step (x_column) as x-axis and replaces the tick labels to match format episode:step

E.g. Run with 3 episodes, each of 10 steps. This results in 30 global steps. The resulting tick labels could be ['0', '4', '9', '14', '19', '24', '29']. After applying this method they will look like ['0:0', '0:4', '1:0', '1:4', '2:0', '2:4', '3:0', '3:4']

Parameters

- **grid** (*sns.FacesGrid*) –
- **plot_index** (*pd.DataFrame*) – The mapping between current tick labels (global step values) and new tick labels joined by ':'. usually the result from generate_global_step
- **x_column** (*str*) – column label to use for looking up tick values
- **x_label_columns** (*[str, ...]*) – columns labels of columns to use for new labels (joined by ':'

dacbench.plotting.**plot**(*plot_function*, *settings: dict*, *title: str = None*, *x_label: str = None*, *y_label: str = None*, ***kwargs*) → seaborn.FacetGrid

Helper function that: create a FacetGrid 1. Updates settings with kwargs (overwrites values) 2. Plots using plot_function(**settings) 3. Set x and y labels of not provided the columns names will converted to pretty strings using space_sep_upper 4. Sets title (some times has to be readjusted afterwards especially in case of large plots e.g. multiple rows/cols)

Parameters

- **plot_function** – function to generate the FacedGrid. E.g. sns.catplot or sns.catplot
- **settings** (*dict*) – a dicts containing all needed default settings.
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.plotting.**plot_performance**(*data*, *title=None*, *x_label=None*, *y_label=None*, ***kwargs*) → seaborn.FacetGrid

Create a line plot of the performance over episodes.

Per default the mean performance and and one stddev over all instances and seeds is shown if you want to change this specify a property to map those attributes to e.g hue='seed' or/and col='instance'. For more details see: https://seaborn.pydata.org/generated/seaborn.relplot.html

For examples refer to examples/plotting/performance_plotting.py

Parameters

- **data** (*pd.DataFrame*) – Dataframe resulting from logging and loading using log2dataframe(logs, wide=True)
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.plotting.**plot_performance_per_instance**(*data*, *title=None*, *x_label=None*, *y_label=None*, ***args*) → seaborn.FacetGrid

Create a bar plot of the mean performance per instance ordered by the performance.

Per default the mean performance seeds is shown if you want to change this specify a property to map seed to e.g. col='seed'. For more details see: https://seaborn.pydata.org/generated/seaborn.catplot.html

For examples refer to examples/plotting/performance_plotting.py

Parameters

- **data** (*pd.DataFrame*) – Dataframe resulting from logging and loading using log2dataframe(logs, wide=True)
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.plotting.**plot_step_time**(*data*, *show_global_step=False*, *interval=1*, *title=None*, *x_label=None*, *y_label=None*, ***args*) → seaborn.FacetGrid

Create a line plot showing the measured time per step.

Per default the mean performance and and one stddev over all instances and seeds is shown if you want to change this specify a property to map those attributes to e.g hue='seed' or/and col='instance'. For more details see: https://seaborn.pydata.org/generated/seaborn.relplot.html

For examples refer to examples/plotting/time_plotting.py

Parameters

- **data** (*pd.DataFrame*) – Dataframe resulting from logging and loading using log2dataframe(logs, wide=True)
- **show_global_step** (*bool*) – If to show the global_step (step enumerated over all episodes) or Episode:Step. (False default)
- **interval** (*int*) – Interval in number of steps to average over. (default = 1)
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.plotting.**plot_episode_time**(*data*, *title=None*, *x_label=None*, *y_label=None*, ***kargs*) → seaborn.FacetGrid

Create a line plot showing the measured time per episode.

Per default the mean performance and and one stddev over all instances and seeds is shown if you want to change this specify a property to map those attributes to e.g hue='seed' or/and col='instance'. For more details see: https://seaborn.pydata.org/generated/seaborn.relplot.html

For examples refer to examples/plotting/time_plotting.py

Parameters

- **data** (*pd.DataFrame*) – Dataframe resulting from logging and loading using log2dataframe(logs, wide=True)
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.plotting.**plot_action**(*data*, *show_global_step=False*, *interval=1*, *title=None*, *x_label=None*, *y_label=None*, ***kargs*)

Create a line plot showing actions over time.

Please be aware that action spaces can be quite large and the plots can become quite messy (and take some time) if you try plot all dimensions at once. It is therefore recommended to select a subset of columns before running the plot method.

Per default the mean performance and and one stddev over all instances and seeds is shown if you want to change this specify a property to map those attributes to e.g hue='seed' or/and col='instance'. For more details see: https://seaborn.pydata.org/generated/seaborn.relplot.html

For examples refer to examples/plotting/action_plotting.py

Parameters

- **data** (*pd.DataFrame*) – Dataframe resulting from logging and loading using log2dataframe(logs, wide=True)
- **show_global_step** (*bool*) – If to show the global_step (step enumerated over all episodes) or Episode:Step. (False default)
- **interval** (*int*) – Interval in number of steps to average over. (default = 1)
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.plotting.**plot_state**(*data*, *show_global_step=False*, *interval=1*, *title=None*, *x_label=None*, *y_label=None*, ***kargs*)

Create a line plot showing state over time.

Please be aware that state can be quite large and the plots can become quite messy (and take some time) if you try plot all dimensions at once. It is therefore recommended to select a subset of columns before running the plot method. Especially for dict state spaces.

Per default the mean performance and and one stddev over all instances and seeds is shown if you want to change this specify a property to map those attributes to e.g hue='seed' or/and col='instance'. For more details see: https://seaborn.pydata.org/generated/seaborn.relplot.html

For examples refer to examples/plotting/state_plotting.py

Parameters

- **data** (*pd.DataFrame*) – Dataframe resulting from logging and loading using log2dataframe(logs, wide=True)
- **show_global_step** (*bool*) – If to show the global_step (step enumerated over all episodes) or Episode:Step. (False default)
- **interval** (*int*) – Interval in number of steps to average over. (default = 1)
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.plotting.**plot_space**(*data*, *space_column_name*, *show_global_step*, *interval=1*, *title=None*, *x_label=None*, *y_label=None*, ***args*) → seaborn.FacetGrid

Create a line plot showing sapce over time.

Please be aware that spaces can be quite large and the plots can become quite messy (and take some time) if you try plot all dimensions at once. It is therefore recommended to select a subset of columns before running the plot method. Especially for dict spaces.

Per default the mean performance and and one stddev over all instances and seeds is shown if you want to change this specify a property to map those attributes to e.g hue='seed' or/and col='instance'. For more details see: https://seaborn.pydata.org/generated/seaborn.relplot.html

For examples refer to
examples/plotting/state_plotting.py or examples/plotting/action_plotting.py

Parameters

- **data** (*pd.DataFrame*) – Dataframe resulting from logging and loading using log2dataframe(logs, wide=True)
- **show_global_step** (*bool*) – If to show the global_step (step enumerated over all episodes) or Episode:Step. (False default)
- **interval** (*int*) – Interval in number of steps to average over. (default = 1)
- **title** (*str*) – Title of the plot (optional)
- **x_label** (*str*) – Label of the x-axis (optional)
- **y_label** (*str*) – Label of the y-axis (optional)
- **kwargs** – Keyword arguments to overwrite default settings.

Returns

Return type
sns.FacedGrid

dacbench.run_baselines

Module Contents

Functions

run_random(results_path, benchmark_name, num_episodes, seeds, fixed)	
run_static(results_path, benchmark_name, action, num_episodes, seeds=np.arange(10))	
run_optimal(results_path, benchmark_name, num_episodes, seeds)	
run_dynamic_policy(results_path, benchmark_name, num_episodes, seeds=np.arange(10))	
run_policy(results_path, benchmark_name, num_episodes, policy, seeds=np.arange(10))	
main(args)	

dacbench.run_baselines.**modea_actions**

dacbench.run_baselines.**DISCRETE_ACTIONS**

dacbench.run_baselines.**run_random**(*results_path*, *benchmark_name*, *num_episodes*, *seeds*, *fixed*)

dacbench.run_baselines.**run_static**(*results_path*, *benchmark_name*, *action*, *num_episodes*, *seeds=np.arange(10)*)

dacbench.run_baselines.**run_optimal**(*results_path*, *benchmark_name*, *num_episodes*, *seeds*)

dacbench.run_baselines.**run_dynamic_policy**(*results_path*, *benchmark_name*, *num_episodes*, *seeds=np.arange(10)*)

dacbench.run_baselines.**run_policy**(*results_path*, *benchmark_name*, *num_episodes*, *policy*, *seeds=np.arange(10)*)

dacbench.run_baselines.**main**(*args*)

dacbench.runner

Module Contents

Functions

run_benchmark(env, agent, num_episodes, logger=None)	Run single benchmark env for a given number of episodes with a given agent
run_dacbench(results_path, agent_method, num_episodes, bench=None, seeds=None)	Run all benchmarks for 10 seeds for a given number of episodes with a given agent and save result

dacbench.runner.**current_palette**

dacbench.runner.**run_benchmark**(*env*, *agent*, *num_episodes*, *logger=None*)

Run single benchmark env for a given number of episodes with a given agent

Parameters

- **env** (*gym.Env*) – Benchmark environment
- **agent** – Any agent implementing the methods act, train and end_episode (see AbstractDACBenchAgent below)
- **num_episodes** (*int*) – Number of episodes to run
- **logger** (*dacbench.logger.Logger: logger to use for logging. Not closed automatically like env*) –

dacbench.runner.**run_dacbench**(*results_path*, *agent_method*, *num_episodes*, *bench=None*, *seeds=None*)

Run all benchmarks for 10 seeds for a given number of episodes with a given agent and save result

Parameters

- **bench** –
- **results_path** (*str*) – Path to where results should be saved
- **agent_method** (*function*) – Method that takes an env as input and returns an agent
- **num_episodes** (*int*) – Number of episodes to run for each benchmark
- **seeds** (*list[int]*) – List of seeds to runs all benchmarks for. If None (default) seeds [1, . . . , 10] are used.

