trixi Documentation

Release 0.1

MIC

Jan 13, 2021

Contents

This is the documentation of **trixi** (Training & Retrospective Insights eXperiment Infrastructure). **trixi** is a python package aiming to facilitate the setup, visualization and comparison of reproducible experiments, currently with a focus on experiments using PyTorch.

You can jump right into the package by looking into our *Quick Start*.

Install Trixi:

```
pip install trixi
```

Install trixi directly via git:

```
git clone https://github.com/MIC-DKFZ/trixi.git
cd trixi
pip install -e .
```

CHAPTER 1

Quick Start

Introduction & Features:

https://github.com/MIC-DKFZ/trixi#features

Install trixi:

```
pip install trixi
```

Have a look and run a simple MNIST example:

https://github.com/MIC-DKFZ/trixi/blob/master/examples/pytorch_experiment.ipynb

CHAPTER 2

License

```
The MIT License (MIT)

Copyright (c) 2018 Medical Image Computing Group, DKFZ

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER 3

Authors

Core Development Team:

- David Zimmerer (d.zimmerer@dkfz.de)
- Jens Petersen (jens.petersen@dkfz.de)
- Gregor Koehler (g.koehler@dkfz.de)

Contributions:

- Jakob Wasserthal
- Sebastian Wirkert
- Lisa Kausch

CHAPTER 4

trixi.experiment

4.1 Experiment

class trixi.experiment.experiment.**Experiment**(*n_epochs=0*)

Bases: object

An abstract Experiment which can be run for a number of epochs.

The basic life cycle of an experiment is:

```
setup()
prepare()

while epoch < n_epochs:
    train()
    validate()
    epoch += 1

end()
```

If you want to use another criterion than number of epochs, e.g. stopping based on validation loss, you can implement that in your validation method and just call .stop() at some point to break the loop. Just set your n_epochs to a high number or np.inf.

The reason there is both *setup()* and *prepare()* is that internally there is also a _setup_internal() method for hidden magic in classes that inherit from this. For example, the *trixi.experiment.pytorchexperiment.PytorchExperiment* uses this to restore checkpoints. Think of *setup()* as an __init__() that is only called when the Experiment is actually asked to do anything. Then use *prepare()* to modify the fully instantiated Experiment if you need to.

To write a new Experiment simply inherit the Experiment class and overwrite the methods. You can then start your Experiment calling *run()*

In Addition the Experiment also has a test function. If you call the *run_test()* method it will call the *test()* and *end_test()* method internally (and if you give the parameter setup = True in run_test is will again call *setup()* and *prepare()*).

Each Experiment also has its current state in `_exp_state`, its start time in `_time_start`, its end time in `_time_end` and the current epoch index in `_epoch_idx`

Parameters **n_epochs** (*int*) – The number of epochs in the Experiment (how often the train and validate method will be called)

end()
Is called at the end of each experiment

end_test()
Is called at the end of each experiment test

epoch
Convenience access property for self._epoch_idx

prepare()
This method is called directly before the experiment training starts

process_err(*e*)
This method is called if an error occurs during the execution of an experiment. Will just raise by default.

Parameters **e** (*Exception*) – The exception which was raised during the experiment life cycle

run(*setup=True*)
This method runs the Experiment. It runs through the basic lifecycle of an Experiment:

```
setup()
prepare()

while epoch < n_epochs:
    train()
    validate()
    epoch += 1

end()
```

run_test(*setup=True*)
This method runs the Experiment.

The test consist of an optional setup and then calls the *test()* and *end_test()*.

Parameters **setup** – If True it will execute the *setup()* and *prepare()* function similar to the run method before calling *test()*.

setup()
Is called at the beginning of each Experiment run to setup the basic components needed for a run.

stop()
If called the Experiment will stop after that epoch and not continue training

test()
The testing part of the Experiment

train(*epoch*)
The training part of the Experiment, it is called once for each epoch

Parameters **epoch** (*int*) – The current epoch the train method is called in

validate(*epoch*)
The evaluation/validation part of the Experiment, it is called once for each epoch (after the training part)

Parameters **epoch** (*int*) – The current epoch the validate method is called in

4.2 PytorchExperiment

class trixi.experiment.pytorchexperiment.**PytorchExperiment**(*config=None*, *name=None*, *n_epochs=None*, *seed=None*, *base_dir=None*, *globs=None*, *resume=None*, *ignore_resume_config=False*, *resume_save_types=('model', 'optimizer', 'simple', 'th_vars', 'results')*, *resume_reset_epochs=True*, *parse_sys_argv=False*, *checkpoint_to_cpu=True*, *save_checkpoint_every_epoch=1*, *explogger_kwargs=None*, *explogger_freq=1*, *loggers=None*, *append_rnd_to_name=False*, *default_save_types=('model', 'optimizer', 'simple', 'th_vars', 'results')*, *save_checkpoints_default=True*)

Bases: *trixi.experiment.experiment.Experiment*

A PytorchExperiment extends the basic functionality of the *Experiment* class with convenience features for PyTorch (and general logging) such as creating a folder structure, saving, plotting results and checkpointing your experiment.

The basic life cycle of a PytorchExperiment is the same as *Experiment*:

```
setup()
prepare()

for epoch in n_epochs:
    train()
    validate()

end()
```

where the distinction between the first two is that between them PytorchExperiment will automatically restore checkpoints and save the `_config_raw` in `_setup_internal()`. Please see below for more information on this. To get your own experiment simply inherit from the PytorchExperiment and overwrite the *setup()*, *prepare()*, *train()*, *validate()* method (or you can use the *very* experimental decorator *experimentify()* to convert your class into a experiment). Then you can run your own experiment by calling the *run()* method.

Internally PytorchExperiment will provide a number of member variables which you can access.

- **n_epochs** Number of epochs.
- **exp_name** Name of your experiment.
- **config** The (initialized) *Config* of your experiment. You can access the uninitialized one via `_config_raw`.
- **result** A dict in which you can store your result values. If a *PytorchExperimentLogger* is used, results will be a *ResultLogDict* that directly automatically writes to a file and also stores the N last entries for each key for quick access (e.g. to quickly get the running mean).
- **elog (if base_dir is given)** A *PytorchExperimentLogger* instance which can log your results to a given folder. Will automatically be created if a base_dir is available.
- **loggers** Contains all loggers you provide, including the experiment logger, accessible by the names you provide.
- **clog** A *CombinedLogger* instance which logs to all loggers with different frequencies (specified with the last entry in the tuple you provide for each logger where 1 means every time and N means every Nth time, e.g. if you only want to send stuff to Visdom every 10th time).

The most important attribute is certainly `config`, which is the initialized *Config* for the experiment. To understand how it needs to be structured to allow for automatic instantiation of types, please refer to its documentation. If you decide not to use this functionality, `config` and `_config_raw` are identical. **Beware however that by default the Pytorchexperiment only saves the raw config** after *setup()*. If you modify `config` during setup, make sure to implement `_setup_internal()` yourself should you want the modified config to be saved:

```
def _setup_internal(self):

    super(YourExperiment, self)._setup_internal() # calls .prepare_resume()
    self.elog.save_config(self.config, "config")
```

Parameters

- **config** (*dict or Config*) – Configures your experiment. If `name`, `n_epochs`, `seed`, `base_dir` are given in the config, it will automatically overwrite the other args/kwargs with the values from the config. In addition (defined by `parse_config_sys_argv`) the config automatically parses the argv arguments and updates its values if a key matches a console argument.
- **name** (*str*) – The name of the PytorchExperiment.
- **n_epochs** (*int*) – The number of epochs (number of times the training cycle will be executed).
- **seed** (*int*) – A random seed (which will set the random, numpy and torch seed).
- **base_dir** (*str*) – A base directory in which the experiment result folder will be created. A *PytorchExperimentLogger* instance will be created if this is given.
- **globs** – The `globals()` of the script which is run. This is necessary to get and save the executed files in the experiment folder.
- **resume** (*str or PytorchExperiment*) – Another PytorchExperiment or path to the result dir from another PytorchExperiment from which it will load the PyTorch modules and other member variables and resume the experiment.
- **ignore_resume_config** (*bool*) – If `True` it will not resume with the config from the resume experiment but take the current/own config.

- **resume_save_types** (*list or tuple*) – A list which can define which values to restore when resuming. Choices are:
  - "model" <– Pytorch models
  - "optimizer" <– Optimizers
  - "simple" <– Simple python variables (basic types and lists/tuples
  - "th_vars" <– torch tensors/variables
  - "results" <– The result dict
- **resume_reset_epochs** (*bool*) – Set epoch to zero if you resume an existing experiment.
- **parse_sys_argv** (*bool*) – Parsing the console arguments (argv) to get a `config path` and/or `resume_path`.
- **parse_config_sys_argv** (*bool*) – Parse argv to update the config (if the keys match).
- **checkpoint_to_cpu** (*bool*) – When checkpointing, transfer all tensors to the CPU beforehand.
- **save_checkpoint_every_epoch** (*int*) – Determines after how many epochs a checkpoint is stored.
- **explogger_kwargs** (*dict*) – Keyword arguments for `elog` instantiation.
- **explogger_freq** (*int*) – The frequency x (meaning one in x) with which the `clog` will call the `elog`.
- **loggers** (*dict*) – Specify additional loggers. Entries should have one of these formats:

```
"name": "identifier" (will default to a frequency of 10)
"name": ("identifier"(, kwargs, frequency)) (last two are optional)
```

  "identifier" is one of "telegram", "tensorboard", "visdom", "slack".
- **append_rnd_to_name** (*bool*) – If `True`, will append a random six digit string to the experiment name.
- **save_checkpoints_default** (*bool*) – By default save the current and the last checkpoint or not.

add_result(*value*, *name*, *counter=None*, *tag=None*, *label=None*, *plot_result=True*, *plot_running_mean=False*)

Saves a results and add it to the result dict, this is similar to results[key] = val, but in addition also logs the value to the combined logger (it also stores in the results-logs file).

This should be your preferred method to log your numeric values

Parameters

- **value** – The value of your variable
- **name** (*str*) – The name/key of your variable
- **counter** (*int or float*) – A counter which can be seen as the x-axis of your value. Normally you would just use the current epoch for this.
- **tag** (*str*) – A label/tag which can group similar values and will plot values with the same label in the same plot

- **label** – deprecated label
- **plot_result** (*bool*) – By default True, will also log all your values to the combined logger (with show_value).

add_result_without_epoch(*val*, *name*)
A faster method to store your results, has less overhead and does not call the combined logger. Will only store to the results dictionary.

Parameters

- **val** – the value you want to add.
- **name** (*str*) – the name/key of your value.

at_exit_func()
Stores the results and checkpoint at the end (if not already stored). This method is also called if an error occurs.

get_pytorch_modules(*from_config=True*)
Returns all torch.nn.Modules stored in the experiment in a dict (even child dicts are stored).

Parameters **from_config** (*bool*) – Also get modules that are stored in the `config` attribute.

Returns Dictionary of PyTorch modules

Return type dict

get_pytorch_optimizers(*from_config=True*)
Returns all torch.optim.Optimizers stored in the experiment in a dict.

Parameters **from_config** (*bool*) – Also get optimizers that are stored in the `config` attribute.

Returns Dictionary of PyTorch optimizers

Return type dict

get_pytorch_tensors(*ignore=()*)
Returns all torch.tensors in the experiment in a dict.

Parameters **ignore** (*list or tuple*) – Iterable of names which will be ignored

Returns Dictionary of PyTorch tensor

Return type dict

get_pytorch_variables(*ignore=()*)
Same as *get_pytorch_tensors()*.

get_result(*name*)
Similar to result[key] this will return the values in the results dictionary with the given name/key.

Parameters **name** (*str*) – the name/key for which a value is stored.

Returns The value with the key 'name' in the results dict.

get_result_without_epoch(*name*)
Similar to result[key] this will return the values in result with the given name/key.

Parameters **name** (*str*) – the name/ key for which a value is stores.

Returns The value with the key 'name' in the results dict.

get_simple_variables(*ignore=()*)
Returns all standard variables in the experiment in a dict. Specifically, this looks for types `int`, `float`, `bytes`, `bool`, `str`, `set`, `list`, `tuple`.

Parameters **ignore** (*list or tuple*) – Iterable of names which will be ignored

Returns Dictionary of variables

Return type dict

load_checkpoint(*name='checkpoint'*, *save_types=('model'*, *'optimizer'*, *'simple'*, *'th_vars'*, *'results')*, *n_iter=None*, *iter_format='{:05d}'*, *prefix=False*, *path=None*)
Loads a checkpoint and restores the experiment.

Make sure you have your torch stuff already on the right devices beforehand, otherwise this could lead to errors e.g. when making a optimizer step (and for some reason the Adam states are not already on the GPU: https://discuss.pytorch.org/t/loading-a-saved-model-for-continue-training/17244/3)

Parameters

- **name** (*str*) – The name of the checkpoint file
- **save_types** (*list or tuple*) – What kind of member variables should be loaded? Choices are: "model" <– Pytorch models, "optimizer" <– Optimizers, "simple" <– Simple python variables (basic types and lists/tuples), "th_vars" <– torch tensors, "results" <– The result dict
- **n_iter** (*int*) – Number of iterations. Together with the name, defined by the iter_format, a file name will be created and searched for.
- **iter_format** (*str*) – Defines how the name and the n_iter will be combined.
- **prefix** (*bool*) – If True, the formatted n_iter will be prepended, otherwise appended.
- **path** (*str*) – If no path is given then it will take the current experiment dir and formatted name, otherwise it will simply use the path and the formatted name to define the checkpoint file.

load_pytorch_models()
Loads all model files from the experiment checkpoint folder.

load_simple_vars()
Restores all simple python member variables from the 'simple_vars.json' file in the log folder.

log_simple_vars()
Logs all simple python member variables as a json file in the experiment log folder. The file will be names 'simple_vars.json'.

prepare_resume()
Tries to resume the experiment by using the defined resume path or PytorchExperiment.

print(**args*)
Calls 'print' on the experiment logger or uses builtin 'print' if former is not available.

process_err(*e*)
This method is called if an error occurs during the execution of an experiment. Will just raise by default.

Parameters **e** (*Exception*) – The exception which was raised during the experiment life cycle

save_checkpoint(*name='checkpoint'*, *save_types=('model'*, *'optimizer'*, *'simple'*, *'th_vars'*, *'results')*, *n_iter=None*, *iter_format='{:05d}'*, *prefix=False*)
Saves a current model checkpoint from the experiment.

Parameters

- **name** (*str*) – The name of the checkpoint file
- **save_types** (*list or tuple*) – What kind of member variables should be stored? Choices are: "model" <– Pytorch models, "optimizer" <– Optimizers, "simple" <– Simple python variables (basic types and lists/tuples), "th_vars" <– torch tensors, "results" <– The result dict
- **n_iter** (*int*) – Number of iterations. Together with the name, defined by the iter_format, a file name will be created.
- **iter_format** (*str*) – Defines how the name and the n_iter will be combined.
- **prefix** (*bool*) – If True, the formatted n_iter will be prepended, otherwise appended.

save_end_checkpoint()
Saves the current checkpoint as checkpoint_last.

save_pytorch_models()
Saves all torch.nn.Modules as model files in the experiment checkpoint folder.

save_results(*name='results.json'*)
Saves the result dict as a json file in the result dir of the experiment logger.

Parameters **name** (*str*) – The name of the json file in which the results are written.

save_temp_checkpoint()
Saves the current checkpoint as checkpoint_current.

slog

tblog

tlog

txlog

update_attributes(*var_dict*, *ignore=()*)
Updates the member attributes with the attributes given in the var_dict

Parameters

- **var_dict** (*dict*) – dict in which the update values stored. If a key matches a member attribute name the member attribute will be updated
- **ignore** (*list or tuple*) – iterable of keys to ignore

vlog

trixi.experiment.pytorchexperiment.**experimentify**(*setup_fn='setup'*, *train_fn='train'*, *validate_fn='validate'*, *end_fn='end'*, *test_fn='test'*, ***deocoargs*)
Experimental decorator which monkey patches your class into a PytorchExperiment. You can then call run on your new *PytorchExperiment* class.

Parameters

- **setup_fn** – The name of your setup() function
- **train_fn** – The name of your train() function
- **validate_fn** – The name of your validate() function
- **end_fn** – The name of your end() function
- **test_fn** – The name of your test() function

trixi.experiment.pytorchexperiment.**get_last_file**(*dir_*, *name=None*)
Returns the most recently created file in the folder which matches the name supplied

Parameters

- **dir** – The base directory to start the search in
- **name** – The name pattern to match with the files

Returns the path to the most recent file

Return type str

trixi.experiment.pytorchexperiment.**get_vars_from_sys_argv**()
Parses the command line args (argv) and looks for –config_path and –resume_path and returns them if found.

Returns a tuple of (config_path, resume_path) , None if it is not found

Return type tuple

CHAPTER 5

trixi.experiment_browser

5.1 browser

trixi.experiment_browser.browser.**combine**(*base_dir*)

trixi.experiment_browser.browser.**create_flask_app**(*base_dir*)

trixi.experiment_browser.browser.**experiment**(*base_dir*)

trixi.experiment_browser.browser.**experiment_log**(*base_dir*)

trixi.experiment_browser.browser.**experiment_plots**(*base_dir*)

trixi.experiment_browser.browser.**experiment_remove**(*base_dir*)

trixi.experiment_browser.browser.**experiment_rename**(*base_dir*)

trixi.experiment_browser.browser.**experiment_star**(*base_dir*)

trixi.experiment_browser.browser.**overview**(*base_dir*)

trixi.experiment_browser.browser.**overview_**(*base_dir*)

trixi.experiment_browser.browser.**parse_args**()

trixi.experiment_browser.browser.**register_url_routes**(*app*, *base_dir*)

trixi.experiment_browser.browser.**start_browser**()

5.2 dataprocessing

trixi.experiment_browser.dataprocessing.**group_images**(*images*)

trixi.experiment_browser.dataprocessing.**make_graphs**(*results*, *trace_options=None*, *layout_options=None*, *color_map=<sphinx.ext.autodoc.importer._MockObject object>*)

Create plot markups.

This converts results into plotly plots in markup form. Results in a common group will be placed in the same plot.

Parameters **results** (*dict*) – Dictionary

trixi.experiment_browser.dataprocessing.**merge_results**(*experiment_names*, *result_list*)

trixi.experiment_browser.dataprocessing.**process_base_dir**(*base_dir*, *view_dir=''*, *default_val='-'*, *short_len=25*, *ignore_keys=('name', 'experiment_dir', 'work_dir', 'config_dir', 'log_dir', 'checkpoint_dir', 'img_dir', 'plot_dir', 'save_dir', 'result_dir', 'time', 'state')*)

Create an overview table of all experiments in the given directory.

Parameters

- **directory** (*str*) – A directory containing experiment folders.
- **default_val** (*str*) – Default value if an entry is missing.
- **short_len** (*int*) – Cut strings to this length. Full string in alt-text.

Returns

{"ccols": Columns for config entries, "rcols": Columns for result entries, "rows": The actual data}

Return type dict

5.3 ExperimentReader

class trixi.experiment_browser.experimentreader.**CombiExperimentReader**(*base_dir*, *exp_dirs=()*, *name=None*, *decode_config_clean_str=True*)

Bases: *trixi.experiment_browser.experimentreader.ExperimentReader*

get_config()

get_result_log_dict()

get_results()

Get the last result item.

Returns The last result item in the experiment.

Return type dict

get_results_full()

get_results_log()
Build result dictionary.

During the experiment result items are written out as a stream of quasi-atomic units. This reads the stream and builds arrays of corresponding items. The resulting dict looks like this:

```
{
    "result group": {
        "result": {
            "counter": x-array,
            "data": y-array
        }
    }
}
```

Returns Result dictionary.

Return type dict

ignore_experiment()
Create a flag file, so the browser ignores this experiment.

read_meta_info()
Reads the meta info of the experiment i.e. new name, stared or ignored

save(*target_dir=None*)

update_meta_info(*name=None*, *star=None*, *ignore=None*)
Updates the meta info i.e. new name, stared or ignored and saves it in the experiment folder

Parameters

- **name** (*str*) – New name of the experiment
- **star** (*bool*) – Flag, if experiment is starred/ favorited
- **ignore** (*boll*) – Flag, if experiment should be ignored

class trixi.experiment_browser.experimentreader.**ExperimentReader**(*base_dir*, *exp_dir=''*, *name=None*, *decode_config_clean_str=True*)

Bases: object

Reader class to read out experiments created by `trixi.experimentlogger.ExperimentLogger`.

Parameters

- **work_dir** (*str*) – Directory with the structure defined by `trixi.experimentlogger.ExperimentLogger`.
- **name** (*str*) – Optional name for the experiment. If None, will try to read name from experiment config.

get_checkpoints()

static get_file_contents(*folder*, *include_subdirs=False*)
Get all files in a folder.

Returns All files joined with folder path.

Return type list

get_images()

get_log_file_content(*file_name*)
Read out log file and HTMLify.

Parameters **file_name** (*str*) – Name of the log file.

Returns Log file contents as HTML ready string.

Return type str

get_logs()

get_plots()

get_results()
Get the last result item.

Returns The last result item in the experiment.

Return type dict

get_results_log()
Build result dictionary.

During the experiment result items are written out as a stream of quasi-atomic units. This reads the stream and builds arrays of corresponding items. The resulting dict looks like this:

```
{
    "result group": {
        "result": {
            "counter": x-array,
            "data": y-array
        }
    }
}
```

Returns Result dictionary.

Return type dict

ignore_experiment()
Create a flag file, so the browser ignores this experiment.

read_meta_info()
Reads the meta info of the experiment i.e. new name, stared or ignored

update_meta_info(*name=None*, *star=None*, *ignore=None*)
Updates the meta info i.e. new name, stared or ignored and saves it in the experiment folder

Parameters

- **name** (*str*) – New name of the experiment
- **star** (*bool*) – Flag, if experiment is starred/ favorited
- **ignore** (*boll*) – Flag, if experiment should be ignored

trixi.experiment_browser.experimentreader.**group_experiments_by**(*exps*, *group_by_list*)

CHAPTER 6

trixi.logger

6.1 experiment

6.1.1 ExperimentLogger

class trixi.logger.experiment.experimentlogger.**ExperimentLogger**(*exp_name*, *base_dir*, *folder_format='%Y%m%d-%H%M%S_{experiment_name}'*, *resume=False*, *text_logger_args=None*, *plot_logger_args=None*, ***kwargs*)

Bases: *trixi.logger.abstractlogger.AbstractLogger*

A single class for logging your experiments to file.

It creates a experiment folder in your base folder and a folder structure to store your experiment files. The folder structure is:

```
base_dir/
    new_experiment_folder/
        checkpoint/
        config/
        img/
        log/
        plot/
        result/
        save/
```

load_checkpoint()

load_config(*name*, ***kwargs*)

Loads a config from a json file from the experiment config dir

Parameters **name** – the name of the config file

Returns: A Config/ dict filled with the json file content

load_dict(*path*)
Loads a json file as dict from a sub path in the experiment save dir

Parameters **path** – sub path to the file (starting from the experiment save dir)

Returns: The restored data as a dict

load_model()

load_numpy_data(*path*)
Loads a numpy file from a sub path in the experiment save dir

Parameters **path** – sub path to the file (starting from the experiment save dir)

Returns: The restored numpy array

load_pickle(*path*)
Loads a object via pickle from a sub path in the experiment save dir

Parameters **path** – sub path to the file (starting from the experiment save dir)

Returns: The restored object

resolve_format(*input_*, *resume*)

Given some input pattern, tries to find the best matching folder name by resolving the format. Options are:

- Run-number: {run_number}
- Time: "%Y%m%d-%H%M%S
- Member variables (e.g experiment_name) : {variable_name} (e.g. {experiment_name})

Parameters

- **input** – The format to be resolved
- **resume** – Flag if folder should be resumed

Returns: The resolved folder name

save_checkpoint()

save_config(*data*, *name*, ***kwargs*)
Saves a config as a json file in the experiment config dir

Parameters

- **data** – The data to be stored as config json
- **name** – The name of the json file in which the data will be stored

save_dict(*data*, *path*, *indent=4*, *separators=(', ', ': ')*, *encoder_cls=<class 'trixi.util.util.MultiTypeEncoder'>*, ***kwargs*)
Saves a dict as a json file in the experiment save dir

Parameters

- **data** – The data to be stored as save file
- **path** – sub path in the save folder (or simply filename)
- **indent** – Indent for the json file

- **separators** – Separators for the json file
- **encoder_cls** – Encoder Class for the encoding to json

save_file(*filepath*, *path=None*)
Copies a file to the experiment save dir

Parameters

- **filepath** – Path to the file to be copied to the experiment save dir
- **path** – sub path to the target file (starting from the experiment save dir, does not have to exist yet)

save_model()

save_numpy_data(*data*, *path*)
Saves a numpy array in the experiment save dir

Parameters

- **data** – The array to be stored as a save file
- **path** – sub path in the save folder (or simply filename)

save_pickle(*data*, *path*)
Saves a object data in the experiment save dir via pickle

Parameters

- **data** – The data to be stored as a save file
- **path** – sub path in the save folder (or simply filename)

save_result(*data*, *name*, *indent=4*, *separators=(', ', ': ')*, *encoder_cls=<class 'trixi.util.util.MultiTypeEncoder'>*, ***kwargs*)
Saves data as a json file in the experiment result dir

Parameters

- **data** – The data to be stored as result json
- **name** – name of the result json file
- **indent** – Indent for the json file
- **separators** – Separators for the json file
- **encoder_cls** – Encoder Class for the encoding to json

show_barplot(*array*, *name*, *file_format='.png'*, ***kwargs*)
This function saves a barplot in the experiment plot folder.

Parameters

- **array** (*np.ndarray*) – array to be plotted
- **name** (*str*) – image title
- **file_format** (*str*) – file format of the image

show_boxplot(*array*, *name*, *file_format='.png'*, ***kwargs*)
This function saves a boxplot in the experiment plot folder.

Parameters

- **array** (*np.ndarray*) – array to be plotted
- **name** (*str*) – image title

- **file_format** (*str*) – file format of the image

show_image(*image*, *name*, *file_format='.png'*, ***kwargs*)
This function saves an image in the experiment img folder.

Parameters

- **image** (*np.ndarray*) – image to be shown
- **name** (*str*) – image title
- **file_format** (*str*) – file format of the image

show_lineplot(*y_vals*, *x_vals=None*, *name='lineplot'*, *file_format='.png'*, ***kwargs*)
This function saves a line plot in the experiment plot folder.

Parameters

- **x_vals** – x values of the line
- **y_vals** – y values of the line
- **name** (*str*) – image title
- **file_format** (*str*) – file format of the image

show_matplot_plt(*figure*, *name*, *file_format='.png'*, **args*, ***kwargs*)
This function saves a custom matplotlib figure in the experiment plot folder.

Parameters

- **figure** (*matplotlib.figure.Figure*) – figure to be plotted
- **name** (*str*) – image title
- **file_format** (*str*) – file format of the image

show_piechart(*array*, *name*, *file_format='.png'*, ***kwargs*)
This function saves a piechart in the experiment plot folder.

Parameters

- **array** (*np.ndarray*) – array to be plotted
- **name** (*str*) – image title
- **file_format** (*str*) – file format of the image

show_scatterplot(*array*, *name*, *file_format='.png'*, ***kwargs*)
This function saves a scatterplot in the experiment plot folder.

Parameters

- **array** (*np.ndarray*) – array to be plotted
- **name** (*str*) – image title
- **file_format** (*str*) – file format of the image

show_text(*text*, *name=None*, *logger='default'*, ***kwargs*)
Logs a text to a log file.

Parameters

- **text** – The text to be logged
- **name** – Name of the text
- **logger** – log file (in the experiment log folder) in which the text will be logged.

- ****kwargs** –

show_value(*value*, *name=None*, *counter=None*, *tag=None*, *file_format='.png'*, ***kwargs*)
This function saves a value as a consequtive line plot.

Parameters

- **value** (*np.ndarray*) – value to be plotted
- **name** (*str*) – image title
- **counter** – y-value of the image (if not suppled simply increases for each call)
- **tag** – group/label for the value. Values with the same tag will be plotted in the same plot
- **file_format** (*str*) – file format of the image

6.1.2 PytorchExperimentLogger

class trixi.logger.experiment.pytorchexperimentlogger.**PytorchExperimentLogger**(**args*, ***kwargs*)
Bases: *trixi.logger.experiment.experimentlogger.ExperimentLogger*

A single class for logging your pytorch experiments to file. Extends the ExperimentLogger also also creates a experiment folder with a file structure:

The folder structure is :

base_dir/

new_experiment_folder/ checkpoint/ config/ img/ log/ plot/ result/ save/

static get_classification_metrics(*tensor*, *labels*, *name=''*, *metric=('roc-auc', 'pr-score')*, *use_sub_process=False*, *tag_name=None*, *results_fn=<function PytorchExperimentLogger.<lambda>>*)
Displays some classification metrics as line plots in a graph (similar to show value (also uses show value for the caluclated values))

Parameters

- **tensor** – Tensor with scores (e.g class probability)
- **labels** – Labels of the samples to which the scores match
- **name** – The name of the window
- **metric** – List of metrics to calculate. Options are: roc-auc, pr-auc, pr-score, mcc, f1
- **tag_name** – Name for the tag, if no given use name
- **use_sub_process** – Use a sub process to do the processing, if true nothing is returned
- **results_fn** – function which is called with the results/ return values. Expected f(val, name, tag)

Returns:

static get_input_gradient(*model*, *inpt*, *err_fn*, *grad_type='vanilla'*, *n_runs=20*, *eps=0.1*, *abs=False*, *results_fn=<function PytorchExperimentLogger.<lambda>>*)
Given a model creates calculates the error and backpropagates it to the image and saves it (saliency map).

Parameters

- **model** – The model to be evaluated

- **inpt** – Input to the model
- **err_fn** – The error function the evaluate the output of the model on
- **grad_type** – Gradient calculation method, currently supports (vanilla, vanilla-smooth, guided,
- **guided-smooth)** (*the guided backprob can lead to segfaults -. -*) –
- **n_runs** – Number of runs for the smooth variants
- **eps** – noise scaling to be applied on the input image (noise is drawn from N(0,1))
- **abs** (*bool*) – Flag, if the gradient should be a absolute value
- **results_fn** – function which is called with the results/ return values. Expected f(grads)

static get_pr_curve(*tensor*, *labels*, *reduce_to_n_samples=None*, *use_sub_process=False*, *results_fn=<function PytorchExperimentLogger.<lambda>>*)

Displays a precision recall curve given a tensor with scores and the coresponding labels

Parameters

- **tensor** – Tensor with scores (e.g class probability)
- **labels** – Labels of the samples to which the scores match
- **reduce_to_n_samples** – Reduce/ downsample to to n samples for fewer data points
- **use_sub_process** – Use a sub process to do the processing, if true nothing is returned
- **results_fn** – function which is called with the results/ return values. Expected f(precision, recall)

static get_roc_curve(*tensor*, *labels*, *reduce_to_n_samples=None*, *use_sub_process=False*, *results_fn=<function PytorchExperimentLogger.<lambda>>*)

Displays a roc curve given a tensor with scores and the coresponding labels

Parameters

- **tensor** – Tensor with scores (e.g class probability)
- **labels** – Labels of the samples to which the scores match
- **reduce_to_n_samples** – Reduce/ downsample to to n samples for fewer data points
- **use_sub_process** – Use a sub process to do the processing, if true nothing is returned
- **results_fn** – function which is called with the results/ return values. Expected f(tpr, fpr)

get_save_checkpoint_fn(*name='checkpoint'*, ***kwargs*)

A function which returns a function which takes n_iter as arguments and saves the current values of the variables given as kwargs as a checkpoint file.

Parameters

- **name** – Base-name of the checkpoint file
- ****kwargs** – dict which is actually saved, when the returned function is called

Returns: Function which takes n_iter as arguments and saves a checkpoint file

load_checkpoint(*name*, *exclude_layer_dict=None*, *warnings=True*, ***kwargs*)

Loads a checkpoint from the checkpoint directory of the experiment folder

Parameters

- **name** – The name of the checkpoint file
- **exclude_layer_dict** – A dict with key 'model_name' and a list of all layers of 'model_name' which should
- **be restored** (*not*) –
- **warnings** – Flag which indicates if method should warn if not everything went perfectlys
- ****kwargs** – dict which is actually loaded (key=name (used to save the checkpoint) , value=variable to be
- **overwritten)** (*loaded/*) –

Returns: The kwargs dict with the loaded/ overwritten values

static load_checkpoint_static(*checkpoint_file*, *exclude_layer_dict=None*, *warnings=True*, ***kwargs*)

Loads a checkpoint/dict in a given directory (using pytorch)

Parameters

- **checkpoint_file** – The checkpoint from which the checkpoint/dict should be loaded
- **exclude_layer_dict** – A dict with key 'model_name' and a list of all layers of 'model_name' which should
- **be restored** (*not*) –
- **warnings** – Flag which indicates if method should warn if not everything went perfectlys
- ****kwargs** – dict which is actually loaded (key=name (used to save the checkpoint) , value=variable to be
- **overwritten)** (*loaded/*) –

Returns: The kwargs dict with the loaded/ overwritten values

load_last_checkpoint(***kwargs*)

Loads the (alphabetically) last checkpoint file in the checkpoint directory in the experiment folder

Parameters

- ****kwargs** – dict which is actually loaded (key=name (used to save the checkpoint) , value=variable to be
- **overwritten)** (*loaded/*) –

Returns: The kwargs dict with the loaded/ overwritten values

static load_last_checkpoint_static(*dir_*, *name=None*, ***kwargs*)

Loads the (alphabetically) last checkpoint file in a given directory

Parameters

- **dir** – The directory to look for the (alphabetically) last checkpoint
- **name** – String pattern which indicates the files to look form
- ****kwargs** – dict which is actually loaded (key=name (used to save the checkpoint) , value=variable to be
- **overwritten)** (*loaded/*) –

Returns: The kwargs dict with the loaded/ overwritten values

load_model(*model*, *name*, *exclude_layers=()*, *warnings=True*)

Loads a pytorch model from the model directory of the experiment folder

Parameters

- **model** – The model to be loaded (whose parameters should be restored)
- **name** – The file name of the model file
- **exclude_layers** – List of layer names which should be excluded from restoring
- **warnings** – Flag which indicates if method should warn if not everything went perfectlys

static load_model_static(**args*, ***kwargs*)

print(**args*)
Prints the given arguments using the text logger print function

Parameters ***args** – Things to be printed

save_at_exit(*name='checkpoint_end'*, ***kwargs*)
Saves a dict as checkpoint if the program exits (not garanteed to work 100%)

Parameters

- **name** – Name of the checkpoint file
- ****kwargs** – dict which is actually saved (key=name, value=variable to be stored)

save_checkpoint(*name*, *n_iter=None*, *iter_format='{:05d}'*, *prefix=False*, ***kwargs*)
