thelper Documentation

Release 0.6.1

Pierre-Luc St-Charles

Sep 15, 2020

Contents

CHAPTER 1

Overview

dependencies	
ci-status	
releases	
packages	

This package provides a training framework and CLI for PyTorch-based machine learning projects. This is free software distributed under the Apache Software License version 2.0 built by researchers and developers from the Centre de Recherche Informatique de Montréal / Computer Research Institute of Montreal (CRIM).

To get a general idea of what this framework can be used for, visit the FAQ page. For installation instructions, refer to the installation guide. For usage instructions, refer to the user guide. The auto-generated documentation is available via readthedocs.io.

1.1 Notes

Development is still on-going — the API and internal classes may change in the future.

The project's structure was originally generated by cookiecutter via ionelmc's template.

CHAPTER 2

FAQ

We answer some of the simple and frequently asked questions about the framework below. If you think of any other question that should be in this list, send a mail to one of the maintainers, and it will be added here.

2.1 What it is...

- This framework is used to simplify the exploration, development, and testing of models that you create yourself, or that you import from other libraries or frameworks.
- This framework is used to enforce good reproducibility standards for your experiments via the use of global configuration files, checkpoints, and logs.
- This framework is used to easily swap, split, scale, combine, and augment datasets used in your experiments.
- This framework can help you fine-tune, debug, measure, visualize, and understand the behavior of your models more easily.

2.2 What it is NOT...

- This framework is **NOT** used to obtain off-the-shelf solutions. In most cases, you will have to put in some work by at least modifying a pre-existing configuration file to get a solution for a new task (e.g. image classification, segmentation, ...).
- This framework is **NOT** a model (or model architecture) zoo. By design, importing models for training from 3rd-party packages is easy, and so is importing a model architecture from a local Python file. However, due to the fast-paced nature of deep learning, we do not plan to keep a repository of "state-of-the-art" models in the framework.

2.3 What it supports...

- **PyTorch.** For now, the entire backend is based on the design patterns, interfaces, and concepts of the PyTorch library ([more info]).
- Image classification, segmentation, object detection, super-resolution, and generic regression tasks. More types of tasks are planned in the future. Users can also implement their own task interfaces and trainers to support custom scenarios if needed (e.g. for multi-task learning).
- Live evaluation and monitoring of predefined metrics. The framework implements *[several types of metrics]*, but custom metrics can also be defined and evaluated at run time.
- Data augmentation. The framework implements basic *[transformation operations and wrappers]* for large augmentation libraries such as `albumentations` ([more info]).
- Model fine-tuning and exportation. Models obtained from the `torchvision` package ([more info]) or pre-trained using the framework can be loaded and fine-tuned directly for any compatible task. They can also be exported in PyTorch-JIT/ONNX format for external inference.
- Tensorboard. Event logs are generated using `tensorboardX` ([more info]) and may contain plots, visualizations, histograms, graph module trees and more.

2.4 How do I...

This section is still a work in progress; see the use case examples *[here]* for a list of code snippets and tutorials on how to use the framework. For high-level documentation, refer to the *[user guide]*.

CHAPTER 3

Installation

3.1 Docker

Starting with v0.3.2, the latest stable version of the framework is pre-built and available from the docker hub. To get a copy, simply pull it via:

```
$ docker pull plstcharles/thelper
```

You should then be able to launch sessions in containers as such:

```
$ docker run -it plstcharles/thelper thelper <CLI_ARGS_HERE>
```

The image is built from `nvidia/cuda`, meaning that it is compatible with `nvidia-docker` and supports CUDA-enabled GPUs. To run a GPU-enabled container, install the runtime using these instructions, and add `--runtime=nvidia` to the arguments given to `docker run`.

3.2 Installing from source

If you wish to modify the framework's source code or develop new modules within the framework itself, follow the installation instructions below.

3.2.1 Linux

You can use the provided Makefile to automatically create a conda environment on your system that will contain the thelper framework and all its dependencies. In your terminal, simply enter:

```
$ cd <THELPER_ROOT>
$ make install
```

If you already have conda installed somewhere, you can force the Makefile to use it for the installation of the new environment by setting the `CONDA_HOME` variable before calling make:

```
$ export CONDA_HOME=/some/path/to/miniconda3
$ cd <THELPER_ROOT>
$ make install
```

The newly created conda environment will be called 'thelper', and can then be activated using:

```
$ conda activate thelper
```

Or, assuming conda is not already in your path:

```
$ source /some/path/to/miniconda3/bin/activate thelper
```

3.2.2 Other systems

If you cannot use the Makefile, you will have to install the dependencies yourself. These dependencies are listed in the requirements file, and can also be installed using the conda environment configuration file provided here. For the latter case, call the following from your terminal:

```
$ conda env create --file <THELPER_ROOT>/conda-env.yml -n thelper
```

Then, simply activate your environment and install the thelper package within it:

```
$ conda activate thelper
$ pip install -e <THELPER_ROOT> --no-deps
```

On the other hand, although it is *not* recommended since it tends to break PyTorch, you can install the dependencies directly through pip:

```
$ pip install -r <THELPER_ROOT>/requirements.txt
$ pip install -e <THELPER_ROOT> --no-deps
```

3.3 Anaconda

Starting with v0.2.5, a stable version of the framework can be installed directly (with its dependencies) via Anaconda. In a conda environment, simply enter:

```
$ conda config --env --add channels plstcharles
$ conda config --env --add channels conda-forge
$ conda config --env --add channels pytorch
$ conda install thelper
```

This should install a stable version of the framework on Windows and Linux for Python 3.6 or 3.7. You can check the release notes on GitHub, and pre-built packages here.

Note that due to Travis build limitations (as of November 2019), conda package builds and deployments have been stalling and have required manual uploads. This means that the conda packages are fairly likely to be out-of-date compared to those on Docker Hub and PyPI. As such, we now recommend users to install the framework through the "Install from source" method above.

3.4 Testing the installation

You should now be able to print the thelper package version number to see if the package is properly installed and that all dependencies can be loaded at runtime:

```
(conda-env:thelper) username@hostname:~/devel/thelper$ python
  Python X.Y.Z |Anaconda, Inc.| (default, YYY XX ZZZ, AA:BB:CC)
  [GCC X.Y.Z] on linux
  Type "help", "copyright", "credits" or "license" for more information.
  >>> import thelper
  >>> print(thelper.__version__)
  x.y.z
```

You can now refer to the [user guide] for more information on how to use the framework.

3.5 Documentation

The sphinx documentation is generated automatically via readthedocs.io, but it might still be incomplete due to buggy apidoc usage/platform limitations. To build it yourself, use the makefile:

```
$ cd <THELPER_ROOT>
$ make docs
```

The HTML documentation should then be generated inside `<THELPER_ROOT>/docs/build/html`. To browse it, simply open the `index.html` file there.

CHAPTER 4

User Guide

This guide provides an overview of the basic functionalities and typical use cases of the thelper framework. For installation instructions, refer to the installation guide *[here]*.

Currently, the framework can be used to tackle image classification, image segmentation, object detection, image super-resolution, and generic regression tasks. Models for all of these tasks can be trained out-of-the-box using PyTorch. More task types are expected to follow in the future. The goal of the framework is not to solve those problems for you; its goal is to facilitate your model exploration and development process. This is achieved by providing a centralized interface for the control of all your experiment settings, by offering a simple solution for model checkpointing and fine-tuning, and by providing debugging tools and visualizations to help you understand your model's behavior. It can also help users working with GPU clusters by keeping track of their jobs more easily. This framework will not directly give you the perfect solution for your particular problem, but it will help you discover a solution while enforcing good reproducibility standards.

If your problem is related to one of the aforementioned tasks, and if you can solve this problem using a standard model architecture already included in PyTorch or in the framework itself, then you might be able to train and export a solution without writing a single line of code. It is however typical to work with a custom model, a custom trainer, or even a custom task/objective. This is also supported by the framework, as most classes can be either imported as-is, or they can derive from and replace the internal classes of the framework.

In the sections below, we introduce the framework's *Command-Line Interface (CLI)* used to launch jobs, the *session configuration files* used to define the settings of these jobs, and the *session directories* that contain job outputs. Use cases that show how to use different functionalities of the framework are available in *[a different section]*.

4.1 Command-Line Interface

The Command-Line Interface (CLI) of the framework offers the main entrypoint from which jobs are executed. A number of different operations are supported; these are detailed in the following subsections, and listed *[in the documentation]*. For now, note that these operations all rely on a configuration dictionary which is typically parsed from a JSON file. The fields of this dictionary that are required by each operation are detailed *in the next section*.

Note that using the framework's CLI is not mandatory. If you prefer bypassing it and creating your own high-level job dispatcher, you can do so by deconstructing one of the already-existing CLI entrypoints, and by calling the same high-level functions it uses to load the components you need. These might include for example *thelper.data.utils.create_loaders()* and *thelper.nn.utils.create_model()*. Calling those functions directly may also be necessary if you intend on embedding the framework inside another application.

4.1.1 Creating a training session

Usage from the terminal:

```
$ thelper new <PATH_TO_CONFIG_FILE.json> <PATH_TO_ROOT_SAVE_DIR>
```

To create a training session, the `new` operation of the CLI is used. This redirects the execution flow to *thelper.cli.create_session()*. The configuration dictionary that is provided must contain all sections required to train a model, namely `datasets`, `loaders`, `model`, and `trainer`. It is also mandatory to provide a `name` field in the global space for the training session to be properly identified later on.

No distinction is made at this stage regarding the task that the training session is tackling. The nature of this task (e.g. image classification) will be deduced from the `datasets` section of the configuration later in the process. This CLI entrypoint can therefore be used to start training sessions for any task.

Finally, note that since starting a training session produces logs and data, the path to a directory where the output can be created must be provided as the second argument.

4.1.2 Resuming a training session

Usage from the terminal:

```
$ thelper resume <PATH_TO_SESSION_DIR_OR_CHECKPT> [-m MAP_LOCATION] [-c OVERRIDE_CFG]␣
↪[...]
```

If a previously created training session was halted for any reason, it is possible to resume it with the `resume` operation of the CLI. To do so, you must provide either the path to the session directory or to a checkpoint created by the framework. If a directory path is given, it will be searched for checkpoints and the latest one will be loaded. The training session will then be resumed using the loaded model and optimizer state, and subsequent outputs will be saved in the original session directory.

A session can be resumed with an overriding configuration dictionary adding e.g. new metrics. If no configuration is provided at all, the original one contained in the loaded checkpoint will be used. Compatibility between an overriding configuration dictionary and the original one must be ensured by the user. A session can also be resumed only to evaluate the (best) trained model performance on the testing set. This is done by adding the `--eval-only` flag at the end of the command line. For more information on the parameters, see the documentation of *thelper.cli.resume_session()*.

4.1.3 Visualizing data

Usage from the terminal:

```
$ thelper viz <PATH_TO_CONFIG_FILE.json>
```

Visualizing the images that will be forwarded to the model during training after applying data augmentation operations can be useful to determine whether they still look natural or not. The `viz` operation of the CLI allows you to do just this. It relies on the dataset parsers or data loaders defined in a configuration dictionary that would normally be given

to the CLI under the `new` or `resume` operation modes. For more information on this mode, see the documentation of *`thelper.cli.visualize_data()`*.

4.1.4 Annotating data

Usage from the terminal:

```
$ thelper annot <PATH_TO_CONFIG_FILE.json> <PATH_TO_ROOT_SAVE_DIR>
```

The `annot` CLI operation allows the user to browse a dataset and annotate individual samples from it using a specialized GUI tool. The configuration dictionary that is provided must contain a `datasets` section to define the parsers that load the data, and an `annotator` section that defines the GUI tool settings used to create annotations. During an annotation session, all annotations that are created by the user will be saved into the session directory. For more information on the parameters, refer to the documentation of *`thelper.cli.annotate_data()`*.

4.1.5 Split data

Usage from the terminal:

```
$ thelper split <PATH_TO_CONFIG_FILE.json> <PATH_TO_ROOT_SAVE_DIR>
```

When traning a model, the framework will typically split the datasets into non-overlapping data loaders. This split must be performed every time a training session is created or resumed. This can be a lengthy process based on the amount of preprocessing and parsing required by the dataset constructors. Using the `split` CLI operation allows the user to pre-compute this split and archive the training, validation, and test sets into a HDF5 archive. This archive can then be parsed by an interface provided in the framework to speed up the creation/resuming of training sessions, or simply for external tests. See *`thelper.data.parsers.HDF5Dataset`* for more information on the dataset interface, or *`thelper.cli.split_data()`* on the operation itself.

4.1.6 Export model

Usage from the terminal:

```
$ thelper export <PATH_TO_CONFIG_FILE.json> <PATH_TO_ROOT_SAVE_DIR>
```

The `export` CLI operation allows the user to export a trained model for external use as defined in a configuration file. The export format is a new checkpoint that may optionally contain an optimized version of the model compiled using PyTorch's JIT engine. This is still an experimental feature. See the documentation of *`thelper.cli.export_model()`* or the *[example here]* for more information.

[to top]

4.2 Configuration Files

Configuration files are at the heart of the framework. These essentially contain all the settings that might affect the behavior of a training session, and therefore of a trained model. The framework itself does not enforce that all parameters must be passed through the configuration file, but it is good to follow this principle, as it helps enforce reproducibility. Configuration files also essentially always contain a dictionary so that parameters can be split into sections. We thus often refer to them as 'configuration dictionaries'.

The framework will automatically skips sections of the configuration file that it does not need to use or that it does not understand. This is useful when sections or subsections are added for custom needs, or when only a portion of the configuration is relevant to some use case (for example, the 'visualization' mode of the CLI will only look at the datasets and data loaders sections).

For now, all configuration files are expected to be in JSON or YAML format. Future versions of the framework should support raw python modules (.py files) that define each subsection as a dictionary. Examples of complete configuration files used for various purposes are available in the `config` directory located with the code ([see them here]).

4.2.1 Datasets section

The `datasets` section of the configuration defines the dataset "parsers" that will be instantiated by the framework and passed to the data loaders. These are responsible for parsing the structure of a dataset and providing the total number of samples that it contains. Dataset parsers should expose a `__getitem__` function that returns an individual data sample when queried by index. The dataset parsers provided in the `torchvision.datasets` package are all fully compatible with these requirements.

The configuration section itself should be built like a dictionary of objects to instantiate. The key associated with each parser is the name that will be used to refer to it internally as well as in the `loaders` section. If a dataset parser that does not derive from *`thelper.data.parsers.Dataset`* is needed, you will have to specify a task object inside its definition. An example configuration based on the CIFAR10 class provided by `torchvision` ([more info here]) is shown below:

```
"datasets": {
    "cifar10_train": {  # name of the first dataset parser
        "type": "torchvision.datasets.CIFAR10",  # class to instantiate
        "params": {  # parameters forwarded to the class constructor
            "root": "data/cifar/train",
            "train": true,
            "download": true
        },
        "task": {  # task defined explicitely due to external type
            "type": "thelper.tasks.Classification",
            "params": { # by default, we just need to know the class names
                "class_names": [
                    "airplane", "car", "bird", "cat", "deer",
                    "dog", "frog", "horse", "ship", "truck"
                ],
                # torchvision loads samples as tuples; we map the indices
                "input_key": "0",  # input = element at index#0 in tuple
                "label_key": "1"   # label = element at index#1 in tuple
            }
        }
    },
    "cifar10_test": {  # name of the second dataset parser
        "type": "torchvision.datasets.CIFAR10",  # class to instantiate
        "params": {  # parameters forwarded to the class constructor
            "root": "data/cifar/test",
            "train": false,  # here, fetch test data instead of train data
            "download": true
        },
        "task": {
            # we use the same task info as above, both will be merged
            "type": "thelper.tasks.Classification",
            "params": {
                "class_names": [
```

(continues on next page)

(continued from previous page)

```
                    "airplane", "car", "bird", "cat", "deer",
                    "dog", "frog", "horse", "ship", "truck"
                ],
                "input_key": "0",
                "label_key": "1"
            }
        }
    }
}
```

The example above defines two dataset parsers, `cifar10_train` and `cifar10_test`, that can now be referred to in the `loaders` section of a configuration file (*described next*). For more information on the instantiation of dataset parsers, refer to *thelper.data.utils.create_parsers()*.

4.2.2 Loaders section

The `loaders` section of the configuration defines all data loader-related settings including split ratios, samplers, batch sizes, base transforms and augmentations, seeds, memory pinning, and async worker count. The first important concept to understand here is that multiple data parsers (*defined earlier*) can be combined or split into one or more data loaders. Moreover, there are exactly three data loaders defined for all experiments: the training data loader, the validation data loader, and the test data loader. For more information on the fundamental role of each loader, see [this link]. In short, data loaders deal with parsers to load and transform data samples efficiently before packing them into batches that we can feed our models.

Some of the settings defined in this section apply to all three data loaders (e.g. memory pinning, base data transforms), while others can be specified for each loader individually (e.g. augmentations, batch size). The meta-settings that should always be set however are the split ratios that define the fraction of samples from each parser to use in a data loader. As shown in the example below, these ratios allow us to split a dataset into different loaders automatically, and without any possibility of data leakage between them. If all RNG seeds are set in this section, then the split will be reproducible between experiments. The split can also be precomputed using the `split` operation of the CLI (*click here for more information*).

Besides, base transformations defined in this section are used to ensure that all samples loaded by parsers are compatible with the input format expected by the model during training. For example, typical image classification pipelines expect images to have a resolution of 224x224 pixels, with each color channel normalized to either the [-1, 1] range, or using pre-computed mean and standard deviation values. We can define such operations directly using the classes available in the *thelper.transforms* module. This is also demonstrated in the example configuration below:

```
# note: this example is tied with the "datasets" example given earlier
"loaders": {
    "batch_size": 32,      # pack 32 images per minibatch (for all loaders)
    "test_seed": 0,        # fix the test set splitting seed
    "valid_seed": 0,       # fix the validation set splitting seed
    "torch_seed": 0,       # fix the PyTorch RNG seed for tranforms/augments
    "numpy_seed": 0,       # fix the numpy RNG seed for transforms/augments
    "random_seed": 0,      # fix the random package RNG seed for transforms/augments
    # note: non-fixed seeds will be initialized randomly and printed in logs
    "workers": 4,          # each loader will be loading 4 minibatches in parallel
    "base_transforms": [   # defines the operations to apply to all loaded samples
        {
            # first, normalize 8-bit images to the [-1, 1] range
            "operation": "thelper.transforms.NormalizeMinMax",
            "params": {
                "min": [127, 127, 127],
```

(continues on next page)

(continued from previous page)

```
                "max": [255, 255, 255]
            }
        },
        {
            # next, resize the CIFAR10 images to 224x224 for the model
            "operation": "thelper.transforms.Resize",
            "params": {
                "dsize": [224, 224]
            }
        },
        {
            # finally, transform the opencv/numpy arrays to torch.Tensor arrays
            "operation": "torchvision.transforms.ToTensor"
        }
    ],
    # we reserve 20% of the samples from the training parser for validation
    "train_split": {
        "cifar10_train": 0.8
    },
    "valid_split": {
        "cifar10_train": 0.2
    },
    # we use 100% of the samples from the test parser for testing
    "test_split": {
        "cifar10_test": 1.0
    }
}
```

The example above prepares the CIFAR10 data using a 80%-20% training-validation split, and keeps all the original CIFAR10 testing data for actual testing. All loaded samples will be normalized and resized to fit the expected input resolution of a typical model, as shown in the next subsection. This example however contains no data augmentation pipelines; refer to the *[relevant sections here]* for actual usage examples. Similarly, no sampler is used above to rebalance the classes; *[see here]* for a use case. Finally, for more information on other parameters that are not discussed here, refer to the documentation of *thelper.data.utils.create_loaders()*.

4.2.3 Model section

The `model` section of the configuration defines the model that will be trained, fine-tuned, evaluated, or exported during the session. The model can be defined in several ways. If you are creating a new model from scratch (i.e. using randomly initialized weights), you simply have to specify the type of the class that implements the model's architecture along with its constructor's parameters. This is shown in the example below for an instance of MobileNet:

```
"model": {
    "type": "thelper.nn.mobilenet.MobileNetV2",
    "params": {
        "input_size": 224
    }
}
```

In this case, the constructor of *thelper.nn.mobilenet.MobileNetV2* will only receive a single argument, `input_size`, i.e. the size of the tensors it should expect as input. Some implementations of model architectures such as those in `torchvision.models` ([see them here]) might allow you to specify a `pretrained` parameter. Setting this parameter to `True` will let you automatically download the weights of that model and thus allow you to fine-tune it directly:

```
"model": {
    "type" : "torchvision.models.resnet.resnet18",
    "params": {
        "pretrained": true
    }
}
```

The second option to fine-tune a model that is not available via `torchvision` is to specify the path to a checkpoint produced by the framework as such:

```
"model": {
    "ckptdata" : "<PATH_TO_ANY_THELPER_CHECKPOINT.pth>"
}
```

When using this approach, the framework will first open the checkpoint and reinstantiate the model using its original fully qualified class name and the parameters originally passed to its constructor. Then, that model will be checked for task compatibility, and its weights will finally be loaded in. For more information on the checkpoints produced by the framework, see the *[relevant section below]*. For more information on the model creation/loading process, refer to *thelper.nn.utils.create_model()*.

4.2.4 Trainer section

The `trainer` section of the configuration defines trainer, optimization, and metric-related settings used in a session. These settings include the type of trainer to use, the number of epochs to train for, the list of metrics to compute during training, the name of the metric to continuously monitor for improvements, the loss function to use, the optimizer, the scheduler, and the device (CUDA or CPU) that the session should be executed on.

First, note here that the type of trainer that is picked must be compatible with the task(s) exposed by the dataset parser(s) listed earlier in the configuration. If no trainer type is provided, the framework will automatically deduce which one to use for the current task. This deduction might fail for custom trainers/task combinations. If you are using a custom task, or if your model relies on multiple loss functions (or any other similar exotic thing), you might have to create your own trainer implementation derived from *thelper.train.base.Trainer*. Otherwise, see the trainers module (*thelper.train*) for a list of all available trainers.

All optimization settings are grouped into the `optimization` subsection of the `trainer` section. While specifying a scheduler is optional, an optimizer and a loss function must always be specified. The loss function can be provided via the typical type/params setup (as shown below), or obtained from the model via a getter function. For more information on the latter option, see *thelper.optim.utils.create_loss_fn()*. On the other hand, the nature of the optimizer and scheduler can only be specified via a type/param setup (as also shown below). The weights of the model specified in the last section will always be passed as the first argument of the optimizer's constructor at runtime. This behavior is compatible with all optimizers defined by PyTorch ([more info here]).

The `trainer` section finally contains another subsection titled `metrics`. This subsection defines a dictionary of named metrics that should be continuously updated during training, and evaluated at the end of each epoch. Numerous types of metrics are already implemented in *thelper.optim.metrics*, and many more will be added in the future. Metrics typically measure the performance of the model based on a specific criteria, but they can also do things like save model predictions and create graphs. A special "monitored" metric can also be defined in the `trainer` section, and it will be used to determine whether the model is improving or not during the training session. This is used to keep track of the "best" model weights while creating checkpoints, and it might also be used for scheduling.

A complete example of a trainer configuration is shown below:

```
"trainer": {
    # this example is in line with the earlier examples; we create a classifier
    "type": "thelper.train.ImageClassifTrainer",  # type could be deduced␣
↪automatically
```

(continues on next page)

(continued from previous page)

```
    "device": "cuda:all",    # by default, run the session on all GPUs in parallel
    "epochs": 50,            # run the session for a maximum of 50 epochs
    "save_freq": 1,          # save the model in a checkpoint every epoch
    "monitor": "accuracy",   # monitor the 'accuracy' metric defined below for␣
↪improvements
    "use_tbx": true,         # activate tensorboardX metric logging in output directory
    "optimization": {
        "loss": {
            "type": "torch.nn.CrossEntropyLoss",
            "params": {}     # empty sections like these can be omitted
        },
        "optimizer": {
            "type": "torch.optim.RMSprop",
            "params": {
                "lr": 0.01, # default learning rate used at the first epoch
                "weight_decay": 0.00004
            }
        },
        "scheduler": {
            # here, we create a fancy scheduler that will check a metric for its steps
            "type": "torch.optim.lr_scheduler.ReduceLROnPlateau",
            "params": {
                "mode": "max",    # since we will monitor accuracy, we want to␣
↪maximize it
                "factor": 0.1,    # when a plateau is detected, decrease lr by 90%
                "patience": 3     # wait three epochs with no improvement before␣
↪stepping
            },
            # now, we just name the metric defined below for the scheduler to use
            "step_metric": "accuracy"
        }
    },
    "metrics": {  # this is the list of all metrics we will be evaluating
        "accuracy": {  # the name of each metric should be unique
            "type": "thelper.optim.Accuracy",
            "params": {
                "top_k": 1
            }
        },
        "confmat": {
            # this is a special consumer used to create confusion matrices
            # (we can't monitor this one, as it is not an actual "metric")
            "type": "thelper.train.ConfusionMatrix"
        }
    },
    "test_metrics": {  # metrics in this section will only be used for testing
        "logger": {
            # (can't monitor this one either, as it is not an actual "metric")
            "type": "thelper.train.ClassifLogger",
            "params": {
                "top_k": 3
            }
        }
    }
}
```

For more information on the metrics available in the framework, see *thelper.optim.metrics*.

4.2.5 Annotator section

The `annotator` section of the configuration is used solely to define GUI-related settings during annotation sessions. For now, it should only contain the type and constructor parameters of the GUI tool that will be instantiated to create the annotations. An example is shown below:

```
"annotator": {
    "type": "thelper.gui.ImageSegmentAnnotator",  # type of annotator to instantiate
    "params": {
        "sample_input_key": "image",  # this key is tied to the data parser's output
        "labels": [
            # for this example, we only use one brush type that draws using solid red
            {"id": 255, "name": "foreground", "color": [0, 0, 255]}
        ]
    }
}
```

In this case, an image segmentation GUI is created that will allow the "image" loaded in each sample to be annotated by user with a brush tool. This section (as well as all GUI tools) are still experimental. For more information on annotators, refer to *thelper.gui.annotators*.

4.2.6 Global parameters

Finally, session configurations can also contain global parameters located outside the main sections detailed so far. For example, the session name is a global flag which is often mandatory as it is used to identify the session and create its output directory. Other global parameters are used to change the behavior of imported package, or are just hacky solutions to problems that should be fixed otherwise.

For now, the global parameters considered "of interest" are the following:

- `name` : specifies the name of the session (mandatory in most operation modes).
- `cudnn_benchmark` : specifies whether to activate/deactivate cuDNN benchmarking mode.
- `cudnn_deterministic` : specifies whether to activate/deactivate cuDNN deterministic mode.

Future global parameters will most likely be handled via *thelper.utils.setup_globals()*.

[to top]

4.3 Session Directories

If the framework is used in a way that requires it to produce outputs, they will always be located somewhere in the "session directory". This directory is created in the root output directory provided to the CLI (also often called the "save" directory), and it is named after the session itself. The session directory contains three main folders that hold checkpoints, logs, and outputs. These are discussed in the following subsections. The general structure of a session directory is shown below:

```
<session_directory_name>
  |
  |-- checkpoints
  |      |-- ckpt.0000.<platform>-<date>-<time>.pth
  |      |-- ckpt.0001.<platform>-<date>-<time>.pth
  |      |-- ckpt.0002.<platform>-<date>-<time>.pth
```

(continues on next page)

(continued from previous page)

```
|     |-- ...
|     \-- ckpt.best.pth
|
|-- logs
|     |-- <dataset1-name>.log
|     |-- <dataset2-name>.log
|     |-- ...
|     |-- config.<platform>-<date>-<time>.json
|     |-- data.log
|     |-- modules.log
|     |-- packages.log
|     |-- task.log
|     \-- trainer.log
|
|-- output
|     \-- <session_directory_name>
|           |-- train-<platform>-<date>-<time>
|           |     |-- events.out.tfevents.<something>.<platform>
|           |     \-- ...
|           |
|           |-- valid-<platform>-<date>-<time>
|           |     |-- events.out.tfevents.<something>.<platform>
|           |     \-- ...
|           |
|           |-- test-<platform>-<date>-<time>
|           |     |-- events.out.tfevents.<something>.<platform>
|           |     \-- ...
|           |
|           \-- ...
|
\-- config.latest.json
```

4.3.1 Checkpoints

The `checkpoints` folder contains the binary files pickled by PyTorch that store all training data required to resume a session. These files are automatically saved at the end of each epoch during a training session. The checkpoints are named using the `ckpt.XXXX.YYYYY-ZZZZZZ-ZZZZZZ.pth` convention, where `XXXX` is the epoch index (0-based), `YYYYY` is the platform or hostname, and `ZZZZZZ-ZZZZZZ` defines the date and time of their creation (in YYYYMMDD-HHMMSS format). All checkpoints created by the framework during training will use this naming convention except for the `best` checkpoint that might be created in monitored training sessions (as part of early stopping and for final model evaluation). In this case, it will simply be named `ckpt.best.pth`. Its content is the same as other checkpoints however, and it is actually just a copy of the corresponding "best" checkpoint in the same directory.

Checkpoints can be opened directly using `torch.load()`. They contain a dictionary with the following fields:

- `name` : the name of the session. Used as a unique identifier for many types of output.
- `epoch` : the epoch index (0-based) at the end of which the checkpoint was saved. This value is optional, and may only be saved in training sessions.
- `iter` : the total number of iterations computed so far in the training session. This value is optional, and may only be saved in training sessions.
- `source` : the name of the host that created the checkpoint and its time of creation.
- `sha1` : the sha1 signature of the framework's latest git commit. Used for debugging purposed only.