21.1.3 Package Contents

Classes

AbstractEnv	Abstract template for environments
AbstractBenchmark	Abstract template for benchmark classes

class dacbench.**AbstractEnv**(*config*)

Bases: gym.Env

Abstract template for environments

step_(*self*)

Pre-step function for step count and cutoff

Returns
End of episode

Return type
bool

reset_(*self*, *instance=None*, *instance_id=None*, *scheme=None*)

Pre-reset function for progressing through the instance set Will either use round robin, random or no progression scheme

use_next_instance(*self*, *instance=None*, *instance_id=None*, *scheme=None*)

Changes instance according to chosen instance progession

Parameters

- **instance** – Instance specification for potentional new instances
- **instance_id** – ID of the instance to switch to
- **scheme** – Update scheme for this progression step (either round robin, random or no progression)

abstract step(*self*, *action*)

Execute environment step

Parameters
action – Action to take

Returns

- *state* – Environment state
- *reward* – Environment reward
- **done** (*bool*) – Run finished flag
- **info** (*dict*) – Additional metainfo

abstract reset(*self*)

Reset environment

Returns
Environment state

Return type
state

get_inst_id(*self*)

Return instance ID

Returns
ID of current instance

Return type
int

get_instance_set(*self*)

Return instance set

Returns
List of instances

Return type
list

get_instance(*self*)

Return current instance

Returns
Currently used instance

Return type
type flexible

set_inst_id(*self*, *inst_id*)

Change current instance ID

Parameters
inst_id (*int*) – New instance index

set_instance_set(*self*, *inst_set*)

Change instance set

Parameters
inst_set (*list*) – New instance set

set_instance(*self*, *instance*)

Change currently used instance

Parameters
instance – New instance

seed_action_space(*self*, *seed=None*)

Seeds the action space. :param seed: if None self.initial_seed is be used :type seed: int, default None

seed(*self*, *seed=None*, *seed_action_space=False*)

Set rng seed

Parameters

- **seed** – seed for rng
- **seed_action_space** (*bool, default False*) – if to seed the action space as well

use_test_set(*self*)

Change to test instance set

use_training_set(*self*)

Change to training instance set

class dacbench.**AbstractBenchmark**(*config_path=None*, *config:* dacbench.abstract_benchmark.objdict = *None*)

Abstract template for benchmark classes

get_config(*self*)

Return current configuration

Returns
Current config

Return type
dict

serialize_config(*self*)

Save configuration to json

Parameters
path (*str*) – File to save config to

process_configspace(*self*, *configuration_space*)

This is largely the builting cs.json.write method, but doesn't save the result directly If this is ever implemented in cs, we can replace this method

classmethod from_json(*cls*, *json_config*)

to_json(*self*)

save_config(*self*, *path*)

jsonify_wrappers(*self*)

dejson_wrappers(*self*, *wrapper_list*)

static __import_from(*module: str*, *name: str*)

Imports the class / function / ... with name from module :param module: :param name:

Returns

Return type
the imported object

classmethod class_to_str(*cls*)

static __decorate_config_with_functions(*conf: dict*)

Replaced the stringified functions with the callable objects :param config:

static __stringify_functions(*conf: dict*) → dict

Replaced all callables in the config with a triple ('function', module_name, function_name)

Parameters
config –

Returns

Return type
modified dict

space_to_list(*self*, *space*)

list_to_space(*self*, *space_list*)

jsonify_dict_space(*self*, *dict_space*)

dictify_json(*self*, *dict_list*)

load_config(*self*, *config:* dacbench.abstract_benchmark.objdict)

read_config_file(*self*, *path*)

Read configuration from file

Parameters
path (*str*) – Path to config file

abstract get_environment(*self*)

Make benchmark environment

Returns
env – Benchmark environment

Return type
gym.Env

set_seed(*self*, *seed*)

Set environment seed

Parameters
seed (*int*) – New seed

set_action_space(*self*, *kind*, *args*)

Change action space

Parameters

- **kind** (*str*) – Name of action space class
- **args** (*list*) – List of arguments to pass to action space class

set_observation_space(*self*, *kind*, *args*, *data_type*)

Change observation_space

Parameters

- **kind** (*str*) – Name of observation space class
- **args** (*list*) – List of arguments to pass to observation space class
- **data_type** (*type*) – Data type of observation space

register_wrapper(*self*, *wrap_func*)

__eq__(*self*, *other*)

Return self==value.

21.2 random_states

21.2.1 Module Contents

Functions

small_random_luby_state(self)	
random_luby_state(self)	
small_random_sigmoid_state(self)	
random_sigmoid_state(self)	

random_states.**small_random_luby_state**(*self*)

random_states.**random_luby_state**(*self*)

random_states.**small_random_sigmoid_state**(*self*)

random_states.**random_sigmoid_state**(*self*)

21.3 reward_functions

21.3.1 Module Contents

Functions

easy_sigmoid(self)	
almost_easy_sigmoid(self)	
sum_reward(self)	
random_reward(self)	
manhattan_distance_reward_geometric(self)	
quadratic_manhattan_distance_reward_geometric(self)	
quadratic_euclidean_distance_reward_geometric(self)	
multiply_reward_geometric(self)	

reward_functions.**easy_sigmoid**(*self*)

reward_functions.**almost_easy_sigmoid**(*self*)

reward_functions.**sum_reward**(*self*)

reward_functions.**random_reward**(*self*)

reward_functions.**manhattan_distance_reward_geometric**(*self*)

reward_functions.**quadratic_manhattan_distance_reward_geometric**(*self*)

reward_functions.**quadratic_euclidean_distance_reward_geometric**(*self*)

reward_functions.**multiply_reward_geometric**(*self*)

21.4 SampleGeometricInstances

21.4.1 Module Contents

Functions

save_geometric_instances(filename: str, config: Dict = FUNCTION_CONFIG, path: str = '')	First delete old isntance_set.
_create_csv_string(index, func_name: str) → str	Create comma separated string with function name and parameter values.
sample_sinus_value()	
sample_sigmoid_value()	
sample_parabel_cubic_value()	

SampleGeometricInstances.**FILE_PATH**

SampleGeometricInstances.**FUNCTION_CONFIG**

SampleGeometricInstances.**FUNCTION_PARAMETER_NUMBERS**

SampleGeometricInstances.**SAMPLE_SIZE = 100**

SampleGeometricInstances.**save_geometric_instances**(*filename: str*, *config: Dict = FUNCTION_CONFIG*, *path: str = ''*)

First delete old isntance_set. Create new instances based on config.

Parameters

- **filename** (*str*) – name of instance set
- **config** (*Dict, optional*) – config that has info about which functions will get selected, by default FUNCTION_CONFIG

SampleGeometricInstances.**_create_csv_string**(*index*, *func_name: str*) → str

Create comma separated string with function name and parameter values. Set 0 for irrelevant params.

Parameters

- **index** – instance index

- **func_name** (*str*) – name of function

Returns
comma separated string

Return type
str

SampleGeometricInstances.**sample_sinus_value**()

SampleGeometricInstances.**sample_sigmoid_value**()

SampleGeometricInstances.**sample_parabel_cubic_value**()

PYTHON MODULE INDEX

d

dacbench, 69
dacbench.abstract_agent, 138
dacbench.abstract_benchmark, 139
dacbench.abstract_env, 141
dacbench.agents, 69
dacbench.agents.dynamic_random_agent, 69
dacbench.agents.generic_agent, 70
dacbench.agents.simple_agents, 71
dacbench.argument_parsing, 144
dacbench.benchmarks, 75
dacbench.benchmarks.cma_benchmark, 75
dacbench.benchmarks.fast_downward_benchmark, 76
dacbench.benchmarks.geometric_benchmark, 77
dacbench.benchmarks.luby_benchmark, 78
dacbench.benchmarks.modcma_benchmark, 79
dacbench.benchmarks.modea_benchmark, 80
dacbench.benchmarks.sgd_benchmark, 81
dacbench.benchmarks.sigmoid_benchmark, 82
dacbench.benchmarks.theory_benchmark, 83
dacbench.benchmarks.toysgd_benchmark, 83
dacbench.container, 87
dacbench.container.container_utils, 87
dacbench.container.remote_env, 89
dacbench.container.remote_runner, 90
dacbench.envs, 92
dacbench.envs.cma_es, 94
dacbench.envs.cma_step_size, 96
dacbench.envs.fast_downward, 97
dacbench.envs.geometric, 99
dacbench.envs.luby, 103
dacbench.envs.modcma, 104
dacbench.envs.modea, 105
dacbench.envs.policies, 92
dacbench.envs.policies.csa_cma, 92
dacbench.envs.policies.optimal_fd, 92
dacbench.envs.policies.optimal_luby, 92
dacbench.envs.policies.optimal_sigmoid, 93
dacbench.envs.policies.sgd_ca, 93
dacbench.envs.sgd, 106
dacbench.envs.sigmoid, 109
dacbench.envs.theory, 110
dacbench.envs.toysgd, 113
dacbench.logger, 144
dacbench.plotting, 150
dacbench.run_baselines, 155
dacbench.runner, 156
dacbench.wrappers, 122
dacbench.wrappers.action_tracking_wrapper, 122
dacbench.wrappers.episode_time_tracker, 123
dacbench.wrappers.instance_sampling_wrapper, 124
dacbench.wrappers.observation_wrapper, 125
dacbench.wrappers.performance_tracking_wrapper, 126
dacbench.wrappers.policy_progress_wrapper, 128
dacbench.wrappers.reward_noise_wrapper, 129
dacbench.wrappers.state_tracking_wrapper, 130

r

random_states, 161
reward_functions, 161

s

SampleGeometricInstances, 162

INDEX

Symbols

__buffer_to_file() (*dacbench.logger.ModuleLogger method*), 148
__call__() (*dacbench.argument_parsing.PathType method*), 144
__call__() (*dacbench.container.remote_runner.RemoteRunnerServerFactory method*), 91
__call__() (*dacbench.envs.sgd.Reward method*), 106
__connect_to_server() (*dacbench.container.remote_runner.RemoteRunner method*), 91
__decorate_config_with_functions() (*dacbench.AbstractBenchmark static method*), 159
__decorate_config_with_functions() (*dacbench.abstract_benchmark.AbstractBenchmark static method*), 140
__default_loss_function() (*in module dacbench.benchmarks.sgd_benchmark*), 81
__del__() (*dacbench.container.remote_runner.RemoteRunner method*), 91
__del__() (*dacbench.logger.Logger method*), 149
__del__() (*dacbench.logger.ModuleLogger method*), 147
__delattr__() (*dacbench.abstract_benchmark.objdict method*), 141
__end_step() (*dacbench.logger.ModuleLogger method*), 147
__eq__() (*dacbench.AbstractBenchmark method*), 160
__eq__() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 141
__eq__() (*dacbench.abstract_benchmark.objdict method*), 141
__getattr__() (*dacbench.abstract_benchmark.objdict method*), 141
__getattribute__() (*dacbench.wrappers.ActionFrequencyWrapper method*), 131
__getattribute__() (*dacbench.wrappers.EpisodeTimeWrapper method*), 132
__getattribute__() (*dacbench.wrappers.InstanceSamplingWrapper method*), 133
__getattribute__() (*dacbench.wrappers.ObservationWrapper method*), 137
__getattribute__() (*dacbench.wrappers.PerformanceTrackingWrapper method*), 137
__getattribute__() (*dacbench.wrappers.PolicyProgressWrapper method*), 134
__getattribute__() (*dacbench.wrappers.RewardNoiseWrapper method*), 135
__getattribute__() (*dacbench.wrappers.StateTrackingWrapper method*), 135
__getattribute__() (*dacbench.wrappers.action_tracking_wrapper.Actio method*), 122
__getattribute__() (*dacbench.wrappers.episode_time_tracker.Episode method*), 123
__getattribute__() (*dacbench.wrappers.instance_sampling_wrapper.In method*), 125
__getattribute__() (*dacbench.wrappers.observation_wrapper.Observa method*), 126
__getattribute__() (*dacbench.wrappers.performance_tracking_wrappe method*), 127
__getattribute__() (*dacbench.wrappers.policy_progress_wrapper.Polic method*), 128
__getattribute__() (*dacbench.wrappers.reward_noise_wrapper.Reward method*), 129
__getattribute__() (*dacbench.wrappers.state_tracking_wrapper.StateT method*), 130
__import_from() (*dacbench.AbstractBenchmark static method*), 159
__import_from() (*dacbench.abstract_benchmark.AbstractBenchmark static method*), 140
__init_dict() (*dacbench.logger.ModuleLogger static method*), 147
__json_default() (*dacbench.logger.ModuleLogger static method*), 147
__log() (*dacbench.logger.ModuleLogger method*), 148
__ne__() (*dacbench.abstract_benchmark.objdict method*), 141
__reset_step() (*dacbench.logger.ModuleLogger method*), 147
__setattr__() (*dacbench.abstract_benchmark.objdict method*), 141
__setattr__() (*dacbench.wrappers.ActionFrequencyWrapper method*), 131