Saves a checkpoint in the checkpoint directory of the experiment folder

Parameters

- **name** – The file name of the checkpoint file
- **n_iter** – The iteration number, formatted with the iter_format and added to the checkpoint name (if not None)
- **iter_format** – The format string, which indicates how n_iter will be formated as a string
- **prefix** – If True, the formated n_iter will be appended as a prefix, otherwise as a suffix
- ****kwargs** – dict which is actually saved (key=name, value=variable to be stored)

static save_checkpoint_static(**args*, ***kwargs*)

save_model(*model*, *name*, *n_iter=None*, *iter_format='{:05d}'*, *prefix=False*)
Saves a pytorch model in the model directory of the experiment folder

Parameters

- **model** – The model to be stored
- **name** – The file name of the model file
- **n_iter** – The iteration number, formatted with the iter_format and added to the model name (if not None)
- **iter_format** – The format string, which indicates how n_iter will be formated as a string
- **prefix** – If True, the formated n_iter will be appended as a prefix, otherwise as a suffix

static save_model_static(**args*, ***kwargs*)

show_gif(*frame_list=None*, *name='frames'*, *scale=1.0*, *fps=25*)
Saves gif in the img folder. Should be a list of arrays with dimension HxWxC.

Parameters

- **frame_list** – The list of image tensors/arrays to be saved as a gif
- **name** – Filename of the gif
- **scale** – Scaling factor of the individual frames
- **fps** – FPS of the gif

show_image_gradient(*name*, **args*, ***kwargs*)
Given a model creates calculates the error and backpropagates it to the image and saves it.

Parameters

- **name** – Name of the file
- **model** – The model to be evaluated
- **inpt** – Input to the model
- **err_fn** – The error function the evaluate the output of the model on
- **grad_type** – Gradient calculation method, currently supports (vanilla, vanilla-smooth, guided,
- **guided-smooth)** (*the guided backprob can lead to segfaults -. -*) –
- **n_runs** – Number of runs for the smooth variants
- **eps** – noise scaling to be applied on the input image (noise is drawn from N(0,1))
- **abs** (*bool*) – Flag, if the gradient should be a absolute value

show_image_grid(*image*, *name*, ***kwargs*)
Saves images in the img folder as a image grid

Parameters

- **images** – The images to be saved
- **name** – file name of the new image file

show_image_grid_heatmap(*heatmap*, *background=None*, *name='heatmap'*, ***kwargs*)
Saves images in the img folder as a image grid

Parameters

- **heatmap** – The images to be converted to a heatmap
- **background** – Context of the heatmap (to be underlayed)
- **name** – file name of the new image file

show_images(*images*, *name*, ***kwargs*)
Saves images in the img folder

Parameters

- **images** – The images to be saved
- **name** – file name of the new image file

show_video(*frame_list=None*, *name='video'*, *dim='LxHxWxC'*, *scale=1.0*, *fps=25*, *extension='.mp4'*, *codec='THEO'*)
Saves video in the img folder. Should be a list of arrays with dimension HxWxC.

Parameters

- **frame_list** – The list of image tensors/arrays to be saved as a video

- **name** – Filename of the video
- **dim** – Dimension of the tensor - should be either LxHxWxC or LxCxHxW
- **fps** – FPS of the video
- **extension** – File extension - should be mp4, ogc, avi or webm

6.2 file

6.2.1 NumpyPlotFileLogger

class trixi.logger.file.numpyplotfilelogger.**NumpyPlotFileLogger**(*img_dir*, *plot_dir*, *switch_backend=True*, ***kwargs*)

Bases: *trixi.logger.plt.numpyseabornplotlogger.NumpySeabornPlotLogger*

NumpyPlotFileLogger is a logger, which can plot/ interpret numpy array as different types (images, lineplots, ...) into an image and plot directory. For the plotting it builds up on the NumpySeabornPlotLogger.

show_barplot(*array*, *name*, *file_format='.png'*, **args*, ***kwargs*)
Method which creates and stores a barplot

Parameters

- **array** – Array of values you want to plot
- **name** – file-name
- **file_format** – output-image (plot) file format

show_boxplot(*array*, *name*, *file_format='.png'*, **args*, ***kwargs*)
Method which creates and stores a boxplot

Parameters

- **array** – Array of values you want to plot
- **name** – file-name
- **file_format** – output-image (plot) file format

show_image(*image*, *name*, *file_format='.png'*, **args*, ***kwargs*)
Method which stores an image as a image file

Parameters

- **image** – Numpy array-image
- **name** – file-name
- **file_format** – output-image file format

show_lineplot(*y_vals*, *x_vals*, *name*, *file_format='.png'*, **args*, ***kwargs*)
Method which creates and stores a lineplot

Parameters

- **y_vals** – Array of y values
- **x_vals** – Array of corresponding x-values
- **name** – file-name

- **file_format** – output-image (plot) file format

show_matplot_plt(*figure*, *name*, *file_format='.png'*, **args*, ***kwargs*)
Method to save a custom matplotlib figure

Parameters

- **figure** – Figure you want to plot
- **name** – file name
- **file_format** – output image (plot) file format

show_piechart(*array*, *name*, *file_format='.png'*, **args*, ***kwargs*)
Method which creates and stores a piechart

Parameters

- **array** – Array of values you want to plot
- **name** – file-name
- **file_format** – output-image (plot) file format

show_scatterplot(*array*, *name*, *file_format='.png'*, **args*, ***kwargs*)
Method which creates and stores a scatter

Parameters

- **array** – Array of values you want to plot
- **name** – file-name
- **file_format** – output-image (plot) file format

show_value(*value*, *name*, *counter=None*, *tag=None*, *file_format='.png'*, **args*, ***kwargs*)
Method which logs a value as a line plot

Parameters

- **value** – Value (y-axis value) you want to display/ plot/ store
- **name** – Name of the value (will also be the filename if no tag is given)
- **counter** – counter, which tells the number of the sample (with the same name –> file-name) (x-axis value)
- **tag** – Tag, grouping similar values. Values with the same tag will be plotted in the same plot
- **file_format** – output-image file format

Returns:

trixi.logger.file.numpyplotfilelogger.**threaded**(*func*)

6.2.2 PytorchPlotFileLogger

class trixi.logger.file.pytorchplotfilelogger.**PytorchPlotFileLogger**(**args*, ***kwargs*)

Bases: *trixi.logger.file.numpyplotfilelogger.NumpyPlotFileLogger*

Visual logger, inherits the NumpyPlotLogger and plots/ logs pytorch tensors and variables as files on the local file system.

process_params(*f*, **args*, ***kwargs*)
Inherited "decorator": convert Pytorch variables and Tensors to numpy arrays

save_image(*tensor*, *name*, *n_iter=None*, *iter_format='{:05d}'*, *prefix=False*, *image_args=None*)
Saves an image into the image directory of the PytorchPlotFileLogger

Parameters

- **tensor** – Tensor containing the image
- **name** – file-name of the image file
- **n_iter** – The iteration number, formatted with the iter_format and added to the model name (if not None)
- **iter_format** – The format string, which indicates how n_iter will be formated as a string
- **prefix** – If True, the formated n_iter will be appended as a prefix, otherwise as a suffix
- **image_args** – Arguments for the tensorvision save image method

save_image_grid(*tensor*, *name*, *n_iter=None*, *prefix=False*, *iter_format='{:05d}'*, *image_args=None*)
Saves images of a 4d- tensor (N, C, H, W) as a image grid into an image file in the image directory of the PytorchPlotFileLogger

Parameters

- **tensor** – 4d- tensor (N, C, H, W)
- **name** – file-name of the image file
- **n_iter** – The iteration number, formatted with the iter_format and added to the model name (if not None)
- **iter_format** – The format string, which indicates how n_iter will be formated as a string
- **prefix** – If True, the formated n_iter will be appended as a prefix, otherwise as a suffix
- **image_args** – Arguments for the tensorvision save image method

static save_image_grid_static(**args*, ***kwargs*)

static save_image_static(**args*, ***kwargs*)

save_images(*tensors*, *n_iter=None*, *iter_format='{:05d}'*, *prefix=False*, *image_args=None*)
Saves an image tensors into the image directory of the PytorchPlotFileLogger

Parameters

- **tensors** – A dict with file-name-> tensor to plot as image
- **n_iter** – The iteration number, formatted with the iter_format and added to the model name (if not None)
- **iter_format** – The format string, which indicates how n_iter will be formated as a string
- **prefix** – If True, the formated n_iter will be appended as a prefix, otherwise as a suffix
- **image_args** – Arguments for the tensorvision save image method

static save_images_static(**args*, ***kwargs*)

show_image(*image*, *name*, *n_iter=None*, *iter_format='{:05d}'*, *prefix=False*, *image_args=None*, ***kwargs*)

Calls the save image method (for abstract logger combatibility)

Parameters

- **image** – Tensor containing the image
- **name** – file-name of the image file
- **n_iter** – The iteration number, formatted with the iter_format and added to the model name (if not None)
- **iter_format** – The format string, which indicates how n_iter will be formated as a string
- **prefix** – If True, the formated n_iter will be appended as a prefix, otherwise as a suffix
- **image_args** – Arguments for the tensorvision save image method

show_image_grid(*images*, *name*, *n_iter=None*, *prefix=False*, *iter_format='{:05d}'*, *image_args=None*, ***kwargs*)

Calls the save image grid method (for abstract logger combatibility)

Parameters

- **images** – 4d- tensor (N, C, H, W)
- **name** – file-name of the image file
- **n_iter** – The iteration number, formatted with the iter_format and added to the model name (if not None)
- **iter_format** – The format string, which indicates how n_iter will be formated as a string
- **prefix** – If True, the formated n_iter will be appended as a prefix, otherwise as a suffix
- **image_args** – Arguments for the tensorvision save image method

show_image_grid_heatmap(*heatmap*, *background=None*, *ratio=0.3*, *normalize=True*, *colormap=<sphinx.ext.autodoc.importer._MockObject object>*, *name='heatmap'*, *n_iter=None*, *prefix=False*, *iter_format='{:05d}'*, *image_args=None*, ***kwargs*)

Creates heat map from the given map and if given combines it with the background and then displays results with as image grid.

Parameters

- **heatmap** – 4d- tensor (N, C, H, W) to be converted to a heatmap
- **background** – 4d- tensor (N, C, H, W) background/ context of the heatmap (to be underlayed)
- **name** – The name of the window
- **ratio** – The ratio to mix the map with the background (0 = only background, 1 = only map)
- **n_iter** – The iteration number, formatted with the iter_format and added to the model name (if not None)
- **iter_format** – The format string, which indicates how n_iter will be formated as a string
- **prefix** – If True, the formated n_iter will be appended as a prefix, otherwise as a suffix

- **image_args** – Arguments for the tensorvision save image method

show_images(*images*, *name*, *n_iter=None*, *iter_format='{:05d}'*, *prefix=False*, *image_args=None*, ***kwargs*)

Calls the save images method (for abstract logger combatibility)

Parameters

- **images** – List of Tensors
- **name** – List of file names (corresponding to the images list)
- **n_iter** – The iteration number, formatted with the iter_format and added to the model name (if not None)
- **iter_format** – The format string, which indicates how n_iter will be formated as a string
- **prefix** – If True, the formated n_iter will be appended as a prefix, otherwise as a suffix
- **image_args** – Arguments for the tensorvision save image method

6.2.3 TextFileLogger

class trixi.logger.file.textfilelogger.**TextFileLogger**(*base_dir=None*, *logging_level=10*, *logging_stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *default_stream_handler=True*, ***kwargs*)

Bases: *trixi.logger.abstractlogger.AbstractLogger*

A Logger for logging text into different text files and output streams (using the python logging framework)

add_file_handler(*name*, *logger='default'*)

Adds a file handler to a logger, thus the logger will log into a log file with a given name

Parameters

- **name** – File name of the log file (in which the logger will log now)
- **logger** – Name of the logger to add the file-hander/ logging file to

add_handler(*handler*, *logger='default'*)

Adds an additional handler to a logger with a name :param handler: Logging handler to be added to a given logger :param logger: Name of the logger to add the hander to

add_logger(*name*, *logging_level=None*, *file_handler=True*, *stream_handler=True*)

Adds a new logger

Parameters

- **name** – Name of the new logger
- **logging_level** – Logging level of the new logger
- **file_handler** – Flag, if it should use a file_handler, if yes creates a new file with the given name in the logging directory
- **stream_handler** – Flag, if the logger should also log to the default stream

Returns:

add_stream_handler(*logger='default'*)
Adds a stream handler to a logger, thus the logger will log into the default logging stream