- `version` : the version of the framework that created this checkpoint. Will be used for data and configuration file migration if necessary when reloading the checkpoint.
- `task` : a copy or string representation of the task the model was being trained for. Used to keep track of expected model input/output mappings (e.g. class names).
- `outputs` : the outputs (e.g. metrics) generated by the trainer for all epochs thus far. This object is optional, and may only be saved in training sessions.
- `model` : the weights (or "state dictionary") of the model, or a path to where these weights may be found. This field can be used to hold a link to an external JIT trace or ONNX version of the model.
- `model_type` : the type (or class name) of the model that may be used to reinstantiate it.
- `model_params` : the constructor parameters of the model that may be used to reinstantiate it.
- `optimizer` : the state of the optimizer at the end of the latest training epoch. This value is optional, and may only be saved in training sessions.
- `scheduler` : the state of the scheduler at the end of the latest training epoch. This value is optional, and may only be saved in training sessions.
- `monitor_best` : the "best" value for the metric being monitorered so far. This value is optional, and may only be saved in training sessions.
- `config` : the full session configuration directionary originally passed to the CLI entrypoint.

By default, these fields do not contain pickled objects directly tied to the framework, meaning any PyTorch installation should be able to open a checkpoint without crashing. This also means that a model trained with this framework can be opened and reused in any other framework, as long as you are willing to extract its weights from the checkpoint yourself. An example of this procedure is given *[here]*.

Experimental support for checkpoint creation outside a training session is available through the CLI's `export` operation. *See the section above for more information*.

4.3.2 Session logs

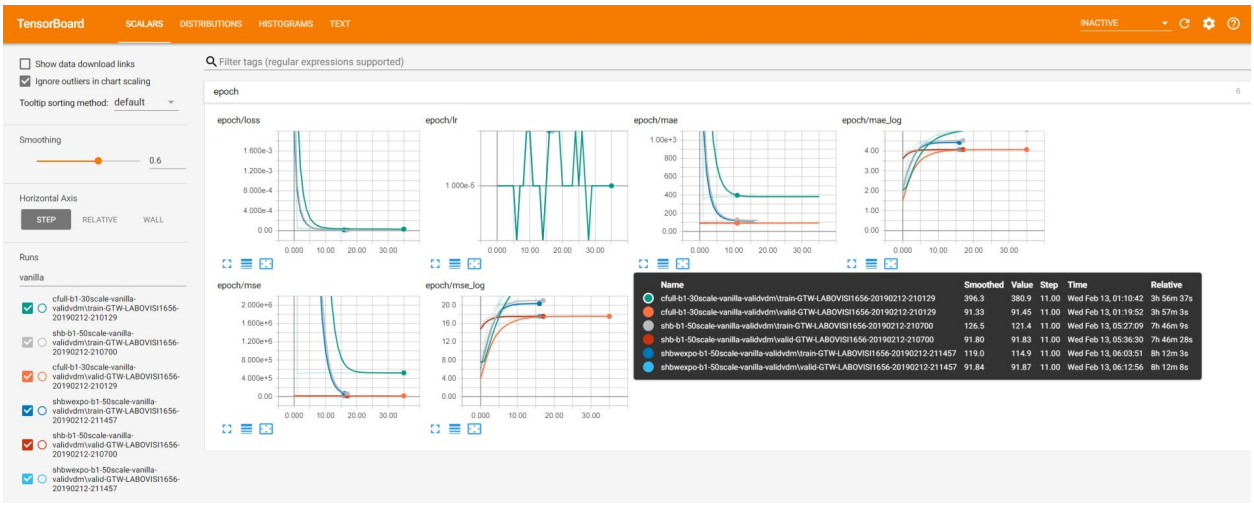
All information printed to the terminal during a session will also automatically be printed to files located in the `logs` folder of the session directory. Moreover, useful information about the training environment and datasets will be printed in other files in the same location. A brief description of these files is provided below:

- `<dataset_name>.log` : contains metadata (in JSON format) of the named dataset, its loaded sample count, and the separation of its sample indices across the train/valid/test sets. Can be used to validate the data split and keep track of which sample is used in which set.
- `config.<PLATFORM>-<DATE>-<TIME>.json` : backup of the (JSON) configuration file of the session that created or modified the current session directory.
- `data.log` : logger output that provides high-level information about the loaded dataset parsers including given names, sizes, task interfaces, and base transforms.
- `modules.log` : logger output that provides details regarding the instantiated model type (class name), the parameters passed to its constructor, and a full list of its layers once constructed.
- `packages.log` : lists all packages installed in the runtime environment as well as their version.
- `task.log` : provides the full string representation of the task object used during the session.
- `trainer.log` : logger output that details the training progress during the session. This file can become very large for long sessions; and might be rotated past a certain size in the future.

Specialized CLI operations and trainers as well as custom implementations might create additional logs in this directory. In all cases, logs are provided as nice-to-have for debugging purposes only, and their content/structure might change in future versions.

4.3.3 Outputs

Finally, the session directory contains an `output` folder that is used to store all the evaluation results generated by the metrics as well as the `tensorboard` event files. The first level of the `output` directory is named after the session itself so that it may easily be copied elsewhere without creating conflicts. This also allows `tensorboard` to display the session name in its UI. That folder then contains the training, validation, and testing outputs generated for each session. These outputs are separated so that individual curves can be turned on and off in `tensorboard`. A typical output directory loaded in `tensorboard` is shown below.



In this example, the training and validation outputs of several sessions are combined. The metrics of each session that produced scalar values were used to generate plots. The scalars are evaluated once every epoch, and are grouped automatically in a section named `epoch`. The loss and learning rates are also automatically plotted in this section. Additional tabs holding model weight histograms and text outputs are also available. If a metric had been used that generated images, those would also be available in another tab.

For more information on available metrics, see *`thelper.optim.metrics`*. For more information about `tensorboard`, visit [the official site].

[to top]

CHAPTER 5

Use Cases

This section is still under construction. Some example configuration files are available in the `config` directory of the repository ([see them here]). For high-level information on generic parts of the framework, refer to the *[user guide]*.

5.1 Image classification

Building an image classifier is probably the simplest thing you can do with the framework. Here, we provide an in-depth look at a JSON configuration used to built a 10-class object classification model based on the CIFAR-10 dataset.

As usual, we must define four different fields in our configuration for everything to work: the datasets, the data loaders, the model, and the trainer. First off, the dataset. As mentionned before, here we will work with the CIFAR-10 dataset provided by `torchvision`. We do so simply to have a configuration that can run "off-the-shelf", but in reality, you will probably be using your own data. To learn how to create your own dataset interface and load your own data, see *[this section]*. So, for our simple task, we define the `datasets` field as follows:

```
"datasets": {
    "cifar10": {
        "type": "torchvision.datasets.CIFAR10",
        "params": {"root": "data/cifar/train", "download": true},
        "task": {
            "type": "thelper.tasks.Classification",
            "params": {
                "class_names": [
                    "airplane", "car", "bird", "cat", "deer",
                    "dog", "frog", "horse", "ship", "truck"
                ],
                "input_key": "0", "label_key": "1"
            }
        }
    }
}
```

In summary, we will instantiate a single dataset named "cifar10" based on the `torchvision.datasets.CIFAR10` class. The constructor of that class will receive two arguments, namely the (relative) path where to save the data, and a boolean indicating that the dataset should be downloaded if not already available. More info on these two arguments can actually be found in the [PyTorch documentation]. Here, since this dataset interface does not explicitly define a "task", we need to provide one ourselves. Therefore, we add a "task" field in which we specify that the task type is related to classification, and provide the task's construction arguments. In this case, this is the list of class names that correspond to the indices that the dataset interface will be associating to each loaded sample, and the key strings. Note that since `torchvision.datasets.CIFAR10` will be loading samples as tuples, so the key strings merely correspond to tuple indices.

Next, we will define how this cute little dataset will be used to load training and/or evaluation data. This is accomplished through the `loaders` fields as follows:

```
"loaders": {
    "train_split": {"cifar10": 0.9},
    "valid_split": {"cifar10": 0.1},
    "batch_size": 32,
    "base_transforms": [
        {
            "operation": "thelper.transforms.NormalizeMinMax",
            "params": {
                "min": [127, 127, 127], "max": [255, 255, 255]
            }
        },
        {
            "operation": "thelper.transforms.Resize",
            "params": {"dsize": [224, 224]}
        },
        {
            "operation": "torchvision.transforms.ToTensor"
        }
    ]
}
```

In this example, we ask the training and validation data loaders to split the "cifar10" dataset we defined above using a 90%-10% ratio. By default, this split will be randomized, but repeatable across experiments. Seeds used to shuffle the data samples will be printed in the logs, and can even be set manually in this section if needed. Next, we set the batch size for all data loaders to 32, and define base transforms to apply to all samples. In this case, the transforms will normalize each 8-bit image by min-maxing it, resize it 224x224 (as required by our model), and finally transform it into a tensor. Many transformation operations can be defined in the `loaders` section, and it may be interesting to visualize the output before trying to feed it to a model. For more information on how to do so, see *[this use case]*.

Next, we will define the model architecture to train. Again, you might want to use your own architecture here. If so, refer to the *[relevant use case]*. Here, we keep things extremely simple and rely on the pre-existing ResNet implementation located inside the framework:

```
"model": {
    "type": "thelper.nn.resnet.ResNet"
}
```

Since this class can be instantiated as-is without having to provide any arguments (it defaults to ResNet-34), we do not even need to specify a "params" field. Once created, this model will be adapted to our classification problem by providing it the "task" object we defined in our dataset interface. This means that its default 1000-class output layer will be updated to produce 10 outputs (since we have 10 classes).

Finally, we define the `trainer` field to provide all remaining training parameters:

```
"trainer": {
    "epochs": 5,
    "monitor": "accuracy",
    "optimization": {
        "loss": {"type": "torch.nn.CrossEntropyLoss"},
        "optimizer": {"type": "torch.optim.Adam"}
    },
    "metrics": {
        "accuracy": {"type": "thelper.optim.Accuracy"}
    }
}
```

Here, we limit the training to 5 epochs. The loss is a traditional cross-entropy, and we use Adam to update model weights via backprop. The loss function and optimizer could both receive extra parameters (using a "params" field once more), but we keep the defaults everywhere. Finally, we define a single metric to be evaluated during training (accuracy), and set it as the "monitoring" metric to use for early stopping.

The complete configuration is shown below:

```
{
    "name": "classif-cifar10",
    "datasets": {
        "cifar10": {
            "type": "torchvision.datasets.CIFAR10",
            "params": {"root": "data/cifar/train", "download": true},
            "task": {
                "type": "thelper.tasks.Classification",
                "params": {
                    "class_names": [
                        "airplane", "car", "bird", "cat", "deer",
                        "dog", "frog", "horse", "ship", "truck"
                    ],
                    "input_key": "0", "label_key": "1"
                }
            }
        }
    },
    "loaders": {
        "batch_size": 32,
        "base_transforms": [
            {
                "operation": "thelper.transforms.NormalizeMinMax",
                "params": {
                    "min": [127, 127, 127], "max": [255, 255, 255]
                }
            },
            {
                "operation": "thelper.transforms.Resize",
                "params": {"dsize": [224, 224]}
            },
            {
                "operation": "torchvision.transforms.ToTensor"
            }
        ],
        "train_split": {"cifar10": 0.9},
        "valid_split": {"cifar10": 0.1}
    },
```

(continues on next page)

(continued from previous page)

```
    "model": {"type": "thelper.nn.resnet.ResNet"},
    "trainer": {
        "epochs": 5,
        "monitor": "accuracy",
        "optimization": {
            "loss": {"type": "torch.nn.CrossEntropyLoss"},
            "optimizer": {"type": "torch.optim.Adam"}
        },
        "metrics": {
            "accuracy": {"type": "thelper.optim.Accuracy"}
        }
    }
}
```

Once saved to a json file, we will be able to launch the training session via:

```
$ thelper new <PATH_TO_CLASSIF_CIFAR10_CONFIG>.json <PATH_TO_OUTPUT_DIR>
```

The dataset will first be downloaded, split, and passed to data loaders. Then, the model will be instantiated, and all objects will be given to the trainer to start the session. Right away, some log files will be created in a new folder named "classif-cifar10" in the directory provided as the second argument on the command line. Once the training is complete, that folder will contain the model checkpoints as well as the final evaluation results.

5.2 Image segmentation

Section statement here @@@@@@

5.3 Object Detection

Section statement here @@@@@@

5.4 Super-resolution

Section statement here @@@@@@

5.5 Creating a new dataset interface

Section statement here @@@@@@

5.6 Dataset/Loader visualization

Section statement here @@@@@@

5.7 Dataset annotation

Section statement here @@@@@@

5.8 Rebalancing a dataset

Section statement here @@@@@@

5.9 Exporting a dataset

Section statement here @@@@@@

5.10 Defining a data augmentation pipeline

Section statement here @@@@@@

5.11 Supporting a custom trainer

Section statement here @@@@@@

5.12 Supporting a custom task

Section statement here @@@@@@

5.13 Supporting a custom model

Section statement here @@@@@@

5.14 Visualizing metrics using `tensorboardX`

Section statement here @@@@@@

5.15 Manually reloading a model

Section statement here @@@@@@

5.16 Exporting a model

Once you have trained a model (using the framework or otherwise), you might want to share it with others. Models are typically exported in two parts: architecture and weights. However, metadata related to the task the model was built for would be missing with only those two components. Here, we show a solution for exporting a classification model trained using the framework under ONNX, TraceScript, or pickle format along with its corresponding index-to-class-name mapping. Further down, we also give tips on similarly exporting a model trained in another framework.

The advantage of ONNX and TraceScript exports is that whoever reloads your model does not need to have the class that you used to define the model's architecture at hand. However, this approach might make fine-tuning or retraining your model more complicated (you should consider it a 'read-only' export).

Models/checkpoints exported this way can be easily reloaded using the framework, and may also be opened manually by others to extract only the information they need.

So, first off, let's start by training a classification model using the following configuration:

```
{
    "name": "classif-cifar10",
    "datasets": {
        "cifar10": {
            "type": "torchvision.datasets.CIFAR10",
            "params": {"root": "data/cifar/train"},
            "task": {
                "type": "thelper.tasks.Classification",
                "params": {
                    "class_names": [
                        "airplane", "car", "bird", "cat", "deer",
                        "dog", "frog", "horse", "ship", "truck"
                    ],
                    "input_key": "0", "label_key": "1"
                }
            }
        }
    },
    "loaders": {
        "batch_size": 32,
        "base_transforms": [
            {
                "operation": "thelper.transforms.NormalizeMinMax",
                "params": {
                    "min": [127, 127, 127], "max": [255, 255, 255]
                }
            },
            {
                "operation": "thelper.transforms.Resize",
                "params": {"dsize": [224, 224]}
            },
            {
                "operation": "torchvision.transforms.ToTensor"
            }
        ],
        "train_split": {"cifar10": 0.9},
        "valid_split": {"cifar10": 0.1}
    },
    "model": {"type": "thelper.nn.resnet.ResNet"},
    "trainer": {
        "epochs": 5,
```

(continues on next page)

(continued from previous page)

```
        "monitor": "accuracy",
        "optimization": {
            "loss": {"type": "torch.nn.CrossEntropyLoss"},
            "optimizer": {"type": "torch.optim.Adam"}
        },
        "metrics": {
            "accuracy": {"type": "thelper.optim.Accuracy"}
        }
    }
}
```

The above configuration essentially means that we will be training a ResNet model with default settings on CIFAR10 using all 10 classes. You can launch the training process via:

```
$ thelper new <PATH_TO_CLASSIF_CIFAR10_CONFIG>.json <PATH_TO_OUTPUT_DIR>
```

See the *[user guide]* for more information on creating training sessions. Once that's done, you should obtain a folder named `classif-cifar10` in your output directory that contains training logs as well as checkpoints. To export this model in a new checkpoint, we will use the following session configuration:

```
{
    "name": "export-classif-cifar10",
    "model": {
        "ckptdata": "<PATH_TO_OUTPUT_DIR>/classif-cifar10/checkpoints/ckpt.best.pth"
    },
    "export": {
        "ckpt_name": "test-export.pth",
        "trace_name": "test-export.zip",
        "save_raw": true,
        "trace_input": "torch.rand(1, 3, 224, 224)"
    }
}
```

This configuration essentially specifies where to find the 'best' checkpoint for the model we just trained, and how to export a trace of it. For more information on the export operation, refer to *[the user guide]*. We now provide the configuration as a JSON to the CLI one more:

```
$ thelper export <PATH_TO_EXPORT_CONFIG>.json <PATH_TO_OUTPUT_DIR>
```

If everything goes well, `<PATH_TO_OUTPUT_DIR>/export-classif-cifar10` should now contain a checkpoint with the exported model trace and all metadata required to reinstantiate it. Note that as of 2019/06, PyTorch exports model traces as zip files, meaning you will have to copy two files from the output session folder. In this case, that would be `test-export.pth` and `test-export.zip`.

Finally, note that if you are attempting to export a model that was trained outside the framework, you will have to specify which task this model was trained for as well as the type of the model to instantiate and possibly the path to its weights in the `model` field of the configuration above. An example configuration is given below:

```
{
    "name": "export-classif-custom",
    "model": {
        "type": "fully.qualified.name.to.model",
        "params": {
            # here, provide all model constructor parameters
        },
        "weights": "path_to_model_state_dictionary.pth"
```

(continues on next page)

(continued from previous page)

```
    },
    "export": {
        "ckpt_name": "test-export.pth",
        "trace_name": "test-export.zip",
        "save_raw": true,
        "trace_input": "torch.rand(1, 3, 224, 224)"
    }
}
```

For more information on model importation, refer to the documentation of *thelper.nn.utils.create_model()*.

[to top]

CHAPTER 6

thelper package

Top-level package for the 'thelper' framework.

Running `import thelper` will recursively import all important subpackages and modules.

6.1 Subpackages

6.1.1 thelper.data package

Dataset parsing/loading package.

This package contains classes and functions whose role is to fetch the data required to train, validate, and test a model. The *thelper.data.utils.create_loaders()* function contained herein is responsible for preparing the task and data loaders for a training session. This package also contains the base interfaces for dataset parsers.

Subpackages

thelper.data.geo package

Geospatial dataset parsing/loading package.

This package contains classes and functions whose role is to fetch the data required to train, validate, and test a model on geospatial data. Importing the modules inside this package requires GDAL.

Submodules

thelper.data.geo.agrivis module

Agricultural Semantic Segentation Challenge Dataset Interface

Original author: David Landry (david.landry@crim.ca) Updated by Pierre-Luc St-Charles (April 2020)

class thelper.data.geo.agrivis.**Hdf5AgricultureDataset**(*hdf5_path: AnyStr, group_name: AnyStr, transforms: Any = None, use_global_normalization: bool = True, keep_file_open: bool = False, load_meta_keys: bool = False, copy_to_slurm_tmpdir: bool = False*)

Bases: *thelper.data.parsers.Dataset*

__getitem__(*idx*)
Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*hdf5_path: AnyStr, group_name: AnyStr, transforms: Any = None, use_global_normalization: bool = True, keep_file_open: bool = False, load_meta_keys: bool = False, copy_to_slurm_tmpdir: bool = False*)
Dataset parser constructor.

In order for derived datasets to be instantiated automatically by the framework from a configuration file, they must minimally accept a 'transforms' argument like the shown one here.

Parameters

- **transforms** – function or object that should be applied to all loaded samples in order to return the data in the requested transformed/augmented state.
- **deepcopy** – specifies whether this dataset interface should be deep-copied inside *thelper.data.loaders.LoaderFactory* so that it may be shared between different threads. This is false by default, as we assume datasets do not contain a state or buffer that might cause problems in multi-threaded data loaders.

thelper.data.geo.bigearthnet module

thelper.data.geo.gdl module

Data parsers & utilities for cross-framework compatibility with Geo Deep Learning (GDL).

Geo Deep Learning (GDL) is a machine learning framework initiative for geospatial projects lead by the wonderful folks at NRCan's CCMEO. See https://github.com/NRCan/geo-deep-learning for more information.

The classes and functions defined here were used for the exploration of research topics and for the validation and testing of new software components.

class thelper.data.geo.gdl.**MetaSegmentationDataset**(*class_names, work_folder, dataset_type, meta_map, max_sample_count=None, dontcare=None, transforms=None*)

Bases: *thelper.data.geo.gdl.SegmentationDataset*

Semantic segmentation dataset interface that appends metadata under new tensor layers.

__getitem__(*index*)
Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*class_names, work_folder, dataset_type, meta_map, max_sample_count=None, dontcare=None, transforms=None*)
Segmentation dataset parser constructor.

This constructor receives all extra arguments necessary to build a segmentation task object.

Parameters

- **class_names** – list of all class names (or labels) that must be predicted in the image.
- **input_key** – key used to index the input image in the loaded samples.
- **label_map_key** – key used to index the label map in the loaded samples.
- **meta_keys** – list of extra keys that will be available in the loaded samples.
- **transforms** – function or object that should be applied to all loaded samples in order to return the data in the requested transformed/augmented state.
- **deepcopy** – specifies whether this dataset interface should be deep-copied inside *thelper.data.loaders.LoaderFactory* so that it may be shared between different threads. This is false by default, as we assume datasets do not contain a state or buffer that might cause problems in multi-threaded data loaders.

static get_meta_value(*map*, *key*)

metadata_handling_modes = ['const_channel', 'scaled_channel']

class thelper.data.geo.gdl.**SegmentationDataset**(*class_names*, *work_folder*, *dataset_type*, *max_sample_count=None*, *dontcare=None*, *transforms=None*)

Bases: *thelper.data.parsers.SegmentationDataset*

Semantic segmentation dataset interface for GDL-based HDF5 parsing.

__getitem__(*index*)

Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*class_names*, *work_folder*, *dataset_type*, *max_sample_count=None*, *dontcare=None*, *transforms=None*)

Segmentation dataset parser constructor.

This constructor receives all extra arguments necessary to build a segmentation task object.

Parameters

- **class_names** – list of all class names (or labels) that must be predicted in the image.
- **input_key** – key used to index the input image in the loaded samples.
- **label_map_key** – key used to index the label map in the loaded samples.
- **meta_keys** – list of extra keys that will be available in the loaded samples.
- **transforms** – function or object that should be applied to all loaded samples in order to return the data in the requested transformed/augmented state.
- **deepcopy** – specifies whether this dataset interface should be deep-copied inside *thelper.data.loaders.LoaderFactory* so that it may be shared between different threads. This is false by default, as we assume datasets do not contain a state or buffer that might cause problems in multi-threaded data loaders.

thelper.data.geo.infer module

class thelper.data.geo.infer.**SlidingWindowTester**(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)

Bases: *thelper.infer.base.Tester*

Tester that satisfies the requirements of the `Tester` in order to run classification inference

__init__(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)
Receives the trainer configuration dictionary, parses it, and sets up the session.

eval_epoch(*model*, *epoch*, *dev*, *loader*, *metrics*, *output_path*)
Computes the pixelwise prediction on an image.

It does the prediction per batch size of N pixels. It returns the class predicted and its probability. The results are saved into two images created with the same size and projection info as the input rasters.

The `class` image gives the class id, a number between 1 and the number of classes for corresponding pixels. Class id 0 is reserved for `nodata`.

The `probs` image contains N-class channels with the probability values of the pixels for each class. The probabilities by default are normalised.

Also, a `config-classes.json` file is created listing the `name-to-class-id` mapping that was used to generate the values in the `class` image (i.e.: class names defined by the pre-trained `model`).

Parameters

- **model** – the model with which to run inference that is already uploaded to the target device(s).
- **epoch** – the epoch index we are training for (0-based, and should normally only be 0 for single test pass).
- **dev** – the target device that tensors should be uploaded to (corresponding to model's device(s)).
- **loader** – the data loader used to get transformed test samples.
- **metrics** – the dictionary of metrics/consumers to report inference results (mostly loggers and basic report generator in this case since there shouldn't be ground truth labels to validate against).
- **output_path** – directory where output files should be written, if necessary.

supports_classification = True

thelper.data.geo.ogc module

Data parsers & utilities module for OGC-related projects.

class thelper.data.geo.ogc.**TB15D104**
Bases: `object`

Wrapper class for OGC Testbed-15 (D104) identifiers.

BACKGROUND_ID = 0

LAKE_ID = 1

TYPECE_LAKE = '21'

TYPECE_RIVER = '10'

class thelper.data.geo.ogc.**TB15D104Dataset**(*raster_path*, *vector_path*, *px_size=None*, *allow_outlying_vectors=True*, *clip_outlying_vectors=True*, *lake_area_min=0.0*, *lake_area_max=inf*, *lake_river_max_dist=inf*, *feature_buffer=1000*, *master_roi=None*, *focus_lakes=True*, *srs_target='3857'*, *force_parse=False*, *reproj_rasters=False*, *reproj_all_cpus=True*, *display_debug=False*, *keep_rasters_open=True*, *parallel=False*, *transforms=None*)

Bases: *thelper.data.geo.parsers.VectorCropDataset*

OGC Testbed-15 dataset parser for D104 (lake/river) segmentation task.

__getitem__(*idx*)

Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*raster_path*, *vector_path*, *px_size=None*, *allow_outlying_vectors=True*, *clip_outlying_vectors=True*, *lake_area_min=0.0*, *lake_area_max=inf*, *lake_river_max_dist=inf*, *feature_buffer=1000*, *master_roi=None*, *focus_lakes=True*, *srs_target='3857'*, *force_parse=False*, *reproj_rasters=False*, *reproj_all_cpus=True*, *display_debug=False*, *keep_rasters_open=True*, *parallel=False*, *transforms=None*)

Dataset parser constructor.

In order for derived datasets to be instantiated automatically by the framework from a configuration file, they must minimally accept a 'transforms' argument like the shown one here.

Parameters

- **transforms** – function or object that should be applied to all loaded samples in order to return the data in the requested transformed/augmented state.
- **deepcopy** – specifies whether this dataset interface should be deep-copied inside *thelper.data.loaders.LoaderFactory* so that it may be shared between different threads. This is false by default, as we assume datasets do not contain a state or buffer that might cause problems in multi-threaded data loaders.

static lake_cleaner(*features*, *area_min*, *area_max*, *lake_river_max_dist*, *parallel=False*)

Flags geometric features as 'clean' based on type and distance to nearest river.

static lake_cropper(*features*, *rasters_data*, *coverage*, *srs_target*, *px_size*, *skew*, *feature_buffer*, *parallel=False*)

Returns the ROI information for a given feature (may be modified in derived classes).

class thelper.data.geo.ogc.**TB15D104DetectLogger**(*conf_threshold=0.5*)

Bases: *thelper.train.utils.DetectLogger*

__init__(*conf_threshold=0.5*)

Receives the logging parameters & the optional class label names used to decorate the log.

report_geojson()

class thelper.data.geo.ogc.**TB15D104TileDataset**(*raster_path*, *vector_path*, *tile_size*, *tile_overlap*, *px_size=None*, *allow_outlying_vectors=True*, *clip_outlying_vectors=True*, *lake_area_min=0.0*, *lake_area_max=inf*, *master_roi=None*, *srs_target='3857'*, *force_parse=False*, *reproj_rasters=False*, *reproj_all_cpus=True*, *display_debug=False*, *keep_rasters_open=True*, *parallel=False*, *transforms=None*)

Bases: *thelper.data.geo.parsers.TileDataset*

OGC Testbed-15 dataset parser for D104 (lake/river) segmentation task.

__getitem__(*idx*)

Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*raster_path*, *vector_path*, *tile_size*, *tile_overlap*, *px_size=None*, *allow_outlying_vectors=True*, *clip_outlying_vectors=True*, *lake_area_min=0.0*, *lake_area_max=inf*, *master_roi=None*, *srs_target='3857'*, *force_parse=False*, *reproj_rasters=False*, *reproj_all_cpus=True*, *display_debug=False*, *keep_rasters_open=True*, *parallel=False*, *transforms=None*)

Dataset parser constructor.

In order for derived datasets to be instantiated automatically by the framework from a configuration file, they must minimally accept a 'transforms' argument like the shown one here.

Parameters

- **transforms** – function or object that should be applied to all loaded samples in order to return the data in the requested transformed/augmented state.
- **deepcopy** – specifies whether this dataset interface should be deep-copied inside *thelper.data.loaders.LoaderFactory* so that it may be shared between different threads. This is false by default, as we assume datasets do not contain a state or buffer that might cause problems in multi-threaded data loaders.

thelper.data.geo.ogc.**postproc_features**(*input_file*, *bboxes_srs*, *orig_geoms_path*, *output_file*, *final_srs=None*, *write_shapefile_copy=False*)

Post-processes bounding box detections produced during an evaluation session into a GeoJSON file.

thelper.data.geo.parsers module

Geospatial data parser & utilities module.

class thelper.data.geo.parsers.**ImageFolderGDataset**(*root*, *transforms=None*, *image_key='image'*, *label_key='label'*, *path_key='path'*, *idx_key='idx'*, *channels=None*)

Bases: *thelper.data.parsers.ImageFolderDataset*

Image folder dataset specialization interface for classification tasks on geospatial images.

This specialization is used to parse simple image subfolders, and it essentially replaces the very basic `torchvision.datasets.ImageFolder` interface with similar functionalities. It it used to provide a proper task interface as well as path metadata in each loaded packet for metrics/logging output.

The difference with the parent class ImageFolderDataset is the used of gdal to manage multi channels images found in remote sensing domain. The user can specify the channels to load. By default the first three channels are loaded [1,2,3].

See also:

thelper.data.parsers.ImageDataset
thelper.data.parsers.ClassificationDataset
thelper.data.parsers.ImageFolderDataset

__getitem__(*idx*)
Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*root*, *transforms=None*, *image_key='image'*, *label_key='label'*, *path_key='path'*, *idx_key='idx'*, *channels=None*)
Image folder dataset parser constructor.

class thelper.data.geo.parsers.**SlidingWindowDataset**(*raster_path*, *raster_bands*, *patch_size*, *transforms=None*, *image_key='image'*)

Bases: *thelper.data.parsers.Dataset*

Sliding window dataset specialization interface for classification tasks over a geospatial image.