__setattr__() (*dacbench.wrappers.EpisodeTimeWrapper method*), 132
__setattr__() (*dacbench.wrappers.InstanceSamplingWrapper method*), 133
__setattr__() (*dacbench.wrappers.ObservationWrapper method*), 137
__setattr__() (*dacbench.wrappers.PerformanceTrackingWrapper method*), 136
__setattr__() (*dacbench.wrappers.PolicyProgressWrapper method*), 134
__setattr__() (*dacbench.wrappers.RewardNoiseWrapper method*), 134
__setattr__() (*dacbench.wrappers.StateTrackingWrapper method*), 135
__setattr__() (*dacbench.wrappers.action_tracking_wrapper.ActionFrequencyWrapper method*), 122
__setattr__() (*dacbench.wrappers.episode_time_tracker.EpisodeTimeWrapper method*), 123
__setattr__() (*dacbench.wrappers.instance_sampling_wrapper.InstanceSamplingWrapper method*), 124
__setattr__() (*dacbench.wrappers.observation_wrapper.ObservationWrapper method*), 125
__setattr__() (*dacbench.wrappers.performance_tracking_wrapper.PerformanceTrackingWrapper method*), 127
__setattr__() (*dacbench.wrappers.policy_progress_wrapper.PolicyProgressWrapper method*), 128
__setattr__() (*dacbench.wrappers.reward_noise_wrapper.RewardNoiseWrapper method*), 129
__setattr__() (*dacbench.wrappers.state_tracking_wrapper.StateTrackingWrapper method*), 130
__space_dict() (*dacbench.logger.ModuleLogger static method*), 148
__start_server() (*dacbench.container.remote_runner.RemoteRunner method*), 91
__stringify_functions() (*dacbench.AbstractBenchmark static method*), 159
__stringify_functions() (*dacbench.abstract_benchmark.AbstractBenchmark static method*), 140
_add_correlation() (*dacbench.envs.geometric.Functions method*), 102
_apply_correlation_update() (*dacbench.envs.geometric.Functions method*), 102
_architecture_constructor() (*dacbench.envs.sgd.SGDEnv method*), 107
_calculate_function_value() (*dacbench.envs.geometric.Functions method*), 102
_constant() (*dacbench.envs.geometric.Functions method*), 102
_create_csv_string() (*in module SampleGeometricInstances*), 162
_cubic() (*dacbench.envs.geometric.Functions method*), 102
_get_alignment() (*dacbench.envs.sgd.SGDEnv method*), 108
_get_full_training_loss() (*dacbench.envs.sgd.SGDEnv method*), 108
_get_gradients() (*dacbench.envs.sgd.SGDEnv method*), 108
_get_loss_features() (*dacbench.envs.sgd.SGDEnv method*), 108
_get_momentum() (*dacbench.envs.sgd.SGDEnv method*), 108
_get_predictive_change_features() (*dacbench.envs.sgd.SGDEnv method*), 108
_get_validation_loss() (*dacbench.envs.sgd.SGDEnv method*), 108
_get_validation_loss_() (*dacbench.envs.sgd.SGDEnv method*), 108
_init_logging_adapter() (*dacbench.logger.AbstractLogger static method*), 146
_linear() (*dacbench.envs.geometric.Functions method*), 102
_logarithmic() (*dacbench.envs.geometric.Functions method*), 102
_norm() (*in module dacbench.envs.cma_es*), 95
_parabel() (*dacbench.envs.geometric.Functions method*), 102
_pre_reward() (*dacbench.envs.GeometricEnv method*), 122
_pre_reward() (*dacbench.envs.geometric.GeometricEnv method*), 101
_process_valid_types() (*dacbench.logger.AbstractLogger static method*), 146
_process_data() (*dacbench.envs.FastDownwardEnv method*), 118
_process_data() (*dacbench.envs.fast_downward.FastDownwardEnv method*), 98
_save_div() (*dacbench.envs.FastDownwardEnv static method*), 117
_save_div() (*dacbench.envs.fast_downward.FastDownwardEnv static method*), 97
_sig() (*dacbench.envs.SigmoidEnv method*), 116
_sig() (*dacbench.envs.sigmoid.SigmoidEnv method*), 109
_sigmoid() (*dacbench.envs.geometric.Functions method*), 102
_sinus() (*dacbench.envs.geometric.Functions method*), 102
_train_batch_() (*dacbench.envs.sgd.SGDEnv method*), 108

A

ABSDIFF (*dacbench.envs.fast_downward.StateType attribute*), 97
AbstractBenchmark (*class in dacbench*), 158
AbstractBenchmark (*class in dacbench.abstract_benchmark*), 7, 139
AbstractDACBenchAgent (*class in dacbench.abstract_agent*), 138
AbstractEnv (*class in dacbench*), 157
AbstractEnv (*class in dacbench.abstract_env*), 9, 141
AbstractLogger (*class in dacbench.logger*), 53, 146
act() (*dacbench.abstract_agent.AbstractDACBenchAgent method*), 138
act() (*dacbench.agents.dynamic_random_agent.DynamicRandomAgent method*), 69
act() (*dacbench.agents.DynamicRandomAgent method*), 74
act() (*dacbench.agents.generic_agent.GenericAgent method*), 70
act() (*dacbench.agents.GenericAgent method*), 72
act() (*dacbench.agents.RandomAgent method*), 73
act() (*dacbench.agents.simple_agents.RandomAgent method*), 71
act() (*dacbench.agents.simple_agents.StaticAgent method*), 72
act() (*dacbench.agents.StaticAgent method*), 74
act() (*dacbench.envs.policies.sgd_ca.CosineAnnealingAgent method*), 93
action_space (*dacbench.container.remote_env.RemoteEnvironmentClient property*), 90
action_space (*dacbench.container.remote_env.RemoteEnvironmentServer property*), 90
ACTION_VALUES (*in module dacbench.benchmarks.geometric_benchmark*), 77
ACTION_VALUES (*in module dacbench.benchmarks.sigmoid_benchmark*), 82
ActionFrequencyWrapper (*class in dacbench.wrappers*), 47, 131
ActionFrequencyWrapper (*class in dacbench.wrappers.action_tracking_wrapper*), 122
ACTIVE (*in module dacbench.benchmarks.modcma_benchmark*), 79
add_agent() (*dacbench.logger.Logger method*), 54, 149
add_benchmark() (*dacbench.logger.Logger method*), 54, 149
add_module() (*dacbench.logger.Logger method*), 54, 149
add_multi_level_ticks() (*in module dacbench.plotting*), 59, 151
add_noise() (*dacbench.wrappers.reward_noise_wrapper.RewardNoiseWrapper method*), 129
add_noise() (*dacbench.wrappers.RewardNoiseWrapper method*), 50, 135
additional_info (*dacbench.logger.AbstractLogger property*), 146
ALL_POLICIES (*in module dacbench.envs.policies*), 94
almost_easy_sigmoid() (*in module reward_functions*), 161
argstring (*dacbench.envs.fast_downward.FastDownwardEnv property*), 97
argstring (*dacbench.envs.FastDownwardEnv property*), 117

B

BASE_SAMPLER (*in module dacbench.benchmarks.modcma_benchmark*), 79
BinaryProblem (*class in dacbench.envs.theory*), 28, 110
BOUND_CORRECTION (*in module dacbench.benchmarks.modcma_benchmark*), 80
build_objective_function() (*dacbench.envs.toysgd.ToySGDEnv method*), 114
build_objective_function() (*dacbench.envs.ToySGDEnv method*), 119

C

calculate_derivative() (*dacbench.envs.geometric.Functions method*), 101
calculate_norm_values() (*dacbench.envs.geometric.Functions method*), 102
class_to_str() (*dacbench.abstract_benchmark.AbstractBenchmark class method*), 140
class_to_str() (*dacbench.AbstractBenchmark class method*), 159
close() (*dacbench.container.remote_env.RemoteEnvironmentClient method*), 90
close() (*dacbench.container.remote_env.RemoteEnvironmentServer method*), 90
close() (*dacbench.container.remote_runner.RemoteRunner method*), 91
close() (*dacbench.envs.cma_es.CMAESEnv method*), 32, 95
close() (*dacbench.envs.cma_step_size.CMAStepSizeEnv method*), 96
close() (*dacbench.envs.fast_downward.FastDownwardEnv method*), 24, 99
close() (*dacbench.envs.FastDownwardEnv method*), 119
close() (*dacbench.envs.geometric.GeometricEnv method*), 20, 100
close() (*dacbench.envs.GeometricEnv method*), 121