Parameters **logger** – Name of the logger to add the stream-hander to

debug(*msg*, *logger='default'*)
Prints and logs a message with the level debug

Parameters

- **msg** – Message to print/ log
- **logger** – Logger which should log

error(*msg*, *logger='default'*)
Prints and logs a message with the level error

Parameters

- **msg** – Message to print/ log
- **logger** – Logger which should log

info(*msg*, *logger='default'*)
Prints and logs a message with the level info

Parameters

- **msg** – Message to print/ log
- **logger** – Logger which should log

log(*msg*, *logger='default'*)
Prints and logs a message with the level info

Parameters

- **msg** – Message to print/ log
- **logger** – Logger which should log

log_to(*msg*, *name*, *log_to_default=False*)
Logs to an existing logger or if logger does not exists creates new one

Parameters

- **msg** – Message to be logged
- **name** – Name of the logger to log to (usually also the logfile-name)
- **log_to_default** – Flag if it should in addition to the logger given by name, log to the default logger

print(**args*, *logger='default'*)
Prints and logs objects

Parameters

- ***args** – Object to print/ log
- **logger** – Logger which should log

show_text(*text*, *name=None*, *logger='default'*, ***kwargs*)
Logs a text. Calls the log function (for combatibility reasons with AbstractLogger)

Parameters

- **text** – Text to be logged

- **name** – Some identifier for the text (will be added infront of the text)
- **logger** – Name of the Logger to log to

show_value(*value*, *name=None*, *logger='default'*, ***kwargs*)
Logs a Value. Calls the log function (for combatibility reasons with AbstractLogger)

Parameters

- **value** – Value to be logged
- **name** – Some identifier for the text (will be added infront of the text)
- **logger** – Name of the Logger to log to

6.3 message

6.3.1 SlackMessageLogger

class trixi.logger.message.slackmessagelogger.**SlackMessageLogger**(*token*, *user_email*, *exp_name=None*, **args*, ***kwargs*)
Bases: *trixi.logger.plt.numpyseabornimageplotlogger.NumpySeabornImagePlotLogger*

Slack logger, inherits the NumpySeabornImagePlotLogger and sends plots/logs to a chat via a Slack bot.

delete_message(*ts*)
Deletes a direct message from the bot with the given timestamp (ts)

Parameters **ts** – Time stamp the message was sent

static find_cid_for_user(*slack_client*, *uid*)
Returns the chat/channel id for a direct message of the bot with the given user

Parameters

- **slack_client** – Slack client (already authorized)
- **uid** – User id of the user

Returns chat/channel id for a direct message

static find_uid_for_email(*slack_client*, *email*)
Returns the slack user id for a given email

Parameters

- **slack_client** – Slack client (already authorized)
- **email** – Workspace email address to get the user id for

Returns Slack workspace user id

print(*text*, **args*, ***kwargs*)
Just calls *show_text()*

process_params(*f*, **args*, ***kwargs*)
Inherited "decorator": convert PyTorch variables and Tensors to numpy arrays

send_message(*message=''*, *file=None*)
Sends a message and a file if one is given

Parameters

- **message** – Message to be sent
- **file** – File to be sent

Returns The timestamp (ts) of the message

show_barplot(*array*, *name='barplot'*, *delete_last=True*, **args*, ***kwargs*)
Sends a barplot to a chat using an existing slack bot.

Parameters

- **array** – array of shape NxM where N is the number of rows and M is the number of elements in the row.
- **name** – The name of the figure
- **delete_last** – If a message with the same name was sent, delete it beforehand

show_image(*image*, **args*, ***kwargs*)
Sends an image file to a chat using an existing slack bot.

Parameters **image** (*str or np array*) – Path to the image file to be sent to the chat.

show_image_grid(*image_array*, *name=None*, *nrow=8*, *padding=2*, *normalize=False*, *range=None*, *scale_each=False*, *pad_value=0*, *delete_last=True*, **args*, ***kwargs*)
Sends an array of images to a chat using an existing Slack bot. (Requires torch and torchvision)

Parameters

- **image_array** (*np.narray / torch.tensor*) – Image array/tensor which will be sent as an image grid
- **make_grid_kargs** – Key word arguments for the torchvision make grid method
- **delete_last** – If a message with the same name was sent, delete it beforehand

show_lineplot(*y_vals*, *x_vals=None*, *name='lineplot'*, *delete_last=True*, **args*, ***kwargs*)
Sends a lineplot to a chat using an existing slack bot.

Parameters

- **y_vals** – Array of shape MxN , where M is the number of points and N is the number of different line
- **x_vals** – Has to have the same shape as Y: MxN. For each point in Y it gives the corresponding X value (if not set the points are assumed to be equally distributed in the interval [0, 1])
- **name** – The name of the figure
- **delete_last** – If a message with the same name was sent, delete it beforehand

show_piechart(*array*, *name='piechart'*, *delete_last=True*, **args*, ***kwargs*)
Sends a piechart to a chat using an existing slack bot.

Parameters

- **array** – Array of positive integers. Each integer will be presented as a part of the pie (with the total as the sum of all integers)
- **name** – The name of the figure
- **delete_last** – If a message with the same name was sent, delete it beforehand

show_scatterplot(*array*, *name='scatterplot'*, *delete_last=True*, **args*, ***kwargs*)
Sends a scatterplot to a chat using an existing slack bot.

Parameters

- **array** – An array with size N x dim, where each element i in N at X[i] results in a 2D (if dim = 2) or 3D (if dim = 3) point.
- **name** – The name of the figure
- **delete_last** – If a message with the same name was sent, delete it beforehand

show_text(*text*, **args*, ***kwargs*)
Sends a text to a chat using an existing slack bot.

Parameters **text** (*str*) – Text message to be sent to the bot.

show_value(*value*, *name*, *counter=None*, *tag=None*, *delete_last=True*, **args*, ***kwargs*)
Sends a value to a chat using an existing slack bot.

Parameters

- **value** – Value to be plotted sent to the chat.
- **name** – Name for the plot.
- **counter** – Optional counter to be sent in conjunction with the value.
- **tag** – Tag to be used as a label for the plot.
- **delete_last** – If a message with the same name was sent, delete it beforehand

6.3.2 TelegramMessageLogger

class trixi.logger.message.telegrammessagelogger.**TelegramMessageLogger**(*token*, *chat_id*, *exp_name=None*, **args*, ***kwargs*)
Bases: *trixi.logger.plt.numpyseabornimageplotlogger.NumpySeabornImagePlotLogger*

Telegram logger, inherits the NumpySeabornImagePlotLogger and sends plots/logs to a chat via a Telegram bot.

print(*text*, **args*, ***kwargs*)
Just calls show_text()

process_params(*f*, **args*, ***kwargs*)
Inherited "decorator": convert PyTorch variables and Tensors to numpy arrays

show_barplot(*array*, *name=None*, **args*, ***kwargs*)
Sends a barplot to a chat using an existing Telegram bot.

Parameters

- **array** – array of shape NxM where N is the number of rows and M is the number of elements in the row.
- **name** – The name of the figure

show_image(*image*, **args*, ***kwargs*)
Sends an image file to a chat using an existing Telegram bot.

Parameters **image** (*str or np array*) – Path to the image file to be sent to the chat.

show_image_grid(*image_array*, *name=None*, *nrow=8*, *padding=2*, *normalize=False*, *range=None*, *scale_each=False*, *pad_value=0*, **args*, ***kwargs*)
Sends an array of images to a chat using an existing Telegram bot. (Requires torch and torchvision)

Parameters

- **image_array** (*np.narray / torch.tensor*) – Image array/ tensor which will be sent as an image grid
- **make_grid_kargs** – Key word arguments for the torchvision make grid method

show_lineplot(*y_vals*, *x_vals=None*, *name=None*, **args*, ***kwargs*)
Sends a lineplot to a chat using an existing Telegram bot.

Parameters

- **y_vals** – Array of shape MxN , where M is the number of points and N is the number of different line
- **x_vals** – Has to have the same shape as Y: MxN. For each point in Y it gives the corresponding X value (if
- **set the points are assumed to be equally distributed in the interval [0, 1])** (*not*) –
- **name** – The name of the figure

show_piechart(*array*, *name=None*, **args*, ***kwargs*)
Sends a piechart to a chat using an existing Telegram bot.

Parameters

- **array** – Array of positive integers. Each integer will be presented as a part of the pie (with the total
- **the sum of all integers)** (*as*) –
- **name** – The name of the figure

show_scatterplot(*array*, *name=None*, **args*, ***kwargs*)
Sends a scatterplot to a chat using an existing Telegram bot.

Parameters

- **array** – A 2d array with size N x dim, where each element i in N at X[i] results in a a 2d (if dim = 2)/
- **3d**(*if dim = 3*) –
- **name** – The name of the figure

show_text(*text*, **args*, ***kwargs*)
Sends a text to a chat using an existing Telegram bot.

Parameters **text** (*str*) – Text message to be sent to the bot.

show_value(*value*, *name*, *counter=None*, *tag=None*, **args*, ***kwargs*)
Sends a value to a chat using an existing Telegram bot.

Parameters

- **value** – Value to be plotted sent to the chat.
- **name** – Name for the plot.
- **counter** – Optional counter to be sent in conjunction with the value.
- **tag** – Tag to be used as a label for the plot.

6.4 plt

6.4.1 NumpySeabornPlotLogger

class trixi.logger.plt.numpyseabornplotlogger.**NumpySeabornPlotLogger**(***kwargs*)
Bases: *trixi.logger.abstractlogger.AbstractLogger*

Visual logger, inherits the AbstractLogger and plots/ logs numpy arrays/ values as matplotlib / seaborn plots.

get_figure(*name*)
Returns a figure with a given name as identifier.

If no figure yet exists with the name a new one is created. Otherwise the existing one is returned

Parameters **name** – Name of the figure

Returns A figure with the given name

show_barplot(*array*, *name=None*, *show=True*, **args*, ***kwargs*)
Creates a bar plot figure from an array

Parameters

- **array** – array of shape NxM where N is the number of rows and M is the number of elements in the row.
- **name** – The name of the figure
- **show** – Flag if it should also display the figure (result might also depend on the matplotlib backend)

Returns A matplotlib figure

show_boxplot(*array*, *name*, *show=True*, **args*, ***kwargs*)
Creates a box plot figure from an array

Parameters

- **array** – array of shape NxM where N is the number of rows and M is the number of elements in the row.
- **name** – The name of the figure
- **show** – Flag if it should also display the figure (result might also depend on the matplotlib backend)

Returns A matplotlib figure

show_image(*image*, *name=None*, *show=True*, **args*, ***kwargs*)
Create an image figure

Parameters

- **image** – The image array to be displayed
- **name** – The name of the image window
- **show** – Flag if it should also display the figure (result might also depend on the matplotlib backend)

Returns A matplotlib figure

show_lineplot(*y_vals*, *x_vals=None*, *name=None*, *show=True*, **args*, ***kwargs*)
Creates a line plot figure with (multiple) lines plot, given values Y (and optional the corresponding X values)

Parameters

- **y_vals** – Array of shape MxN , where M is the number of points and N is the number of different line
- **x_vals** – Has to have the same shape as Y: MxN. For each point in Y it gives the corresponding X value (if
- **set the points are assumed to be equally distributed in the interval [0, 1])** (*not*) –
- **name** – The name of the figure
- **show** – Flag if it should also display the figure (result might also depend on the matplotlib backend)

Returns A matplotlib figure

show_piechart(*array*, *name=None*, *show=True*, **args*, ***kwargs*)
Creates a scatter plot figure

Parameters

- **array** – Array of positive integers. Each integer will be presented as a part of the pie (with the total
- **the sum of all integers)** (*as*) –
- **name** – The name of the figure
- **show** – Flag if it should also display the figure (result might also depend on the matplotlib backend)

Returns A matplotlib figure

show_scatterplot(*array*, *name=None*, *show=True*, **args*, ***kwargs*)
Creates a scatter plot figure with the points given in array

Parameters

- **array** – A 2d array with size N x dim, where each element i in N at X[i] results in a a 2d (if dim = 2)/
- **3d** (*if dim = 3*) –
- **name** – The name of the figure
- **show** – Flag if it should also display the figure (result might also depend on the matplotlib backend)

Returns A matplotlib figure

show_value(*value*, *name*, *counter=None*, *tag=None*, *show=True*, **args*, ***kwargs*)
Creates a line plot that is automatically appended with new values and returns it as a figure.

Parameters

- **value** – Value to be plotted / appended to the graph (y-axis value)
- **name** – The name of the window
- **counter** – counter, which tells the number of the sample (with the same name) (x-axis value)
- **tag** – Tag, grouping similar values. Values with the same tag will be plotted in the same plot