The dataset runs a sliding window over the whole geospatial image in order to return tile patches. The operation can be accomplished over multiple raster bands if they can be found in the provided raster container.

__getitem__(*idx*)
Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*raster_path*, *raster_bands*, *patch_size*, *transforms=None*, *image_key='image'*)
Dataset parser constructor.

In order for derived datasets to be instantiated automatically by the framework from a configuration file, they must minimally accept a 'transforms' argument like the shown one here.

Parameters

- **transforms** – function or object that should be applied to all loaded samples in order to return the data in the requested transformed/augmented state.
- **deepcopy** – specifies whether this dataset interface should be deep-copied inside *thelper.data.loaders.LoaderFactory* so that it may be shared between different threads. This is false by default, as we assume datasets do not contain a state or buffer that might cause problems in multi-threaded data loaders.

class thelper.data.geo.parsers.**TileDataset**(*raster_path*, *vector_path*, *tile_size*, *tile_overlap=0*, *skip_empty_tiles=False*, *skip_nodata_tiles=True*, *px_size=None*, *allow_outlying_vectors=True*, *clip_outlying_vectors=True*, *vector_area_min=0.0*, *vector_area_max=inf*, *vector_target_prop=None*, *master_roi=None*, *srs_target='3857'*, *raster_key='raster'*, *mask_key='mask'*, *cleaner=None*, *force_parse=False*, *reproj_rasters=False*, *reproj_all_cpus=True*, *keep_rasters_open=True*, *transforms=None*)

Bases: *thelper.data.geo.parsers.VectorCropDataset*

Abstract dataset used to systematically tile vector data and rasters.

__init__(*raster_path*, *vector_path*, *tile_size*, *tile_overlap=0*, *skip_empty_tiles=False*, *skip_nodata_tiles=True*, *px_size=None*, *allow_outlying_vectors=True*, *clip_outlying_vectors=True*, *vector_area_min=0.0*, *vector_area_max=inf*, *vector_target_prop=None*, *master_roi=None*, *srs_target='3857'*, *raster_key='raster'*, *mask_key='mask'*, *cleaner=None*, *force_parse=False*, *reproj_rasters=False*, *reproj_all_cpus=True*, *keep_rasters_open=True*, *transforms=None*)

Dataset parser constructor.

In order for derived datasets to be instantiated automatically by the framework from a configuration file, they must minimally accept a 'transforms' argument like the shown one here.

Parameters

- **transforms** – function or object that should be applied to all loaded samples in order to return the data in the requested transformed/augmented state.
- **deepcopy** – specifies whether this dataset interface should be deep-copied inside *thelper.data.loaders.LoaderFactory* so that it may be shared between different threads. This is false by default, as we assume datasets do not contain a state or buffer that might cause problems in multi-threaded data loaders.

class thelper.data.geo.parsers.**VectorCropDataset**(*raster_path*, *vector_path*, *px_size=None*, *skew=None*, *allow_outlying_vectors=True*, *clip_outlying_vectors=True*, *vector_area_min=0.0*, *vector_area_max=inf*, *vector_target_prop=None*, *feature_buffer=None*, *master_roi=None*, *srs_target='3857'*, *raster_key='raster'*, *mask_key='mask'*, *cleaner=None*, *cropper=None*, *force_parse=False*, *reproj_rasters=False*, *reproj_all_cpus=True*, *keep_rasters_open=True*, *transforms=None*)

Bases: *thelper.data.parsers.Dataset*

Abstract dataset used to combine geojson vector data and rasters.

__getitem__(*idx*)

Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*raster_path*, *vector_path*, *px_size=None*, *skew=None*, *allow_outlying_vectors=True*, *clip_outlying_vectors=True*, *vector_area_min=0.0*, *vector_area_max=inf*, *vector_target_prop=None*, *feature_buffer=None*, *master_roi=None*, *srs_target='3857'*, *raster_key='raster'*, *mask_key='mask'*, *cleaner=None*, *cropper=None*, *force_parse=False*, *reproj_rasters=False*, *reproj_all_cpus=True*, *keep_rasters_open=True*, *transforms=None*)

Dataset parser constructor.

In order for derived datasets to be instantiated automatically by the framework from a configuration file, they must minimally accept a 'transforms' argument like the shown one here.

Parameters

- **transforms** – function or object that should be applied to all loaded samples in order to return the data in the requested transformed/augmented state.
- **deepcopy** – specifies whether this dataset interface should be deep-copied inside `thelper.data.loaders.LoaderFactory` so that it may be shared between different threads. This is false by default, as we assume datasets do not contain a state or buffer that might cause problems in multi-threaded data loaders.

thelper.data.geo.utils module

thelper.data.geo.utils.**export_geojson_with_crs**(*features*, *srs_target*)
Exports a list of features along with their SRS into a GeoJSON-compat string.

thelper.data.geo.utils.**export_geotiff**(*filepath*, *crop*, *srs*, *geotransform*)

thelper.data.geo.utils.**get_feature_bbox**(*geom*, *offsets=None*)

thelper.data.geo.utils.**get_feature_roi**(*geom*, *px_size*, *skew*, *roi_buffer=None*, *crop_img_size=None*, *crop_real_size=None*)

thelper.data.geo.utils.**get_geocoord**(*geotransform*, *x*, *y*)

thelper.data.geo.utils.**get_geoextent**(*geotransform*, *x*, *y*, *cols*, *rows*)

thelper.data.geo.utils.**get_pxcoord**(*geotransform*, *x*, *y*)

thelper.data.geo.utils.**open_rasterfile**(*raster_data*, *keep_rasters_open=False*)

thelper.data.geo.utils.**parse_geojson**(*geojson*, *srs_target=None*, *roi=None*, *allow_outlying=False*, *clip_outlying=False*)

thelper.data.geo.utils.**parse_geojson_crs**(*body*)
Imports a coordinate reference system (CRS) from a GeoJSON tree.

thelper.data.geo.utils.**parse_raster_metadata**(*raster_metadata*, *raster_dataset=None*)
Parses the provided raster metadata and updates it by adding extra details required for later use.

The provided raster metadata is updated directly. Metadata is validated against the matching data storage. If any important, required or requested (bands) metadata is missing, the function raises the issue immediately.

Parameters

- **raster_metadata** (*dict*) – raster metadata dictionary with minimally a file 'path' and list of 'bands' indices to process.
- **raster_dataset** (*gdal.Dataset*) – (optional) preloaded dataset object corresponding to the raster metadata.

Raises

- `ValueError` – at least one input raster was missing a required metadata parameter or a parameter is erroneous.
- `IOError` – the raster path could not be found or reading it did not generate a valid raster using GDAL.

thelper.data.geo.utils.**parse_rasters**(*raster_paths*, *srs_target=None*, *reproj=False*)

thelper.data.geo.utils.**parse_roi**(*roi_path*, *srs_target=None*)

thelper.data.geo.utils.**parse_shapefile**(*shapefile_path*, *srs_target=None*, *roi=None*, *allow_outlying=False*, *clip_outlying=False*, *layer_id=0*)

thelper.data.geo.utils.**reproject_coords**(*coords*, *src_srs*, *tgt_srs*)

thelper.data.geo.utils.**reproject_crop**(*raster*, *crop_raster*, *crop_size*, *crop_datatype*, *crop_nodataval=None*, *reproj_opt=None*, *fill_nodata=False*)

Submodules

thelper.data.loaders module

Dataset loaders module.

This module contains a dataset loader specialization used to properly seed samplers and workers.

class thelper.data.loaders.**DataLoader**(**args*, *seeds=None*, *epoch=0*, *collate_fn=<function default_collate>*, ***kwargs*)

Bases: sphinx.ext.autodoc.importer._MockObject

Specialized data loader used to load minibatches from a dataset parser.

This specialization handles the seeding of samplers and workers.

See torch.utils.data.DataLoader for more information on attributes/methods.

__init__(**args*, *seeds=None*, *epoch=0*, *collate_fn=<function default_collate>*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

sample_count

set_epoch(*epoch=0*)

Sets the current epoch number in order to offset RNG states for the workers and the sampler.

class thelper.data.loaders.**DataLoaderWrapper**(*loader*, *callback*)

Bases: *thelper.data.loaders.DataLoader*

Data loader wrapper used to transform all loaded samples with an external function.

This can be useful to convert the samples before the user gets to access them, or to upload them on a specific device.

The wrapped data loader should be compatible with *thelper.data.loaders.DataLoader*.

__init__(*loader*, *callback*)

Initialize self. See help(type(self)) for accurate signature.

class thelper.data.loaders.**LoaderFactory**(*config*)

Bases: object

Factory used for preparing and splitting dataset parsers into usable data loader objects.

This class is responsible for parsing the parameters contained in the 'loaders' field of a configuration dictionary, instantiating the data loaders, and shuffling/splitting the samples. An example configuration is presented in *thelper.data.utils.create_loaders()*.

See also:

thelper.data.utils.create_loaders()
thelper.transforms.utils.load_augments()
thelper.transforms.utils.load_transforms()

__init__(*config*)
Receives and parses the data configuration dictionary.

create_loaders(*datasets*, *train_idxs*, *valid_idxs*, *test_idxs*)
Returns the data loaders for the train/valid/test sets based on a prior split.

This function essentially takes the dataset parser interfaces and indices maps, and instantiates data loaders that are ready to produce samples for training or evaluation. Note that the dataset parsers will be deep-copied in each data loader, meaning that they should ideally not contain a persistent loading state or a large buffer.

Parameters

- **datasets** – the map of dataset parsers, where each has a name (key) and a parser (value).
- **train_idxs** – training data samples indices map.
- **valid_idxs** – validation data samples indices map.
- **test_idxs** – test data samples indices map.

Returns A three-element tuple containing the training, validation, and test data loaders, respectively.

get_base_transforms()
Returns the (global) sample transformation operations parsed in the data configuration.

get_split(*datasets*, *task*)
Returns the train/valid/test sample indices split for a given dataset (name-parser) map.

Note that the returned indices are unique, possibly shuffled, and never duplicated between sets. If the samples have a class attribute (i.e. the task is related to classification), the split will respect the initial distribution and apply the ratios within the classes themselves. For example, consider a dataset of three classes (A, B, and C) that contains 100 samples such as:

$$|A| = 50,\ |B| = 30,\ |C| = 20$$

If we require a 80%-10%-10% ratio distribution for the training, validation, and test loaders respectively, the resulting split will contain the following sample counts:

$$\text{training loader} = 40A + 24B + 16C$$

$$\text{validation loader} = 5A + 3B + 2C$$

$$\text{test loader} = 5A + 3B + 2C$$

In the case of multi-label classification datasets, there is no guarantee that the classes will be balanced across the training/validation/test sets. Instead, for a given class list, the classes with fewer samples will be split first.

Parameters

- **datasets** – the map of datasets to split, where each has a name (key) and a parser (value).
- **task** – a task object that should be compatible with all provided datasets (can be `None`).

Returns A three-element tuple containing the maps of the training, validation, and test sets respectively. These maps associate dataset names to a list of sample indices.

thelper.data.loaders.**default_collate**(*batch*, *force_tensor=True*)
Puts each data field into a tensor with outer dimension batch size.

This function is copied from PyTorch's *torch.utils.data._utils.collate.default_collate*, but additionally supports custom objects from the framework (such as bounding boxes). These will not be converted to tensors, and it will be up to the trainer to handle them accordingly.

See `torch.utils.data.DataLoader` for more information.

thelper.data.parsers module

Dataset parsers module.

This module contains dataset parser interfaces and base classes that define basic i/o operations so that the framework can automatically interact with training data.

class thelper.data.parsers.**ClassificationDataset**(*class_names*, *input_key*, *label_key*, *meta_keys=None*, *transforms=None*, *deepcopy=False*)

Bases: *thelper.data.parsers.Dataset*

Classification dataset specialization interface.

This specialization receives some extra parameters in its constructor and automatically defines a *thelper.tasks.classif.Classification* task based on those. The derived class must still implement *thelper.data.parsers.ClassificationDataset.__getitem__()*, and it must still store its samples as dictionaries in `self.samples` to behave properly.

See also:

thelper.data.parsers.Dataset

__getitem__(*idx*)
Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*class_names*, *input_key*, *label_key*, *meta_keys=None*, *transforms=None*, *deepcopy=False*)
Classification dataset parser constructor.

This constructor receives all extra arguments necessary to build a classification task object.

Parameters

- **class_names** – list of all class names (or labels) that will be associated with the samples.
- **input_key** – key used to index the input data in the loaded samples.
- **label_key** – key used to index the label (or class name) in the loaded samples.
- **meta_keys** – list of extra keys that will be available in the loaded samples.
- **transforms** – function or object that should be applied to all loaded samples in order to return the data in the requested transformed/augmented state.
- **deepcopy** – specifies whether this dataset interface should be deep-copied inside *thelper.data.loaders.LoaderFactory* so that it may be shared between different threads. This is false by default, as we assume datasets do not contain a state or buffer that might cause problems in multi-threaded data loaders.

class thelper.data.parsers.**Dataset**(*transforms=None*, *deepcopy=False*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

Abstract dataset parsing interface that holds a task and a list of sample dictionaries.

This interface helps fix a failure of PyTorch's dataset interface (`torch.utils.data.Dataset`): the lack of identity associated with the components of a sample. In short, a data sample loaded by a dataset typically contains the input data that should be forwarded to a model as well as the expected prediction of the model (i.e. the 'groundtruth') that will be used to compute the loss. These two elements are typically paired in a tuple that can then be provided to the data loader for batching. Problems however arise when the model has multiple inputs or outputs, when the sample needs to carry supplemental metadata to simplify debugging, or when transformation operations need to be applied only to specific elements of the sample. Here, we fix this issue by specifying that all samples must be provided to data loaders as dictionaries. The keys of these dictionaries explicitly define which value(s) should be transformed, which should be forwarded to the model, which are the expected model predictions, and which are only used for debugging. The keys are defined via the task object that is generated by the dataset or specified via the configuration file (see *thelper.tasks.utils.Task* for more information).

To properly use this interface, a derived class must implement *thelper.data.parsers.Dataset.__getitem__()*, as well as provide proper `task` and `samples` attributes. The `task` attribute must derive from *thelper.tasks.utils.Task*, and `samples` must be an array-like object holding already-parsed information about the dataset samples (in dictionary format). The length of the `samples` array will automatically be returned as the size of the dataset in this interface. For class-based datasets, it is recommended to parse the classes in the dataset constructor so that external code can directly peek into the `samples` attribute to see their distribution without having to call `__getitem__`. This is done for example in *thelper.data.loaders.LoaderFactory.get_split()* to automatically rebalance classes without having to actually load the samples one by one, which speeds up the process dramatically.

Variables

- ***transforms*** – function or object that should be applied to all loaded samples in order to return the data in the requested transformed/augmented state.
- ***deepcopy*** – specifies whether this dataset interface should be deep-copied inside *thelper.data.loaders.LoaderFactory* so that it may be shared between different threads. This is false by default, as we assume datasets do not contain a state or buffer that might cause problems in multi-threaded data loaders.
- ***samples*** – list of dictionaries containing the data that is ready to be forwarded to the data loader. Note that relatively costly operations (such as reading images from a disk or pre-transforming them) should be delayed until the *thelper.data.parsers.Dataset.__getitem__()* function is called, as they will most likely then be accomplished in a separate thread. Once loaded, these samples should never be modified by another part of the framework. For example, transformation and augmentation operations will always be applied to copies of these samples.
- ***task*** – object used to define what keys are used to index the loaded data into sample dictionaries.

See also:

thelper.data.parsers.ExternalDataset

__getitem__(*idx*)
Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*transforms=None*, *deepcopy=False*)
Dataset parser constructor.

In order for derived datasets to be instantiated automatically by the framework from a configuration file, they must minimally accept a 'transforms' argument like the shown one here.

Parameters

- **transforms** – function or object that should be applied to all loaded samples in order to return the data in the requested transformed/augmented state.
- **deepcopy** – specifies whether this dataset interface should be deep-copied inside *thelper.data.loaders.LoaderFactory* so that it may be shared between different threads. This is false by default, as we assume datasets do not contain a state or buffer that might cause problems in multi-threaded data loaders.

deepcopy
specifies whether this dataset interface should be deep-copied inside *thelper.data.loaders.LoaderFactory* so that it may be shared between different threads. This is false by default, as we assume datasets do not contain a state or buffer that might cause problems in multi-threaded data loaders.

samples
Returns the list of internal samples held by this dataset interface.

task
Returns the task object associated with this dataset interface.

transforms
Returns the transformation operations to apply to this dataset's loaded samples.

class thelper.data.parsers.**ExternalDataset**(*dataset*, *task*, *transforms=None*, *deepcopy=False*, ***kwargs*)

Bases: *thelper.data.parsers.Dataset*

External dataset interface.

This interface allows external classes to be instantiated automatically in the framework through a configuration file, as long as they themselves provide implementations for `__getitem__` and `__len__`. This includes all derived classes of `torch.utils.data.Dataset` such as `torchvision.datasets.ImageFolder`, and the specialized versions such as `torchvision.datasets.CIFAR10`.

Note that for this interface to be compatible with our runtime instantiation rules, the constructor needs to receive a fully constructed task object. This object is currently constructed in *thelper.data.utils.create_parsers()* based on extra parameters; see the code there for more information.

Variables

- **dataset_type** – type of the internally instantiated or provided dataset object.
- **warned_dictionary** – specifies whether the user was warned about missing keys in the output samples dictionaries.

See also:

thelper.data.parsers.Dataset

__getitem__(*idx*)
Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*dataset*, *task*, *transforms=None*, *deepcopy=False*, ***kwargs*)
External dataset parser constructor.

Parameters

- **dataset** – fully qualified name of the dataset object to instantiate, or the dataset itself.
- **task** – fully constructed task object providing key information for sample loading.

- **transforms** – function or object that should be applied to all loaded samples in order to return the data in the requested transformed/augmented state.
- **deepcopy** – specifies whether this dataset interface should be deep-copied inside *thelper.data.loaders.LoaderFactory* so that it may be shared between different threads. This is false by default, as we assume datasets do not contain a state or buffer that might cause problems in multi-threaded data loaders.

class thelper.data.parsers.**HDF5Dataset**(*root*, *subset='train'*, *transforms=None*)

Bases: *thelper.data.parsers.Dataset*

HDF5 dataset specialization interface.

This specialization is compatible with the HDF5 packages made by the CLI's "split" operation. The archives it loads contains pre-split datasets that can be reloaded without having to resplit their data. The archive also contains useful metadata, and a task interface.

Variables

- **archive** – file descriptor for the opened hdf5 dataset.
- **subset** – hdf5 group section representing the targeted set.
- **target_args** – list decompression args required for each sample key.
- **source** – source logstamp of the hdf5 dataset.
- **git_sha1** – framework git tag of the hdf5 dataset.
- **version** – version of the framework that saved the hdf5 dataset.
- **orig_config** – configuration used to originally generate the hdf5 dataset.

See also:

thelper.cli.split_data()
thelper.data.utils.create_hdf5()

__getitem__(*idx*)

Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*root*, *subset='train'*, *transforms=None*)

HDF5 dataset parser constructor.

This constructor receives the path to the HDF5 archive as well as a subset indicating which section of the archive to load. By default, it loads the training set.

close()

Closes the internal HDF5 file.

class thelper.data.parsers.**ImageDataset**(*root*, *transforms=None*, *image_key='image'*, *path_key='path'*, *idx_key='idx'*)

Bases: *thelper.data.parsers.Dataset*

Image dataset specialization interface.

This specialization is used to parse simple image folders, and it does not fulfill the requirements of any specialized task constructors due to the lack of groundtruth data support. Therefore, it returns a basic task object (*thelper.tasks.utils.Task*) with no set value for the groundtruth key, and it cannot be used to directly train a model. It can however be useful when simply visualizing, annotating, or testing raw data from a simple directory structure.

See also:

thelper.data.parsers.Dataset

__getitem__(*idx*)
Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*root*, *transforms=None*, *image_key='image'*, *path_key='path'*, *idx_key='idx'*)
Image dataset parser constructor.

This constructor exposes some of the configurable keys used to index sample dictionaries.

class thelper.data.parsers.**ImageFolderDataset**(*root*, *transforms=None*, *image_key='image'*, *label_key='label'*, *path_key='path'*, *idx_key='idx'*)
Bases: *thelper.data.parsers.ClassificationDataset*

Image folder dataset specialization interface for classification tasks.

This specialization is used to parse simple image subfolders, and it essentially replaces the very basic `torchvision.datasets.ImageFolder` interface with similar functionalities. It it used to provide a proper task interface as well as path metadata in each loaded packet for metrics/logging output.

See also:

thelper.data.parsers.ImageDataset
thelper.data.parsers.ClassificationDataset

__getitem__(*idx*)
Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*root*, *transforms=None*, *image_key='image'*, *label_key='label'*, *path_key='path'*, *idx_key='idx'*)
Image folder dataset parser constructor.

class thelper.data.parsers.**SegmentationDataset**(*class_names*, *input_key*, *label_map_key*, *meta_keys=None*, *dontcare=None*, *transforms=None*, *deepcopy=False*)
Bases: *thelper.data.parsers.Dataset*

Segmentation dataset specialization interface.

This specialization receives some extra parameters in its constructor and automatically defines its task (*thelper.tasks.segm.Segmentation*) based on those. The derived class must still implement *thelper.data.parsers.SegmentationDataset.__getitem__()*, and it must still store its samples as dictionaries in `self.samples` to behave properly.

See also:

thelper.data.parsers.Dataset

__getitem__(*idx*)
Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*class_names*, *input_key*, *label_map_key*, *meta_keys=None*, *dontcare=None*, *transforms=None*, *deepcopy=False*)
Segmentation dataset parser constructor.

This constructor receives all extra arguments necessary to build a segmentation task object.

Parameters

- **class_names** – list of all class names (or labels) that must be predicted in the image.
- **input_key** – key used to index the input image in the loaded samples.
- **label_map_key** – key used to index the label map in the loaded samples.
- **meta_keys** – list of extra keys that will be available in the loaded samples.
- **transforms** – function or object that should be applied to all loaded samples in order to return the data in the requested transformed/augmented state.
- **deepcopy** – specifies whether this dataset interface should be deep-copied inside *thelper.data.loaders.LoaderFactory* so that it may be shared between different threads. This is false by default, as we assume datasets do not contain a state or buffer that might cause problems in multi-threaded data loaders.

class thelper.data.parsers.**SuperResFolderDataset**(*root*, *downscale_factor=2.0*, *rescale_lowres=True*, *center_crop=None*, *transforms=None*, *lowres_image_key='lowres_image'*, *highres_image_key='highres_image'*, *path_key='path'*, *idx_key='idx'*, *label_key='label'*)
Bases: *thelper.data.parsers.Dataset*

Image folder dataset specialization interface for super-resolution tasks.

This specialization is used to parse simple image subfolders, and it essentially replaces the very basic `torchvision.datasets.ImageFolder` interface with similar functionalities. It it used to provide a proper task interface as well as path/class metadata in each loaded packet for metrics/logging output.

__getitem__(*idx*)
Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*root*, *downscale_factor=2.0*, *rescale_lowres=True*, *center_crop=None*, *transforms=None*, *lowres_image_key='lowres_image'*, *highres_image_key='highres_image'*, *path_key='path'*, *idx_key='idx'*, *label_key='label'*)
Image folder dataset parser constructor.

thelper.data.pascalvoc module

PASCAL VOC dataset parser module.

This module contains a dataset parser used to load the PASCAL Visual Object Classes (VOC) dataset for semantic segmentation or object detection. See http://host.robots.ox.ac.uk/pascal/VOC/ for more info.

class thelper.data.pascalvoc.**PASCALVOC**(*root*, *task='segm'*, *subset='trainval'*, *target_labels=None*, *download=False*, *preload=True*, *use_difficult=False*, *use_occluded=True*, *use_truncated=True*, *transforms=None*, *image_key='image'*, *sample_name_key='name'*, *idx_key='idx'*, *image_path_key='image_path'*, *gt_path_key='gt_path'*, *bboxes_key='bboxes'*, *label_map_key='label_map'*)

Bases: *thelper.data.parsers.Dataset*

PASCAL VOC dataset parser.

This class can be used to parse the PASCAL VOC dataset for either semantic segmentation or object detection. The task object it exposes will be changed accordingly. In all cases, the 2012 version of the dataset will be used.

TODO: Add support for semantic instance segmentation.

See also:

thelper.data.parsers.Dataset

__getitem__(*idx*)
Returns the data sample (a dictionary) for a specific (0-based) index.

__init__(*root*, *task='segm'*, *subset='trainval'*, *target_labels=None*, *download=False*, *preload=True*, *use_difficult=False*, *use_occluded=True*, *use_truncated=True*, *transforms=None*, *image_key='image'*, *sample_name_key='name'*, *idx_key='idx'*, *image_path_key='image_path'*, *gt_path_key='gt_path'*, *bboxes_key='bboxes'*, *label_map_key='label_map'*)
Dataset parser constructor.

In order for derived datasets to be instantiated automatically by the framework from a configuration file, they must minimally accept a 'transforms' argument like the shown one here.

Parameters

- **transforms** – function or object that should be applied to all loaded samples in order to return the data in the requested transformed/augmented state.
- **deepcopy** – specifies whether this dataset interface should be deep-copied inside *thelper.data.loaders.LoaderFactory* so that it may be shared between different threads. This is false by default, as we assume datasets do not contain a state or buffer that might cause problems in multi-threaded data loaders.

decode_label_map(*label_map*)
Returns a color image from a label indices map.

encode_label_map(*label_map*)
Returns a map of label indices from a color image.

thelper.data.samplers module

Samplers module.

This module contains classes used for raw dataset rebalancing or augmentation.

All samplers here should aim to be compatible with PyTorch's sampling interface (torch.utils.data.sampler.Sampler) so that they can be instantiated at runtime through a configuration file and used as the input of a data loader.

class thelper.data.samplers.**FixedWeightSubsetSampler**(*indices*, *labels*, *weights*, *seeds=None*, *epoch=0*)

Bases: sphinx.ext.autodoc.importer._MockObject

Provides a rebalanced list of sample indices to use in a data loader.

Given a list of sample indices and the corresponding list of class labels, this sampler will produce a new list of indices that rebalances the distribution of samples according to a provided array of weights.

Example configuration file:

```
# ...
# the sampler is defined inside the 'loaders' field
"loaders": {
    # ...
    # this field is completely optional, and can be omitted entirely
    "sampler": {
        # the type of the sampler we want to instantiate
        "type": "thelper.data.samplers.FixedWeightSubsetSampler",
        # the parameters passed to the sampler's constructor
        "params": {
            "weights": {
                # the weights must be provided using class name pairs
                "class_A": 0.1,
                "class_B": 5.0,
                "class_C": 1.0,
                # ...
            }
        },
        # specifies whether the sampler should receive class labels
        "pass_labels": true
    },
    # ...
},
# ...
```

Variables

- **nb_samples** – total number of samples to rebalance (i.e. scaled size of original dataset).
- **weights** – weight map to use for sampling each class
- **indices** – copy of the original list of sample indices provided in the constructor.
- **seeds** – dictionary of seeds to use when initializing RNG state.
- **epoch** – epoch number used to reinitialize the RNG to an epoch-specific state.

See also:

thelper.data.utils.create_loaders()
thelper.data.utils.get_class_weights()

__init__(*indices*, *labels*, *weights*, *seeds=None*, *epoch=0*)

Receives sample indices, labels, rebalancing strategy, and dataset scaling factor.

This function will validate all input arguments, parse and categorize samples according to labels, initialize rebalancing parameters, and determine sample counts for each valid class. Note that an empty list

of indices is an acceptable input; the resulting object will also create and empty list of samples when `__iter__` is called.

Parameters

- **indices** – list of integers representing the indices of samples of interest in the dataset.
- **labels** – list of labels tied to the list of indices (must be the same length).
- **weights** – weight map to use for sampling each class.
- **seeds** – dictionary of seeds to use when initializing RNG state.
- **epoch** – epoch number used to reinitialize the RNG to an epoch-specific state.

set_epoch(*epoch=0*)
Sets the current epoch number in order to offset the RNG state for sampling.

class thelper.data.samplers.**SubsetRandomSampler**(*indices*, *seeds=None*, *epoch=0*, *scale=1.0*)
Bases: sphinx.ext.autodoc.importer._MockObject

Samples elements randomly from a given list of indices, without replacement.

This specialization handles seeding based on the epoch number, and scaling (via duplication/decimation) of samples.

Parameters

- **indices** (*list*) – a list of indices
- **seeds** (*dict*) – dictionary of seeds to use when initializing RNG state.
- **epoch** (*int*) – epoch number used to reinitialize the RNG to an epoch-specific state.
- **scale** (*float*) – scaling factor used to increase/decrease the final number of samples.

__init__(*indices*, *seeds=None*, *epoch=0*, *scale=1.0*)
Initialize self. See help(type(self)) for accurate signature.

set_epoch(*epoch=0*)
Sets the current epoch number in order to offset the RNG state for sampling.

class thelper.data.samplers.**SubsetSequentialSampler**(*indices*)
Bases: sphinx.ext.autodoc.importer._MockObject

Samples element indices sequentially, always in the same order.

Parameters **indices** (*list*) – a list of indices

__init__(*indices*)
Initialize self. See help(type(self)) for accurate signature.

class thelper.data.samplers.**WeightedSubsetRandomSampler**(*indices*, *labels*, *stype='uniform'*, *scale=1.0*, *seeds=None*, *epoch=0*)
Bases: sphinx.ext.autodoc.importer._MockObject

Provides a rebalanced list of sample indices to use in a data loader.

Given a list of sample indices and the corresponding list of class labels, this sampler will produce a new list of indices that rebalances the distribution of samples according to a specified strategy. It can also optionally scale the dataset's total sample count to avoid undersampling large classes as smaller ones get bigger.