close() (*dacbench.envs.luby.LubyEnv method*), 16, 103
close() (*dacbench.envs.LubyEnv method*), 116
close() (*dacbench.envs.modcma.ModCMAEnv method*), 37, 104
close() (*dacbench.envs.modea.ModeaEnv method*), 35, 105
close() (*dacbench.envs.sgd.SGDEnv method*), 40, 108
close() (*dacbench.envs.sigmoid.SigmoidEnv method*), 12, 109
close() (*dacbench.envs.SigmoidEnv method*), 117
close() (*dacbench.envs.theory.RLSEnv method*), 29, 113
close() (*dacbench.envs.toysgd.ToySGDEnv method*), 115
close() (*dacbench.envs.ToySGDEnv method*), 120
close() (*dacbench.logger.AbstractLogger method*), 53, 146
close() (*dacbench.logger.Logger method*), 54, 148
close() (*dacbench.logger.ModuleLogger method*), 55, 147
CMAES_DEFAULTS (*in module dacbench.benchmarks.cma_benchmark*), 75
CMAESBenchmark (*class in dacbench.benchmarks.cma_benchmark*), 31, 75
CMAESEnv (*class in dacbench.envs.cma_es*), 32, 95
CMAStepSizeEnv (*class in dacbench.envs.cma_step_size*), 96
combine() (*dacbench.envs.theory.BinaryProblem method*), 28, 111
config (*in module dacbench.envs.sigmoid*), 110
ContinuousSigmoidEnv (*class in dacbench.envs.sigmoid*), 12, 110
ContinuousStateSigmoidEnv (*class in dacbench.envs.sigmoid*), 12, 110
copy() (*dacbench.abstract_benchmark.objdict method*), 9, 141
CosineAnnealingAgent (*class in dacbench.envs.policies.sgd_ca*), 93
crash (*dacbench.envs.sgd.SGDEnv property*), 107
create() (*dacbench.container.remote_runner.RemoteRunnerServerFactory method*), 91
create_correlation_table() (*dacbench.benchmarks.geometric_benchmark.GeometricBenchmark method*), 19, 78
create_correlation_table() (*dacbench.benchmarks.GeometricBenchmark method*), 86
create_observation_space_from_description() (*dacbench.benchmarks.theory_benchmark.TheoryBenchmark method*), 27, 83
create_polynomial_instance_set() (*in module dacbench.envs.toysgd*), 114
crossover() (*dacbench.envs.theory.BinaryProblem method*), 28, 111
csa() (*in module dacbench.envs.policies*), 94
csa() (*in module dacbench.envs.policies.csa_cma*), 92
current_palette (*in module dacbench.runner*), 156
current_palette (*in module dacbench.wrappers.action_tracking_wrapper*), 122
current_palette (*in module dacbench.wrappers.episode_time_tracker*), 123
current_palette (*in module dacbench.wrappers.performance_tracking_wrapper*), 126
current_palette (*in module dacbench.wrappers.state_tracking_wrapper*), 130
current_validation_loss (*dacbench.envs.sgd.SGDEnv property*), 108

D

dacbench
    module, 69
dacbench.abstract_agent
    module, 138
dacbench.abstract_benchmark
    module, 7, 139
dacbench.abstract_env
    module, 9, 141
dacbench.agents
    module, 69
dacbench.agents.dynamic_random_agent
    module, 69
dacbench.agents.generic_agent
    module, 70
dacbench.agents.simple_agents
    module, 71
dacbench.argument_parsing
    module, 144
dacbench.benchmarks
    module, 75
dacbench.benchmarks.cma_benchmark
    module, 31, 75
dacbench.benchmarks.fast_downward_benchmark
    module, 23, 76
dacbench.benchmarks.geometric_benchmark
    module, 19, 77
dacbench.benchmarks.luby_benchmark
    module, 15, 78
dacbench.benchmarks.modcma_benchmark
    module, 37, 79
dacbench.benchmarks.modea_benchmark
    module, 35, 80
dacbench.benchmarks.sgd_benchmark

module, 39, 81
dacbench.benchmarks.sigmoid_benchmark
module, 11, 82
dacbench.benchmarks.theory_benchmark
module, 27, 83
dacbench.benchmarks.toysgd_benchmark
module, 83
dacbench.container
module, 87
dacbench.container.container_utils
module, 87
dacbench.container.remote_env
module, 89
dacbench.container.remote_runner
module, 90
dacbench.envs
module, 92
dacbench.envs.cma_es
module, 32, 94
dacbench.envs.cma_step_size
module, 96
dacbench.envs.fast_downward
module, 24, 97
dacbench.envs.geometric
module, 20, 99
dacbench.envs.luby
module, 16, 103
dacbench.envs.modcma
module, 37, 104
dacbench.envs.modea
module, 35, 105
dacbench.envs.policies
module, 92
dacbench.envs.policies.csa_cma
module, 92
dacbench.envs.policies.optimal_fd
module, 92
dacbench.envs.policies.optimal_luby
module, 92
dacbench.envs.policies.optimal_sigmoid
module, 93
dacbench.envs.policies.sgd_ca
module, 93
dacbench.envs.sgd
module, 40, 106
dacbench.envs.sigmoid
module, 12, 109
dacbench.envs.theory
module, 28, 110
dacbench.envs.toysgd
module, 113
dacbench.logger
module, 53, 144
dacbench.plotting
module, 59, 150
dacbench.run_baselines
module, 155
dacbench.runner
module, 156
dacbench.wrappers
module, 47, 122
dacbench.wrappers.action_tracking_wrapper
module, 122
dacbench.wrappers.episode_time_tracker
module, 123
dacbench.wrappers.instance_sampling_wrapper
module, 124
dacbench.wrappers.observation_wrapper
module, 125
dacbench.wrappers.performance_tracking_wrapper
module, 126
dacbench.wrappers.policy_progress_wrapper
module, 128
dacbench.wrappers.reward_noise_wrapper
module, 129
dacbench.wrappers.state_tracking_wrapper
module, 130
decode_space() (*dacbench.container.container_utils.Decoder method*), 88
Decoder (*class in dacbench.container.container_utils*), 88
DEFAULT_CFG_SPACE (*in module dacbench.benchmarks.cma_benchmark*), 75
DEFAULT_CFG_SPACE (*in module dacbench.benchmarks.fast_downward_benchmark*), 76
DEFAULT_CFG_SPACE (*in module dacbench.benchmarks.geometric_benchmark*), 77
DEFAULT_CFG_SPACE (*in module dacbench.benchmarks.luby_benchmark*), 78
DEFAULT_CFG_SPACE (*in module dacbench.benchmarks.modcma_benchmark*), 79
DEFAULT_CFG_SPACE (*in module dacbench.benchmarks.sgd_benchmark*), 81
DEFAULT_CFG_SPACE (*in module dacbench.benchmarks.sigmoid_benchmark*), 82
DEFAULT_CFG_SPACE (*in module dacbench.benchmarks.toysgd_benchmark*), 83
DefaultJsonable (*in module dacbench.container.remote_env*), 89
DEFAULTS (*in module dacbench.benchmarks.toysgd_benchmark*), 84

dejson_wrappers() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 139
dejson_wrappers() (*dacbench.AbstractBenchmark method*), 159
deserialize_random_state() (*in module dacbench.container.container_utils*), 88
determineRegime() (*dacbench.envs.modea.ModeaEnv method*), 105
dictify_json() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 140
dictify_json() (*dacbench.AbstractBenchmark method*), 160
DIFF (*dacbench.envs.fast_downward.StateType attribute*), 97
DiffTraining (*dacbench.envs.sgd.Reward attribute*), 106
DiffValidation (*dacbench.envs.sgd.Reward attribute*), 106
DISCRETE_ACTIONS (*in module dacbench.run_baselines*), 155
DynamicRandomAgent (*class in dacbench.agents*), 74
DynamicRandomAgent (*class in dacbench.agents.dynamic_random_agent*), 69

E

easy_sigmoid() (*in module reward_functions*), 161
ELITIST (*in module dacbench.benchmarks.modcma_benchmark*), 79
encode() (*dacbench.container.container_utils.Encoder method*), 88
encode_space() (*dacbench.container.container_utils.Encoder static method*), 88
Encoder (*class in dacbench.container.container_utils*), 88
end_episode() (*dacbench.abstract_agent.AbstractDACBenchAgent method*), 139
end_episode() (*dacbench.agents.dynamic_random_agent.DynamicRandomAgent method*), 70
end_episode() (*dacbench.agents.DynamicRandomAgent method*), 75
end_episode() (*dacbench.agents.generic_agent.GenericAgent method*), 70
end_episode() (*dacbench.agents.GenericAgent method*), 73
end_episode() (*dacbench.agents.RandomAgent method*), 73
end_episode() (*dacbench.agents.simple_agents.RandomAgent method*), 71
end_episode() (*dacbench.agents.simple_agents.StaticAgent method*), 72
end_episode() (*dacbench.agents.StaticAgent method*), 74
end_episode() (*dacbench.envs.policies.sgd_ca.CosineAnnealingAgent method*), 93
ensureFullLengthRepresentation() (*dacbench.envs.modea.ModeaEnv method*), 35, 105
EpisodeTimeWrapper (*class in dacbench.wrappers*), 47, 132
EpisodeTimeWrapper (*class in dacbench.wrappers.episode_time_tracker*), 123
eval() (*dacbench.envs.theory.BinaryProblem method*), 111
eval() (*dacbench.envs.theory.LeadingOne method*), 112
excepthook (*in module dacbench.container.remote_runner*), 90

F

FACTORY_NAME (*dacbench.container.remote_runner.RemoteRunner attribute*), 91
FastDownwardBenchmark (*class in dacbench.benchmarks*), 86
FastDownwardBenchmark (*class in dacbench.benchmarks.fast_downward_benchmark*), 23, 76
FastDownwardEnv (*class in dacbench.envs*), 117
FastDownwardEnv (*class in dacbench.envs.fast_downward*), 24, 97
FD_DEFAULTS (*in module dacbench.benchmarks.fast_downward_benchmark*), 76
FILE_PATH (*in module dacbench.benchmarks.geometric_benchmark*), 77
FILE_PATH (*in module SampleGeometricInstances*), 162
fit_dist() (*dacbench.wrappers.instance_sampling_wrapper.InstanceSam method*), 125
fit_dist() (*dacbench.wrappers.InstanceSamplingWrapper method*), 48, 134
flatten() (*dacbench.wrappers.observation_wrapper.ObservationWrappe method*), 126
flatten() (*dacbench.wrappers.ObservationWrapper method*), 138
flatten_log_entry() (*in module dacbench.logger*), 56, 145
flip() (*dacbench.envs.theory.BinaryProblem method*), 28, 111
from_json() (*dacbench.abstract_benchmark.AbstractBenchmark class method*), 139
from_json() (*dacbench.AbstractBenchmark class method*), 159
FullTraining (*dacbench.envs.sgd.Reward attribute*), 106
FUNCTION_CONFIG (*in module SampleGeometricInstances*), 162

FUNCTION_PARAMETER_NUMBERS (*in module SampleGeometricInstances*), 162
Functions (*class in dacbench.envs.geometric*), 101