- **show** – Flag if it should also display the figure (result might also depend on the matplotlib backend)

  Returns: A matplotlib figure

6.4.2 NumpySeabornImagePlotLogger

class trixi.logger.plt.numpyseabornimageplotlogger.**NumpySeabornImagePlotLogger**(***kwargs*)

Bases: *trixi.logger.plt.numpyseabornplotlogger.NumpySeabornPlotLogger*

Wrapper around *NumpySeabornPlotLogger* that renders figures into numpy arrays.

show_barplot(*array*, *name=None*, **args*, ***kwargs*)

Creates a bar plot figure from an array

Parameters

- **array** – array of shape NxM where N is the number of rows and M is the number of elements in the row.
- **name** – The name of the figure

Returns A numpy array image of the figure

show_image(*image*, *name=None*, **args*, ***kwargs*)

Create an image figure

Parameters

- **image** – The image array to be displayed
- **name** – The name of the image window

Returns A numpy array image of the figure

show_lineplot(*y_vals*, *x_vals=None*, *name=None*, **args*, ***kwargs*)

Creates a line plot figure with (multiple) lines plot, given values Y (and optional the corresponding X values)

Parameters

- **y_vals** – Array of shape MxN , where M is the number of points and N is the number of different line
- **x_vals** – Has to have the same shape as Y: MxN. For each point in Y it gives the corresponding X value (if
- **set the points are assumed to be equally distributed in the interval [0, 1])** (*not*) –
- **name** – The name of the figure

Returns A numpy array image of the figure

show_piechart(*array*, *name=None*, **args*, ***kwargs*)

Creates a scatter plot figure

Parameters

- **array** – Array of positive integers. Each integer will be presented as a part of the pie (with the total
- **the sum of all integers)** (*as*) –
- **name** – The name of the figure

Returns A numpy array image of the figure

show_scatterplot(*array*, *name=None*, **args*, ***kwargs*)
Creates a scatter plot figure with the points given in array

Parameters

- **array** – A 2d array with size N x dim, where each element i in N at X[i] results in a a 2d (if dim = 2)/
- **3d**(*if dim = 3*) –
- **name** – The name of the figure

Returns: A numpy array image of the figure

show_value(*value*, *name*, *counter=None*, *tag=None*, **args*, ***kwargs*)
Creates a line plot that is automatically appended with new values and returns it as a figure.

Parameters

- **value** – Value to be plotted / appended to the graph (y-axis value)
- **name** – The name of the window
- **counter** – counter, which tells the number of the sample (with the same name) (x-axis value)
- **tag** – Tag, grouping similar values. Values with the same tag will be plotted in the same plot

Returns: A numpy array image of the figure

6.5 tensorboard

6.5.1 TensorboardLogger

6.5.2 PytorchTensorboardLogger

6.6 visdom

6.6.1 NumpyVisdomLogger

class trixi.logger.visdom.numpyvisdomlogger.**NumpyVisdomLogger**(*exp_name='main'*, *server='http://localhost'*, *port=8080*, *auto_close=True*, *auto_start=False*, *auto_start_ports=(8080, 8000)*, ***kwargs*)

Bases: *trixi.logger.abstractlogger.AbstractLogger*

Visual logger, inherits the AbstractLogger and plots/ logs numpy arrays/ values on a Visdom server.

add_to_graph(**args*, ***kwargs*)

close_all()
Closes all visdom windows.

exit()
Kills the internal process.

save_vis()

send_data(**args*, ***kwargs*)

show_barplot(**args*, ***kwargs*)

show_boxplot(**args*, ***kwargs*)

show_contourplot(**args*, ***kwargs*)

show_histogram(**args*, ***kwargs*)

show_image(**args*, ***kwargs*)

show_images(**args*, ***kwargs*)

show_lineplot(**args*, ***kwargs*)

show_matplot_plt(**args*, ***kwargs*)

show_piechart(**args*, ***kwargs*)

show_plotly_plt(**args*, ***kwargs*)

show_progress(**args*, ***kwargs*)

show_scatterplot(**args*, ***kwargs*)

show_surfaceplot(**args*, ***kwargs*)

show_svg(**args*, ***kwargs*)

show_text(**args*, ***kwargs*)

show_value(**args*, ***kwargs*)

show_values(*val_dict*)
A util function for multiple values. Simple plots all values in a dict, there the window name is the key in the dict and the plotted value is the value of a dict (Simply calls the show_value function).

Parameters **val_dict** – Dict with key, values pairs which will be plotted

trixi.logger.visdom.numpyvisdomlogger.**add_to_queue**(*func*)

trixi.logger.visdom.numpyvisdomlogger.**start_visdom**(*port_list=(8080, 8000)*)
Starts a visdom server on a given port

Parameters **port_list** – Priority list of port. Will start a visdom server on the first available port.

6.6.2 PytorchVisdomLogger

class trixi.logger.visdom.pytorchvisdomlogger.**PytorchVisdomLogger**(**args*, ***kwargs*)
Bases: *trixi.logger.visdom.numpyvisdomlogger.NumpyVisdomLogger*

Visual logger, inherits the NumpyVisdomLogger and plots/ logs pytorch tensors and variables on a Visdom server.

plot_model_gradient_flow(*model*, *name='model'*, *title=None*)
Plots statstics (mean, std, abs(max)) of the weights or the corresponding gradients of a model as a barplot.

Parameters

- **model** – Model with the weights.

- **env_appendix** – Visdom environment name appendix, if none is given, it uses "-histogram".
- **model_name** – Name of the model (is used as window name).
- **plot_grad** – If false plots weight statistics, if true plot the gradients of the weights.

plot_model_statistics(***kwargs*)

plot_model_statistics_grads(*model*, *env_appendix=''*, *model_name=''*, ***kwargs*)
Plots statstics (mean, std, abs(max)) of the gradients of a model as a barplot (uses plot model statistics with plot_grad=True).

Parameters

- **model** – Model with the weights and the corresponding gradients (have to calculated previously).
- **env_appendix** – Visdom environment name appendix
- **model_name** – Name of the model (is used as window name).

plot_model_statistics_weights(*model*, *env_appendix=''*, *model_name=''*, ***kwargs*)
Plots statstics (mean, std, abs(max)) of the weights of a model as a barplot (uses plot model statistics with plot_grad=False).

Parameters

- **model** – Model with the weights.
- **env_appendix** – Visdom environment name appendix
- **model_name** – Name of the model (is used as window name).

plot_model_structure(*model*, *input_size*, *name='model_structure'*, *use_cuda=True*, *delete_tmp_on_close=False*, *forward_kwargs=None*, ***kwargs*)
Plots the model structure/ model graph of a pytorch module (this only works correctly with pytorch 0.2.0).

Parameters

- **model** – The graph of this model will be plotted.
- **input_size** – Input size of the model (with batch dim).
- **name** – The name of the window in the visdom env.
- **use_cuda** – Perform model dimension calculations on the gpu (cuda).
- **delete_tmp_on_close** – Determines if the tmp file will be deleted on close. If set true, can cause problems due to the multi threadded plotting.

plot_mutliple_models_statistics_grads(*model_dict*, *env_appendix=''*, ***kwargs*)
Given models in a dict, plots the gradient statistics of the models.

Parameters

- **model_dict** – Dict with models, the key is assumed to be the name, while the value is the model.
- **env_appendix** – Visdom environment name appendix

plot_mutliple_models_statistics_weights(*model_dict*, *env_appendix=None*, ***kwargs*)
Given models in a dict, plots the weight statistics of the models.

Parameters

- **model_dict** – Dict with models, the key is assumed to be the name, while the value is the model.

- **env_appendix** – Visdom environment name appendix

process_params(*f*, **args*, ***kwargs*)
Inherited "decorator": convert Pytorch variables and Tensors to numpy arrays.

show_classification_metrics(*tensor*, *labels*, *name*, *metric=('roc-auc'*, *'pr-score')*, *use_sub_process=False*, *tag_name=None*)
Displays some classification metrics as line plots in a graph (similar to show value (also uses show value for the caluclated values))

Parameters

- **tensor** – Tensor with scores (e.g class probability)
- **labels** – Labels of the samples to which the scores match
- **name** – The name of the window
- **metric** – List of metrics to calculate. Options are: roc-auc, pr-auc, pr-score, mcc, f1

Returns:

show_embedding(*tensor*, *labels=None*, *name=None*, *method='tsne'*, *n_dims=2*, *n_neigh=30*, *meth_args=None*, **args*, ***kwargs*)
Displays a tensor a an embedding

Parameters

- **tensor** – Tensor to be embedded an then displayed
- **labels** – Labels of the entries in the tensor (first dimension)
- **name** – The name of the window
- **method** – Method used for embedding, options are: tsne, standard, ltsa, hessian, modified, isomap, mds,
- **umap** (*spectral,*) –
- **n_dims** – dimensions to embed the data into
- **n_neigh** – Neighbour parameter to kind of determin the embedding (see t-SNE for more information)
- **meth_args** – Further arguments which can be passed to the embedding method

show_image_gradient(*model*, *inpt*, *err_fn*, *grad_type='vanilla'*, *n_runs=20*, *eps=0.1*, *abs=False*, ***image_grid_params*)
Given a model creates calculates the error and backpropagates it to the image and saves it (saliency map).

Parameters

- **model** – The model to be evaluated
- **inpt** – Input to the model
- **err_fn** – The error function the evaluate the output of the model on
- **grad_type** – Gradient calculation method, currently supports (vanilla, vanilla-smooth, guided,
- **guided-smooth)** (*the guided backprob can lead to segfaults -. -*) –
- **n_runs** – Number of runs for the smooth variants
- **eps** – noise scaling to be applied on the input image (noise is drawn from N(0,1))

- **abs** (*bool*) – Flag, if the gradient should be a absolute value
- ****image_grid_params** – Params for make image grid.

show_image_grid(***kwargs*)

show_image_grid_heatmap(**args*, ***kwargs*)

show_pr_curve(*tensor*, *labels*, *name*, *reduce_to_n_samples=None*, *use_sub_process=False*)
Displays a precision recall curve given a tensor with scores and the coresponding labels

Parameters

- **tensor** – Tensor with scores (e.g class probability)
- **labels** – Labels of the samples to which the scores match
- **name** – The name of the window
- **reduce_to_n_samples** – Reduce/ downsample to to n samples for fewer data points
- **use_sub_process** – Use a sub process to do the processing

show_roc_curve(*tensor*, *labels*, *name*, *reduce_to_n_samples=None*, *use_sub_process=False*)
Displays a roc curve given a tensor with scores and the coresponding labels

Parameters

- **tensor** – Tensor with scores (e.g class probability)
- **labels** – Labels of the samples to which the scores match
- **name** – The name of the window
- **reduce_to_n_samples** – Reduce/ downsample to to n samples for fewer data points
- **use_sub_process** – Use a sub process to do the processing

show_video(*frame_list=None*, *name='frames'*, *dim='LxHxWxC'*, *scale=1.0*, *fps=25*)

trixi.logger.visdom.pytorchvisdomlogger.**move_to_cpu**(*fn*)
Decorator to call the process_params method of the class.

6.7 AbstractLogger

class trixi.logger.abstractlogger.**AbstractLogger**(**args*, ***kwargs*)
Bases: object

Abstract interface for visual logger.

process_params(*f*, **args*, ***kwargs*)
Implement this to handle data conversions in your logger.

Example: Implement logger for numpy data, then implement torch logger as child of numpy logger and just use the process_params method to convert from torch to numpy.

show_barplot(**args*, ***kwargs*)
Abstract method which should handle and somehow log/ store a barplot

show_image(**args*, ***kwargs*)
Abstract method which should handle and somehow log/ store an image

show_lineplot(**args*, ***kwargs*)
Abstract method which should handle and somehow log/ store a lineplot

show_piechart(**args*, ***kwargs*)
Abstract method which should handle and somehow log/ store a piechart

show_scatterplot(**args*, ***kwargs*)
Abstract method which should handle and somehow log/ store a scatterplot

show_text(**args*, ***kwargs*)
Abstract method which should handle and somehow log/ store a text

show_value(**args*, ***kwargs*)
Abstract method which should handle and somehow log/ store a value

trixi.logger.abstractlogger.**convert_params**(*f*)
Decorator to call the process_params method of the class.

trixi.logger.abstractlogger.**threaded**(*f*)
Decorator to run the process in an extra thread.

6.8 CombinedLogger

class trixi.logger.combinedlogger.**CombinedLogger**(**loggers*)
Bases: object

A Logger which can combine all other logger and if called calls all the sub loggers

trixi.logger.combinedlogger.**create_function**(*self*, *sub_methods*)

CHAPTER 7

trixi.util

class trixi.util.util.**CustomJSONDecoder**(*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

Bases: json.decoder.JSONDecoder

decode(*obj*)

Return the Python representation of s (a str instance containing a JSON document).

class trixi.util.util.**CustomJSONEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: json.encoder.JSONEncoder

encode(*obj*)

Return a JSON string representation of a Python data structure.

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

iterencode(*obj*, **args*, ***kwargs*)

Encode the given object and yield each string representation as available.

For example:

```
for chunk in JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

class trixi.util.util.**LogDict**(*file_name*, *base_dir=None*, *to_console=False*, *mode='a'*)

Bases: dict

log_complete_content()

Logs the current content of the dict to the output file as a whole.

class trixi.util.util.**ModuleMultiTypeDecoder**(*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

Bases: *trixi.util.util.MultiTypeDecoder*

class trixi.util.util.**ModuleMultiTypeEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: *trixi.util.util.MultiTypeEncoder*

class trixi.util.util.**MultiTypeDecoder**(*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

Bases: *trixi.util.util.CustomJSONDecoder*

class trixi.util.util.**MultiTypeEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: *trixi.util.util.CustomJSONEncoder*

class trixi.util.util.**PyLock**(*name*, *timeout*, *check_interval=0.25*)

Bases: object

class trixi.util.util.**ResultElement**(*data=None*, *label=None*, *epoch=None*, *counter=None*)

Bases: dict

class trixi.util.util.**ResultLogDict**(*file_name*, *base_dir=None*, *running_mean_length=10*, ***kwargs*)

Bases: *trixi.util.util.LogDict*

close()

load(*reload_dict*)

print_to_file(*text*)

class trixi.util.util.**SafeDict**

Bases: dict

class trixi.util.util.**Singleton**(*decorated*)

Bases: object

A non-thread-safe helper class to ease implementing singletons. This should be used as a decorator – not a metaclass – to the class that should be a singleton.

The decorated class can define one *__init__* function that takes only the *self* argument. Also, the decorated class cannot be inherited from. Other than that, there are no restrictions that apply to the decorated class.

To get the singleton instance, use the *Instance* method. Trying to use *__call__* will result in a *TypeError* being raised.

get_instance(***kwargs*)

Returns the singleton instance. Upon its first call, it creates a new instance of the decorated class and calls its *__init__* method. On all subsequent calls, the already created instance is returned.

class trixi.util.util.**StringMultiTypeDecoder**(*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

Bases: *trixi.util.util.CustomJSONDecoder*

trixi.util.util.**chw_to_hwc**(*np_array*)

trixi.util.util.**create_folder**(*path*)
Creates a folder if not already exists :param : param path: The folder to be created

Returns

return True if folder was newly created, false if folder already exists

trixi.util.util.**figure_to_image**(*figures*, *close=True*)
Render matplotlib figure to numpy format.

Note that this requires the `matplotlib` package. (https://tensorboardx.readthedocs.io/en/latest/_modules/tensorboardX/utils.html#figure_to_image)

Parameters

- **figure** (*matplotlib.pyplot.figure*) – figure or a list of figures
- **close** (*bool*) – Flag to automatically close the figure

Returns image in [CHW] order

Return type numpy.array

trixi.util.util.**get_image_as_buffered_file**(*image_array*)
Returns a images as file pointer in a buffer

Parameters **image_array** – (C,W,H) To be returned as a file pointer

Returns Buffer file-pointer object containing the image file

trixi.util.util.**get_tensor_embedding**(*tensor*, *method='tsne'*, *n_dims=2*, *n_neigh=30*, ***meth_args*)
Return a embedding of a tensor (in a lower dimensional space, e.g. t-SNE)

Parameters

- **tensor** – Tensor to be embedded
- **method** – Method used for embedding, options are: tsne, standard, ltsa, hessian, modified, isomap, mds,
- **umap** (*spectral,*) –
- **n_dims** – dimensions to embed the data into
- **n_neigh** – Neighbour parameter to kind of determin the embedding (see t-SNE for more information)
- ****meth_args** – Further arguments which can be passed to the embedding method

Returns The embedded tensor

trixi.util.util.**is_picklable**(*obj*)

trixi.util.util.**name_and_iter_to_filename**(*name*, *n_iter*, *ending*, *iter_format='{:05d}'*, *prefix=False*)

trixi.util.util.**np_make_grid**(*np_array*, *nrow=8*, *padding=2*, *normalize=False*, *range=None*, *scale_each=False*, *pad_value=0*, *to_int=False*, *standardize=False*)
Make a grid of images.

Parameters

- **np_array** (*numpy array*) – 4D mini-batch Tensor of shape (B x C x H x W) or a list of images all of the same size.

- **nrow** (*int, optional*) – Number of images displayed in each row of the grid. The Final grid size is (B / nrow, nrow). Default is 8.
- **padding** (*int, optional*) – amount of padding. Default is 2.
- **normalize** (*bool, optional*) – If True, shift the image to the range (0, 1), by subtracting the minimum and dividing by the maximum pixel value.
- **range** (*tuple, optional*) – tuple (min, max) where min and max are numbers, then these numbers are used to normalize the image. By default, min and max are computed from the tensor.
- **scale_each** (*bool, optional*) – If True, scale each image in the batch of images separately rather than the (min, max) over all images.
- **pad_value** (*float, optional*) – Value for the padded pixels.
- **to_int** (*bool*) – Transforms the np array to a unit8 array with min 0 and max 255

Example

See this notebook here

trixi.util.util.**random_string**(*length*)

trixi.util.util.**savefig_and_close**(*figure*, *filename*, *close=True*)

7.1 Config

class trixi.util.config.**Config**(*file_=None*, *config=None*, *update_from_argv=False*, *deep=False*, ***kwargs*)

Bases: dict

Config is the main object used to store configurations. As a rule of thumb, anything you might want to change in your experiment should go into the Config. It's basically a dict, but vastly more powerful. Key features are

- **Access keys as attributes** Config["a"]["b"]["c"] is the same as Config.a.b.c. Can also be used for setting if the second to last key exists. Only works for keys that conform with Python syntax (Config.myattr-1 is not allowed).
- **Advanced de-/serialization** Using specialized JSON encoders and decoders, almost anything can be serialized and deserialized. This includes types, functions (except lambdas) and modules. For example, you could have something like:

```
c = Config(model=MyModel)
c.dump("somewhere")
```

and end up with a JSON file that looks like this:

```
{
    "model": "__type__(your.model.module.MyModel)"
}
```

and vice versa. We use double underscores and parentheses for serialization, so it's probably a good idea to not use this pattern for other stuff!

- **Automatic CLI exposure** If desired, the Config will create an ArgumentParser that contains all keys in the Config as arguments in the form "- - key", so you can run your experiment from the command

line and manually overwrite certain values. Deeper levels are also accessible via dot notation "- -key_with_dict_value.inner_key".

- **Comparison** Compare any number of Configs and get a new Config containing only the values that differ among input Configs.

  Parameters

  - **file** (*str*) – Load Config from this file.
  - **config** (Config) – Update with values from this Config (can be combined with file_). Will by default only make shallow copies, see deep.
  - **update_from_argv** (*bool*) – Update values from argv. Will automatically expose keys to the CLI as '- - key'.
  - **deep** (*bool*) – Make deep copies if config is given.

contains(*dict_like*)
Check whether all items in a dictionary-like object match the ones in this Config.

Parameters **dict_like** (*dict or derivative thereof*) – Returns True if this is contained in this Config.

Returns True if dict_like is contained in self, otherwise False.

Return type bool

deepcopy()
Get a deep copy of this Config.

Returns A deep copy of self.

Return type *Config*

deepupdate(*dict_like*, *ignore=None*, *allow_dict_overwrite=True*)
Identical to *update()* with *deep=True*.

Parameters

- **dict_like** (*dict or derivative thereof*) – Update source.
- **ignore** (*iterable*) – Iterable of keys to ignore in update.
- **allow_dict_overwrite** (*bool*) – Allow overwriting with dict. Regular dicts only update on the highest level while we recurse and merge Configs. This flag decides whether it is possible to overwrite a 'regular' value with a dict/Config at lower levels. See examples for an illustration of the difference

difference_config(**other_configs*)
Get the difference of this and any number of other configs. See *difference_config_static()* for more information.