The currently implemented strategies are:

- `random`: will return a list of randomly picked samples based on the multinomial distribution of the initial class weights. This sampling is done with replacement, meaning that each index is picked independently of the already-picked ones.
- `uniform`: will rebalance the dataset by normalizing the sample count of all classes, oversampling and undersampling as required to distribute all samples equally. All removed or duplicated samples are selected randomly without replacement whenever possible.
- `root`: will rebalance the dataset by normalizing class weight using an n-th degree root. More specifically, for a list of initial class weights $W^0 = \{w_1^0, w_2^0, ...w_n^0\}$, we compute the adjusted weight w_i of each class via:

$$w_i = \frac{\sqrt[n]{w_i^0}}{\sum_j \sqrt[n]{w_j^0}}$$

  Then, according to the new distribution of weights, all classes are oversampled and undersampled as required to reobtain the dataset's total sample count (which may be scaled). All removed or duplicated samples are selected randomly without replacement whenever possible.

  Note that with the `root` strategy, if a very large root degree `n` is used, this strategy is equivalent to `uniform`. If the degree is one, the original weights will be used for sampling. The `root` strategy essentially provides a flexible solution to rebalance very uneven label sets where uniform over/undersampling would be too aggressive.

By default, this interface will try to keep the dataset size constant and balance oversampling with undersampling. If undersampling is undesired, the user can increase the total dataset size via a scale factor. Finally, note that the rebalanced list of indices is generated by this interface every time the `__iter__` function is called, meaning two consecutive lists might not contain the exact same indices.

Example configuration file:

```
# ...
# the sampler is defined inside the 'loaders' field
"loaders": {
    # ...
    # this field is completely optional, and can be omitted entirely
    "sampler": {
        # the type of the sampler we want to instantiate
        "type": "thelper.data.samplers.WeightedSubsetRandomSampler",
        # the parameters passed to the sampler's constructor
        "params": {
            "stype": "root3",
            "scale": 1.2
        },
        # specifies whether the sampler should receive class labels
        "pass_labels": true
    },
    # ...
},
# ...
```

Variables

- **nb_samples** – total number of samples to rebalance (i.e. scaled size of original dataset).
- **label_groups** – map that splits all samples indices into groups based on labels.
- **stype** – name of the rebalancing strategy to use.
- **indices** – copy of the original list of sample indices provided in the constructor.

- **sample_weights** – list of weights used for random sampling.
- **label_counts** – number of samples in each class for the `uniform` and `root` strategies.
- **seeds** – dictionary of seeds to use when initializing RNG state.
- **epoch** – epoch number used to reinitialize the RNG to an epoch-specific state.

See also:

thelper.data.utils.create_loaders()
thelper.data.utils.get_class_weights()

__init__ (*indices*, *labels*, *stype='uniform'*, *scale=1.0*, *seeds=None*, *epoch=0*)
Receives sample indices, labels, rebalancing strategy, and dataset scaling factor.

This function will validate all input arguments, parse and categorize samples according to labels, initialize rebalancing parameters, and determine sample counts for each valid class. Note that an empty list of indices is an acceptable input; the resulting object will also create and empty list of samples when `__iter__` is called.

Parameters

- **indices** – list of integers representing the indices of samples of interest in the dataset.
- **labels** – list of labels tied to the list of indices (must be the same length).
- **stype** – rebalancing strategy given as a string. Should be either "random", "uniform", or "rootX", where the 'X' is the degree to use in the root computation (float).
- **scale** – scaling factor used to increase/decrease the final number of sample indices to generate while rebalancing.
- **seeds** – dictionary of seeds to use when initializing RNG state.
- **epoch** – epoch number used to reinitialize the RNG to an epoch-specific state.

set_epoch (*epoch=0*)
Sets the current epoch number in order to offset the RNG state for sampling.

thelper.data.utils module

Dataset utility functions and tools.

This module contains utility functions and tools used to instantiate data loaders and parsers.

thelper.data.utils.**create_hdf5** (*archive_path*, *task*, *train_loader*, *valid_loader*, *test_loader*, *compression=None*, *config_backup=None*)
Saves the samples loaded from train/valid/test data loaders into an HDF5 archive.

The loaded minibatches are decomposed into individual samples. The keys provided via the task interface are used to fetch elements (input, groundtruth, ...) from the samples, and save them in the archive. The archive will contain three groups (*train*, *valid*, and *test*), and each group will contain a dataset for each element originally found in the samples.

Note that the compression operates at the sample level, not at the dataset level. This means that elements of each sample will be compressed individually, not as an array. Therefore, if you are trying to compress very correlated samples (e.g. frames in a video sequence), this approach will be pretty bad.

Parameters

- **archive_path** – path pointing where the HDF5 archive should be created.
- **task** – task object that defines the input, groundtruth, and meta keys tied to elements that should be parsed from loaded samples and saved in the HDF5 archive.
- **train_loader** – training data loader (can be *None*).
- **valid_loader** – validation data loader (can be *None*).
- **test_loader** – testing data loader (can be *None*).
- **compression** – the compression configuration dictionary that will be parsed to determine how sample elements should be compressed. If a mapping is missing, that element will not be compressed.
- **config_backup** – optional session configuration file that should be saved in the HDF5 archive.

Example compression configuration:

```
# the config is given as a dictionary
{
    # each field is a key that corresponds to an element in each sample
    "key1": {
        # the 'type' identifies the compression approach to use
        # (see thelper.utils.encode_data for more information)
        "type": "jpg",
        # extra parameters might be needed to encode the data
        # (see thelper.utils.encode_data for more information)
        "encode_params": {}
        # these parameters are packed and kept for decoding
        # (see thelper.utils.decode_data for more information)
        "decode_params": {"flags": "cv.IMREAD_COLOR"}
    },
    "key2": {
        # this explicitly means that no encoding should be performed
        "type": "none"
    },
    ...
    # if a key is missing, its elements will not be compressed
}
```

See also:

thelper.cli.split_data()
thelper.data.parsers.HDF5Dataset
thelper.utils.encode_data()
thelper.utils.decode_data()

thelper.data.utils.**create_loaders**(*config*, *save_dir=None*)

Prepares the task and data loaders for a model trainer based on a provided data configuration.

This function will parse a configuration dictionary and extract all the information required to instantiate the requested dataset parsers. Then, combining the task metadata of all these parsers, it will evenly split the available samples into three sets (training, validation, test) to be handled by different data loaders. These will finally be returned along with the (global) task object.

The configuration dictionary is expected to contain two fields: `loaders`, which specifies all parameters required for establishing the dataset split, shuffling seeds, and batch size (these are listed and detailed below); and `datasets`, which lists the dataset parser interfaces to instantiate as well as their parameters. For more information on the `datasets` field, refer to *thelper.data.utils.create_parsers()*.

The parameters expected in the 'loaders' configuration field are the following:

- `<train_/valid_/test_>batch_size` (mandatory): specifies the (mini)batch size to use in data loaders. If you get an 'out of memory' error at runtime, try reducing it.
- `<train_/valid_/test_>collate_fn` (optional): specifies the collate function to use in data loaders. The default one is typically fine, but some datasets might require a custom function.
- `shuffle` (optional, default=True): specifies whether the data loaders should shuffle their samples or not.
- `test_seed` (optional): specifies the RNG seed to use when splitting test data. If no seed is specified, the RNG will be initialized with a device-specific or time-related seed.
- `valid_seed` (optional): specifies the RNG seed to use when splitting validation data. If no seed is specified, the RNG will be initialized with a device-specific or time-related seed.
- `torch_seed` (optional): specifies the RNG seed to use for torch-related stochastic operations (e.g. for data augmentation). If no seed is specified, the RNG will be initialized with a device-specific or time-related seed.
- `numpy_seed` (optional): specifies the RNG seed to use for numpy-related stochastic operations (e.g. for data augmentation). If no seed is specified, the RNG will be initialized with a device-specific or time-related seed.
- `random_seed` (optional): specifies the RNG seed to use for stochastic operations with python's 'random' package. If no seed is specified, the RNG will be initialized with a device-specific or time-related seed.
- `workers` (optional, default=1): specifies the number of threads to use to preload batches in parallel; can be 0 (loading will be on main thread), or an integer >= 1.
- `pin_memory` (optional, default=False): specifies whether the data loaders will copy tensors into CUDA-pinned memory before returning them.
- `drop_last` (optional, default=False): specifies whether to drop the last incomplete batch or not if the dataset size is not a multiple of the batch size.
- `sampler` (optional): specifies a type of sampler and its constructor parameters to be used in the data loaders. This can be used for example to help rebalance a dataset based on its class distribution. See *thelper.data.samplers* for more information.
- `augments` (optional): provides a list of transformation operations used to augment all samples of a dataset. See *thelper.transforms.utils.load_augments()* for more info.
- `train_augments` (optional): provides a list of transformation operations used to augment the training samples of a dataset. See *thelper.transforms.utils.load_augments()* for more info.
- `valid_augments` (optional): provides a list of transformation operations used to augment the validation samples of a dataset. See *thelper.transforms.utils.load_augments()* for more info.
- `test_augments` (optional): provides a list of transformation operations used to augment the test samples of a dataset. See *thelper.transforms.utils.load_augments()* for more info.
- `eval_augments` (optional): provides a list of transformation operations used to augment the validation and test samples of a dataset. See *thelper.transforms.utils.load_augments()* for more info.

- `base_transforms` (optional): provides a list of transformation operations to apply to all loaded samples. This list will be passed to the constructor of all instantiated dataset parsers. See *thelper.transforms.utils.load_transforms()* for more info.
- `train_split` (optional): provides the proportion of samples of each dataset to hand off to the training data loader. These proportions are given in a dictionary format (`name: ratio`).
- `valid_split` (optional): provides the proportion of samples of each dataset to hand off to the validation data loader. These proportions are given in a dictionary format (`name: ratio`).
- `test_split` (optional): provides the proportion of samples of each dataset to hand off to the test data loader. These proportions are given in a dictionary format (`name: ratio`).
- `skip_verif` (optional, default=True): specifies whether the dataset split should be verified if resuming a session by parsing the log files generated earlier.
- `skip_split_norm` (optional, default=False): specifies whether the question about normalizing the split ratios should be skipped or not.
- `skip_class_balancing` (optional, default=False): specifies whether the balancing of class labels should be skipped in case the task is classification-related.

Example configuration file:

```
# ...
"loaders": {
    "batch_size": 128,  # batch size to use in data loaders
    "shuffle": true,  # specifies that the data should be shuffled
    "workers": 4,  # number of threads to pre-fetch data batches with
    "train_sampler": {  # we can use a data sampler to rebalance classes␣
↪(optional)
        # see e.g. 'thelper.data.samplers.WeightedSubsetRandomSampler'
        # ...
    },
    "train_augments": { # training data augmentation operations
        # see 'thelper.transforms.utils.load_augments'
        # ...
    },
    "eval_augments": { # evaluation (valid/test) data augmentation operations
        # see 'thelper.transforms.utils.load_augments'
        # ...
    },
    "base_transforms": { # global sample transformation operations
        # see 'thelper.transforms.utils.load_transforms'
        # ...
    },
    # optionally indicate how to resolve dataset loader task vs model task␣
↪incompatibility if any
    # leave blank to get more details about each case during runtime if this␣
↪situation happens
    "task_compat_mode": "old|new|compat",
    # finally, we define a 80%-10%-10% split for our data
    # (we could instead use one dataset for training and one for testing)
    "train_split": {
        "dataset_A": 0.8
        "dataset_B": 0.8
    },
    "valid_split": {
        "dataset_A": 0.1
        "dataset_B": 0.1
```

(continues on next page)

(continued from previous page)

```
    },
    "test_split": {
        "dataset_A": 0.1
        "dataset_B": 0.1
    }
    # (note that the dataset names above are defined in the field below)
},
"datasets": {
    "dataset_A": {
        # type of dataset interface to instantiate
        "type": "...",
        "params": {
            # ...
        }
    },
    "dataset_B": {
        # type of dataset interface to instantiate
        "type": "...",
        "params": {
            # ...
        },
        # if it does not derive from 'thelper.data.parsers.Dataset', a task is␣
↪needed:
        "task": {
            # this type must derive from 'thelper.tasks.Task'
            "type": "...",
            "params": {
                # ...
            }
        }
    },
    # ...
},
# ...
```

Parameters

- **config** – a dictionary that provides all required data configuration information under two fields, namely 'datasets' and 'loaders'.
- **save_dir** – the path to the root directory where the session directory should be saved. Note that this is not the path to the session directory itself, but its parent, which may also contain other session directories.

Returns *A 4-element tuple that contains* – 1) the global task object to specialize models and trainers with; 2) the training data loader; 3) the validation data loader; and 4) the test data loader.

See also:

thelper.data.utils.create_parsers()
thelper.transforms.utils.load_augments()
thelper.transforms.utils.load_transforms()

thelper.data.utils.**create_parsers**(*config*, *base_transforms=None*)

Instantiates dataset parsers based on a provided dictionary.

This function will instantiate dataset parsers as defined in a name-type-param dictionary. If multiple datasets are instantiated, this function will also verify their task compatibility and return the global task. The dataset interfaces themselves should be derived from *thelper.data.parsers.Dataset*, be compatible with *thelper.data.parsers.ExternalDataset*, or should provide a 'task' field specifying all the information related to sample dictionary keys and model i/o.

The provided configuration will be parsed for a 'datasets' dictionary entry. The keys in this dictionary are treated as unique dataset names and are used for lookups. The value associated to each key (or dataset name) should be a type-params dictionary that can be parsed to instantiate the dataset interface.

An example configuration dictionary is given in *thelper.data.utils.create_loaders()*.

Parameters

- **config** – a dictionary that provides unique dataset names and parameters needed for instantiation under the 'datasets' field.
- **base_transforms** – the transform operation that should be applied to all loaded samples, and that will be provided to the constructor of all instantiated dataset parsers.

Returns *A 2-element tuple that contains* – 1) the list of dataset interfaces/parsers that were instantiated; and 2) a task object compatible with all of those (see *thelper.tasks.utils.Task* for more information).

See also:

thelper.data.utils.create_loaders()
thelper.data.parsers.Dataset
thelper.data.parsers.ExternalDataset
thelper.tasks.utils.Task

thelper.data.utils.**get_class_weights**(*label_map*, *stype='linear'*, *maxw=inf*, *minw=0.0*, *norm=True*, *invmax=False*)

Returns a map of label weights that may be adjusted based on a given rebalancing strategy.

Parameters

- **label_map** – map of index lists or sample counts tied to class labels.
- **stype** – weighting strategy ('uniform', 'linear', or 'rootX'). Using 'uniform' will provide a uniform map of weights. Using 'linear' will return the actual weights, unmodified. Using 'rootX' will rebalance the weights according to factor 'X'. See *thelper.data.samplers.WeightedSubsetRandomSampler* for more information on these strategies.
- **maxw** – maximum allowed weight value (applied after invmax, if required).
- **minw** – minimum allowed weight value (applied after invmax, if required).
- **norm** – specifies whether the returned weights should be normalized (default=True, i.e. normalized).
- **invmax** – specifies whether to max-invert the weight vector (thus creating cost factors) or not. Not compatible with `norm` (it would return weights again instead of factors).

Returns Map of weights tied to class labels.

See also:

thelper.data.samplers.WeightedSubsetRandomSampler

6.1.2 thelper.gui package

Graphical User Interface (GUI) package.

This package contains various tools and annotators used to simplify user interactions with data or models. Since the training framework is CLI-based, these tools are not used to train models, but can be helpful when debugging them. They can also be used to annotate and explore datasets.

Submodules

thelper.gui.annotators module

Graphical User Interface (GUI) annotator module.

This module contains various annotators that define interactive ways to visualize and annotate data loaded via a dataset parser.

class thelper.gui.annotators.**Annotator**(*session_name*, *config*, *save_dir*, *datasets*)

Bases: object

Abstract annotation tool used to define common functions for all GUI-based I/O.

Example configuration file:

```
# ...
"annotator": {
    # type of annotator to instantiate
    "type": "thelper.gui.ImageSegmentAnnotator",
    # ...
    # provide all extra parameters to the specialized anntator here
    "params": {
        # ...
    }
},
# ...
```

See also:

thelper.gui.annotators.ImageSegmentAnnotator
thelper.gui.utils.create_annotator()

__init__(*session_name*, *config*, *save_dir*, *datasets*)

Receives the annotator configuration dictionary, parses it, and sets up the basic session attributes.

run()

Displays the GUI tool and blocks until it it closed by the user.

class thelper.gui.annotators.**ImageSegmentAnnotator**(*session_name*, *config*, *save_dir*, *datasets*)

Bases: *thelper.gui.annotators.Annotator*

Annotator interface specialized for image segmentation annotation generation.

This interface will create a GUI tool with a brush and a zoomed tooltip window that allows images to be painted over with predetermined class labels. The generated masks will then be saved in the session directory as PNG images.

The configuration is expected to provide values for at least the following parameters:

- `sample_input_key`: specifies the key to use when extracting images from loaded samples. This is typically a string defined by the dataset parser.
- `labels`: provides a list of labels that will be available to use in the GUI. These labels are expected to be given as dictionaries that each define an `id` (uint8 value used in output masks), a `name` (string used for display/lookup purposes), and a color (3-element integer tuple).

Other parameters can also be provided to alter the GUI's default behavior:

- `default_brush_radius`: the size of the brush at startup (default=10).
- `background_id`: the integer id to use for the background label (default=0).
- `window_zoom_crop_size`: the crop size displayed in the zoom tooltip (default=250x250).
- `window_zoom_size`: the size of the zoom tooltip window (default=500x500).
- `zoom_interp_type`: the interpolation type to use when zooming (default=cv2.INTER_NEAREST).
- `start_sample_idx`: the index of the first sample to display (default=0).
- `window_name`: the name of the main display window (default=image-segm-annotator).
- `window_size`: the size of the main display window (default=1000).
- `brush_thickness`: the size of the GUI brush tooltip border display (default=2).
- `gui_bar_size`: the width of the GUI bar displayed on top of the main window (default=50).
- `default_mask_opacity`: the default opacity of the segmentation mask (default=0.3).
- `default_fill_id`: the label id to fill all new masks with (default=0).

See also:

thelper.gui.annotators.Annotator

BRUSH_SIZE = None

class Brush(*config*)

Bases: `object`

Brush manager used to refresh/draw mask contents based on mouse input.

__init__(*config*)

Parses the input config and extracts brush-related parameters.

static draw_stroke(*mask*, *label_id*, *start*, *end*)

Draws a brush stroke on the mask with a given label id between two points.

refresh(*mask*, *label*)

Fetches the latest mouse state and updates the mask if necessary.

CURRENT_KEY = -1

GUI_BAR_SIZE = None

GUI_DIRTY = True

LATEST_PT = (-1, -1)

LATEST_RAW_PT = (-1, -1)

MASK_DIRTY = True

MOUSE_FLAGS = 0

WINDOW_SIZE = None

class ZoomTooltip(*config*)
Bases: object

Zoom tooltip manager used to visualize image details based on mouse location.

__init__(*config*)
Parses the input config and extracts zoom-related parameters.

refresh(*image*, *mask*, *mask_colormap*, *mask_opacity*, *coords*)
Fetches the latest mouse position and updates the zoom window tooltip.

__init__(*session_name*, *config*, *save_dir*, *datasets*)
Parses the input samples and initializes the anntator GUI elements.

get_mask_path(*index*)
Returns the path where the mask of a specific sample should be located.

handle_keys()
Fetches the latest keyboard press and updates the annotator state accordingly.

load(*index*)
Loads the image and mask associated to a specific sample.

static on_mouse(*event*, *x*, *y*, *flags*, *param*)
Callback entrypoint for opencv to register mouse movement/clicks.

static on_press(*key*)
Callback entrypoint for pynput to register keyboard presses.

refresh_gui()
Updates and displays the main window based on the latest changes.

refresh_layers()
Updates the image, mask, and tool display layers based on the latest changes.

run()
Displays the main window and other GUI elements in a loop until it is closed by the user.

thelper.gui.utils module

Graphical User Interface (GUI) utility module.

This module contains various tools and utilities used to instantiate annotators and GUI elements.

thelper.gui.utils.**create_annotator**(*session_name*, *save_dir*, *config*, *datasets*)
Instantiates a GUI annotation tool based on the type contained in the config dictionary.

The tool type is expected to be in the configuration dictionary's *annotator* field, under the *type* key. For more information on the configuration, refer to *thelper.gui.annotators.Annotator*. The instantiated type must be compatible with the constructor signature of *thelper.gui.annotators.Annotator*. The object's constructor will be given the full config dictionary.

Parameters

- **session_name** – name of the annotation session used for printing and to create output directories.
- **save_dir** – path to the session directory where annotations and other outputs will be saved.
- **config** – full configuration dictionary that will be parsed for annotator parameters.
- **datasets** – map of named dataset parsers that will provide the data to annotate.

Returns The fully-constructed annotator object, ready to begin annotation via its `run()` function.

See also:

thelper.gui.annotators.Annotator

thelper.gui.utils.**create_key_listener**(*callback*)
Returns a key press listener based on pynput.keyboard (used for mocking).

6.1.3 thelper.infer package

Infer package.

This package contains classes specialized for inference of models on various tasks.

Submodules

thelper.infer.base module

class thelper.infer.base.**Tester**(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)
Bases: *thelper.train.base.Trainer*

Base interface of a session runner for testing.

This call mostly delegates calls to existing Trainer implementation, but limiting their use to 'eval' methods to make sure that 'train' operations are not called by mistake.

See also:

thelper.train.base.Trainer

__init__(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)
Receives the trainer configuration dictionary, parses it, and sets up the session.

eval_epoch(*model*, *epoch*, *dev*, *loader*, *metrics*, *output_path*)
Evaluates the model using the provided objects.

Parameters

- **model** – the model with which to run inference that is already uploaded to the target device(s).
- **epoch** – the epoch index we are training for (0-based, and should normally only be 0 for single test pass).

- **dev** – the target device that tensors should be uploaded to (corresponding to model's device(s)).
- **loader** – the data loader used to get transformed test samples.
- **metrics** – the dictionary of metrics/consumers to report inference results (mostly loggers and basic report generator in this case since there shouldn't be ground truth labels to validate against).
- **output_path** – directory where output files should be written, if necessary.

test()

test_epoch(**args*, ***kwargs*)

train()

Starts the training process.

This function will train the model until the required number of epochs is reached, and then evaluate it on the test data. The setup of loggers, tensorboard writers is done here, so is model improvement tracking via monitored metrics. However, the code related to loss computation and back propagation is implemented in a derived class via *thelper.train.base.Trainer.train_epoch()*.

train_epoch(*model*, *epoch*, *dev*, *loss*, *optimizer*, *loader*, *metrics*, *output_path*)

Trains the model for a single epoch using the provided objects.

Parameters

- **model** – the model to train that is already uploaded to the target device(s).
- **epoch** – the epoch index we are training for (0-based).
- **dev** – the target device that tensors should be uploaded to.
- **loss** – the loss function used to evaluate model fidelity.
- **optimizer** – the optimizer used for back propagation.
- **loader** – the data loader used to get transformed training samples.
- **metrics** – the dictionary of metrics/consumers to update every iteration.
- **output_path** – directory where output files should be written, if necessary.

thelper.infer.base.**make_tester_from_trainer**(*trainer*)

thelper.infer.impl module

Explicit Tester definitions from existing Trainers.

thelper.infer.utils module

thelper.infer.utils.**create_tester**(*session_name*, *save_dir*, *config*, *model*, *task*, *loaders*, *ckpt_data=None*)

Instantiates the tester object based on the type contained in the config dictionary.

The tester type is expected to be in the configuration dictionary's *tester* field, under the *type* key. For backward compatibility, the fields *runner* and *trainer* will also be looked for. For more information on the configuration, refer to *thelper.train.base.Trainer*. The instantiated type must be compatible with the constructor signature of *thelper.train.base.Trainer*. The object's constructor will be given the full config dictionary and the checkpoint data for resuming the session (if available).

If the trainer type is missing, it will be automatically deduced based on the task object.

Parameters

- **session_name** – name of the training session used for printing and to create internal tensorboardX directories.
- **save_dir** – path to the session directory where logs and checkpoints will be saved.
- **config** – full configuration dictionary that will be parsed for trainer parameters and saved in checkpoints.
- **model** – model to train/evaluate; should be compatible with *thelper.nn.utils.Module*.
- **task** – global task interface defining the type of model and training goal for the session.
- **loaders** – a tuple containing the training/validation/test data loaders (a loader can be `None` if empty).
- **ckptdata** – raw checkpoint to parse data from when resuming a session (if `None`, will start from scratch).

Returns The fully-constructed trainer object, ready to begin model training/evaluation.

See also:

thelper.infer.base.Tester

6.1.4 thelper.nn package

Neural network and model package.

This package contains classes that define blocks and modules used in various neural network architectures. Most of these classes have been adapted from external sources; see their individual headers for more information.

Subpackages

thelper.nn.segmentation package

Neural network and model package for segemnation.

This package contains classes that define blocks and modules used in various neural network for segmentation

Submodules

thelper.nn.segmentation.base module

class thelper.nn.segmentation.base.**SegmModelBase**(*task*, *pretrained=False*)

Bases: *thelper.nn.utils.Module*

Base wrapper class for specialized segmentation models.

__init__(*task*, *pretrained=False*)

Note:

-

forward(*x*)
Transforms an input tensor in order to generate a prediction.

in_channels = None

model_cls = None

set_task(*task*)
Adapts the model to support a new task, replacing layers if needed.

thelper.nn.segmentation.deeplabv3 module

class thelper.nn.segmentation.deeplabv3.**DeepLabV3ResNet101**(*task*, *pretrained=False*)
Bases: *thelper.nn.segmentation.base.SegmModelBase*

This class is a thin wrapper for `torchvision.models.segmentation.deeplabv3_resnet101()` (`torchvision > 0.6`).

Note: Contributed by Mario Beaulieu <mario.beaulieu@crim.ca>.

See also:

Liang-Chieh et al., Rethinking Atrous Convolution for Semantic Image Segmentation [arXiv], 2017.

__init__(*task*, *pretrained=False*)

-

class thelper.nn.segmentation.deeplabv3.**DeepLabV3ResNet50**(*task*, *pretrained=False*)
Bases: *thelper.nn.segmentation.base.SegmModelBase*

This class is a thin wrapper for `torchvision.models.segmentation.deeplabv3_resnet101()` (`torchvision > 0.6`).

Note: Contributed by Mario Beaulieu <mario.beaulieu@crim.ca>.

See also:

Liang-Chieh et al., Rethinking Atrous Convolution for Semantic Image Segmentation [arXiv], 2017.

__init__(*task*, *pretrained=False*)

-

thelper.nn.segmentation.fcn module

class thelper.nn.segmentation.fcn.**FCNResNet101**(*task*, *pretrained=False*)
Bases: *thelper.nn.segmentation.base.SegmModelBase*

This class is a thin wrapper for `torchvision.models.segmentation.fcn_resnet50()` (`torchvision > 0.6`).

Note: Contributed by Mario Beaulieu <mario.beaulieu@crim.ca>.

See also:

Liang-Chieh et al., Rethinking Atrous Convolution for Semantic Image Segmentation [arXiv], 2017.

__init__(*task*, *pretrained=False*)

-

class thelper.nn.segmentation.fcn.**FCNResNet50**(*task*, *pretrained=False*)
Bases: *thelper.nn.segmentation.base.SegmModelBase*

This class is a thin wrapper for `torchvision.models.segmentation.fcn_resnet50()` (`torchvision > 0.6`).

Note: Contributed by Mario Beaulieu <mario.beaulieu@crim.ca>.

See also:

Liang-Chieh et al., Rethinking Atrous Convolution for Semantic Image Segmentation [arXiv], 2017.

__init__(*task*, *pretrained=False*)

-

thelper.nn.sr package

Neural network and model package for super resolution.

This package contains classes that define blocks and modules used in various neural network for super resolution architectures. Most of these classes have been adapted from external sources; see their individual headers for more information.

Submodules

thelper.nn.sr.srcnn module

class thelper.nn.sr.srcnn.**SRCNN**(*task*, *num_channels=1*, *base_filter=64*, *groups=1*)
Bases: *thelper.nn.utils.Module*

Implements the SRCNN architecture.

See also:

Dong et al., Image Super-Resolution Using Deep Convolutional Networks [arXiv], 2014.

__init__(*task*, *num_channels=1*, *base_filter=64*, *groups=1*)
Receives a task object to hold internally for model specialization.

forward(*x*)
Transforms an input tensor in order to generate a prediction.

set_task(*task*)
Adapts the model to support a new task, replacing layers if needed.

weight_init()

thelper.nn.sr.vdsr module

class thelper.nn.sr.vdsr.**VDSR**(*task*, *num_channels=1*, *base_filter=64*, *kernel_size0=3*, *num_residuals=18*, *groups=1*, *activation='relu'*, *norm='batch'*)
Bases: *thelper.nn.utils.Module*

Implements the VDSR architecture.

See also:

Kim et al., Accurate Image Super-Resolution Using Very Deep Convolutional Networks [arXiv], 2015.