G

generate_global_step() (*in module dacbench.plotting*), 59, 150
generate_instance_file() (*dacbench.envs.sgd.SGDEnv method*), 108
GenericAgent (*class in dacbench.agents*), 72
GenericAgent (*class in dacbench.agents.generic_agent*), 70
geo_bench (*in module dacbench.benchmarks.geometric_benchmark*), 78
GEOMETRIC_DEFAULTS (*in module dacbench.benchmarks.geometric_benchmark*), 77
GeometricBenchmark (*class in dacbench.benchmarks*), 86
GeometricBenchmark (*class in dacbench.benchmarks.geometric_benchmark*), 19, 77
GeometricEnv (*class in dacbench.envs*), 120
GeometricEnv (*class in dacbench.envs.geometric*), 20, 99
get_actions() (*dacbench.wrappers.action_tracking_wrapper.ActionFrequencyWrapper method*), 123
get_actions() (*dacbench.wrappers.ActionFrequencyWrapper method*), 47, 132
get_adam_direction() (*dacbench.envs.sgd.SGDEnv method*), 108
get_benchmark() (*dacbench.benchmarks.cma_benchmark.CMAESBenchmark method*), 31, 75
get_benchmark() (*dacbench.benchmarks.fast_downward_benchmark.FastDownwardBenchmark method*), 23, 76
get_benchmark() (*dacbench.benchmarks.FastDownwardBenchmark method*), 87
get_benchmark() (*dacbench.benchmarks.geometric_benchmark.GeometricBenchmark method*), 19, 77
get_benchmark() (*dacbench.benchmarks.GeometricBenchmark method*), 86
get_benchmark() (*dacbench.benchmarks.luby_benchmark.LubyBenchmark method*), 15, 79
get_benchmark() (*dacbench.benchmarks.LubyBenchmark method*), 85
get_benchmark() (*dacbench.benchmarks.modcma_benchmark.ModCMABenchmark method*), 80
get_benchmark() (*dacbench.benchmarks.sgd_benchmark.SGDBenchmark method*), 39, 81
get_benchmark() (*dacbench.benchmarks.sigmoid_benchmark.SigmoidBenchmark method*), 11, 82
get_benchmark() (*dacbench.benchmarks.SigmoidBenchmark method*), 85
get_config() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 7, 139
get_config() (*dacbench.AbstractBenchmark method*), 159
get_coordinates() (*dacbench.envs.geometric.Functions method*), 101
get_coordinates_at_time_step() (*dacbench.envs.geometric.Functions method*), 101
get_default_reward() (*dacbench.envs.cma_es.CMAESEnv method*), 32, 95
get_default_reward() (*dacbench.envs.cma_step_size.CMAStepSizeEnv method*), 97
get_default_reward() (*dacbench.envs.geometric.GeometricEnv method*), 20, 100
get_default_reward() (*dacbench.envs.GeometricEnv method*), 121
get_default_reward() (*dacbench.envs.luby.LubyEnv method*), 103
get_default_reward() (*dacbench.envs.LubyEnv method*), 116
get_default_reward() (*dacbench.envs.modcma.ModCMAEnv method*), 104
get_default_reward() (*dacbench.envs.modea.ModeaEnv method*), 105
get_default_reward() (*dacbench.envs.sigmoid.SigmoidEnv method*), 109
get_default_reward() (*dacbench.envs.SigmoidEnv method*), 117
get_default_state() (*dacbench.envs.cma_es.CMAESEnv method*), 32, 96
get_default_state() (*dacbench.envs.cma_step_size.CMAStepSizeEnv method*), 97
get_default_state() (*dacbench.envs.geometric.GeometricEnv method*), 20, 100
get_default_state() (*dacbench.envs.GeometricEnv method*), 121
get_default_state() (*dacbench.envs.luby.LubyEnv method*), 103
get_default_state() (*dacbench.envs.LubyEnv method*), 116
get_default_state() (*dacbench.envs.modcma.ModCMAEnv method*), 104
get_default_state()

(*dacbench.envs.modea.ModeaEnv method*), 105
get_default_state() (*dacbench.envs.sgd.SGDEnv method*), 40, 108
get_default_state() (*dacbench.envs.sigmoid.SigmoidEnv method*), 109
get_default_state() (*dacbench.envs.SigmoidEnv method*), 117
get_diff_training_reward() (*dacbench.envs.sgd.SGDEnv method*), 107
get_diff_validation_reward() (*dacbench.envs.sgd.SGDEnv method*), 107
get_environment() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 8, 140
get_environment() (*dacbench.AbstractBenchmark method*), 160
get_environment() (*dacbench.benchmarks.cma_benchmark.CMAESBenchmark method*), 31, 75
get_environment() (*dacbench.benchmarks.fast_downward_benchmark.FastDownwardBenchmark method*), 23, 76
get_environment() (*dacbench.benchmarks.FastDownwardBenchmark method*), 87
get_environment() (*dacbench.benchmarks.geometric_benchmark.GeometricBenchmark method*), 19, 77
get_environment() (*dacbench.benchmarks.GeometricBenchmark method*), 86
get_environment() (*dacbench.benchmarks.luby_benchmark.LubyBenchmark method*), 15, 78
get_environment() (*dacbench.benchmarks.LubyBenchmark method*), 84
get_environment() (*dacbench.benchmarks.modcma_benchmark.ModCMABenchmark method*), 37, 80
get_environment() (*dacbench.benchmarks.modea_benchmark.ModeaBenchmark method*), 35, 80
get_environment() (*dacbench.benchmarks.sgd_benchmark.SGDBenchmark method*), 39, 81
get_environment() (*dacbench.benchmarks.sigmoid_benchmark.SigmoidBenchmark method*), 11, 82
get_environment() (*dacbench.benchmarks.SigmoidBenchmark method*), 85
get_environment() (*dacbench.benchmarks.theory_benchmark.TheoryBenchmark method*), 27, 83
get_environment() (*dacbench.benchmarks.toysgd_benchmark.ToySGDBenchmark method*), 84
get_environment() (*dacbench.benchmarks.ToySGDBenchmark method*), 86
get_environment() (*dacbench.container.remote_runner.RemoteRunner method*), 91
get_environment() (*dacbench.container.remote_runner.RemoteRunnerServer method*), 91
get_fitness_after_crossover() (*dacbench.envs.theory.BinaryProblem method*), 28, 111
get_fitness_after_crossover() (*dacbench.envs.theory.LeadingOne method*), 29, 112
get_fitness_after_flipping() (*dacbench.envs.theory.BinaryProblem method*), 28, 111
get_fitness_after_flipping() (*dacbench.envs.theory.LeadingOne method*), 29, 112
get_full_training_loss() (*dacbench.envs.sgd.SGDEnv method*), 107
get_full_training_reward() (*dacbench.envs.sgd.SGDEnv method*), 107
get_initial_position() (*dacbench.envs.toysgd.ToySGDEnv method*), 114
get_initial_position() (*dacbench.envs.ToySGDEnv method*), 119
get_inst_id() (*dacbench.abstract_env.AbstractEnv method*), 9, 142
get_inst_id() (*dacbench.AbstractEnv method*), 157
get_instance() (*dacbench.abstract_env.AbstractEnv method*), 9, 143
get_instance() (*dacbench.AbstractEnv method*), 158
get_instance_set() (*dacbench.abstract_env.AbstractEnv method*), 9, 142
get_instance_set() (*dacbench.AbstractEnv method*), 158
get_log_diff_training_reward() (*dacbench.envs.sgd.SGDEnv method*), 107
get_log_diff_validation_reward() (*dacbench.envs.sgd.SGDEnv method*), 107
get_log_training_reward() (*dacbench.envs.sgd.SGDEnv method*), 107
get_log_validation_reward() (*dacbench.envs.sgd.SGDEnv method*), 107
get_logfile() (*dacbench.logger.ModuleLogger method*), 75, 147
get_momentum_direction() (*dacbench.envs.sgd.SGDEnv method*), 108
get_obs_domain_from_name() (*dacbench.envs.theory.RLSEnv method*), 29, 112
get_optimal() (*dacbench.envs.theory.BinaryProblem method*), 111
get_optimal() (*dacbench.envs.theory.LeadingOne method*), 112
get_optimal_policy() (*dacbench.envs.geometric.GeometricEnv method*), 20, 99
get_optimal_policy() (*dacbench.envs.GeometricEnv method*), 120
get_optimum() (*in module dacbench.envs.policies.optimal_fd*), 92

get_optimum() (*in module dacbench.envs.policies.optimal_luby*), 92
get_optimum() (*in module dacbench.envs.policies.optimal_sigmoid*), 93
get_performance() (*dacbench.wrappers.performance_tracking_wrapper.PerformanceTrackingWrapper method*), 127
get_performance() (*dacbench.wrappers.PerformanceTrackingWrapper method*), 49, 137
get_reward() (*dacbench.envs.sgd.SGDEnv method*), 107
get_rmsprop_direction() (*dacbench.envs.sgd.SGDEnv method*), 108
get_state() (*dacbench.envs.theory.RLSEnv method*), 112
get_states() (*dacbench.wrappers.state_tracking_wrapper.StateTrackingWrapper method*), 131
get_states() (*dacbench.wrappers.StateTrackingWrapper method*), 50, 136
get_times() (*dacbench.wrappers.episode_time_tracker.EpisodeTimeWrapper method*), 124
get_times() (*dacbench.wrappers.EpisodeTimeWrapper method*), 48, 133
get_training_reward() (*dacbench.envs.sgd.SGDEnv method*), 107
get_validation_reward() (*dacbench.envs.sgd.SGDEnv method*), 107

H

HEURISTIC (*in module dacbench.benchmarks.fast_downward_benchmark*), 76
HEURISTICS (*in module dacbench.benchmarks.fast_downward_benchmark*), 76
hint() (*dacbench.container.container_utils.Encoder static method*), 88
HISTORY_LENGTH (*in module dacbench.benchmarks.cma_benchmark*), 75
HISTORY_LENGTH (*in module dacbench.benchmarks.luby_benchmark*), 78
HISTORY_LENGTH (*in module dacbench.envs.theory*), 112

I

id_generator() (*dacbench.container.remote_runner.RemoteRunner static method*), 91
INFO (*in module dacbench.benchmarks.cma_benchmark*), 75
INFO (*in module dacbench.benchmarks.fast_downward_benchmark*), 76
INFO (*in module dacbench.benchmarks.geometric_benchmark*), 77
INFO (*in module dacbench.benchmarks.luby_benchmark*), 78
INFO (*in module dacbench.benchmarks.modcma_benchmark*), 80
INFO (*in module dacbench.benchmarks.modea_benchmark*), 80
INFO (*in module dacbench.benchmarks.sgd_benchmark*), 81
INFO (*in module dacbench.benchmarks.sigmoid_benchmark*), 82
INFO (*in module dacbench.benchmarks.theory_benchmark*), 83
INFO (*in module dacbench.benchmarks.toysgd_benchmark*), 84
initialise_with_fixed_number_of_bits() (*dacbench.envs.theory.BinaryProblem method*), 110
INPUT_DIM (*in module dacbench.benchmarks.cma_benchmark*), 75
instance_mapper() (*in module dacbench.logger*), 146
InstanceSamplingWrapper (*class in dacbench.wrappers*), 48, 133
InstanceSamplingWrapper (*class in dacbench.wrappers.instance_sampling_wrapper*), 124
is_of_valid_type() (*dacbench.logger.AbstractLogger method*), 146
is_optimal() (*dacbench.envs.theory.BinaryProblem method*), 111
is_optimal() (*dacbench.envs.theory.LeadingOne method*), 112