Parameters ***other_configs** (Config) – Compare these configs and self.

Returns Difference of self and the other configs.

Return type *Config*

static difference_config_static(**configs*, *only_set=False*, *encode=True*)
Make a Config of all elements that differ between N configs.

The resulting Config looks like this:

```
{
    key: (config1[key], config2[key], ...)
}
```

If the key is missing, None will be inserted. The inputs will not be modified.

Parameters

- **configs** (*Config*) – Any number of Configs
- **only_set** (*bool*) – If only the set of different values hould be returned or for each config the
- **one** (*corresponding*) –
- **encode** (*bool*) – If True, values will be encoded the same way as they are when exported to disk (e.g."__type__(MyClass)")

Returns Possibly empty Config

Return type *Config*

dump (*file_*, *indent=4*, *separators=(', ', ': ')*, ***kwargs*)
Write config to file using `json.dump()`.

Parameters

- **file** (*str or File*) – Write to this location.
- **indent** (*int*) – Formatting option.
- **separators** (*iterable*) – Formatting option.
- ****kwargs** – Will be passed to `json.dump()`.

dumps (*indent=4*, *separators=(', ', ': ')*, ***kwargs*)
Get string representation using `json.dumps()`.

Parameters

- **indent** (*int*) – Formatting option.
- **separators** (*iterable*) – Formatting option.
- ****kwargs** – Will be passed to `json.dumps()`.

flat (*keep_lists=True*, *max_split_size=10*, *flatten_int=False*)
Returns a flattened version of the Config as dict.

Nested Configs and lists will be replaced by concatenated keys like so:

```
{
    "a": 1,
    "b": [2, 3],
    "c": {
        "x": 4,
        "y": {
            "z": 5
        }
    },
    "d": (6, 7)
}
```

Becomes:

```
{
    "a": 1,
    "b": [2, 3], # if keep_lists is True
    "b.0": 2,
    "b.1": 3,
    "c.x": 4,
    "c.y.z": 5,
    "d": (6, 7)
}
```

We return a dict because dots are disallowed within Config keys.

Parameters

- **keep_lists** – Keeps list along with unpacked values
- **max_split_size** – List longer than this will not be unpacked
- **flatten_int** – Integer keys will be treated as strings

Returns A flattened version of self

Return type dict

hasattr_not_none(*key*)

static init_objects(*config*)

Returns a new Config with types converted to instances.

Any value that is a Config and contains a type key will be converted to an instance of that type:

```
{
    "stuff": "also_stuff",
    "convert_me": {
        type: {
            "param": 1,
            "other_param": 2
        },
        "something_else": "hopefully_useless"
    }
}
```

becomes:

```
{
    "stuff": "also_stuff",
    "convert_me": type(param=1, other_param=2)
}
```

Note that additional entries can be lost as shown above.

Parameters **config** (Config) – New Config will be built from this one

Returns A new config with instances made from type entries.

Return type *Config*

load(*file_*, *raise_=True*, *decoder_cls_=<class 'trixi.util.util.ModuleMultiTypeDecoder'>*, ***kwargs*)

Load config from file using `json.load()`.

Parameters

- **file** (*str or File*) – Read from this location.

- **raise** (*bool*) – Raise errors.
- **decoder_cls** (*type*) – Class that is used to decode JSON string.
- ****kwargs** – Will be passed to `json.load()`.

loads(*json_str*, *decoder_cls_=<class 'trixi.util.util.ModuleMultiTypeDecoder'>*, ***kwargs*)
Load config from JSON string using `json.loads()`.

Parameters

- **json_str** (*str*) – Interpret this string.
- **decoder_cls** (*type*) – Class that is used to decode JSON string.
- ****kwargs** – Will be passed to `json.loads()`.

set_from_string(*str_*, *stringify_value=False*)
Set a value from a single string, separated with "=". Uses :meth:´set_with_decode´.

Parameters **str** (*str*) – String that looks like "key=value".

set_with_decode(*key*, *value*, *stringify_value=False*)
Set single value, using *ModuleMultiTypeDecoder* to interpret key and value strings by creating a temporary JSON string.

Parameters

- **key** (*str*) – Config key.
- **value** (*str*) – New value key will map to.
- **stringify_value** (*bool*) – If *True*, will insert the value into the temporary JSON as a real string. See examples!

Examples

Example for when you need to set *stringify_value=True*:

```
config.set_with_decode("key", "__type__(trixi.util.config.Config)", stringify_
↪value=True)
```

Example for when you need to set *stringify_value=False*:

```
config.set_with_decode("key", "[1, 2, 3]")
```

to_cmd_args_str()
Create a string representing what one would need to pass to the command line. Does not yet use JSON encoding!

Returns Command line string

Return type str

update(*dict_like*, *deep=False*, *ignore=None*, *allow_dict_overwrite=True*)
Update entries in the Config.

Parameters

- **dict_like** (*dict or derivative thereof*) – Update source.
- **deep** (*bool*) – Make deep copies of all references in the source.
- **ignore** (*iterable*) – Iterable of keys to ignore in update.

- **allow_dict_overwrite** (*bool*) – Allow overwriting with dict. Regular dicts only update on the highest level while we recurse and merge Configs. This flag decides whether it is possible to overwrite a 'regular' value with a dict/Config at lower levels. See examples for an illustration of the difference

Examples

The following illustrates the update behaviour if :obj:allow_dict_overwrite is active. If it isn't, an AttributeError would be raised, originating from trying to update "string":

```
config1 = Config(config={
    "lvl0": {
        "lvl1": "string",
        "something": "else"
    }
})

config2 = Config(config={
    "lvl0": {
        "lvl1": {
            "lvl2": "string"
        }
    }
})

config1.update(config2, allow_dict_overwrite=True)

>>>config1
{
    "lvl0": {
        "lvl1": {
            "lvl2": "string"
        },
        "something": "else"
    }
}
```

update_missing(*dict_like*, *deep=False*, *ignore=None*)

Recursively insert values that do not yet exist.

Parameters

- **dict_like** (*dict or derivative thereof*) – Update source.
- **deep** (*bool*) – Make deep copies of all references in the source.
- **ignore** (*iterable*) – Iterable of keys to ignore in update.

trixi.util.config.**monkey_patch_fn_args_as_config**(*f*)

Decorator: Monkey patches, aka addes a variable 'fn_args_as_config' to globals, so that it can be accessed by the decorated function. Adds all function parameters to a dict 'fn_args_as_config', which can be accessed by the method. Be careful using it!

trixi.util.config.**update_from_sys_argv**(*config*, *warn=False*)

Updates Config with the arguments passed as args when running the program. Keys will be converted to command line options, then matching options in *sys.argv* will be used to update the Config.

Parameters

- **config** (Config) – Update this Config.

- **warn** (*bool*) – Raise warnings if there are unknown options. Turn this on if you don't use any `argparse.ArgumentParser` after to check for possible errors.

7.2 ExtraVisdom

class trixi.util.extravisdom.**ExtraVisdom**(**args*, ***kwargs*)

Bases: sphinx.ext.autodoc.importer._MockObject

histogram_3d(*X*, *win=None*, *env=None*, *opts=None*)

Given an array it plots the histrograms of the entries.

Parameters

- **X** – An array of at least 2 dimensions, where the first dimensions gives the number of histograms.
- **win** – Window name.
- **env** – Env name.
- **opts** – dict with options, especially opts['numbins'] (number of histogram bins) and opts['mutiplier']

(factor to stretch / queeze the values on the x axis) should be considered.

Returns The send result.

7.3 GridSearch

class trixi.util.gridsearch.**GridSearch**

Bases: dict

all_combinations()

read(*file_*, *raise_=True*, *decoder_cls_=<class 'trixi.util.util.ModuleMultiTypeDecoder'>*, ***kwargs*)

7.4 pytorchutils

trixi.util.pytorchutils.**get_guided_image_gradient**(*model: <sphinx.ext.autodoc.importer._MockObject object at 0x7f8886120450>*, *inpt*, *err_fn*, *abs=False*)

trixi.util.pytorchutils.**get_input_gradient**(*model*, *inpt*, *err_fn*, *grad_type='vanilla'*, *n_runs=20*, *eps=0.1*, *abs=False*, *results_fn=<function <lambda>>*)

Given a model creates calculates the error and backpropagates it to the image and saves it (saliency map).

Parameters

- **model** – The model to be evaluated
- **inpt** – Input to the model
- **err_fn** – The error function the evaluate the output of the model on

- **grad_type** – Gradient calculation method, currently supports (vanilla, vanilla-smooth, guided,
- **guided-smooth)** (*the guided backprob can lead to segfaults -.-*) –
- **n_runs** – Number of runs for the smooth variants
- **eps** – noise scaling to be applied on the input image (noise is drawn from N(0,1))
- **abs** (*bool*) – Flag, if the gradient should be a absolute value
- **results_fn** – function which is called with the results/ return values. Expected f(grads)

trixi.util.pytorchutils.**get_smooth_image_gradient**(*model*, *inpt*, *err_fn*, *abs=True*, *n_runs=20*, *eps=0.1*, *grad_type='vanilla'*)

trixi.util.pytorchutils.**get_vanilla_image_gradient**(*model*, *inpt*, *err_fn*, *abs=False*)

trixi.util.pytorchutils.**set_seed**(*seed*)
Sets the seed

trixi.util.pytorchutils.**update_model**(*original_model*, *update_dict*, *exclude_layers=()*, *do_warnings=True*)

7.5 SourcePacker

class trixi.util.sourcepacker.**SourcePacker**
Bases: object

Inspired by https://github.com/IDSIA/sacred

static create_source_or_dep(*mod*, *sources*)

static gather_sources_and_dependencies(*globs*)

static git_info(*file_*)

static is_source(*filename*)

static iter_prefixes(*path*)
Iterate through all (non-empty) prefixes of a dotted path. Example: >>> list(iter_prefixes('foo.bar.baz')) ['foo', 'foo.bar', 'foo.bar.baz']

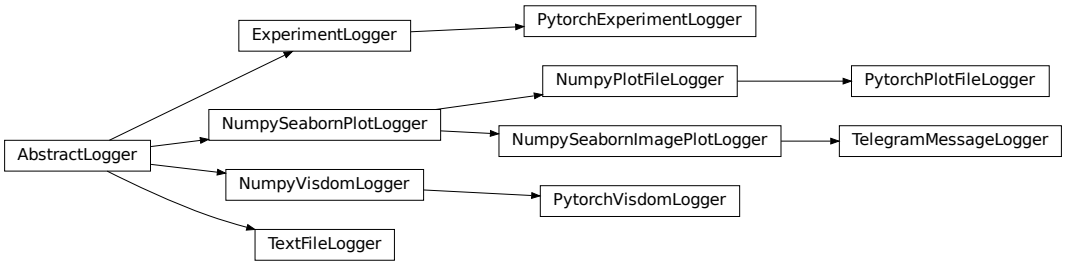
static join_paths(**parts*)
Join different parts together to a valid dotted path.

static zip_sources(*globs*, *filename*)

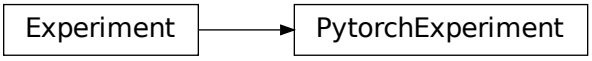
CHAPTER 8

Class Diagram

8.1 Logger



8.2 Experiment



CHAPTER 9

Indices and tables

- genindex
- modindex

Python Module Index

t

Index

F

G

H

I

J

L

M

N

O

P