__init__(*task*, *num_channels=1*, *base_filter=64*, *kernel_size0=3*, *num_residuals=18*, *groups=1*, *activation='relu'*, *norm='batch'*)
Receives a task object to hold internally for model specialization.

forward(*x*)
Transforms an input tensor in order to generate a prediction.

set_task(*task*)
Adapts the model to support a new task, replacing layers if needed.

weight_init()

Submodules

thelper.nn.common module

class thelper.nn.common.**ConvBlock**(*input_size*, *output_size*, *kernel_size=4*, *stride=2*, *padding=1*, *bias=True*, *activation='relu'*, *norm='batch'*, *groups=1*, *prelu_params=1*)
Bases: sphinx.ext.autodoc.importer._MockObject

__init__(*input_size*, *output_size*, *kernel_size=4*, *stride=2*, *padding=1*, *bias=True*, *activation='relu'*, *norm='batch'*, *groups=1*, *prelu_params=1*)
Initialize self. See help(type(self)) for accurate signature.

forward_act_bn(*x*)

forward_bn_act(*x*)

class thelper.nn.common.**DeconvBlock**(*input_size*, *output_size*, *kernel_size=4*, *stride=2*, *padding=1*, *bias=True*, *activation='relu'*, *norm='batch'*)
Bases: sphinx.ext.autodoc.importer._MockObject

__init__(*input_size*, *output_size*, *kernel_size=4*, *stride=2*, *padding=1*, *bias=True*, *activation='relu'*, *norm='batch'*)
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

class thelper.nn.common.**DenseBlock**(*input_size*, *output_size*, *bias=True*, *activation='relu'*, *norm='batch'*)
Bases: sphinx.ext.autodoc.importer._MockObject

__init__(*input_size*, *output_size*, *bias=True*, *activation='relu'*, *norm='batch'*)
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

class thelper.nn.common.**PSBlock**(*input_size*, *output_size*, *scale_factor*, *kernel_size=3*, *stride=1*, *padding=1*, *bias=True*, *activation='relu'*, *norm='batch'*)
Bases: sphinx.ext.autodoc.importer._MockObject

__init__(*input_size*, *output_size*, *scale_factor*, *kernel_size=3*, *stride=1*, *padding=1*, *bias=True*, *activation='relu'*, *norm='batch'*)
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

class thelper.nn.common.**ResNetBlock**(*num_filter*, *kernel_size=3*, *stride=1*, *padding=1*, *bias=True*, *activation='relu'*, *norm='batch'*)
Bases: sphinx.ext.autodoc.importer._MockObject

__init__(*num_filter*, *kernel_size=3*, *stride=1*, *padding=1*, *bias=True*, *activation='relu'*, *norm='batch'*)
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

class thelper.nn.common.**Upsample2xBlock**(*input_size*, *output_size*, *bias=True*, *upsample='deconv'*, *activation='relu'*, *norm='batch'*)
Bases: sphinx.ext.autodoc.importer._MockObject

__init__(*input_size*, *output_size*, *bias=True*, *upsample='deconv'*, *activation='relu'*, *norm='batch'*)
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

thelper.nn.common.**shave**(*imgs*, *border_size=0*)

thelper.nn.common.**weights_init_kaiming**(*m*)

thelper.nn.common.**weights_init_xavier**(*m*)

thelper.nn.coordconv module

class thelper.nn.coordconv.**AddCoords**(*centered=True*, *normalized=True*, *noise=None*, *radius_channel=False*, *scale=None*)
Bases: sphinx.ext.autodoc.importer._MockObject

Creates a torch-compatible layer that adds intrinsic coordinate layers to input tensors.

__init__(*centered=True*, *normalized=True*, *noise=None*, *radius_channel=False*, *scale=None*)
Initialize self. See help(type(self)) for accurate signature.

forward(*in_tensor*)

class thelper.nn.coordconv.**CoordConv2d**(*in_channels*, **args*, *centered=True*, *normalized=True*, *noise=None*, *radius_channel=False*, *scale=None*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

CoordConv-equivalent of torch's default Conv2d model layer.

See also:

Liu et al., An Intriguing Failing of Convolutional Neural Networks and the CoordConv Solution' <https://arxiv.org/abs/1807.03247>'_ [arXiv], 2018.

__init__(*in_channels*, **args*, *centered=True*, *normalized=True*, *noise=None*, *radius_channel=False*, *scale=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

forward(*in_tensor*)

class thelper.nn.coordconv.**CoordConvTranspose2d**(*in_channels*, **args*, *centered=True*, *normalized=True*, *noise=None*, *radius_channel=False*, *scale=None*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

CoordConv-equivalent of torch's default ConvTranspose2d model layer.

See also:

Liu et al., An Intriguing Failing of Convolutional Neural Networks and the CoordConv Solution' <https://arxiv.org/abs/1807.03247>'_ [arXiv], 2018.

__init__(*in_channels*, **args*, *centered=True*, *normalized=True*, *noise=None*, *radius_channel=False*, *scale=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

forward(*in_tensor*)

thelper.nn.coordconv.**get_coords_map**(*height*, *width*, *centered=True*, *normalized=True*, *noise=None*, *dtype=<sphinx.ext.autodoc.importer._MockObject object>*)
Returns a HxW intrinsic coordinates map tensor (shape=2xHxW).

thelper.nn.coordconv.**make_conv2d**(**args*, *coordconv=False*, *centered=True*, *normalized=True*, *noise=None*, *radius_channel=False*, *scale=None*, ***kwargs*)
Creates a 2D convolution layer with optional CoordConv support.

thelper.nn.coordconv.**swap_coordconv_layers**(*module*, *centered=True*, *normalized=True*, *noise=None*, *radius_channel=False*, *scale=None*)
Modifies the provided module by swapping Conv2d layers for CoordConv-equivalent layers.

thelper.nn.densenet module

class thelper.nn.densenet.**DenseNet**(*growth_rate=32*, *block_config=(6, 12, 24, 16)*, *num_init_features=64*, *bn_size=4*, *drop_rate=0*, *num_classes=1000*)

Bases: sphinx.ext.autodoc.importer._MockObject

Densenet-BC model class, based on "Densely Connected Convolutional Networks"

Parameters

- **growth_rate** (*int*) – how many filters to add each layer (*k* in paper)
- **block_config** (*list of 4 ints*) – how many layers in each pooling block
- **num_init_features** (*int*) – the number of filters to learn in the first convolution layer
- **bn_size** (*int*) – multiplicative factor for number of bottle neck layers (i.e. *bn_size * k features* in the bottleneck layer)
- **drop_rate** (*float*) – dropout rate after each dense layer
- **num_classes** (*int*) – number of classification classes

__init__(*growth_rate=32*, *block_config=(6, 12, 24, 16)*, *num_init_features=64*, *bn_size=4*, *drop_rate=0*, *num_classes=1000*)

Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

thelper.nn.densenet.**densenet121**(*pretrained=False*, ***kwargs*)

Densenet-121 model from "Densely Connected Convolutional Networks"

Parameters **pretrained** (*bool*) – If True, returns a model pre-trained on ImageNet

thelper.nn.densenet.**densenet169**(*pretrained=False*, ***kwargs*)

Densenet-169 model from "Densely Connected Convolutional Networks"

Parameters **pretrained** (*bool*) – If True, returns a model pre-trained on ImageNet

thelper.nn.densenet.**densenet201**(*pretrained=False*, ***kwargs*)

Densenet-201 model from "Densely Connected Convolutional Networks"

Parameters **pretrained** (*bool*) – If True, returns a model pre-trained on ImageNet

thelper.nn.densenet.**densenet161**(*pretrained=False*, ***kwargs*)

Densenet-161 model from "Densely Connected Convolutional Networks"

Parameters **pretrained** (*bool*) – If True, returns a model pre-trained on ImageNet

thelper.nn.efficientnet module

class thelper.nn.efficientnet.**EfficientNet**(*task*, *num*, *pretrained=False*)

Bases: *thelper.nn.utils.Module*

__init__(*task*, *num*, *pretrained=False*)

Receives a task object to hold internally for model specialization.

forward(*x*)

Transforms an input tensor in order to generate a prediction.

set_task(*task*)

Adapts the model to support a new task, replacing layers if needed.

class thelper.nn.efficientnet.**FCEfficientNet**(*task*, *ckptdata*, *map_location='cpu'*, *avgpool_size=0*)

Bases: *thelper.nn.efficientnet.EfficientNet*

__init__(*task*, *ckptdata*, *map_location='cpu'*, *avgpool_size=0*)
Receives a task object to hold internally for model specialization.

forward(*x*)
Transforms an input tensor in order to generate a prediction.

set_task(*task*)
Adapts the model to support a new task, replacing layers if needed.

thelper.nn.fcn module

class thelper.nn.fcn.**FCN32s**(*task*, *init_vgg16=True*)

Bases: *thelper.nn.utils.Module*

__init__(*task*, *init_vgg16=True*)
Receives a task object to hold internally for model specialization.

forward(*x*)
Transforms an input tensor in order to generate a prediction.

init_vgg16_params(*vgg16*, *copy_fc8=True*)

set_task(*task*)
Adapts the model to support a new task, replacing layers if needed.

thelper.nn.fcn.**get_upsampling_weight**(*in_channels*, *out_channels*, *kernel_size*)

thelper.nn.inceptionresnetv2 module

class thelper.nn.inceptionresnetv2.**BasicConv2d**(*in_planes*, *out_planes*, *kernel_size*, *stride*, *padding=0*)

Bases: sphinx.ext.autodoc.importer._MockObject

__init__(*in_planes*, *out_planes*, *kernel_size*, *stride*, *padding=0*)
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

class thelper.nn.inceptionresnetv2.**Block17**(*scale=1.0*)

Bases: sphinx.ext.autodoc.importer._MockObject

__init__(*scale=1.0*)
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

class thelper.nn.inceptionresnetv2.**Block35**(*scale=1.0*)

Bases: sphinx.ext.autodoc.importer._MockObject

__init__(*scale=1.0*)
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

class thelper.nn.inceptionresnetv2.**Block8**(*scale=1.0*, *noReLU=False*)

Bases: sphinx.ext.autodoc.importer._MockObject

__init__(*scale=1.0*, *noReLU=False*)
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

class thelper.nn.inceptionresnetv2.**InceptionResNetV2**(*task*, *input_channels=3*)
Bases: *thelper.nn.utils.Module*

__init__(*task*, *input_channels=3*)
Receives a task object to hold internally for model specialization.

features(*input*)

forward(*input*)
Transforms an input tensor in order to generate a prediction.

logits(*features*)

set_task(*task*)
Adapts the model to support a new task, replacing layers if needed.

class thelper.nn.inceptionresnetv2.**Mixed_5b**
Bases: sphinx.ext.autodoc.importer._MockObject

__init__()
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

class thelper.nn.inceptionresnetv2.**Mixed_6a**
Bases: sphinx.ext.autodoc.importer._MockObject

__init__()
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

class thelper.nn.inceptionresnetv2.**Mixed_7a**
Bases: sphinx.ext.autodoc.importer._MockObject

__init__()
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

thelper.nn.lenet module

class thelper.nn.lenet.**LeNet**(*task*, *input_shape=(1, 28, 28)*, *conv1_filters=6*, *conv2_filters=16*, *hidden1_size=120*, *hidden2_size=84*, *output_size=10*)
Bases: *thelper.nn.utils.Module*

LeNet CNN implementation.

See http://yann.lecun.com/exdb/lenet/ for more information.

This is NOT a modern architecture; it is only provided here for tutorial purposes.

__init__(*task*, *input_shape=(1, 28, 28)*, *conv1_filters=6*, *conv2_filters=16*, *hidden1_size=120*, *hidden2_size=84*, *output_size=10*)
Receives a task object to hold internally for model specialization.

forward(*x*)
Transforms an input tensor in order to generate a prediction.

set_task(*task*)
Adapts the model to support a new task, replacing layers if needed.

thelper.nn.mobilenet module

class thelper.nn.mobilenet.**InvertedResidual**(*inp*, *oup*, *stride*, *expand_ratio*)
Bases: sphinx.ext.autodoc.importer._MockObject

__init__(*inp*, *oup*, *stride*, *expand_ratio*)
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

class thelper.nn.mobilenet.**MobileNetV2**(*task*, *input_size=224*, *width_mult=1.0*)
Bases: *thelper.nn.utils.Module*

__init__(*task*, *input_size=224*, *width_mult=1.0*)
Receives a task object to hold internally for model specialization.

forward(*x*)
Transforms an input tensor in order to generate a prediction.

set_task(*task*)
Adapts the model to support a new task, replacing layers if needed.

thelper.nn.mobilenet.**conv_1x1_bn**(*inp*, *oup*)

thelper.nn.mobilenet.**conv_bn**(*inp*, *oup*, *stride*)

thelper.nn.resnet module

class thelper.nn.resnet.**AutoEncoderResNet**(*task*, *output_pads=None*, ***kwargs*)
Bases: *thelper.nn.resnet.ResNet*

Autoencoder-classifier architecture based on ResNet blocks+layers configurations.

__init__(*task*, *output_pads=None*, ***kwargs*)
Receives a task object to hold internally for model specialization.

forward(*input*)
Transforms an input tensor in order to generate a prediction.

class thelper.nn.resnet.**AutoEncoderSkipResNet**(*task*, *output_pads=None*, *decoder_dropout=False*, *dropout_prob=0.1*, ***kwargs*)
Bases: *thelper.nn.resnet.ResNet*

Autoencoder-U-Net architecture based on ResNet blocks+layers configurations.

__init__(*task*, *output_pads=None*, *decoder_dropout=False*, *dropout_prob=0.1*, ***kwargs*)
Receives a task object to hold internally for model specialization.

forward(*input*)
Transforms an input tensor in order to generate a prediction.

class thelper.nn.resnet.**BasicBlock**(*inplanes*, *planes*, *stride=1*, *downsample=None*, *coordconv=False*, *radius_channel=True*, *activation='relu'*)
Bases: *thelper.nn.resnet.Module*

__init__(*inplanes*, *planes*, *stride=1*, *downsample=None*, *coordconv=False*, *radius_channel=True*, *activation='relu'*)
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

class thelper.nn.resnet.**Bottleneck**(*inplanes*, *planes*, *stride=1*, *downsample=None*, *coordconv=False*, *radius_channel=True*, *activation='relu'*)
Bases: *thelper.nn.resnet.Module*

__init__(*inplanes*, *planes*, *stride=1*, *downsample=None*, *coordconv=False*, *radius_channel=True*, *activation='relu'*)
Initialize self. See help(type(self)) for accurate signature.

expansion = 4

forward(*x*)

class thelper.nn.resnet.**ConvTailNet**(*n_inputs*, *num_classes*)
Bases: sphinx.ext.autodoc.importer._MockObject

DEPRECATED. Will be removed in a future version.

__init__(*n_inputs*, *num_classes*)
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

class thelper.nn.resnet.**FCResNet**(*task*, *ckptdata*, *map_location='cpu'*, *avgpool_size=0*)
Bases: *thelper.nn.resnet.ResNet*

Fully Convolutional ResNet converter for pre-trained classification models.

__init__(*task*, *ckptdata*, *map_location='cpu'*, *avgpool_size=0*)
Receives a task object to hold internally for model specialization.

forward(*x*)
Transforms an input tensor in order to generate a prediction.

set_task(*task*)
Adapts the model to support a new task, replacing layers if needed.

class thelper.nn.resnet.**FakeModule**(**args*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*inputs*)

class thelper.nn.resnet.**Module**(*inplanes*, *planes*, *stride=1*, *downsample=None*, *coordconv=False*, *radius_channel=True*)
Bases: sphinx.ext.autodoc.importer._MockObject

__init__(*inplanes*, *planes*, *stride=1*, *downsample=None*, *coordconv=False*, *radius_channel=True*)
Initialize self. See help(type(self)) for accurate signature.

expansion = 1

class thelper.nn.resnet.**ResNet**(*task, block='thelper.nn.resnet.BasicBlock', layers=[3, 4, 6, 3], strides=[1, 2, 2, 2], input_channels=3, flexible_input_res=False, pool_size=7, head_type=None, coordconv=False, radius_channel=True, activation='relu', skip_max_pool=False, pretrained=False, conv1_config=[7, 2, 3]*)
Bases: *thelper.nn.utils.Module*

__init__(*task, block='thelper.nn.resnet.BasicBlock', layers=[3, 4, 6, 3], strides=[1, 2, 2, 2], input_channels=3, flexible_input_res=False, pool_size=7, head_type=None, coordconv=False, radius_channel=True, activation='relu', skip_max_pool=False, pretrained=False, conv1_config=[7, 2, 3]*)
Receives a task object to hold internally for model specialization.

forward(*x*)
Transforms an input tensor in order to generate a prediction.

get_embedding(*x*, *pool=True*)

set_task(*task*)
Adapts the model to support a new task, replacing layers if needed.

class thelper.nn.resnet.**ResNetFullyConv**(*task, block='thelper.nn.resnet.BasicBlock', layers=[3, 4, 6, 3], strides=[1, 2, 2, 2], input_channels=3, flexible_input_res=False, pool_size=7, coordconv=False, radius_channel=True, pretrained=False*)
Bases: *thelper.nn.resnet.ResNet*

DEPRECATED. Will be removed in a future version. Use the torchvision segmentation models or the ResNet above instead.

__init__(*task, block='thelper.nn.resnet.BasicBlock', layers=[3, 4, 6, 3], strides=[1, 2, 2, 2], input_channels=3, flexible_input_res=False, pool_size=7, coordconv=False, radius_channel=True, pretrained=False*)
Receives a task object to hold internally for model specialization.

forward(*x*)
Transforms an input tensor in order to generate a prediction.

set_task(*task*)
Adapts the model to support a new task, replacing layers if needed.

class thelper.nn.resnet.**SqueezeExcitationBlock**(*inplanes, planes, stride=1, downsample=None, reduction=16, coordconv=False, radius_channel=True, activation='relu'*)
Bases: *thelper.nn.resnet.Module*

__init__(*inplanes, planes, stride=1, downsample=None, reduction=16, coordconv=False, radius_channel=True, activation='relu'*)
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

class thelper.nn.resnet.**SqueezeExcitationLayer**(*channel, reduction=16, activation='relu'*)
Bases: sphinx.ext.autodoc.importer._MockObject

__init__(*channel, reduction=16, activation='relu'*)
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

thelper.nn.resnet.**get_activation_layer**(*name: AnyStr*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f425057fe80>

thelper.nn.srm module

class thelper.nn.srm.**SRMWrapper**(*base_model: <sphinx.ext.autodoc.importer._MockObject object at 0x7f42500ed240>, input_channels: int = 3*)
Bases: sphinx.ext.autodoc.importer._MockObject

Wraps a base model for Steganalysis Rich Model (SRM)-based noise analysis.

__init__(*base_model: <sphinx.ext.autodoc.importer._MockObject object at 0x7f42500ed240>*, *input_channels: int = 3*)
Creates a SRM analysis layer and prepares internal params.

forward(*img: <sphinx.ext.autodoc.importer._MockObject object at 0x7f42500ed0f0>*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f42500ed208>
Adds a stack of noise channels to the input tensor, and processes it using the base model.

thelper.nn.srm.**setup_srm_layer**(*input_channels: int = 3*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f42500ed588>
Creates a SRM convolution layer for noise analysis.

thelper.nn.srm.**setup_srm_weights**(*input_channels: int = 3*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f42500ed3c8>
Creates the SRM kernels for noise analysis.

thelper.nn.unet module

class thelper.nn.unet.**BasicBlock**(*in_channels*, *out_channels*, *coordconv=False*, *kernel_size=3*, *padding=1*)
Bases: sphinx.ext.autodoc.importer._MockObject

Default (double-conv) block used in U-Net layers.

__init__(*in_channels*, *out_channels*, *coordconv=False*, *kernel_size=3*, *padding=1*)
Initialize self. See help(type(self)) for accurate signature.

forward(*x*)

class thelper.nn.unet.**UNet**(*task*, *in_channels=3*, *mid_channels=512*, *coordconv=False*, *srm=False*)
Bases: *thelper.nn.utils.Module*

U-Net implementation. Not identical to the original.

This version includes batchnorm and transposed conv2d layers for upsampling. Coordinate Convolutions (CoordConv) can also be toggled on if requested (see *thelper.nn.coordconv* for more information).

__init__(*task*, *in_channels=3*, *mid_channels=512*, *coordconv=False*, *srm=False*)
Receives a task object to hold internally for model specialization.

forward(*x*)
Transforms an input tensor in order to generate a prediction.

set_task(*task*)
Adapts the model to support a new task, replacing layers if needed.

thelper.nn.utils module

Neural network utility functions and classes.

This module contains base interfaces and utility functions used to define and instantiate neural network models.

class thelper.nn.utils.**ExternalClassifModule**(*model_type*, *task*, ***kwargs*)
Bases: *thelper.nn.utils.ExternalModule*

External model interface specialization for classification tasks.

This interface will try to 'rewire' the last fully connected layer of the models it instantiates to match the number of classes to predict defined in the task object.

See also:

thelper.nn.utils.Module
thelper.nn.utils.ExternalModule
thelper.nn.utils.create_model()
thelper.tasks.classif.Classification

__init__(*model_type*, *task*, ***kwargs*)
Receives a task object to hold internally for model specialization, and tries to rewire the last 'fc' layer.

set_task(*task*)
Rewires the last fully connected layer of the wrapped network to fit the given number of classification targets.

class thelper.nn.utils.**ExternalDetectModule**(*model_type*, *task*, ***kwargs*)
Bases: *thelper.nn.utils.ExternalModule*

External model interface specialization for object detection tasks.

This interface will try to 'rewire' the last fully connected layer of the models it instantiates to match the number of classes to predict defined in the task object.

See also:

thelper.nn.utils.Module
thelper.nn.utils.ExternalModule
thelper.nn.utils.create_model()
thelper.tasks.detect.Detection

__init__(*model_type*, *task*, ***kwargs*)
Receives a task object to hold internally for model specialization, and tries to rewire the last 'fc' layer.

set_task(*task*)
Rewires the last fully connected layer of the wrapped network to fit the given number of classification targets.

class thelper.nn.utils.**ExternalModule**(*model_type*, *task*, ***kwargs*)
Bases: *thelper.nn.utils.Module*

Model inteface used to hold a task object for an external implementation.

This interface is built on top of `torch.nn.Module` and should remain fully compatible with it. It is automatically used when instantiating a model via *thelper.nn.utils.create_model()* that is not derived from *thelper.nn.utils.Module*. Its only purpose is to hold the task object, and redirect *thelper.nn.utils.Module.forward()* to the actual model's transformation function. It can also be specialized to automatically adapt some external models after their construction using the knowledge contained in the task object.

See also:

thelper.nn.utils.Module
thelper.nn.utils.ExternalClassifModule
thelper.nn.utils.create_model()

thelper.tasks.utils.Task

__init__(*model_type*, *task*, ***kwargs*)
Receives a task object to hold internally for model specialization.

forward(**input*, ***kwargs*)
Transforms an input tensor in order to generate a prediction.

get_name()
Returns the name of this module (by default, the fully qualified class name of the external model).

load_state_dict(*state_dict*, *strict=True*)
Loads the state dict of an external model.

set_task(*task*)
Stores the new task internally.

Note that since this external module handler is generic, it does not know what to do with the task, so it just assumes that the model is already set up. Specialized external module handlers will instead attempt to modify the model they wrap.

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)
Returns the state dict of the external model.

summary()
Prints a summary of the model using the `thelper.nn` logger.

class thelper.nn.utils.**Module**(*task*, ***kwargs*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

Model interface used to hold a task object.

This interface is built on top of `torch.nn.Module` and should remain fully compatible with it.

All models used in the framework should derive from this interface, and therefore expect a task object as the first argument of their constructor. Their implementation may decide to ignore this task object when building their internal layers, but using it should help specialize the network by specifying e.g. the number of classes to support.

See also:

thelper.nn.utils.create_model()
thelper.tasks.utils.Task

__init__(*task*, ***kwargs*)
Receives a task object to hold internally for model specialization.

forward(**input*)
Transforms an input tensor in order to generate a prediction.

get_name()
Returns the name of this module (by default, its fully qualified class name).

set_task(*task*)
Adapts the model to support a new task, replacing layers if needed.

summary()
Prints a summary of the model using the `thelper.nn` logger.

thelper.nn.utils.**create_model**(*config*, *task*, *save_dir=None*, *ckptdata=None*)

Instantiates a model based on a provided task object.

The configuration must be given as a dictionary object. This dictionary will be parsed for a 'model' field. This field is expected to be a dictionary itself. It may then specify a type to instantiate as well as the parameters to provide to that class constructor, or a path to a checkpoint from which a model should be loaded.

All models must derive from *thelper.nn.utils.Module*, or they must be instantiable through *thelper.nn.utils.ExternalModule* (or one of its specialized classes). The provided task object will be used to make sure that the model has the required input/output layers for the requested objective.

If checkpoint data is provided by the caller, the weights it contains will be loaded into the returned model.

Usage examples inside a session configuration file:

```
# ...
# the function will look for a 'model' field in the provided config dict
"model": {
    # the type provides the class name to instantiate an object from
    "type": "thelper.nn.mobilenet.MobileNetV2",
    # the parameters listed below are passed to the model's constructor
    "params": {
        # ...
    }
# ...
```

Parameters

- **config** – a session dictionary that provides a 'model' field containing a dictionary.
- **task** – a task object that will be passed to the model's constructor in order to specialize it. Can be `None` if a checkpoint is provided, and if the previous task is wanted instead of a new one.
- **save_dir** – if not `None`, a log file containing model information will be created there.
- **ckptdata** – raw checkpoint data loaded via `torch.load()`; the model will be given its previous state.

Returns The instantiated model, compatible with the interface of both *thelper.nn.utils.Module* and `torch.nn.Module`.

See also:

thelper.nn.utils.Module
thelper.nn.utils.ExternalModule
thelper.tasks.utils.Task
thelper.utils.load_checkpoint()

thelper.nn.utils.**get_learnable_param_count**(*model: <sphinx.ext.autodoc.importer._MockObject object at 0x7f42504c3d30>*) → int

Returns the learnable (grad-enabled) parameter count in a module.

6.1.5 thelper.optim package

Optimization/metrics package.

This package contains metrics implementations used to monitor training sessions and evaluate models, and optimization methods used to control the learning behavior of these models.

Submodules

thelper.optim.eval module

Evaluation classes/funcs module.

This module contains procedures used to evaluate models and prediction results on specific tasks or datasets. These procedures may be used as part of metric classes (defined in `thelper.optim.metrics`) or high-level debug/drawing utilities.

`thelper.optim.eval.`**`compute_average_precision`**(*precision*, *recall*, *method='all-points'*)

Computes the average precision given an array of precision and recall values.

This function is inspired from the 'Object Detection Metrics' repository of Rafael Padilla. See https://github.com/rafaelpadilla/Object-Detection-Metrics for more information. The original code is distributed under the MIT License, Copyright (c) 2018 Rafael Padilla.

Parameters

- **precision** – list of precision values for the evaluated predictions of a class.
- **recall** – list of recall values for the evaluated predictions of a class.
- **method** – the evaluation method to use; can be the the latest & official PASCAL VOC toolkit approach ("all-points"), or the 11-point approach ("11-points") described in the original paper ("The PASCAL Visual Object Classes(VOC) Challenge").

Returns A 4-element tuple containing the average precision, rectified precision/recall arrays, and the indices used for the integral.

`thelper.optim.eval.`**`compute_bbox_iou`**(*bbox1*, *bbox2*)

Computes and returns the Intersection over Union (IoU) of two bounding boxes.

`thelper.optim.eval.`**`compute_mask_iou`**(*mask1*, *mask2*, *class_indices=None*, *dontcare=None*)

Computes and returns a map of Intersection over Union (IoU) scores for two segmentation masks.

`thelper.optim.eval.`**`compute_pascalvoc_metrics`**(*pred_bboxes*, *gt_bboxes*, *task*, *iou_threshold=0.5*, *method='all-points'*)

Computes the metrics used by the VOC Pascal 2012 challenge.

This function is inspired from the 'Object Detection Metrics' repository of Rafael Padilla. See https://github.com/rafaelpadilla/Object-Detection-Metrics for more information. The original code is distributed under the MIT License, Copyright (c) 2018 Rafael Padilla.

Parameters

- **pred_bboxes** – list of bbox predictions generated by the model under evaluation.
- **gt_bboxes** – list of groundtruth bounding boxes defined by the dataset.
- **task** – task definition object that holds a vector of all class names.
- **iou_threshold** – Intersection Over Union (IOU) threshold for true/false positive classification.
- **method** – the evaluation method to use; can be the the latest & official PASCAL VOC toolkit approach ("all-points"), or the 11-point approach ("11-points") described in the original paper ("The PASCAL Visual Object Classes(VOC) Challenge").

Returns

A dictionary containing evaluation information and metrics for each class. Each entry contains –

- `precision`: array with the precision values;
- `recall`: array with the recall values;
- `AP`: average precision;
- `interpolated precision`: interpolated precision values;
- `interpolated recall`: interpolated recall values;
- `total positives`: total number of ground truth positives;
- `total TP`: total number of True Positive detections;
- `total FP`: total number of False Negative detections.

thelper.optim.losses module

class thelper.optim.losses.**FocalLoss**(*gamma=2*, *alpha=0.5*, *weight=None*, *ignore_index=255*)

Bases: `sphinx.ext.autodoc.importer._MockObject`

Note: Contributed by Mario Beaulieu <mario.beaulieu@crim.ca>.

See also:

Focal Loss for Dense Object Detection, *Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, Piotr Dollár*, arXiv article.

__init__(*gamma=2*, *alpha=0.5*, *weight=None*, *ignore_index=255*)

Initialize self. See help(type(self)) for accurate signature.

forward(*preds*, *labels*)

thelper.optim.metrics module

Metrics module.

This module contains classes that implement metrics used to monitor training sessions and evaluate models. These metrics should all inherit from *thelper.optim.metrics.Metric* to allow them to be dynamically instantiated by the framework from a configuration file, and evaluated automatically inside a training session. For more information on this, refer to *thelper.train.base.Trainer*.

class thelper.optim.metrics.**Accuracy**(*top_k=1*, *max_win_size=None*)

Bases: *thelper.optim.metrics.Metric*

Classification accuracy metric interface.

This is a scalar metric used to monitor the label prediction accuracy of a model. By default, it works in `top-k` mode, meaning that the evaluation result is given by:

$$\text{accuracy} = \frac{\text{nb. correct predictions}}{\text{nb. total predictions}} \cdot 100$$

When $k > 1$, a 'correct' prediction is obtained if any of the model's top k predictions (i.e. the k predictions with the highest score) match the groundtruth label. Otherwise, if $k = 1$, then only the top prediction is compared to the groundtruth label. Note that for binary classification problems, k should always be set to 1.