J

json_decode() (*in module dacbench.container.remote_env*), 89
json_encode() (*in module dacbench.container.remote_env*), 89
Jsonable (*in module dacbench.container.remote_env*), 89
jsonify_dict_space() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 140
jsonify_dict_space() (*dacbench.AbstractBenchmark method*), 159
jsonify_wrappers() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 139
jsonify_wrappers() (*dacbench.AbstractBenchmark method*), 159

K

kill_connection() (*dacbench.envs.fast_downward.FastDownwardEnv method*), 24, 99

kill_connection() (*dacbench.envs.FastDownwardEnv method*), 118

L

LeadingOne (*class in dacbench.envs.theory*), 29, 112
list_to_space() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 140
list_to_space() (*dacbench.AbstractBenchmark method*), 159
list_to_tuple() (*in module dacbench.logger*), 57, 145
load_benchmark() (*dacbench.container.remote_runner.RemoteRunner method*), 91
load_config() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 140
load_config() (*dacbench.AbstractBenchmark method*), 160
load_logs() (*in module dacbench.logger*), 57, 144
LOCAL_RESTART (*in module dacbench.benchmarks.modcma_benchmark*), 80
log() (*dacbench.logger.AbstractLogger method*), 146
log() (*dacbench.logger.Logger method*), 149
log() (*dacbench.logger.ModuleLogger method*), 148
log2dataframe() (*in module dacbench.logger*), 57, 145
log_dict() (*dacbench.logger.AbstractLogger method*), 53, 147
log_dict() (*dacbench.logger.Logger method*), 55, 149
log_dict() (*dacbench.logger.ModuleLogger method*), 55, 148
LOG_LEVEL (*in module dacbench.container.remote_runner*), 90
log_level_str (*in module dacbench.container.remote_runner*), 90
log_space() (*dacbench.logger.AbstractLogger method*), 53, 147
log_space() (*dacbench.logger.Logger method*), 55, 149
log_space() (*dacbench.logger.ModuleLogger method*), 56, 148
LogDiffTraining (*dacbench.envs.sgd.Reward attribute*), 106
LogDiffValidation (*dacbench.envs.sgd.Reward attribute*), 106
Logger (*class in dacbench.logger*), 54, 148
logger (*in module dacbench.container.remote_runner*), 90
LogTrainingLoss (*dacbench.envs.sgd.Reward attribute*), 106
LogValidationLoss (*dacbench.envs.sgd.Reward attribute*), 106
LR (*in module dacbench.benchmarks.sgd_benchmark*), 81
LR (*in module dacbench.benchmarks.toysgd_benchmark*), 83
LUBY_DEFAULTS (*in module dacbench.benchmarks.luby_benchmark*), 78
luby_gen() (*in module dacbench.envs*), 116
luby_gen() (*in module dacbench.envs.luby*), 17, 104
luby_gen() (*in module dacbench.envs.policies.optimal_luby*), 92
LUBY_SEQUENCE (*in module dacbench.benchmarks.luby_benchmark*), 78
LubyBenchmark (*class in dacbench.benchmarks*), 84
LubyBenchmark (*class in dacbench.benchmarks.luby_benchmark*), 15, 78
LubyEnv (*class in dacbench.envs*), 116
LubyEnv (*class in dacbench.envs.luby*), 16, 103

M

main() (*in module dacbench.run_baselines*), 156
manhattan_distance_reward_geometric() (*in module reward_functions*), 162
MAX_INT (*in module dacbench.envs.theory*), 112
MAX_STEPS (*in module dacbench.benchmarks.luby_benchmark*), 78
MAX_TRIES (*in module dacbench.container.remote_runner*), 90
MIRRORED (*in module dacbench.benchmarks.modcma_benchmark*), 79
MODCMA_DEFAULTS (*in module dacbench.benchmarks.modcma_benchmark*), 80
ModCMABenchmark (*class in dacbench.benchmarks.modcma_benchmark*), 37, 80
ModCMAEnv (*class in dacbench.envs.modcma*), 37, 104
modea_actions (*in module dacbench.run_baselines*), 155
MODEA_DEFAULTS (*in module dacbench.benchmarks.modea_benchmark*), 80
ModeaBenchmark (*class in dacbench.benchmarks.modea_benchmark*), 35, 80
ModeaEnv (*class in dacbench.envs.modea*), 35, 105
module
    dacbench, 69
    dacbench.abstract_agent, 138
    dacbench.abstract_benchmark, 7, 139
    dacbench.abstract_env, 9, 141
    dacbench.agents, 69
    dacbench.agents.dynamic_random_agent, 69
    dacbench.agents.generic_agent, 70
    dacbench.agents.simple_agents, 71
    dacbench.argument_parsing, 144
    dacbench.benchmarks, 75

dacbench.benchmarks.cma_benchmark, 31, 75
dacbench.benchmarks.fast_downward_benchmark, 23, 76
dacbench.benchmarks.geometric_benchmark, 19, 77
dacbench.benchmarks.luby_benchmark, 15, 78
dacbench.benchmarks.modcma_benchmark, 37, 79
dacbench.benchmarks.modea_benchmark, 35, 80
dacbench.benchmarks.sgd_benchmark, 39, 81
dacbench.benchmarks.sigmoid_benchmark, 11, 82
dacbench.benchmarks.theory_benchmark, 27, 83
dacbench.benchmarks.toysgd_benchmark, 83
dacbench.container, 87
dacbench.container.container_utils, 87
dacbench.container.remote_env, 89
dacbench.container.remote_runner, 90
dacbench.envs, 92
dacbench.envs.cma_es, 32, 94
dacbench.envs.cma_step_size, 96
dacbench.envs.fast_downward, 24, 97
dacbench.envs.geometric, 20, 99
dacbench.envs.luby, 16, 103
dacbench.envs.modcma, 37, 104
dacbench.envs.modea, 35, 105
dacbench.envs.policies, 92
dacbench.envs.policies.csa_cma, 92
dacbench.envs.policies.optimal_fd, 92
dacbench.envs.policies.optimal_luby, 92
dacbench.envs.policies.optimal_sigmoid, 93
dacbench.envs.policies.sgd_ca, 93
dacbench.envs.sgd, 40, 106
dacbench.envs.sigmoid, 12, 109
dacbench.envs.theory, 28, 110
dacbench.envs.toysgd, 113
dacbench.logger, 53, 144
dacbench.plotting, 59, 150
dacbench.run_baselines, 155
dacbench.runner, 156
dacbench.wrappers, 47, 122
dacbench.wrappers.action_tracking_wrapper, 122
dacbench.wrappers.episode_time_tracker, 123
dacbench.wrappers.instance_sampling_wrapper, 124
dacbench.wrappers.observation_wrapper, 125
dacbench.wrappers.performance_tracking_wrapper, 126
dacbench.wrappers.policy_progress_wrapper, 128
dacbench.wrappers.reward_noise_wrapper, 129
dacbench.wrappers.state_tracking_wrapper, 130
random_states, 161
reward_functions, 161
SampleGeometricInstances, 162
ModuleLogger (*class in dacbench.logger*), 55, 147
MOMENTUM (*in module dacbench.benchmarks.toysgd_benchmark*), 83
multiply_reward_geometric() (*in module reward_functions*), 162
mutate() (*dacbench.envs.theory.BinaryProblem method*), 28, 111
mutate_rls() (*dacbench.envs.theory.BinaryProblem method*), 29, 111

N

next_episode() (*dacbench.logger.AbstractLogger method*), 54, 146
next_episode() (*dacbench.logger.Logger method*), 55, 149
next_episode() (*dacbench.logger.ModuleLogger method*), 56, 148
next_step() (*dacbench.logger.AbstractLogger method*), 54, 146
next_step() (*dacbench.logger.Logger method*), 55, 149
next_step() (*dacbench.logger.ModuleLogger method*), 56, 147
NON_OPTIMAL_POLICIES (*in module dacbench.envs.policies*), 94
NORMABSDIFF (*dacbench.envs.fast_downward.StateType attribute*), 97
NORMAL (*dacbench.envs.fast_downward.StateType attribute*), 97
NORMDIFF (*dacbench.envs.fast_downward.StateType attribute*), 97
NumpyTypes (*in module dacbench.container.remote_env*), 89

O

objdict (*class in dacbench.abstract_benchmark*), 8, 141
object_hook() (*dacbench.container.container_utils.Decoder method*), 88
ObservationWrapper (*class in dacbench.wrappers*), 48, 137
ObservationWrapper (*class in dacbench.wrappers.observation_wrapper*), 125
optimal_fd() (*in module dacbench.envs.policies*), 94
optimal_luby() (*in module dacbench.envs.policies*), 94

OPTIMAL_POLICIES (*in module dacbench.envs.policies*), 94
optimal_sigmoid() (*in module dacbench.envs.policies*), 94
ORTHOGONAL (*in module dacbench.benchmarks.modcma_benchmark*), 79

P

parser (*in module dacbench.container.remote_runner*), 91
PathType (*class in dacbench.argument_parsing*), 144
PerformanceTrackingWrapper (*class in dacbench.wrappers*), 49, 136
PerformanceTrackingWrapper (*class in dacbench.wrappers.performance_tracking_wrapper*), 126
plot() (*in module dacbench.plotting*), 59, 151
plot_action() (*in module dacbench.plotting*), 60, 153
plot_episode_time() (*in module dacbench.plotting*), 60, 153
plot_performance() (*in module dacbench.plotting*), 61, 151
plot_performance_per_instance() (*in module dacbench.plotting*), 61, 152
plot_space() (*in module dacbench.plotting*), 62, 154
plot_state() (*in module dacbench.plotting*), 62, 154
plot_step_time() (*in module dacbench.plotting*), 63, 152
PolicyProgressWrapper (*class in dacbench.wrappers*), 49, 134
PolicyProgressWrapper (*class in dacbench.wrappers.policy_progress_wrapper*), 128
port (*dacbench.envs.fast_downward.FastDownwardEnv property*), 97
port (*dacbench.envs.FastDownwardEnv property*), 117
process_configspace() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 8, 139
process_configspace() (*dacbench.AbstractBenchmark method*), 159