This metric's goal is to maximize its value $\in [0, 100]$ (a percentage is returned).

Usage example inside a session configuration file:

```
# ...
# lists all metrics to instantiate as a dictionary
"metrics": {
    # ...
    # this is the name of the example metric; it is used for lookup/printing only
    "top_5_accuracy": {
        # this type is used to instantiate the accuracy metric
        "type": "thelper.optim.metrics.Accuracy",
        # these parameters are passed to the wrapper's constructor
        "params": {
            # the top prediction count to check for a match with the groundtruth
            "top_k": 5
        }
    },
    # ...
}
# ...
```

Todo: add support for 'dont care' target value?

Variables

- **top_k** – number of top predictions to consider when matching with the groundtruth (default=1).
- **max_win_size** – maximum moving average window size to use (default=None, which equals dataset size).
- **correct** – total number of correct predictions stored using an array for window-based averaging.
- **total** – total number of predictions stored using an array for window-based averaging.
- **warned_eval_bad** – toggles whether the division-by-zero warning has been flagged or not.

__init__(*top_k=1*, *max_win_size=None*)

Receives the number of predictions to consider for matches (`top_k`) and the moving average window size (`window_size`).

Note that by default, if `max_win_size` is not provided here, the value given to `max_iters` on the first update call will be used instead to fix the sliding window length. In any case, the smallest of `max_iters` and `max_win_size` will be used to determine the actual window size.

eval()

Returns the current accuracy (in percentage) based on the accumulated prediction counts.

Will issue a warning if no predictions have been accumulated yet.

goal

Returns the scalar optimization goal of this metric (maximization).

reset()

Toggles a reset of the metric's internal state, deallocating count arrays.

supports_classification = True

supports_segmentation = True

update(*task*, *input*, *pred*, *target*, *sample*, *loss*, *iter_idx*, *max_iters*, *epoch_idx*, *max_epochs*, *output_path*, ***kwargs*)

Receives the latest class prediction and groundtruth labels from the training session.

This function computes and accumulate the number of correct and total predictions in the internal arrays, cycling over the iteration index if the maximum window length is reached.

The exact signature of this function should match the one of the callbacks defined in *thelper.train.base.Trainer* and specified by `thelper.typedefs.IterCallbackParams`.

class thelper.optim.metrics.**AveragePrecision**(*target_class=None*, *iou_threshold=0.5*, *method='all-points'*, *max_win_size=None*)

Bases: *thelper.optim.metrics.Metric*

Object detection average precision score from PascalVOC.

This metric is computed based on the evaluator function implemented in *thelper.optim.eval*. It can target a single class at a time, or produce the mean average precision for all classes.

Usage example inside a session configuration file:

```
# ...
# lists all metrics to instantiate as a dictionary
"metrics": {
    # ...
    # this is the name of the example metric; it is used for lookup/printing only
    "mAP": {
        # this type is used to instantiate the AP metric
        "type": "thelper.optim.metrics.AveragePrecision",
        # these parameters are passed to the wrapper's constructor
        "params": {
            # no parameters means we will compute the mAP
        }
    },
    # ...
}
# ...
```

Variables

- **target_class** – name of the class to target; if 'None', will compute mAP instead of AP.
- **iou_threshold** – Intersection Over Union (IOU) threshold for true/false positive classification.
- **method** – the evaluation method to use; can be the the latest & official PASCAL VOC toolkit approach ("all-points"), or the 11-point approach ("11-points") described in the original paper ("The PASCAL Visual Object Classes(VOC) Challenge").
- **max_win_size** – maximum moving average window size to use (default=None, which equals dataset size).
- **preds** – array holding the predicted bounding boxes for all input samples.
- **targets** – array holding the target bounding boxes for all input samples.

__init__ (*target_class=None*, *iou_threshold=0.5*, *method='all-points'*, *max_win_size=None*)
Initializes metric attributes.

Note that by default, if `max_win_size` is not provided here, the value given to `max_iters` on the first update call will be used instead to fix the sliding window length. In any case, the smallest of `max_iters` and `max_win_size` will be used to determine the actual window size.

eval()
Returns the current accuracy (in percentage) based on the accumulated prediction counts.

Will issue a warning if no predictions have been accumulated yet.

goal
Returns the scalar optimization goal of this metric (maximization).

live_eval
Returns whether this metric can/should be evaluated at every backprop iteration or not.

reset()
Toggles a reset of the metric's internal state, deallocating bbox arrays.

supports_detection = True

update (*task*, *input*, *pred*, *target*, *sample*, *loss*, *iter_idx*, *max_iters*, *epoch_idx*, *max_epochs*, *output_path*, ***kwargs*)
Receives the latest bbox predictions and targets from the training session.

The exact signature of this function should match the one of the callbacks defined in *thelper.train.base.Trainer* and specified by `thelper.typedefs.IterCallbackParams`.

class thelper.optim.metrics.**ExternalMetric**(*metric_name*, *metric_type*, *metric_goal*, *metric_params=None*, *target_name=None*, *class_names=None*, *max_win_size=None*, *force_softmax=True*, *live_eval=True*)
Bases: *thelper.optim.metrics.Metric*, *thelper.ifaces.ClassNamesHandler*

External metric wrapping interface.

This interface is used to wrap external metrics and use them in the training framework. The metrics of `sklearn.metrics` are good candidates that have been used extensively with this interface in the past, but those of other libraries might also be compatible.

Along with the name of the class to import and its constructor's parameters, the user must provide a handling mode that specifies how prediction and groundtruth data should be handled in this wrapper. Also, extra arguments such as target label names, goal information, and window sizes can be provided for specific use cases related to the selected handling mode.

For now, two metric handling modes (both related to classification) are supported:

- `classif_best`: the wrapper will accumulate the predicted and groundtruth classification labels forwarded by the trainer and provide them to the external metric for evaluation. If a target label name is specified, then only classifications related to that label will be accumulated. This is the handling mode required for count-based classification metrics such as accuracy, F-Measure, precision, recall, etc.
- `classif_score`: the wrapper will accumulate the prediction score of the targeted label along with a boolean that indicates whether this label was the groundtruth label or not. This is the handling mode required for score-based classification metrics such as when computing the area under the ROC curve (AUC).

Usage examples inside a session configuration file:

```
# ...
# lists all metrics to instantiate as a dictionary
"metrics": {
    # ...
    # this is the name of the first example metric; it is used for lookup/
↪printing only
    "f1_score_reject": {
        # this type is used to instantiate the wrapper
        "type": "thelper.optim.metrics.ExternalMetric",
        # these parameters are passed to the wrapper's constructor
        "params": {
            # the external class to import
            "metric_name": "sklearn.metrics.f1_score",
            # the parameters passed to the external class's constructor
            "metric_params": {},
            # the wrapper metric handling mode
            "metric_type": "classif_best",
            # the target class name (note: dataset-specific)
            "target_name": "reject",
            # the goal type of the external metric
            "metric_goal": "max"
        }
    },
    # this is the name of the second example metric; it is used for lookup/
↪printing only
    "roc_auc_accept": {
        # this type is used to instantiate the wrapper
        "type": "thelper.optim.metrics.ExternalMetric",
        # these parameters are passed to the wrapper's constructor
        "params": {
            # the external class to import
            "metric_name": "sklearn.metrics.roc_auc_score",
            # the parameters passed to the external class's constructor
            "metric_params": {},
            # the wrapper metric handling mode
            "metric_type": "classif_score",
            # the target class name (note: dataset-specific)
            "target_name": "accept",
            # the goal type of the external metric
            "metric_goal": "max"
        }
    },
    # ...
}
# ...
```

Variables

- **metric_goal** – goal of the external metric, used for monitoring. Can be `min` or `max`.
- **metric_type** – handling mode of the external metric. Can only be one of the predetermined values.
- **metric** – type of the external metric that will be instantiated when `eval` is called.
- **metric_params** – dictionary of parameters passed to the external metric on instantiation.
- **target_name** – name of the targeted label. Used only in handling modes related to classification.

- **target_idx** – index of the targeted label. Used only in handling modes related to classification.
- ***class_names*** – holds the list of class label names provided by the dataset parser. If it is not provided when the constructor is called, it will be set by the trainer at runtime.
- **force_softmax** – specifies whether a softmax operation should be applied to the prediction scores obtained from the trainer. Only used with the "classif_score" handling mode.
- **max_win_size** – maximum moving average window size to use (default=None, which equals dataset size).
- **pred** – queue used to store predictions-related values for window-based averaging.
- **target** – queue used to store groundtruth-related values for window-based averaging.

__init__(*metric_name*, *metric_type*, *metric_goal*, *metric_params=None*, *target_name=None*, *class_names=None*, *max_win_size=None*, *force_softmax=True*, *live_eval=True*)
Receives all necessary arguments for wrapper initialization and external metric instantiation.

See *thelper.optim.metrics.ExternalMetric* for information on arguments.

class_names
Returns the list of class names considered "of interest" by the derived class.

eval()
Returns the external metric's evaluation result.

goal
Returns the scalar optimization goal of this metric (user-defined).

live_eval
Returns whether this metric can/should be evaluated at every backprop iteration or not.

By default, this returns `True`, but implementations that are quite slow may return `False`.

reset()
Toggles a reset of the metric's internal state, emptying pred/target queues.

supports_classification = True

supports_segmentation = True

update(*task*, *input*, *pred*, *target*, *sample*, *loss*, *iter_idx*, *max_iters*, *epoch_idx*, *max_epochs*, *output_path*, ***kwargs*)
Receives the latest predictions and target values from the training session.

The handling of the data received here will depend on the current metric's handling mode.

The exact signature of this function should match the one of the callbacks defined in *thelper.train.base.Trainer* and specified by `thelper.typedefs.IterCallbackParams`.

class thelper.optim.metrics.**IntersectionOverUnion**(*target_names=None*, *global_score=True*, *max_win_size=None*)
Bases: *thelper.optim.metrics.Metric*

Computes the intersection over union over image classes.

It can target a single class at a time, or produce the mean IoU (mIoU) for a number of classes. It can also average IoU scores from each images, or sum up all intersection and union areas and compute a global score.

Usage example inside a session configuration file:

```
# ...
# lists all metrics to instantiate as a dictionary
"metrics": {
    # ...
    # this is the name of the example metric; it is used for lookup/printing only
    "mIoU": {
        # this type is used to instantiate the IoU metric
        "type": "thelper.optim.metrics.IntersectionOverUnion",
        # these parameters are passed to the wrapper's constructor
        "params": {
            # no parameters means we will compute the mIoU with global scoring
        }
    },
    # ...
}
# ...
```

Variables

- **target_names** – name(s) of the class(es) to target; if 'None' or list, will compute mIoU instead of IoU.
- **max_win_size** – maximum moving average window size to use (default=None, which equals dataset size).
- **inters** – array holding the intesection areas or IoU scores for all input samples.
- **unions** – array holding the union areas for all input samples.

__init__(*target_names=None*, *global_score=True*, *max_win_size=None*)
Initializes metric attributes.

Note that by default, if `max_win_size` is not provided here, the value given to `max_iters` on the first update call will be used instead to fix the sliding window length. In any case, the smallest of `max_iters` and `max_win_size` will be used to determine the actual window size.

eval()
Returns the current IoU ratio based on the accumulated counts.

Will issue a warning if no predictions have been accumulated yet.

goal
Returns the scalar optimization goal of this metric (maximization).

reset()
Toggles a reset of the metric's internal state, deallocating bbox arrays.

supports_segmentation = True

update(*task*, *input*, *pred*, *target*, *sample*, *loss*, *iter_idx*, *max_iters*, *epoch_idx*, *max_epochs*, *output_path*, ***kwargs*)
Receives the latest bbox predictions and targets from the training session.

The exact signature of this function should match the one of the callbacks defined in *thelper.train.base.Trainer* and specified by `thelper.typedefs.IterCallbackParams`.

class thelper.optim.metrics.**MeanAbsoluteError**(*reduction='mean'*, *max_win_size=None*)
Bases: *thelper.optim.metrics.Metric*

Mean absolute error metric interface.

This is a scalar metric used to monitor the mean absolute deviation (or error) for a model's predictions. This regression metric can be described as:

$$e(x, y) = E = \{e_1, \dots, e_N\}^\top, \quad e_n = |x_n - y_n|,$$

where N is the batch size. If `reduction` is not `'none'`, then:

$$\text{MAE}(x, y) = \begin{cases} \text{mean}(E), & \text{if reduction } = \text{mean.} \\ \text{sum}(E), & \text{if reduction } = \text{sum.} \end{cases}$$

x and *y* are tensors of arbitrary shapes with a total of *n* elements each.

Usage example inside a session configuration file:

```
# ...
# lists all metrics to instantiate as a dictionary
"metrics": {
    # ...
    # this is the name of the example metric; it is used for lookup/printing only
    "mae": {
        # this type is used to instantiate the error metric
        "type": "thelper.optim.metrics.MeanAbsoluteError",
        "params": {
            "reduction": "mean"
        }
    },
    # ...
}
# ...
```

Todo: add support for 'dont care' target value?

Variables

- **max_win_size** – maximum moving average window size to use (default=None, which equals dataset size).
- **reduction** – string representing the tensor reduction strategy to use.
- **errors** – array of error values stored for window-based averaging.
- **warned_eval_bad** – toggles whether the division-by-zero warning has been flagged or not.

__init__(*reduction='mean'*, *max_win_size=None*)

Receives the reduction strategy and the moving average window size (`window_size`).

Note that by default, if `max_win_size` is not provided here, the value given to `max_iters` on the first update call will be used instead to fix the sliding window length. In any case, the smallest of `max_iters` and `max_win_size` will be used to determine the actual window size.

eval()

Returns the current (average) mean absolute error based on the accumulated values.

Will issue a warning if no predictions have been accumulated yet.

goal

Returns the scalar optimization goal of this metric (minimization).

reset()

Toggles a reset of the metric's internal state, deallocating the errors array.

supports_regression = True

update(*task*, *input*, *pred*, *target*, *sample*, *loss*, *iter_idx*, *max_iters*, *epoch_idx*, *max_epochs*, *output_path*, ***kwargs*)

Receives the latest predictions and target values from the training session.

This function computes and accumulates the L1 distance between predictions and targets in the internal array, cycling over the iteration index if the maximum window length is reached.

The exact signature of this function should match the one of the callbacks defined in *thelper.train.base.Trainer* and specified by `thelper.typedefs.IterCallbackParams`.

class thelper.optim.metrics.**MeanSquaredError**(*reduction='mean'*, *max_win_size=None*)

Bases: *thelper.optim.metrics.Metric*

Mean squared error metric interface.

This is a scalar metric used to monitor the mean squared deviation (or error) for a model's predictions. This regression metric can be described as:

$$e(x, y) = E = \{e_1, \dots, e_N\}^\top, \quad e_n = (x_n - y_n)^2,$$

where N is the batch size. If `reduction` is not `'none'`, then:

$$\text{MSE}(x, y) = \begin{cases} \text{mean}(E), & \text{if reduction } = \text{mean}. \\ \text{sum}(E), & \text{if reduction } = \text{sum}. \end{cases}$$

x and *y* are tensors of arbitrary shapes with a total of *n* elements each.

Usage example inside a session configuration file:

```
# ...
# lists all metrics to instantiate as a dictionary
"metrics": {
    # ...
    # this is the name of the example metric; it is used for lookup/printing only
    "mse": {
        # this type is used to instantiate the error metric
        "type": "thelper.optim.metrics.MeanSquaredError",
        "params": {
            "reduction": "mean"
        }
    },
    # ...
}
# ...
```

Todo: add support for 'dont care' target value?

Variables

- **max_win_size** – maximum moving average window size to use (default=None, which equals dataset size).
- **reduction** – string representing the tensor reduction strategy to use.
- **errors** – array of error values stored for window-based averaging.
- **warned_eval_bad** – toggles whether the division-by-zero warning has been flagged or not.

__init__ (*reduction='mean'*, *max_win_size=None*)
Receives the reduction strategy and the moving average window size (`window_size`).

Note that by default, if `max_win_size` is not provided here, the value given to `max_iters` on the first update call will be used instead to fix the sliding window length. In any case, the smallest of `max_iters` and `max_win_size` will be used to determine the actual window size.

eval()
Returns the current (average) mean squared error based on the accumulated values.

Will issue a warning if no predictions have been accumulated yet.

goal
Returns the scalar optimization goal of this metric (minimization).

reset()
Toggles a reset of the metric's internal state, deallocating the errors array.

supports_regression = True

update (*task*, *input*, *pred*, *target*, *sample*, *loss*, *iter_idx*, *max_iters*, *epoch_idx*, *max_epochs*, *output_path*, ***kwargs*)
Receives the latest predictions and target values from the training session.

This function computes and accumulates the mean squared error between predictions and targets in the internal array, cycling over the iteration index if the maximum window length is reached.

The exact signature of this function should match the one of the callbacks defined in *thelper.train.base.Trainer* and specified by `thelper.typedefs.IterCallbackParams`.

class thelper.optim.metrics.**Metric**
Bases: *thelper.ifaces.PredictionConsumer*

Abstract metric interface.

This interface defines basic functions required so that *thelper.train.base.Trainer* can figure out how to instantiate, update, and optimize a given metric while training/evaluating a model.

All metrics, by definition, must be 'optimizable'. This means that they should return a scalar value when 'evaluated' and define an optimal goal (-inf or +inf). If this is not possible, then the class should probably be derived using the more generic *thelper.ifaces.PredictionConsumer* instead.

eval()
Returns the metric's evaluation result.

The returned value should be a scalar. As a model improves, this scalar should get closer to the optimization goal (defined through the 'goal' attribute). This value will be queried at the end of each training epoch by the trainer.

goal
Returns the scalar optimization goal of the metric.

The returned goal can be the `minimize` or `maximize` members of `thelper.optim.metrics.Metric` if the class's evaluation returns a scalar value, and `None` otherwise. The trainer will check this value to see if monitoring the metric's evaluation result progression is possible.

live_eval
Returns whether this metric can/should be evaluated at every backprop iteration or not.

By default, this returns `True`, but implementations that are quite slow may return `False`.

maximize = inf
Possible value of the `goal` attribute of this metric.

minimize = -inf
Possible value of the `goal` attribute of this metric.

update(*task*, *input*, *pred*, *target*, *sample*, *loss*, *iter_idx*, *max_iters*, *epoch_idx*, *max_epochs*, *output_path*, ***kwargs*)
Receives the latest prediction and groundtruth tensors from the training session.

The data given here will be "consumed" internally, but it should NOT be modified. For example, a classification accuracy metric might accumulate the correct number of predictions in comparison to groundtruth labels, but never alter those predictions. The iteration/epoch indices may be used to 'reset' the internal state of this object when needed (for example, at the start of each new epoch).

Remember that input, prediction, and target tensors received here will all have a batch dimension!

The exact signature of this function should match the one of the callbacks defined in *thelper.train.base.Trainer* and specified by `thelper.typedefs.IterCallbackParams`.

class thelper.optim.metrics.**PSNR**(*data_range=1.0*, *max_win_size=None*)
Bases: *thelper.optim.metrics.Metric*

Peak Signal-to-Noise Ratio (PSNR) metric interface.

This is a scalar metric used to monitor the change in quality of a signal (or image) following a transformation. For more information, see its definition on [Wikipedia].

The PSNR (in decibels, dB) between a modified signal x and its original version y is defined as:

$$\text{PSNR}(x, y) = 10 * \log_{10}\left(\frac{R^2}{\text{MSE}(x, y)}\right)$$

where $\text{MSE}(x, y)$ returns the mean squared error (see *thelper.optim.metrics.MeanSquaredError* for more information), and R is the maximum possible value for a single element in the input signal (i.e. its maximum "range").

Usage example inside a session configuration file:

```
# ...
# lists all metrics to instantiate as a dictionary
"metrics": {
    # ...
    # this is the name of the example metric; it is used for lookup/printing only
    "psnr": {
        # this type is used to instantiate the metric
        "type": "thelper.optim.metrics.PSNR",
        "params": {
            "data_range": "255"
        }
    },
    # ...
}
# ...
```

Variables

- **max_win_size** – maximum moving average window size to use (default=None, which equals dataset size).
- **data_range** – maximum value of an element in the target signal.
- **psnrs** – array of psnr values stored for window-based averaging.

- **warned_eval_bad** – toggles whether the division-by-zero warning has been flagged or not.

__init__(*data_range=1.0*, *max_win_size=None*)
Receives all necessary initialization arguments to compute signal PSNRs,

See *thelper.optim.metrics.PSNR* for information on arguments.

eval()
Returns the current (average) PSNR based on the accumulated values.

Will issue a warning if no predictions have been accumulated yet.

goal
Returns the scalar optimization goal of this metric (maximization).

reset()
Toggles a reset of the metric's internal state, deallocating the psnrs array.

supports_regression = True

update(*task*, *input*, *pred*, *target*, *sample*, *loss*, *iter_idx*, *max_iters*, *epoch_idx*, *max_epochs*, *output_path*, ***kwargs*)
Receives the latest predictions and target values from the training session.

The exact signature of this function should match the one of the callbacks defined in *thelper.train.base.Trainer* and specified by `thelper.typedefs.IterCallbackParams`.

class thelper.optim.metrics.**ROCCurve**(*target_name*, *target_tpr=None*, *target_fpr=None*, *class_names=None*, *force_softmax=True*, *sample_weight=None*, *drop_intermediate=True*)
Bases: *thelper.optim.metrics.Metric*, *thelper.ifaces.ClassNamesHandler*

Receiver operating characteristic (ROC) computation interface.

This class provides an interface to `sklearn.metrics.roc_curve` and `sklearn.metrics.roc_auc_score` that can produce various types of ROC-related information including the area under the curve (AUC), the false positive and negative rates for various operating points, and the ROC curve itself as an image (also compatible with tensorboardX).

By default, evaluating this metric returns the Area Under the Curve (AUC). If a target operating point is set, it will instead return the false positive/negative prediction rate of the model at that point.

Usage examples inside a session configuration file:

```
# ...
# lists all metrics to instantiate as a dictionary
"metrics": {
    # ...
    # this is the name of the first example; it will output the AUC of the "reject
↪" class
    "roc_reject_auc": {
        # this type is used to instantiate the ROC metric
        "type": "thelper.optim.metrics.ROCCurve",
        # these parameters are passed to the constructor
        "params": {
            # the name of the class to evaluate
            "target_name": "reject"
        }
    },
    # this is the name of the second example; it will output the FPR at TPR=0.99
    "roc_reject_0.99tpr": {
```

(continues on next page)

(continued from previous page)

```
        # this type is used to instantiate the ROC metric
        "type": "thelper.optim.metrics.ROCCurve",
        # these parameters are passed to the constructor
        "params": {
            # the name of the class to evaluate
            "target_name": "reject",
            # the target true positive rate (TPR) operating point
            "target_tpr": 0.99
        }
    },
    # ...
}
# ...
```

Variables

- **target_inv** – used to target all classes except the named one(s); experimental!
- **target_name** – name of targeted class to generate the roc curve/auc information for.
- **target_tpr** – target operating point in terms of true positive rate (provided in constructor).
- **target_fpr** – target operating point in terms of false positive rate (provided in constructor).
- **target_idx** – index of the targeted class, mapped from target_name using the class_names list.
- ***class_names*** – holds the list of class label names provided by the dataset parser. If it is not provided when the constructor is called, it will be set by the trainer at runtime.
- **force_softmax** – specifies whether a softmax operation should be applied to the prediction scores obtained from the trainer.
- **curve** – roc curve generator function, called at evaluation time to generate the output string.
- **auc** – auc score generator function, called at evaluation time to generate the output string.
- **score** – queue used to store prediction score values for window-based averaging.
- **true** – queue used to store groundtruth label values for window-based averaging.

__init__(*target_name*, *target_tpr=None*, *target_fpr=None*, *class_names=None*, *force_softmax=True*, *sample_weight=None*, *drop_intermediate=True*)

Receives the target class/operating point info, log parameters, and roc computation arguments.

Parameters

- **target_name** – name of targeted class to generate the roc curve/auc information for.
- **target_tpr** – target operating point in terms of true positive rate (provided in constructor).
- **target_fpr** – target operating point in terms of false positive rate (provided in constructor).
- **class_names** – holds the list of class label names provided by the dataset parser. If it is not provided when the constructor is called, it will be set by the trainer at runtime.
- **force_softmax** – specifies whether a softmax operation should be applied to the prediction scores obtained from the trainer.

- **sample_weight** – passed to `sklearn.metrics.roc_curve` and `sklearn.metrics.roc_auc_score`.
- **drop_intermediate** – passed to `sklearn.metrics.roc_curve`.

class_names
Returns the list of class names considered "of interest" by the derived class.

eval()
Returns the evaluation result (AUC/TPR/FPR).

If no target operating point is set, the returned value is the AUC for the target class. If a target TPR is set, the returned value is the FPR for that operating point. If a target FPR is set, the returned value is the TPR for that operating point.

goal
Returns the scalar optimization goal of this metric (variable based on target op point).

live_eval
Returns whether this metric can/should be evaluated at every backprop iteration or not.

render()
Returns the ROC curve as a numpy-compatible RGBA image drawn by pyplot.

reset()
Toggles a reset of the metric's internal state, emptying queues.

supports_classification = True

supports_segmentation = True

update(*task*, *input*, *pred*, *target*, *sample*, *loss*, *iter_idx*, *max_iters*, *epoch_idx*, *max_epochs*, *output_path*, ***kwargs*)
Receives the latest predictions and target values from the training session.

The exact signature of this function should match the one of the callbacks defined in *thelper.train.base.Trainer* and specified by `thelper.typedefs.IterCallbackParams`.

thelper.optim.schedulers module

Schedulers module.

This module contains classes used for scheduling learning rate changes while training a model. All classes defined here should derive from `torch.optim.lr_scheduler._LRScheduler` to remain torch- compatible.

class thelper.optim.schedulers.**CustomStepLR**(*optimizer*, *milestones*, *last_epoch=-1*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

Sets the learning rate of each parameter group using a dictionary of preset scaling factors for epoch-based milestones.

This class can be useful for tuning the learning rate scheduling behavior of a training session beyond what is already possible using PyTorch's existing LR scheduler classes. Note that all epoch indices are assumed to be 0-based.

Usage example in Python:

```
# Assuming the optimizer uses lr = 0.05, we hard-code a slow startup...
# lr = 0.00625   if epoch < 2        (1/8 scale before epoch 2)
# lr = 0.0125    if 2 <= epoch < 3   (1/4 scale before epoch 3)
# lr = 0.025     if 3 <= epoch < 4   (1/2 scale before epoch 4)
# lr = 0.05      if 4 <= epoch < 30  (default scale between epoch 4 and 30)
```

(continues on next page)

(continued from previous page)

```
# lr = 0.005     if 30 <= epoch < 80 (1/10 scale past epoch 30)
# lr = 0.0005    if epoch >= 80      (1/100 scale past epoch 80)
scheduler = CustomStepLR(optimizer, milestones={
    0: 1/8,
    2: 1/4,
    3: 1/2,
    4: 1,
    30: 0.1,
    80: 0.01
})
for epoch in range(100):
    # note: we can call the scheduler's step function before optimizing, as
    # we provide it with the epoch index directly --- this is not the case
    # typically with default pytorch schedulers (and it changed beyond v1.1)
    scheduler.step(epoch)
    train(...)
    validate(...)
```

Usage example inside a session configuration file:

```
# ...
# lists the model optimization parameters for the training session
"optimization": {
    # lists the optimizer arguments (type, parameters, LR, ...)
    "optimizer": {
        # ...
    },
    # lists the scheduler arguments (field can be omitted if no scheduler is␣
↪needed)
    "scheduler": {
        # the type used to instantiate the scheduler
        "type": "thelper.optim.schedulers.CustomStepLR",
        # the parameters passed to the scheduler's constructor
        "params": {
            # by default, the optimizer is passed automatically;
            # we only need to specify the extra parameters here
            "milestones": {
                "1": 1,  # after epoch 1, scale the LR by 1
                "10": 0.1, # after epoch 10, scale the LR by 0.1
                "20": 0.01,  # ... and so on
                "30": 0.001,
                "40": 0.0001
            }
        }
    }
},
# ...
```

Variables

- **stages** – list of epochs where a new scaling factor is to be applied.
- **scales** – list of scaling factors to apply at each stage.
- **milestones** – original milestones map provided in the constructor.

See also:

thelper.train.base.Trainer

__init__(*optimizer*, *milestones*, *last_epoch=-1*)
Receives the optimizer, milestone scaling factor, and initialization state.

If the milestones do not include the first epoch (idx = 0), then its scaling factor is set to 1. When last_epoch is -1, the training is assumed to start from scratch.

Parameters

- **optimizer** – Wrapped optimizer (PyTorch-compatible object).
- **milestones** – Map of epoch indices tied to scaling factors. Keys must be increasing.
- **last_epoch** – The index of last epoch. Default: -1.

get_lr()
Returns the learning rate to use given the current epoch and scaling factors.

thelper.optim.utils module

Optimization/metrics utility module.

This module contains utility functions and tools used by the other modules of this package.

thelper.optim.utils.**create_loss_fn**(*config*, *model*, *loader=None*, *uploader=None*)
Instantiates and returns the loss function to use for training.

The default way to specify the loss function to use is to provide a callable type to instantiate as well as its initialization parameters. For example:

```
# ...
"loss": {
    # if we want to use PyTorch's cross entropy loss:
    #  >>> loss_fn = torch.nn.CrossEntropyLoss(**params)
    #  >>> ...
    #  >>> loss = loss_fn(pred, target)
    "type": "torch.nn.CrossEntropyLoss"
    "params": {
        "weight": [ ... ],
        "reduction": "mean",
        # ...
    }
},
# ...
```

The loss function can also be queried from a member function of the model class, as such:

```
# ...
"loss": {
    # to query the model for the loss function:
    #  >>> loss_fn = model.get_loss_fn(**params)
    #  >>> ...
    #  >>> loss = loss_fn(pred, target)
    "model_getter": "get_loss_fn"
    "params": {
        # ...
    }
```

(continues on next page)

(continued from previous page)

```
},
# ...
```

If the model is supposed to compute its own loss, we suggest creating a specialized trainer class. In that case only, the 'loss' field can be omitted from the session configuration file.

If the task is related to image classification or semantic segmentation, the classes can be weighted based on extra parameters in the loss configuration. The strategy used to compute the weights is related to the one in *thelper.data.samplers.WeightedSubsetRandomSampler*. The exact parameters that are expected for class reweighting are the following:

- `weight_distribution` (mandatory, toggle): the dictionary of weights assigned to each class, or the rebalancing strategy to use. If omitted entirely, no class weighting will be performed.
- `weight_param_name` (optional, default="weight"): name of the constructor parameter that expects the weight list.
- `weight_max` (optional, default=inf): the maximum weight that can be assigned to a class.
- `weight_min` (optional, default=0): the minimum weight that can be assigned to a class.
- `weight_norm` (optional, default=True): specifies whether the weights should be normalized or not.

This function also supports an extra special parameter if the task is related to semantic segmentation: `ignore_index`. If this parameter is found and not `None` (integer), then the loss function will ignore the given value when computing the loss of a sample. The exact parameters that are expected in this case are the following:

- `ignore_index_param_name` (optional, default="ignore_index"): name of the constructor parameter that expects the ignore value.
- `ignore_index_label_name` (optional, default="dontcare"): name of the label to pass the ignore value from.

`thelper.optim.utils.`**`create_metrics`**(*config*)
Instantiates and returns the metrics defined in the configuration dictionary.

All arguments are expected to be handed in through the configuration via a dictionary named 'params'.

`thelper.optim.utils.`**`create_optimizer`**(*config*, *model*)
Instantiates and returns the optimizer to use for training.

By default, the optimizer will be instantiated with the model parameters given as the first argument of its constructor. All supplementary arguments are expected to be handed in through the configuration via a dictionary named 'params'.

`thelper.optim.utils.`**`create_scheduler`**(*config*, *optimizer*)
Instantiates and returns the learning rate scheduler to use for training.

All arguments are expected to be handed in through the configuration via a dictionary named 'params'.

`thelper.optim.utils.`**`get_lr`**(*optimizer*)
Returns the optimizer's learning rate, or 0 if not found.

6.1.6 thelper.session package

Submodules

thelper.session.base module

class thelper.session.base.**SessionRunner**(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)

Bases: object

Abstract session runner interface that defines basic session i/o and setup operations.

This class offers the most basic methods that can be employed by more specialized training or inference sessions. By itself, it doesn't actually run anything.

Variables

- **checkpoint_dir** – session checkpoint output directory (located within the 'session directory').
- **config** – session configuration dictionary holding all original settings, including trainer configuration.
- **devices** – list of (cuda) device IDs to upload the model/tensors to; can be empty if only the CPU is available.
- **epochs** – number of epochs to train the model for.
- **logger** – used to output debug/warning/error messages to session log.
- **model** – reference to the model being trained or used for evaluation/prediction.
- **monitor** – name of the training/validation metric that should be monitored for model improvement.
- **name** – name of the session, used for printing and creating log folders.
- **optimization_config** – dictionary of optim-related parameters, parsed at training time.
- **output_paths** – map of session output paths where training/evaluation results should be saved.
- **save_freq** – frequency of checkpoint saves while training (i.e. save every X epochs).
- **save_raw** – specifies whether to save raw types or thelper objects in checkpoints.
- **skip_eval_iter** – number of evaluation iterations to skip (useful for resuming a session).
- **skip_tbx_histograms** – flag used to skip the generation of graph histograms in tbx (useful for large models).
- ***task*** – reference to the object used to specialize the model and that holds task metainformation.
- **tbx_histogram_freq** – frequency of tbx histogram saves while training (i.e. save every X epochs).
- **use_tbx** – defines whether to use tensorboardX writers for logging or not.
- **writers** – map of tbx writers used to save training/evaluation events.

See also:

thelper.train.base.Trainer

__init__(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)
Receives the trainer configuration dictionary, parses it, and sets up the session.

6.1.7 thelper.tasks package

Task definition package.

This package contains classes and functions whose role is to define the input/output formats and operations expected from a trained model. This essentially defines the 'goal' of the model, and is used to specialize and automate its training.

Submodules

thelper.tasks.classif module

Classification task interface module.

This module contains a class that defines the objectives of models/trainers for classification tasks.

class thelper.tasks.classif.**Classification**(*class_names: Iterable[AnyStr], input_key: collections.abc.Hashable, label_key: collections.abc.Hashable, meta_keys: Optional[Iterable[collections.abc.Hashable]] = None, multi_label: bool = False*)
Bases: *thelper.tasks.utils.Task*, *thelper.ifaces.ClassNamesHandler*

Interface for input labeling/classification tasks.

This specialization requests that when given an input tensor, the trained model should provide prediction scores for each predefined label (or class). The label names are used here to help categorize samples, and to assure that two tasks are only identical when their label counts and ordering match.

Variables

- ***class_names*** – list of label (class) names to predict (each name should be a string).
- ***input_key*** – the key used to fetch input tensors from a sample dictionary.
- **label_key** – the key used to fetch label (class) names/indices from a sample dictionary.
- ***meta_keys*** – the list of extra keys provided by the data parser inside each sample.

See also:

thelper.tasks.utils.Task
thelper.train.classif.ImageClassifTrainer

__init__(*class_names: Iterable[AnyStr], input_key: collections.abc.Hashable, label_key: collections.abc.Hashable, meta_keys: Optional[Iterable[collections.abc.Hashable]] = None, multi_label: bool = False*)
Receives and stores the class (or label) names to predict, the input tensor key, the groundtruth label (class) key, and the extra (meta) keys produced by the dataset parser(s).

The class names can be provided as a list of strings, or as a path to a json file that contains such a list. The list must contain at least two items. All other arguments are used as-is to index dictionaries, and must therefore be key-compatible types.

If the *multi_label* is activated, samples with non-scalar class labels will be allowed in the *get_class_sizes* and *get_class_sample_map* functions.

check_compat(*task: thelper.tasks.utils.Task*, *exact: bool = False*) → bool
Returns whether the current task is compatible with the provided one or not.

This is useful for sanity-checking, and to see if the inputs/outputs of two models are compatible. If `exact = True`, all fields will be checked for exact (perfect) compatibility (in this case, matching meta keys and class name order).

get_class_sample_map(*samples: Iterable*, *unset_key: collections.abc.Hashable = None*) → Dict[AnyStr, List[int]]
Splits a list of samples based on their labels into a map of sample lists.

This function is useful if we need to split a dataset based on its label categories in order to sort it, augment it, or re-balance it. The samples do not need to be fully loaded for this to work, as only their label (gt) value will be queried. If a sample is missing its label, it will be ignored and left out of the generated dictionary unless a value is given for `unset_key`.

Parameters

- **samples** – the samples to split, where each sample is provided as a dictionary.
- **unset_key** – a key under which all unlabeled samples should be kept (`None` = ignore).

Returns A dictionary that maps each class label to its corresponding list of samples.

get_class_sizes(*samples: Iterable*) → Dict[AnyStr, int]
Given a list of samples, returns a map of sample counts for each class label.

get_compat(*task: thelper.tasks.utils.Task*) → thelper.tasks.utils.Task
Returns a task instance compatible with the current task and the given one.

supports_classification = True

thelper.tasks.detect module

Detection task interface module.

This module contains classes that define object detection utilities and task interfaces.

class thelper.tasks.detect.**BoundingBox**(*class_id*, *bbox*, *include_margin=True*, *difficult=False*, *occluded=False*, *truncated=False*, *iscrowd=False*, *confidence=None*, *image_id=None*, *task=None*)
Bases: `object`

Interface used to hold instance metadata for object detection tasks.

Object detection trainers and display utilities in the framework will expect this interface to be used when parsing a predicted detection or an annotation. The base contents are based on the PASCALVOC metadata structure, and this class can be derived if necessary to contain more metadata.

Variables

- ***class_id*** – type identifier for the underlying object instance.
- ***bbox*** – four-element tuple holding the (xmin,ymin,xmax,ymax) bounding box parameters.
- ***include_margin*** – defines whether xmax/ymax is included in the bounding box area or not.
- ***difficult*** – defines whether this instance is considered "difficult" (false by default).

- ***occluded*** – defines whether this instance is considered "occluded" (false by default).
- ***truncated*** – defines whether this instance is considered "truncated" (false by default).
- ***iscrowd*** – defines whether this instance covers a "crowd" of objects or not (false by default).
- ***confidence*** – scalar or array of prediction confidence values tied to class types (empty by default).
- ***image_id*** – string used to identify the image containing this bounding box (i.e. file path or uuid).
- ***task*** – reference to the task object that holds extra metadata regarding the content of the bbox (None by default).

See also:

thelper.tasks.utils.Task
thelper.tasks.detect.Detection

__init__ (*class_id*, *bbox*, *include_margin=True*, *difficult=False*, *occluded=False*, *truncated=False*, *iscrowd=False*, *confidence=None*, *image_id=None*, *task=None*)
Receives and stores low-level input detection metadata for later access.

area
Returns a scalar indicating the total surface of the annotation (may be None if unknown/unspecified).

bbox
Returns the bounding box tuple $(x_min, y_min, x_max, y_max)$.

bottom
Returns the bottom bounding box edge origin offset value.

bottom_right
Returns the bottom right bounding box corner coordinates (x, y).

centroid
Returns the bounding box centroid coordinates (x, y).

class_id
Returns the object class type identifier.

confidence
Returns the confidence value (or array of confidence values) associated to the predicted class types.

static decode (*vec*, *format=None*)
Returns a BoundingBox object from a vectorized representation in a specified format.

Note: The input bbox is expected to be a 4 element array $(x_min, y_min, x_max, y_max)$.

difficult
Returns whether this bounding box is considered *difficult* by the dataset (false by default).

encode (*format=None*)
Returns a vectorizable representation of this bounding box in a specified format.

WARNING: Encoding might cause information loss (e.g. task reference is discarded).

height
Returns the height of the bounding box.

image_id
Returns the image string identifier.

include_margin
Returns whether x_max and y_max are included in the bounding box area or not

intersects(*geom*)
Returns whether the bounding box intersects a geometry (i.e. a 2D point or another bbox).

iscrowd
Returns whether this instance covers a *crowd* of objects or not (false by default).

json()
Gets a JSON-serializable representation of the bounding box parameters.

left
Returns the left bounding box edge origin offset value.

occluded
Returns whether this bounding box is considered *occluded* by the dataset (false by default).

right
Returns the right bounding box edge origin offset value.

supports_detection = True

task
Returns the reference to the task object that holds extra metadata regarding the content of the bbox.

tolist()
Gets a `list` representation of the underlying bounding box tuple $(x_min, y_min, x_max, y_max)$.

This ensures that `Tensor` objects are converted to native *Python* types.

top
Returns the top bounding box edge origin offset value.

top_left
Returns the top left bounding box corner coordinates (x, y).

totuple()
Gets a `tuple` representation of the underlying bounding box tuple $(x_min, y_min, x_max, y_max)$.

This ensures that `Tensor` objects are converted to native *Python* types.

truncated
Returns whether this bounding box is considered *truncated* by the dataset (false by default).

width
Returns the width of the bounding box.

class thelper.tasks.detect.**Detection**(*class_names*, *input_key*, *bboxes_key*, *meta_keys=None*, *input_shape=None*, *target_shape=None*, *target_min=None*, *target_max=None*, *background=None*, *color_map=None*)

Bases: *thelper.tasks.regr.Regression*, *thelper.ifaces.ClassNamesHandler*

Interface for object detection tasks.

This specialization requests that when given an input image, the trained model should provide a list of bounding box (bbox) proposals that correspond to probable objects detected in the image.

This specialized regression interface is currently used to help display functions.

Variables

- ***class_names*** – map of class name-value pairs for object types to detect.
- ***input_key*** – the key used to fetch input tensors from a sample dictionary.
- **bboxes_key** – the key used to fetch target (groundtruth) bboxes from a sample dictionary.
- ***meta_keys*** – the list of extra keys provided by the data parser inside each sample.
- ***input_shape*** – a numpy-compatible shape to expect input images to possess.
- ***target_shape*** – a numpy-compatible shape to expect the predictions to be in.
- ***target_min*** – a 2-dim tensor containing minimum (x,y) bounding box corner values.
- ***target_max*** – a 2-dim tensor containing maximum (x,y) bounding box corner values.
- ***background*** – value of the 'background' label (if any) used in the class map.
- ***color_map*** – map of class name-color pairs to use when displaying results.

See also:

thelper.tasks.utils.Task
thelper.tasks.regr.Regression
thelper.train.detect.ObjDetectTrainer

__init__(*class_names*, *input_key*, *bboxes_key*, *meta_keys=None*, *input_shape=None*, *target_shape=None*, *target_min=None*, *target_max=None*, *background=None*, *color_map=None*)

Receives and stores the bbox types to detect, the input tensor key, the groundtruth bboxes list key, the extra (meta) keys produced by the dataset parser(s), and the color map used to color bboxes when displaying results.

The class names can be provided as a list of strings, as a path to a json file that contains such a list, or as a map of predefined name-value pairs to use in gt maps. This list/map must contain at least two elements (background and one class). All other arguments are used as-is to index dictionaries, and must therefore be key- compatible types.

background

Returns the 'background' label value used in loss functions (can be `None`).

check_compat(*task*, *exact=False*)

Returns whether the current task is compatible with the provided one or not.

This is useful for sanity-checking, and to see if the inputs/outputs of two models are compatible. If `exact = True`, all fields will be checked for exact (perfect) compatibility (in this case, matching meta keys and class maps).

color_map

Returns the color map used to swap label indices for colors when displaying results.

get_class_sizes(*samples*, *bbox_format=None*)

Given a list of samples, returns a map of element counts for each object type.

get_compat(*task*)

Returns a task instance compatible with the current task and the given one.

supports_detection = True

thelper.tasks.regr module

Regression task interface module.

This module contains a class that defines the objectives of models/trainers for regression tasks.

class thelper.tasks.regr.**Regression**(*input_key*, *target_key*, *meta_keys=None*, *input_shape=None*, *target_shape=None*, *target_type=None*, *target_min=None*, *target_max=None*)

Bases: *thelper.tasks.utils.Task*

Interface for n-dimension regression tasks.

This specialization requests that when given an input tensor, the trained model should provide an n-dimensional target prediction. This is a fairly generic task that (unlike image classification and semantic segmentation) is not linked to a pre-existing set of possible solutions. The task interface is used to carry useful metadata for this task, e.g. input/output shapes, types, and min/max values for rounding/saturation.

Variables

- ***input_shape*** – a numpy-compatible shape to expect model inputs to be in.
- ***target_shape*** – a numpy-compatible shape to expect the predictions to be in.
- ***target_type*** – a numpy-compatible type to cast the predictions to (if needed).
- ***target_min*** – an n-dim tensor containing minimum target values (if applicable).
- ***target_max*** – an n-dim tensor containing maximum target values (if applicable).
- ***input_key*** – the key used to fetch input tensors from a sample dictionary.
- **target_key** – the key used to fetch target (groundtruth) values from a sample dictionary.
- ***meta_keys*** – the list of extra keys provided by the data parser inside each sample.

See also:

thelper.tasks.utils.Task
thelper.train.regr.RegressionTrainer
thelper.tasks.regr.SuperResolution
thelper.tasks.detect.Detection

__init__(*input_key*, *target_key*, *meta_keys=None*, *input_shape=None*, *target_shape=None*, *target_type=None*, *target_min=None*, *target_max=None*)
Receives and stores the keys produced by the dataset parser(s).

check_compat(*task*, *exact=False*)
Returns whether the current task is compatible with the provided one or not.

This is useful for sanity-checking, and to see if the inputs/outputs of two models are compatible. If `exact = True`, all fields will be checked for exact (perfect) compatibility (in this case, matching meta keys).

get_compat(*task*)
Returns a task instance compatible with the current task and the given one.

input_shape
Returns the shape of the input tensors to be processed by the model.

supports_regression = True

target_max
Returns the maximum target value(s) to be generated by the model.

target_min
Returns the minimum target value(s) to be generated by the model.

target_shape
Returns the shape of the output tensors to be generated by the model.

target_type
Returns the type of the output tensors to be generated by the model.

class thelper.tasks.regr.**SuperResolution**(*input_key*, *target_key*, *meta_keys=None*, *input_shape=None*, *target_type=None*, *target_min=None*, *target_max=None*)

Bases: *thelper.tasks.regr.Regression*

Interface for super-resolution tasks.

This specialization requests that when given an input tensor, the trained model should provide an identically-shape target prediction that essentially contains more (or more adequate) high-frequency spatial components.

This specialized regression interface is currently used to help display functions.

Variables

- ***input_shape*** – a numpy-compatible shape to expect model inputs/outputs to be in.
- ***target_type*** – a numpy-compatible type to cast the predictions to (if needed).
- ***target_min*** – an n-dim tensor containing minimum target values (if applicable).
- ***target_max*** – an n-dim tensor containing maximum target values (if applicable).
- ***input_key*** – the key used to fetch input tensors from a sample dictionary.
- **target_key** – the key used to fetch target (groundtruth) values from a sample dictionary.
- ***meta_keys*** – the list of extra keys provided by the data parser inside each sample.

See also:

thelper.tasks.utils.Task
thelper.tasks.regr.Regression
thelper.train.regr.RegressionTrainer

__init__(*input_key*, *target_key*, *meta_keys=None*, *input_shape=None*, *target_type=None*, *target_min=None*, *target_max=None*)
Receives and stores the keys produced by the dataset parser(s).

supports_regression = True

thelper.tasks.segm module

Segmentation task interface module.

This module contains a class that defines the objectives of models/trainers for segmentation tasks.

class thelper.tasks.segm.**Segmentation**(*class_names*, *input_key*, *label_map_key*, *meta_keys=None*, *dontcare=None*, *color_map=None*)

Bases: *thelper.tasks.utils.Task*, *thelper.ifaces.ClassNamesHandler*

Interface for pixel-level labeling/classification (segmentation) tasks.

This specialization requests that when given an input tensor, the trained model should provide prediction scores for each predefined label (or class), for each element of the input tensor. The label names are used here to help categorize samples, and to assure that two tasks are only identical when their label counts and ordering match.

Variables

- ***class_names*** – map of class name-value pairs to predict for each pixel.
- ***input_key*** – the key used to fetch input tensors from a sample dictionary.
- **label_map_key** – the key used to fetch label (class) maps from a sample dictionary.
- ***meta_keys*** – the list of extra keys provided by the data parser inside each sample.
- ***dontcare*** – value of the 'dontcare' label (if any) used in the class map.
- ***color_map*** – map of class name-color pairs to use when displaying results.

See also:

thelper.tasks.utils.Task
thelper.train.segm.ImageSegmTrainer

__init__(*class_names*, *input_key*, *label_map_key*, *meta_keys=None*, *dontcare=None*, *color_map=None*)

Receives and stores the class (or label) names to predict, the input tensor key, the groundtruth label (class) map key, the extra (meta) keys produced by the dataset parser(s), the 'dontcare' label value that might be present in gt maps (if any), and the color map used to swap label indices for colors when displaying results.

The class names can be provided as a list of strings, as a path to a json file that contains such a list, or as a map of predefined name-value pairs to use in gt maps. This list/map must contain at least two elements. All other arguments are used as-is to index dictionaries, and must therefore be key-compatible types.

check_compat(*task*, *exact=False*)

Returns whether the current task is compatible with the provided one or not.

This is useful for sanity-checking, and to see if the inputs/outputs of two models are compatible. If `exact = True`, all fields will be checked for exact (perfect) compatibility (in this case, matching meta keys and class maps).

color_map

Returns the color map used to swap label indices for colors when displaying results.

dontcare

Returns the 'dontcare' label value used in loss functions (can be `None`).

get_class_sizes(*samples*)

Given a list of samples, returns a map of element counts for each class label.

get_compat(*task*)

Returns a task instance compatible with the current task and the given one.

supports_segmentation = True

thelper.tasks.utils module

Task utility functions & base interface module.

This module contains utility functions used to instantiate tasks and check their compatibility, and the base interface used to define new tasks.

class thelper.tasks.utils.**Task**(*input_key: collections.abc.Hashable*, *gt_key: collections.abc.Hashable = None*, *meta_keys: Optional[Iterable[collections.abc.Hashable]] = None*)

Bases: object

Basic task interface that defines a training objective and that holds sample i/o keys.

Since the framework's data loaders expect samples to be passed in as dictionaries, keys are required to obtain the input that should be forwarded to a model, and to obtain the groundtruth required for the evaluation of model predictions. Other keys might also be kept by this interface for reference (these are considered meta keys).

Note that while this interface can be instantiated directly, trainers and models might not be provided enough information about their goal to be correctly instantiated. Thus, specialized task objects derived from this base class should be used if possible.

Variables

- ***input_key*** – the key used to fetch input tensors from a sample dictionary.
- ***gt_key*** – the key used to fetch gt tensors from a sample dictionary.
- ***meta_keys*** – the list of extra keys provided by the data parser inside each sample.

See also:

thelper.tasks.classif.Classification
thelper.tasks.segm.Segmentation
thelper.tasks.regr.Regression
thelper.tasks.detect.Detection

__init__(*input_key: collections.abc.Hashable*, *gt_key: collections.abc.Hashable = None*, *meta_keys: Optional[Iterable[collections.abc.Hashable]] = None*)
Receives and stores the keys used to index dataset sample contents.

check_compat(*task: thelper.tasks.utils.Task*, *exact: bool = False*) → bool
Returns whether the current task is compatible with the provided one or not.

This is useful for sanity-checking, and to see if the inputs/outputs of two models are compatible. It should be overridden in derived classes to specialize the compatibility verification. If `exact = True`, all fields will be checked for exact compatibility.

get_compat(*task: thelper.tasks.utils.Task*) → thelper.tasks.utils.Task
Returns a task instance compatible with the current task and the given one.

gt_key
Returns the key used to fetch groundtruth data tensors from a sample dictionary.

input_key
Returns the key used to fetch input data tensors from a sample dictionary.

keys
Returns a list of all keys used to carry tensors and metadata in samples.

meta_keys
Returns the list of keys used to carry meta/auxiliary data in samples.

thelper.tasks.utils.**create_global_task**(*tasks: Optional[Iterable[Task]]*) → Optional[thelper.tasks.utils.Task]
Returns a new task object that is compatible with a list of subtasks.

When different datasets must be combined in a session, the tasks they define must also be merged. This functions allows us to do so as long as the tasks all share a common objective. If creating a globally-compatible task is impossible, this function will raise an exception. Otherwise, the returned task object can be used to replace the subtasks of all used datasets.

See also:

thelper.tasks.utils.Task
thelper.tasks.utils.create_task()
thelper.data.utils.create_parsers()

thelper.tasks.utils.**create_task**(*config: Union[Dict[AnyStr, Union[AnyStr, float, int, List[Any], Dict[Any, Any], ConfigDict]], AnyStr]*) → Task
Parses a configuration dictionary or repr string and instantiates a task from it.

If a string is provided, it will first be parsed to get the task type, and then the object will be instantiated by forwarding the parameters contained in the string to the constructor of that type. Note that it is important for this function to work that the constructor argument names match the names of parameters printed in the task's `__repr__` function.

If a dict is provided, it should contain a 'type' and a 'params' field with the values required for direct instantiation.

If a *Task* instance was specified, it is directly returned.

See also:

thelper.tasks.utils.Task

6.1.8 thelper.train package

Trainer package.

This package contains classes specialized for training models on various tasks.

Submodules

thelper.train.ae module

class thelper.train.ae.**AutoEncoderTrainer**(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)
Bases: *thelper.train.base.Trainer*

__init__(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)
Receives session parameters, parses image/label keys from task object, and sets up metrics.

eval_epoch(*model*, *epoch*, *dev*, *loader*, *metrics*, *output_path*)
Evaluates the model using the provided objects.

Parameters

- **model** – the model to evaluate that is already uploaded to the target device(s).
- **epoch** – the epoch index we are training for (0-based).
- **dev** – the target device that tensors should be uploaded to.
- **loader** – the data loader used to get transformed valid/test samples.
- **metrics** – the dictionary of metrics/consumers to update every iteration.
- **output_path** – directory where output files should be written, if necessary.

supports_classification = True

supports_segmentation = True

train_epoch(*model*, *epoch*, *dev*, *classif_loss*, *optimizer*, *loader*, *metrics*, *output_path*)
Trains the model for a single epoch using the provided objects.

Parameters

- **model** – the model to train that is already uploaded to the target device(s).
- **epoch** – the epoch index we are training for (0-based).
- **dev** – the target device that tensors should be uploaded to.
- **loss** – the loss function used to evaluate model fidelity.
- **optimizer** – the optimizer used for back propagation.
- **loader** – the data loader used to get transformed training samples.
- **metrics** – the dictionary of metrics/consumers to update every iteration.
- **output_path** – directory where output files should be written, if necessary.

thelper.train.base module

Model training/evaluation base interface module.

This module contains the interface required to train and/or evaluate a model based on different tasks. The trainers based on this interface are instantiated in launched sessions based on configuration dictionaries.

class thelper.train.base.**Trainer**(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckpt_data=None*)
Bases: *thelper.session.base.SessionRunner*

Abstract trainer interface that defines basic session i/o and setup operations.

This interface defines the general behavior of a training session which includes configuration parsing, tensorboard setup, metrics and goal setup, and loss/optimizer setup. It also provides utilities for uploading models and tensors on specific devices, and for saving the state of a session. This interface should be specialized for every task by implementing the `train_epoch` and `eval_epoch` functions in a derived class. For better support from visualization utilities, the derived class should also implement `to_tensor`. See *thelper.train.classif.ImageClassifTrainer* for a complete example.

Any trainer derived from this class will alternate between training and validation epochs. It will also support post-training (final) evaluation using a separate test set. If requested, visualizations can be computed after the

validation epoch during training (e.g. sample activation maps, or t-SNE plots). See *thelper.viz* for more information on these.

The main parameters that will be parsed by this interface from a configuration dictionary are the following:

- `epochs` (mandatory if training): number of epochs to train for; one epoch is one iteration over all mini-batches.
- `optimization` (mandatory if training): sub-dictionary containing types and extra parameters required for instantiating the loss, optimizer, and scheduler objects. See the code of each related loading function for more information on special parameters.
- `save_freq` (optional, default=1): checkpoint save frequency (will save every epoch multiple of given number).
- `save_raw` (optional, default=True): specifies whether to save raw types or thelper objects in checkpoints.
- `use_tbx` (optional, default=False): defines whether to use tensorboardX writers for logging or not.
- `device` (optional): specifies which device to train/evaluate the model on (default=all available).
- `metrics`: list of metrics to instantiate and update during training/evaluation; see related loading function for more information.
- `monitor`: specifies the name of the metric that should be monitored on the validation set for model improvement.

Example configuration file:

```
# ...
"trainer": {
    # type of trainer to instantiate (linked to task type)
    "type": "thelper.train.ImageClassifTrainer",
    # train for 40 epochs
    "epochs": 40,
    # save every 5 epochs
    "save_freq": 5,
    # monitor validation accuracy and save best model based on that
    "monitor": "accuracy",
    # optimization parameters block
    "optimization": {
        # all types & params below provided by PyTorch
        "loss": {
            "type": "torch.nn.CrossEntropyLoss"
        },
        "optimizer": {
            "type": "torch.optim.SGD",
            "params": {
                "lr": 0.1,
                "momentum": 0.9,
                "weight_decay": 1e-06,
                "nesterov": true
            }
        },
        "scheduler": {
            "type": "torch.optim.lr_scheduler.StepLR",
            "params": {
                "step_size": 10,
                "step_size": 0.1
            }
        }
```

(continues on next page)

(continued from previous page)

```
    },
    # visualization block (optional)
    "viz": {
        # multiple visualization techniques can be toggled by name
        "tsne": {
            # visualization parameters would be provided here
        }
    },
    # in this example, we use two consumers in total
    # (one metric for monitoring, and one for logging)
    "metrics": {
        "accuracy": {
            "type": "thelper.optim.Accuracy"
        },
        "fullreport": {
            "type": "thelper.train.ClassifReport"
        }
    }
}
# ...
```

Variables

- **config** – session configuration dictionary holding all original settings, including trainer configuration.
- **model** – reference to the model being trained or used for evaluation/prediction.
- ***task*** – reference to the object used to specialize the model and that holds task meta-information.

TODO: move static utils to their related modules

See also:

thelper.session.base.SessionRunner
thelper.train.classif.ImageClassifTrainer
thelper.train.segm.ImageSegmTrainer
thelper.train.detect.ObjDetectTrainer
thelper.train.regr.RegressionTrainer
thelper.train.utils.create_trainer()

__init__(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)
Receives the trainer configuration dictionary, parses it, and sets up the session.

eval()
Starts the evaluation process.

This function will evaluate the model using the test data (or the validation data, if no test data is available), and return the results. Note that the code related to the forwarding of samples inside the model itself is implemented in a derived class via *thelper.train.base.Trainer.eval_epoch()*.

eval_epoch(*model*, *epoch*, *device*, *loader*, *metrics*, *output_path*)
Evaluates the model using the provided objects.

Parameters

- **model** – the model to evaluate that is already uploaded to the target device(s).
- **epoch** – the epoch index we are training for (0-based).
- **device** – the target device that tensors should be uploaded to.
- **loader** – the data loader used to get transformed valid/test samples.
- **metrics** – the dictionary of metrics/consumers to update every iteration.
- **output_path** – directory where output files should be written, if necessary.

train()
Starts the training process.