Q

quadratic_euclidean_distance_reward_geometric() (*in module reward_functions*), 162
quadratic_manhattan_distance_reward_geometric() (*in module reward_functions*), 162

R

random_luby_state() (*in module random_states*), 161
random_reward() (*in module reward_functions*), 162
random_sigmoid_state() (*in module random_states*), 161
random_states
    module, 161
RandomAgent (*class in dacbench.agents*), 73
RandomAgent (*class in dacbench.agents.simple_agents*), 71
RAW (*dacbench.envs.fast_downward.StateType attribute*), 97
read_config_file() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 8, 140
read_config_file() (*dacbench.AbstractBenchmark method*), 160
read_instance_set() (*dacbench.benchmarks.cma_benchmark.CMAESBenchmark method*), 31, 75
read_instance_set() (*dacbench.benchmarks.fast_downward_benchmark.FastDownwar method*), 24, 76
read_instance_set() (*dacbench.benchmarks.FastDownwardBenchmark method*), 87
read_instance_set() (*dacbench.benchmarks.geometric_benchmark.GeometricBenchma method*), 20, 77
read_instance_set() (*dacbench.benchmarks.GeometricBenchmark method*), 86
read_instance_set() (*dacbench.benchmarks.luby_benchmark.LubyBenchmark method*), 16, 79
read_instance_set() (*dacbench.benchmarks.LubyBenchmark method*), 85
read_instance_set() (*dacbench.benchmarks.modcma_benchmark.ModCMABenchmark method*), 80
read_instance_set() (*dacbench.benchmarks.modea_benchmark.ModeaBenchmark method*), 35, 80
read_instance_set() (*dacbench.benchmarks.sgd_benchmark.SGDBenchmark method*), 39, 81
read_instance_set() (*dacbench.benchmarks.sigmoid_benchmark.SigmoidBenchmark method*), 12, 82
read_instance_set() (*dacbench.benchmarks.SigmoidBenchmark method*), 85
read_instance_set() (*dacbench.benchmarks.theory_benchmark.TheoryBenchmark method*), 27, 83
read_instance_set() (*dacbench.benchmarks.toysgd_benchmark.ToySGDBenchmark*

method), 84
read_instance_set() (*dacbench.benchmarks.ToySGDBenchmark method*), 86
recv_msg() (*dacbench.envs.fast_downward.FastDownwardEnv method*), 24, 98
recv_msg() (*dacbench.envs.FastDownwardEnv method*), 118
recvall() (*dacbench.envs.fast_downward.FastDownwardEnv method*), 24, 98
recvall() (*dacbench.envs.FastDownwardEnv method*), 118
register_wrapper() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 141
register_wrapper() (*dacbench.AbstractBenchmark method*), 160
RemoteEnvironmentClient (*class in dacbench.container.remote_env*), 90
RemoteEnvironmentServer (*class in dacbench.container.remote_env*), 89
RemoteRunner (*class in dacbench.container.remote_runner*), 91
RemoteRunnerServer (*class in dacbench.container.remote_runner*), 90
RemoteRunnerServerFactory (*class in dacbench.container.remote_runner*), 91
render() (*dacbench.container.remote_env.RemoteEnvironmentServer method*), 89
render() (*dacbench.envs.cma_es.CMAESEnv method*), 32, 95
render() (*dacbench.envs.fast_downward.FastDownwardEnv method*), 24, 99
render() (*dacbench.envs.FastDownwardEnv method*), 119
render() (*dacbench.envs.geometric.GeometricEnv method*), 21, 100
render() (*dacbench.envs.GeometricEnv method*), 121
render() (*dacbench.envs.luby.LubyEnv method*), 16, 103
render() (*dacbench.envs.LubyEnv method*), 116
render() (*dacbench.envs.sgd.SGDEnv method*), 40, 108
render() (*dacbench.envs.sigmoid.SigmoidEnv method*), 13, 109
render() (*dacbench.envs.SigmoidEnv method*), 117
render() (*dacbench.envs.toysgd.ToySGDEnv method*), 114
render() (*dacbench.envs.ToySGDEnv method*), 119
render_3d_dimensions() (*dacbench.envs.geometric.GeometricEnv method*), 21, 101
render_3d_dimensions() (*dacbench.envs.GeometricEnv method*), 121
render_action_tracking() (*dacbench.wrappers.action_tracking_wrapper.ActionFrequencyWrapper method*), 123
render_action_tracking() (*dacbench.wrappers.ActionFrequencyWrapper method*), 47, 132
render_episode_time() (*dacbench.wrappers.episode_time_tracker.EpisodeTimeWrapper method*), 124
render_episode_time() (*dacbench.wrappers.EpisodeTimeWrapper method*), 48, 133
render_instance_performance() (*dacbench.wrappers.performance_tracking_wrapper.PerformanceTrackingWrapper method*), 127
render_instance_performance() (*dacbench.wrappers.PerformanceTrackingWrapper method*), 49, 137
render_performance() (*dacbench.wrappers.performance_tracking_wrapper.PerformanceTrackingWrapper method*), 127
render_performance() (*dacbench.wrappers.PerformanceTrackingWrapper method*), 49, 137
render_policy_progress() (*dacbench.wrappers.policy_progress_wrapper.PolicyProgressWrapper method*), 128
render_policy_progress() (*dacbench.wrappers.PolicyProgressWrapper method*), 49, 134
render_state_tracking() (*dacbench.wrappers.state_tracking_wrapper.StateTrackingWrapper method*), 131
render_state_tracking() (*dacbench.wrappers.StateTrackingWrapper method*), 50, 136
render_step_time() (*dacbench.wrappers.episode_time_tracker.EpisodeTimeWrapper method*), 124
render_step_time() (*dacbench.wrappers.EpisodeTimeWrapper method*), 48, 133
reset() (*dacbench.abstract_env.AbstractEnv method*), 9, 142
reset() (*dacbench.AbstractEnv method*), 157
reset() (*dacbench.container.remote_env.RemoteEnvironmentClient method*), 90
reset() (*dacbench.container.remote_env.RemoteEnvironmentServer method*), 89
reset() (*dacbench.envs.cma_es.CMAESEnv method*), 32, 95
reset() (*dacbench.envs.cma_step_size.CMAStepSizeEnv method*), 96
reset() (*dacbench.envs.fast_downward.FastDownwardEnv method*), 24, 98
reset() (*dacbench.envs.FastDownwardEnv method*), 118
reset() (*dacbench.envs.geometric.GeometricEnv*

method), 21, 100
reset() (*dacbench.envs.GeometricEnv method*), 121
reset() (*dacbench.envs.luby.LubyEnv method*), 16, 103
reset() (*dacbench.envs.LubyEnv method*), 116
reset() (*dacbench.envs.modcma.ModCMAEnv method*), 37, 104
reset() (*dacbench.envs.modea.ModeaEnv method*), 36, 105
reset() (*dacbench.envs.sgd.SGDEnv method*), 40, 107
reset() (*dacbench.envs.sigmoid.SigmoidEnv method*), 13, 109
reset() (*dacbench.envs.SigmoidEnv method*), 117
reset() (*dacbench.envs.theory.RLSEnv method*), 29, 112
reset() (*dacbench.envs.toysgd.ToySGDEnv method*), 114
reset() (*dacbench.envs.ToySGDEnv method*), 119
reset() (*dacbench.wrappers.instance_sampling_wrapper.InstanceSamplingWrapper method*), 125
reset() (*dacbench.wrappers.InstanceSamplingWrapper method*), 48, 133
reset() (*dacbench.wrappers.observation_wrapper.ObservationWrapper method*), 126
reset() (*dacbench.wrappers.ObservationWrapper method*), 48, 138
reset() (*dacbench.wrappers.state_tracking_wrapper.StateTrackingWrapper method*), 130
reset() (*dacbench.wrappers.StateTrackingWrapper method*), 51, 136
reset_() (*dacbench.abstract_env.AbstractEnv method*), 9, 142
reset_() (*dacbench.AbstractEnv method*), 157
reset_episode() (*dacbench.logger.Logger method*), 149
reset_episode() (*dacbench.logger.ModuleLogger method*), 56, 147
restart() (*dacbench.envs.modea.ModeaEnv method*), 105
Reward (*class in dacbench.envs.sgd*), 40, 106
reward_functions
    module, 161
reward_range (*in module dacbench.benchmarks.sgd_benchmark*), 81
reward_range() (*in module dacbench.envs.sgd*), 106
RewardNoiseWrapper (*class in dacbench.wrappers*), 50, 134
RewardNoiseWrapper (*class in dacbench.wrappers.reward_noise_wrapper*), 129
RLSEnv (*class in dacbench.envs.theory*), 29, 112
RLSEnvDiscrete (*class in dacbench.envs.theory*), 30, 113
root (*in module dacbench.container.remote_runner*), 90
run() (*dacbench.container.remote_runner.RemoteRunner method*), 91
run_benchmark() (*in module dacbench.runner*), 156
run_dacbench() (*in module dacbench.runner*), 156
run_dynamic_policy() (*in module dacbench.run_baselines*), 156
run_optimal() (*in module dacbench.run_baselines*), 156
run_policy() (*in module dacbench.run_baselines*), 156
run_random() (*in module dacbench.run_baselines*), 155
run_static() (*in module dacbench.run_baselines*), 155

S

sample_coefficients() (*in module dacbench.envs.toysgd*), 114
sample_parabel_cubic_value() (*in module SampleGeometricInstances*), 163
sample_sigmoid_value() (*in module SampleGeometricInstances*), 163
sample_sinus_value() (*in module SampleGeometricInstances*), 163
SAMPLE_SIZE (*in module SampleGeometricInstances*), 162
SampleGeometricInstances
    module, 162
save_config() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 139
save_config() (*dacbench.AbstractBenchmark method*), 159
save_geometric_instances() (*in module SampleGeometricInstances*), 162
seed() (*dacbench.abstract_env.AbstractEnv method*), 9, 143
seed() (*dacbench.AbstractEnv method*), 158
seed() (*dacbench.envs.sgd.SGDEnv method*), 40, 107
seed_action_space() (*dacbench.abstract_env.AbstractEnv method*), 9, 143
seed_action_space() (*dacbench.AbstractEnv method*), 158
seed_mapper() (*in module dacbench.logger*), 146
send_msg() (*dacbench.envs.fast_downward.FastDownwardEnv method*), 25, 98
send_msg() (*dacbench.envs.FastDownwardEnv method*), 117
SEQ (*in module dacbench.benchmarks.luby_benchmark*), 78
SEQUENTIAL (*in module dacbench.benchmarks.modcma_benchmark*), 79
serialize_config() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 8, 139
serialize_config() (*dacbench.AbstractBenchmark method*), 159

serialize_random_state() (*in module dacbench.container.container_utils*), 88
SERVERTYPE (*in module dacbench.container.remote_runner*), 90
set_action_description() (*dacbench.benchmarks.geometric_benchmark.GeometricBenchmark method*), 20, 78
set_action_description() (*dacbench.benchmarks.GeometricBenchmark method*), 86
set_action_space() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 8, 140
set_action_space() (*dacbench.AbstractBenchmark method*), 160
set_action_values() (*dacbench.benchmarks.geometric_benchmark.GeometricBenchmark method*), 20, 78
set_action_values() (*dacbench.benchmarks.GeometricBenchmark method*), 86
set_action_values() (*dacbench.benchmarks.sigmoid_benchmark.SigmoidBenchmark method*), 12, 82
set_action_values() (*dacbench.benchmarks.SigmoidBenchmark method*), 85
set_additional_info() (*dacbench.logger.Logger method*), 149
set_additional_info() (*dacbench.logger.ModuleLogger method*), 56, 148
set_cutoff() (*dacbench.benchmarks.luby_benchmark.LubyBenchmark method*), 16, 78
set_cutoff() (*dacbench.benchmarks.LubyBenchmark method*), 84
set_env() (*dacbench.logger.AbstractLogger method*), 54, 146
set_env() (*dacbench.logger.Logger method*), 55, 148
set_heuristics() (*dacbench.benchmarks.fast_downward_benchmark.FastDownwardBenchmark method*), 76
set_heuristics() (*dacbench.benchmarks.FastDownwardBenchmark method*), 87
set_history_length() (*dacbench.benchmarks.luby_benchmark.LubyBenchmark method*), 16, 79
set_history_length() (*dacbench.benchmarks.LubyBenchmark method*), 84
set_inst_id() (*dacbench.abstract_env.AbstractEnv method*), 10, 143
set_inst_id() (*dacbench.AbstractEnv method*), 158
set_instance() (*dacbench.abstract_env.AbstractEnv method*), 10, 143
set_instance() (*dacbench.AbstractEnv method*), 158
set_instance() (*dacbench.envs.geometric.Functions method*), 101
set_instance_set() (*dacbench.abstract_env.AbstractEnv method*), 10, 143
set_instance_set() (*dacbench.AbstractEnv method*), 158
set_observation_space() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 8, 141
set_observation_space() (*dacbench.AbstractBenchmark method*), 160
set_seed() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 8, 140
set_seed() (*dacbench.AbstractBenchmark method*), 160
set_writer() (*dacbench.envs.sgd.SGDEnv method*), 108
setConfigurationParameters() (*dacbench.envs.modea.ModeaEnv method*), 105
SGD_DEFAULTS (*in module dacbench.benchmarks.sgd_benchmark*), 81
SGDBenchmark (*class in dacbench.benchmarks.sgd_benchmark*), 39, 81
SGDEnv (*class in dacbench.envs.sgd*), 40, 106
sig() (*in module dacbench.envs.policies.optimal_sigmoid*), 93
SIGMOID_DEFAULTS (*in module dacbench.benchmarks.sigmoid_benchmark*), 82
SigmoidBenchmark (*class in dacbench.benchmarks*), 85
SigmoidBenchmark (*class in dacbench.benchmarks.sigmoid_benchmark*), 11, 82
SigmoidEnv (*class in dacbench.envs*), 116
SigmoidEnv (*class in dacbench.envs.sigmoid*), 12, 109
small_random_luby_state() (*in module random_states*), 161
small_random_sigmoid_state() (*in module random_states*), 161
socket (*dacbench.container.remote_runner.RemoteRunner property*), 91
socket_from_id() (*dacbench.container.remote_runner.RemoteRunner static method*), 91
SOCKET_PATH (*in module dacbench.container.remote_runner*), 90
space_sep_upper() (*in module dacbench.plotting*), 63, 150
space_to_list() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 140
space_to_list() (*dacbench.AbstractBenchmark method*), 159
split() (*in module dacbench.logger*), 57, 145

start() (*dacbench.container.remote_runner.RemoteRunnerClient method*), 90
StateTrackingWrapper (*class in dacbench.wrappers*), 50, 135
StateTrackingWrapper (*class in dacbench.wrappers.state_tracking_wrapper*), 130
StateType (*class in dacbench.envs.fast_downward*), 25, 97
StaticAgent (*class in dacbench.agents*), 74
StaticAgent (*class in dacbench.agents.simple_agents*), 71
step() (*dacbench.abstract_env.AbstractEnv method*), 10, 142
step() (*dacbench.AbstractEnv method*), 157
step() (*dacbench.container.remote_env.RemoteEnvironmentClient method*), 90
step() (*dacbench.container.remote_env.RemoteEnvironmentServer method*), 89
step() (*dacbench.envs.cma_es.CMAESEnv method*), 32, 95
step() (*dacbench.envs.cma_step_size.CMAStepSizeEnv method*), 96
step() (*dacbench.envs.fast_downward.FastDownwardEnv method*), 25, 98
step() (*dacbench.envs.FastDownwardEnv method*), 118
step() (*dacbench.envs.geometric.GeometricEnv method*), 21, 100
step() (*dacbench.envs.GeometricEnv method*), 121
step() (*dacbench.envs.luby.LubyEnv method*), 16, 103
step() (*dacbench.envs.LubyEnv method*), 116
step() (*dacbench.envs.modcma.ModCMAEnv method*), 38, 104
step() (*dacbench.envs.modea.ModeaEnv method*), 36, 105
step() (*dacbench.envs.sgd.SGDEnv method*), 40, 107
step() (*dacbench.envs.sigmoid.ContinuousSigmoidEnv method*), 12, 110
step() (*dacbench.envs.sigmoid.ContinuousStateSigmoidEnv method*), 12, 110
step() (*dacbench.envs.sigmoid.SigmoidEnv method*), 13, 109
step() (*dacbench.envs.SigmoidEnv method*), 116
step() (*dacbench.envs.theory.RLSEnv method*), 30, 112
step() (*dacbench.envs.theory.RLSEnvDiscrete method*), 30, 113
step() (*dacbench.envs.toysgd.ToySGDEnv method*), 114
step() (*dacbench.envs.ToySGDEnv method*), 119
step() (*dacbench.wrappers.action_tracking_wrapper.ActionFrequencyWrapper method*), 122
step() (*dacbench.wrappers.ActionFrequencyWrapper method*), 47, 132
step() (*dacbench.wrappers.episode_time_tracker.EpisodeTimeWrapper method*), 124
step() (*dacbench.wrappers.EpisodeTimeWrapper method*), 48, 132
step() (*dacbench.wrappers.observation_wrapper.ObservationWrapper method*), 126
step() (*dacbench.wrappers.ObservationWrapper method*), 49, 138
step() (*dacbench.wrappers.performance_tracking_wrapper.PerformanceT method*), 127
step() (*dacbench.wrappers.PerformanceTrackingWrapper method*), 49, 137
step() (*dacbench.wrappers.policy_progress_wrapper.PolicyProgressWrap method*), 128
step() (*dacbench.wrappers.PolicyProgressWrapper method*), 49, 134
step() (*dacbench.wrappers.reward_noise_wrapper.RewardNoiseWrapper method*), 129
step() (*dacbench.wrappers.RewardNoiseWrapper method*), 50, 135
step() (*dacbench.wrappers.state_tracking_wrapper.StateTrackingWrapper method*), 130
step() (*dacbench.wrappers.StateTrackingWrapper method*), 51, 136
step_() (*dacbench.abstract_env.AbstractEnv method*), 10, 142
step_() (*dacbench.AbstractEnv method*), 157
STEP_SIZE (*in module dacbench.benchmarks.cma_benchmark*), 75
STEP_SIZE_ADAPTION (*in module dacbench.benchmarks.modcma_benchmark*), 79
sum_reward() (*in module reward_functions*), 162
switchConfiguration() (*dacbench.envs.modea.ModeaEnv method*), 105

T

THEORY_DEFAULTS (*in module dacbench.benchmarks.theory_benchmark*), 83
TheoryBenchmark (*class in dacbench.benchmarks.theory_benchmark*), 27, 83
THRESHOLD_CONVERGENCE (*in module dacbench.benchmarks.modcma_benchmark*), 79
to_json() (*dacbench.abstract_benchmark.AbstractBenchmark method*), 139
to_json() (*dacbench.AbstractBenchmark method*), 159
ToySGDBenchmark (*class in dacbench.benchmarks*), 85
ToySGDBenchmark (*class in dacbench.benchmarks.toysgd_benchmark*), 84
ToySGDEnv (*class in dacbench.envs*), 119

ToySGDEnv (*class in dacbench.envs.toysgd*), 114
train() (*dacbench.abstract_agent.AbstractDACBenchAgent method*), 138
train() (*dacbench.agents.dynamic_random_agent.DynamicRandomAgent method*), 70
train() (*dacbench.agents.DynamicRandomAgent method*), 74
train() (*dacbench.agents.generic_agent.GenericAgent method*), 70
train() (*dacbench.agents.GenericAgent method*), 73
train() (*dacbench.agents.RandomAgent method*), 73
train() (*dacbench.agents.simple_agents.RandomAgent method*), 71
train() (*dacbench.agents.simple_agents.StaticAgent method*), 72
train() (*dacbench.agents.StaticAgent method*), 74
train() (*dacbench.envs.policies.sgd_ca.CosineAnnealingAgent method*), 93
train_network() (*dacbench.envs.sgd.SGDEnv method*), 108
TrainingLoss (*dacbench.envs.sgd.Reward attribute*), 106

U

update_parameters() (*dacbench.envs.modea.ModeaEnv method*), 105
use_next_instance() (*dacbench.abstract_env.AbstractEnv method*), 10, 142
use_next_instance() (*dacbench.AbstractEnv method*), 157
use_test_set() (*dacbench.abstract_env.AbstractEnv method*), 10, 143
use_test_set() (*dacbench.AbstractEnv method*), 158
use_training_set() (*dacbench.abstract_env.AbstractEnv method*), 10, 143
use_training_set() (*dacbench.AbstractEnv method*), 158

V

val_model (*dacbench.envs.sgd.SGDEnv attribute*), 41, 106
valid_types (*dacbench.logger.AbstractLogger attribute*), 146
ValidationLoss (*dacbench.envs.sgd.Reward attribute*), 106

W

wait_for_port() (*in module dacbench.container.container_utils*), 89
wait_for_unixsocket() (*in module dacbench.container.container_utils*), 88
WEIGHTS_OPTION (*in module dacbench.benchmarks.modcma_benchmark*), 79
write() (*dacbench.logger.AbstractLogger method*), 54, 146
write() (*dacbench.logger.Logger method*), 55, 149
write() (*dacbench.logger.ModuleLogger method*), 56, 148

X

X (*in module dacbench.benchmarks.sigmoid_benchmark*), 82