This function will train the model until the required number of epochs is reached, and then evaluate it on the test data. The setup of loggers, tensorboard writers is done here, so is model improvement tracking via monitored metrics. However, the code related to loss computation and back propagation is implemented in a derived class via *thelper.train.base.Trainer.train_epoch()*.

train_epoch(*model*, *epoch*, *dev*, *loss*, *optimizer*, *loader*, *metrics*, *output_path*)
Trains the model for a single epoch using the provided objects.

Parameters

- **model** – the model to train that is already uploaded to the target device(s).
- **epoch** – the epoch index we are training for (0-based).
- **dev** – the target device that tensors should be uploaded to.
- **loss** – the loss function used to evaluate model fidelity.
- **optimizer** – the optimizer used for back propagation.
- **loader** – the data loader used to get transformed training samples.
- **metrics** – the dictionary of metrics/consumers to update every iteration.
- **output_path** – directory where output files should be written, if necessary.

thelper.train.classif module

Classification trainer/evaluator implementation module.

class thelper.train.classif.**ImageClassifTrainer**(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)

Bases: *thelper.train.base.Trainer*

Trainer interface specialized for image classification.

This class implements the abstract functions of *thelper.train.base.Trainer* required to train/evaluate a model for image classification or recognition. It also provides a utility function for fetching i/o packets (images, class labels) from a sample, and that converts those into tensors for forwarding and loss estimation.

See also:

thelper.train.base.Trainer

__init__(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)
Receives session parameters, parses image/label keys from task object, and sets up metrics.

eval_epoch(*model*, *epoch*, *dev*, *loader*, *metrics*, *output_path*)
Evaluates the model using the provided objects.

Parameters

- **model** – the model to evaluate that is already uploaded to the target device(s).
- **epoch** – the epoch index we are training for (0-based).
- **dev** – the target device that tensors should be uploaded to.
- **loader** – the data loader used to get transformed valid/test samples.
- **metrics** – the dictionary of metrics/consumers to update every iteration.
- **output_path** – directory where output files should be written, if necessary.

supports_classification = True

train_epoch(*model*, *epoch*, *dev*, *loss*, *optimizer*, *loader*, *metrics*, *output_path*)
Trains the model for a single epoch using the provided objects.

Parameters

- **model** – the model to train that is already uploaded to the target device(s).
- **epoch** – the epoch index we are training for (0-based).
- **dev** – the target device that tensors should be uploaded to.
- **loss** – the loss function used to evaluate model fidelity.
- **optimizer** – the optimizer used for back propagation.
- **loader** – the data loader used to get transformed training samples.
- **metrics** – the dictionary of metrics/consumers to update every iteration.
- **output_path** – directory where output files should be written, if necessary.

thelper.train.detect module

Object detection trainer/evaluator implementation module.

class thelper.train.detect.**ObjDetectTrainer**(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)
Bases: *thelper.train.base.Trainer*

Trainer interface specialized for object detection.

This class implements the abstract functions of *thelper.train.base.Trainer* required to train/evaluate a model for object detection (i.e. 2D bounding box regression). It also provides a utility function for fetching i/o packets (input images, bounding boxes) from a sample, and that converts those into tensors for forwarding and loss estimation.

See also:

thelper.train.base.Trainer

__init__(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)
Receives session parameters, parses tensor/target keys from task object, and sets up metrics.

eval_epoch(*model*, *epoch*, *dev*, *loader*, *metrics*, *output_path*)
Evaluates the model using the provided objects.

Parameters

- **model** – the model to evaluate that is already uploaded to the target device(s).
- **epoch** – the epoch index we are training for (0-based).
- **dev** – the target device that tensors should be uploaded to.
- **loader** – the data loader used to get transformed valid/test samples.
- **metrics** – the dictionary of metrics/consumers to update every iteration.
- **output_path** – directory where output files should be written, if necessary.

supports_detection = True

train_epoch(*model*, *epoch*, *dev*, *loss*, *optimizer*, *loader*, *metrics*, *output_path*)
Trains the model for a single epoch using the provided objects.

Parameters

- **model** – the model to train that is already uploaded to the target device(s).
- **epoch** – the epoch index we are training for (0-based).
- **dev** – the target device that tensors should be uploaded to.
- **loss** – the loss function used to evaluate model fidelity.
- **optimizer** – the optimizer used for back propagation.
- **loader** – the data loader used to get transformed training samples.
- **metrics** – the dictionary of metrics/consumers to update every iteration.
- **output_path** – directory where output files should be written, if necessary.

thelper.train.regr module

Regression trainer/evaluator implementation module.

class thelper.train.regr.**RegressionTrainer**(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)
Bases: *thelper.train.base.Trainer*

Trainer interface specialized for generic (n-dim) regression.

This class implements the abstract functions of *thelper.train.base.Trainer* required to train/evaluate a model for generic regression (i.e. n-dim target value prediction). It also provides a utility function for fetching i/o packets (input tensors, target values) from a sample, and that converts those into tensors for forwarding and loss estimation.

See also:

thelper.train.base.Trainer

__init__(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)
Receives session parameters, parses tensor/target keys from task object, and sets up metrics.

eval_epoch(*model*, *epoch*, *dev*, *loader*, *metrics*, *output_path*)
Evaluates the model using the provided objects.

Parameters

- **model** – the model to evaluate that is already uploaded to the target device(s).
- **epoch** – the epoch index we are training for (0-based).
- **dev** – the target device that tensors should be uploaded to.
- **loader** – the data loader used to get transformed valid/test samples.
- **metrics** – the dictionary of metrics/consumers to update every iteration.
- **output_path** – directory where output files should be written, if necessary.

supports_regression = True

train_epoch(*model*, *epoch*, *dev*, *loss*, *optimizer*, *loader*, *metrics*, *output_path*)
Trains the model for a single epoch using the provided objects.

Parameters

- **model** – the model to train that is already uploaded to the target device(s).
- **epoch** – the epoch index we are training for (0-based).
- **dev** – the target device that tensors should be uploaded to.
- **loss** – the loss function used to evaluate model fidelity.
- **optimizer** – the optimizer used for back propagation.
- **loader** – the data loader used to get transformed training samples.
- **metrics** – the dictionary of metrics/consumers to update every iteration.
- **output_path** – directory where output files should be written, if necessary.

thelper.train.segm module

Segmentation trainer/evaluator implementation module.

class thelper.train.segm.**ImageSegmTrainer**(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)

Bases: *thelper.train.base.Trainer*

Trainer interface specialized for image segmentation.

This class implements the abstract functions of *thelper.train.base.Trainer* required to train/evaluate a model for image segmentation (i.e. pixel-level classification/labeling). It also provides a utility function for fetching i/o packets (images, class labels) from a sample, and that converts those into tensors for forwarding and loss estimation.

See also:

thelper.train.base.Trainer

__init__(*session_name*, *session_dir*, *model*, *task*, *loaders*, *config*, *ckptdata=None*)
Receives session parameters, parses image/label keys from task object, and sets up metrics.

eval_epoch(*model*, *epoch*, *dev*, *loader*, *metrics*, *output_path*)
Evaluates the model using the provided objects.

Parameters

- **model** – the model to evaluate that is already uploaded to the target device(s).
- **epoch** – the epoch index we are training for (0-based).
- **dev** – the target device that tensors should be uploaded to.
- **loader** – the data loader used to get transformed valid/test samples.
- **metrics** – the dictionary of metrics/consumers to update every iteration.
- **output_path** – directory where output files should be written, if necessary.

supports_segmentation = True

train_epoch(*model*, *epoch*, *dev*, *loss*, *optimizer*, *loader*, *metrics*, *output_path*)
Trains the model for a single epoch using the provided objects.

Parameters

- **model** – the model to train that is already uploaded to the target device(s).
- **epoch** – the epoch index we are training for (0-based).
- **dev** – the target device that tensors should be uploaded to.
- **loss** – the loss function used to evaluate model fidelity.
- **optimizer** – the optimizer used for back propagation.
- **loader** – the data loader used to get transformed training samples.
- **metrics** – the dictionary of metrics/consumers to update every iteration.
- **output_path** – directory where output files should be written, if necessary.

thelper.train.utils module

Training/evaluation utilities module.

This module contains utilities and tools used to instantiate training sessions. It also contains the prediction consumer interface used by metrics and loggers to receive iteration data during training. See *thelper.optim.metrics* for more information on metrics.

class thelper.train.utils.**ClassifLogger**(*top_k=1*, *conf_threshold=None*, *class_names=None*, *target_name=None*, *viz_count=0*, *report_count=None*, *log_keys=None*, *force_softmax=True*, *format=None*)
Bases: *thelper.ifaces.PredictionConsumer*, *thelper.ifaces.ClassNamesHandler*, *thelper.ifaces.FormatHandler*

Classification output logger.

This class provides a simple logging interface for accumulating and saving the predictions of a classifier. By default, all predictions will be logged. However, a confidence threshold can be set to focus on "hard" samples if necessary. It also optionally offers tensorboardX-compatible output images that can be saved locally or posted to tensorboard for browser-based visualization.

Usage examples inside a session configuration file:

```
# ...
# lists all metrics to instantiate as a dictionary
"metrics": {
    # ...
    # this is the name of the example consumer; it is used for lookup/printing␣
↪only
    "logger": {
        # this type is used to instantiate the confusion matrix report object
        "type": "thelper.train.utils.ClassifLogger",
        "params": {
            # log the three 'best' predictions for each sample
            "top_k": 3,
            # keep updating a set of 10 samples for visualization via tensorboardX
            "viz_count": 10
        }
    },
    # ...
}
# ...
```

Variables

- **top_k** – number of 'best' predictions to keep for each sample (along with the gt label).
- **conf_threshold** – threshold used to eliminate uncertain predictions.
- ***class_names*** – holds the list of class label names provided by the dataset parser. If it is not provided when the constructor is called, it will be set by the trainer at runtime.
- **target_name** – name of the targeted label (may be 'None' if all classes are used).
- **target_idx** – index of the targeted label (may be 'None' if all classes are used).
- **viz_count** – number of tensorboardX images to generate and update at each epoch.
- **report_count** – number of samples to print in reports (use 'None' if all samples must be printed).
- **log_keys** – list of metadata field keys to copy from samples into the log for each prediction.
- **force_softmax** – specifies whether a softmax operation should be applied to the prediction scores obtained from the trainer.
- **score** – array used to store prediction scores for logging.
- **true** – array used to store groundtruth labels for logging.
- **meta** – array used to store metadata pulled from samples for logging.
- **format** – output format of the produced log (supports: text, CSV)

__init__(*top_k=1*, *conf_threshold=None*, *class_names=None*, *target_name=None*, *viz_count=0*, *report_count=None*, *log_keys=None*, *force_softmax=True*, *format=None*)

Receives the logging parameters & the optional class label names used to decorate the log.

class_names

Returns the list of class names considered "of interest" by the derived class.

render()

Returns an image of predicted outputs as a numpy-compatible RGBA image drawn by pyplot.

report_csv()
Returns the logged metadata of predicted samples.

The returned object is a print-friendly CSV string that can be consumed directly by tensorboardX. Note that this string might be very long if the dataset is large (i.e. it will contain one line per sample).

report_json()

report_text()
Must be implemented by inheriting classes. Default report text representation.

reset()
Toggles a reset of the internal state, emptying storage arrays.

supports_classification = True

update(*task*, *input*, *pred*, *target*, *sample*, *loss*, *iter_idx*, *max_iters*, *epoch_idx*, *max_epochs*, *output_path*, ***kwargs*)
Receives the latest predictions and target values from the training session.

The exact signature of this function should match the one of the callbacks defined in *thelper.train.base.Trainer* and specified by `thelper.typedefs.IterCallbackParams`.

class thelper.train.utils.**ClassifReport**(*class_names=None*, *sample_weight=None*, *digits=4*, *format=None*)
Bases: *thelper.ifaces.PredictionConsumer*, *thelper.ifaces.ClassNamesHandler*, *thelper.ifaces.FormatHandler*

Classification report interface.

This class provides a simple interface to `sklearn.metrics.classification_report` so that all count-based metrics can be reported at once under a string-based representation.

Usage example inside a session configuration file:

```
# ...
# lists all metrics to instantiate as a dictionary
"metrics": {
    # ...
    # this is the name of the example consumer; it is used for lookup/printing only
    "report": {
        # this type is used to instantiate the classification report object
        "type": "thelper.train.utils.ClassifReport",
        # we do not need to provide any parameters to the constructor, defaults are fine
        "params": {
            # optional parameter that will indicate output as JSON is desired, plain 'text' otherwise
            "format": "json"
        }
    },
    # ...
}
# ...
```

Variables

- ***class_names*** – holds the list of class label names provided by the dataset parser. If it is not provided when the constructor is called, it will be set by the trainer at runtime.
- **pred** – queue used to store the top-1 (best) predicted class indices at each iteration.

- **format** – output format of the produced log (supports: text, JSON)

__init__(*class_names=None*, *sample_weight=None*, *digits=4*, *format=None*)
Receives the optional class names and arguments passed to the report generator function.

Parameters

- **class_names** – holds the list of class label names provided by the dataset parser. If it is not provided when the constructor is called, it will be set by the trainer at runtime.
- **sample_weight** – sample weights, forwarded to `sklearn.metrics.classification_report`.
- **digits** – metrics output digit count, forwarded to `sklearn.metrics.classification_report`.
- **format** – output format of the produced log.

gen_report(*as_dict=False*)

report_json()
Returns the classification report as a JSON formatted string.

report_text()
Returns the classification report as a multi-line print-friendly string.

reset()
Toggles a reset of the metric's internal state, emptying queues.

supports_classification = True

update(*task*, *input*, *pred*, *target*, *sample*, *loss*, *iter_idx*, *max_iters*, *epoch_idx*, *max_epochs*, *output_path*, ***kwargs*)
Receives the latest predictions and target values from the training session.

The exact signature of this function should match the one of the callbacks defined in *thelper.train.base.Trainer* and specified by `thelper.typedefs.IterCallbackParams`.

class thelper.train.utils.**ConfusionMatrix**(*class_names=None*, *draw_normalized=True*)
Bases: *thelper.ifaces.PredictionConsumer*, *thelper.ifaces.ClassNamesHandler*

Confusion matrix report interface.

This class provides a simple interface to `sklearn.metrics.confusion_matrix` so that a full confusion matrix can be easily reported under a string-based representation. It also offers a tensorboardX-compatible output image that can be saved locally or posted to tensorboard for browser-based visualization.

Usage example inside a session configuration file:

```
# ...
# lists all metrics to instantiate as a dictionary
"metrics": {
    # ...
    # this is the name of the example consumer; it is used for lookup/printing␣
↪only
    "confmat": {
        # this type is used to instantiate the confusion matrix report object
        "type": "thelper.train.utils.ConfusionMatrix",
        # we do not need to provide any parameters to the constructor, defaults␣
↪are fine
        "params": {
            # optional parameter that will indicate output as JSON is desired,␣
↪plain 'text' otherwise
```

(continues on next page)

(continued from previous page)

```
            "format": "json"
        }
    },
    # ...
}
# ...
```

Variables

- **matrix** – report generator function, called at evaluation time to generate the output string.
- ***class_names*** – holds the list of class label names provided by the dataset parser. If it is not provided when the constructor is called, it will be set by the trainer at runtime.
- **draw_normalized** – defines whether rendered confusion matrices should be normalized or not.
- **pred** – queue used to store the top-1 (best) predicted class indices at each iteration.

__init__(*class_names=None*, *draw_normalized=True*)
Receives the optional class label names used to decorate the output string.

Parameters

- **class_names** – holds the list of class label names provided by the dataset parser. If it is not provided when the constructor is called, it will be set by the trainer at runtime.
- **draw_normalized** – defines whether rendered confusion matrices should be normalized or not.

render()
Returns the confusion matrix as a numpy-compatible RGBA image drawn by pyplot.

report()
Returns the confusion matrix as a multi-line print-friendly string.

reset()
Toggles a reset of the metric's internal state, emptying queues.

supports_classification = True

supports_segmentation = True

update(*task*, *input*, *pred*, *target*, *sample*, *loss*, *iter_idx*, *max_iters*, *epoch_idx*, *max_epochs*, *output_path*, ***kwargs*)
Receives the latest predictions and target values from the training session.

The exact signature of this function should match the one of the callbacks defined in *thelper.train.base.Trainer* and specified by `thelper.typedefs.IterCallbackParams`.

class thelper.train.utils.**DetectLogger**(*top_k=None*, *conf_threshold=None*, *iou_threshold=None*, *class_names=None*, *target_name=None*, *viz_count=0*, *report_count=None*, *log_keys=None*, *format=None*)
Bases: *thelper.ifaces.PredictionConsumer*, *thelper.ifaces.ClassNamesHandler*, *thelper.ifaces.FormatHandler*

Detection output logger.

This class provides a simple logging interface for accumulating and saving the bounding boxes of an object detector. By default, all detections will be logged. However, a confidence threshold can be set to focus on strong predictions if necessary.

Usage examples inside a session configuration file:

```
# ...
# lists all metrics to instantiate as a dictionary
"metrics": {
    # ...
    # this is the name of the example consumer; it is used for lookup/printing␣
↪only
    "logger": {
        # this type is used to instantiate the confusion matrix report object
        "type": "thelper.train.utils.DetectLogger",
        "params": {
            # (optional) log the three 'best' detections for each target
            "top_k": 3
        }
    },
    # ...
}
# ...
```

Variables

- **top_k** – number of 'best' detections to keep for each target bbox (along with the target label). If omitted, lists all bounding box predictions by the model after applying IoU and confidence thresholds.
- **conf_threshold** – threshold used to eliminate uncertain predictions (if they support confidence). If confidence is not supported by the model bbox predictions, this parameter is ignored.
- **iou_threshold** – threshold used to eliminate predictions too far from target (regardless of confidence). If omitted, will ignore only completely non-overlapping predicted bounding boxes ($IoU = 0$). If no target bounding boxes are provided (prediction-only), this parameter is ignored.
- ***class_names*** – holds the list of class label names provided by the dataset parser. If it is not provided when the constructor is called, it will be set by the trainer at runtime.
- **target_name** – name of the targeted label (may be 'None' if all classes are used).
- **target_idx** – index of the targeted label (may be 'None' if all classes are used).
- **viz_count** – number of tensorboardX images to generate and update at each epoch.
- **report_count** – number of samples to print in reports (use 'None' if all samples must be printed).
- **log_keys** – list of metadata field keys to copy from samples into the log for each prediction.
- ***bbox*** – array used to store prediction bounding boxes for logging.
- **true** – array used to store groundtruth labels for logging.
- **meta** – array used to store metadata pulled from samples for logging.
- **format** – output format of the produced log (supports: text, CSV, JSON)

__init__(*top_k=None*, *conf_threshold=None*, *iou_threshold=None*, *class_names=None*, *target_name=None*, *viz_count=0*, *report_count=None*, *log_keys=None*, *format=None*)

Receives the logging parameters & the optional class label names used to decorate the log.

class_names
Returns the list of class names considered "of interest" by the derived class.

gen_report()
Returns the logged metadata of predicted bounding boxes per sample target in a JSON-like structure.

For every target bounding box, the corresponding *best*-sorted detections are returned. Sample metadata is appended to every corresponding sub-target if any where requested.

If `report_count` was specified, the returned report will be limited to that requested amount of targets.

See also:

DetectLogger.group_bbox() for formatting, sorting and filtering details.

group_bbox(*target_bboxes*, *detect_bboxes*)
Groups a sample's detected bounding boxes with target bounding boxes according to configuration parameters.

Returns a list of detections grouped by target(s) with following format:

```
[
    {
        "target": <associated-target-bbox>,
        "detect": [
            {
                "bbox": <detection-bbox>,
                "iou": <IoU(detect-bbox, target-bbox)>
            },
            ...
        ]
    },
    ...
]
```

The associated target bounding box and IoU can be `None` if no target was provided (ie: during inference). In this case, the returned list will have only a single element with all detections associated to it. A single element list can also be returned if only one target was specified for this sample.

When multiple ground truth targets were provided, the returned list will have the same length and ordering as these targets. The associated detected bounding boxes will depend on IoU between target/detection combinations.

All filtering thresholds specified as configuration parameter will be applied for the returned list. Detected bounding boxes will also be sorted by highest confidence (if available) or by highest IoU as fallback.

render()
Returns an image of predicted outputs as a numpy-compatible RGBA image drawn by pyplot.

report_csv()
Returns the logged metadata of predicted bounding boxes.

The returned object is a print-friendly CSV string.

Note that this string might be very long if the dataset is large or if the model tends to generate a lot of detections. The string will contain at least $N_sample \cdot N_target$ lines and each line will have up to N_box detections, unless limited by configuration parameters.

report_json()
Returns the logged metadata of predicted bounding boxes as a JSON formatted string.

report_text()
Must be implemented by inheriting classes. Default report text representation.

reset()
Toggles a reset of the internal state, emptying storage arrays.

supports_detection = True

update(*task*, *input*, *pred*, *target*, *sample*, *loss*, *iter_idx*, *max_iters*, *epoch_idx*, *max_epochs*, *output_path*, ***kwargs*)
Receives the latest predictions and target values from the training session.

The exact signature of this function should match the one of the callbacks defined in *thelper.train.base.Trainer* and specified by thelper.typedefs.IterCallbackParams.

class thelper.train.utils.**PredictionCallback**(*callback_func*, *callback_kwargs=None*)
Bases: *thelper.ifaces.PredictionConsumer*

Callback function wrapper compatible with the consumer interface.

This interface is used to hide user-defined callbacks into the list of prediction consumers given to trainer implementations. The callbacks must always be compatible with the list of arguments defined by thelper.typedefs.IterCallbackParams, but may also receive extra arguments defined in advance and passed to the constructor of this class.

Variables

- **callback_func** – user-defined function to call on every update from the trainer.
- **callback_kwargs** – user-defined extra arguments to provide to the callback function.

__init__(*callback_func*, *callback_kwargs=None*)
Initialize self. See help(type(self)) for accurate signature.

update(**args*, ***kwargs*)
Forwards the latest prediction data from the training session to the user callback.

thelper.train.utils.**create_consumers**(*config*)
Instantiates and returns the prediction consumers defined in the configuration dictionary.

All arguments are expected to be handed in through the configuration via a dictionary named 'params'.

thelper.train.utils.**create_trainer**(*session_name*, *save_dir*, *config*, *model*, *task*, *loaders*, *ckpt_data=None*)
Instantiates the trainer object based on the type contained in the config dictionary.

The trainer type is expected to be in the configuration dictionary's *trainer* field, under the *type* key. For more information on the configuration, refer to *thelper.train.base.Trainer*. The instantiated type must be compatible with the constructor signature of *thelper.train.base.Trainer*. The object's constructor will be given the full config dictionary and the checkpoint data for resuming the session (if available).

If the trainer type is missing, it will be automatically deduced based on the task object.

Parameters

- **session_name** – name of the training session used for printing and to create internal tensorboardX directories.
- **save_dir** – path to the session directory where logs and checkpoints will be saved.
- **config** – full configuration dictionary that will be parsed for trainer parameters and saved in checkpoints.

- **model** – model to train/evaluate; should be compatible with *thelper.nn.utils.Module*.
- **task** – global task interface defining the type of model and training goal for the session.
- **loaders** – a tuple containing the training/validation/test data loaders (a loader can be `None` if empty).
- **ckptdata** – raw checkpoint to parse data from when resuming a session (if `None`, will start from scratch).

Returns The fully-constructed trainer object, ready to begin model training/evaluation.

See also:

thelper.train.base.Trainer

6.1.9 thelper.transforms package

Transformation operations package.

This package contains data transformation classes and wrappers for preprocessing, augmentation, and normalization of data samples.

Submodules

thelper.transforms.composers module

Transformation composers module.

All transforms should aim to be compatible with both numpy arrays and PyTorch tensors. By default, images are processed using `__call__`, meaning that for a given transformation `t`, we apply it via:

```
image_transformed = t(image)
```

All important parameters for an operation should also be passed in the constructor and exposed in the operation's `__repr__` function so that external parsers can discover exactly how to reproduce their behavior. For now, these representations are used for debugging more than anything else.

class thelper.transforms.composers.**Compose**(*transforms*)

Bases: sphinx.ext.autodoc.importer._MockObject

Composes several transforms together (with support for invert ops).

This interface is fully compatible with `torchvision.transforms.Compose`.

See also:

thelper.transforms.composers.CustomStepCompose

__getitem__(*idx*)

Returns the idx-th operation wrapped by the composer.

__init__(*transforms*)

Forwards the list of transformations to the base class.

invert(*sample*)
Tries to invert the transformations applied to a sample.

Will throw if one of the transformations cannot be inverted.

set_epoch(*epoch=0*)
Sets the current epoch number in order to change the behavior of some suboperations.

set_seed(*seed*)
Sets the internal seed to use for stochastic ops.

class thelper.transforms.composers.**CustomStepCompose**(*milestones*, *last_epoch=-1*)
Bases: sphinx.ext.autodoc.importer._MockObject

Composes several transforms together based on an epoch schedule.

This interface is fully compatible with `torchvision.transforms.Compose`. It can be useful if some operations should change their behavior over the course of a training session. Note that all epoch indices are assumed to be 0-based.

Usage example in Python:

```
# We will scale the resolution of input patches based on an arbitrary schedule
# dsize = (16, 16)   if epoch < 2         (16x16 patches before epoch 2)
# dsize = (32, 32)   if 2 <= epoch < 4    (32x32 patches before epoch 4)
# dsize = (64, 64)   if 4 <= epoch < 8    (64x64 patches before epoch 8)
# dsize = (112, 112) if 8 <= epoch < 12   (112x112 patches before epoch 12)
# dsize = (160, 160) if 12 <= epoch < 15  (160x160 patches before epoch 15)
# dsize = (196, 196) if 15 <= epoch < 18  (196x196 patches before epoch 18)
# dsize = (224, 224) if epoch >= 18       (224x224 patches past epoch 18)
transforms = CustomStepCompose(milestones={
    0: thelper.transforms.Resize(dsize=(16, 16)),
    2: thelper.transforms.Resize(dsize=(32, 32)),
    4: thelper.transforms.Resize(dsize=(64, 64)),
    8: thelper.transforms.Resize(dsize=(112, 112)),
    12: thelper.transforms.Resize(dsize=(160, 160)),
    15: thelper.transforms.Resize(dsize=(196, 196)),
    18: thelper.transforms.Resize(dsize=(224, 224)),
})
for epoch in range(100):
    transforms.set_epoch(epoch)
    for sample in loader:
        sample = transforms(sample)
        train(...)
```

Variables

- **stages** – list of epochs where a new scaling factor is to be applied.
- ***transforms*** – list of transformation to apply at each stage.
- **milestones** – original milestones map provided in the constructor.
- **epoch** – index of the current epoch.

See also:

thelper.transforms.composers.Compose

__call__(*img*)
Applies the current stage of transformation operations to a sample.

__getitem__(*idx*)
Returns the idx-th operation wrapped by the composer.

__init__(*milestones*, *last_epoch=-1*)
Receives the milestone stages (or stage lists), and the initialization state.

If the milestones do not include the first epoch (idx = 0), then no transform will be applied until the next specified epoch index. When last_epoch is -1, the training is assumed to start from scratch.

Parameters

- **milestones** – Map of epoch indices tied to transformation stages. Keys must be increasing.
- **last_epoch** – The index of last epoch. Default: -1.

invert(*sample*)
Tries to invert the transformations applied to a sample.

Will throw if one of the transformations cannot be inverted.

set_epoch(*epoch=0*)
Sets the current epoch number in order to change the behavior of some suboperations.

set_seed(*seed*)
Sets the internal seed to use for stochastic ops.

step(*epoch=None*)
Advances the epoch tracker in order to change the behavior of some suboperations.

thelper.transforms.operations module

Transformation operations module.

All transforms should aim to be compatible with both numpy arrays and PyTorch tensors. By default, images are processed using `__call__`, meaning that for a given transformation `t`, we apply it via:

```
image_transformed = t(image)
```

All important parameters for an operation should also be passed in the constructor and exposed in the operation's `__repr__` function so that external parsers can discover exactly how to reproduce their behavior. For now, these representations are used for debugging more than anything else.

class thelper.transforms.operations.**Affine**(*transf*, *out_size=None*, *flags=None*, *border_mode=None*, *border_val=None*)
Bases: `object`

Warps a given image using an affine matrix via OpenCV and numpy.

This operation is deterministic. The code relies on OpenCV, meaning the border arguments must be compatible with `cv2.warpAffine`.

Variables

- **transf** – the 2x3 transformation matrix passed to `cv2.warpAffine`.
- **out_size** – target image size (tuple of width, height). If None, same as original.
- **flags** – extra warp flags forwarded to `cv2.warpAffine`.
- **border_mode** – border extrapolation mode forwarded to `cv2.warpAffine`.

- **border_val** – border constant extrapolation value forwarded to `cv2.warpAffine`.

__call__(*sample*)
Warps a given image using an affine matrix.

Parameters **sample** – the image to warp; should be a 2d or 3d numpy array.

Returns The warped image.

__init__(*transf*, *out_size=None*, *flags=None*, *border_mode=None*, *border_val=None*)
Validates and initializes affine warp parameters.

Parameters

- **transf** – the 2x3 transformation matrix passed to `cv2.warpAffine`.
- **out_size** – target image size (tuple of width, height). If None, same as original.
- **flags** – extra warp flags forwarded to `cv2.warpAffine`.
- **border_mode** – border extrapolation mode forwarded to `cv2.warpAffine`.
- **border_val** – border constant extrapolation value forwarded to `cv2.warpAffine`.

invert(*sample*)
Inverts the warp transformation, but only is the output image has not been cropped before.

class thelper.transforms.operations.**CenterCrop**(*size*, *bordertype=<sphinx.ext.autodoc.importer._MockObject object>*, *borderval=0*)

Bases: `object`

Returns a center crop from a given image via OpenCV and numpy.

This operation is deterministic. The code relies on OpenCV, meaning the border arguments must be compatible with `cv2.copyMakeBorder`.

Variables

- **size** – the size of the target crop (tuple of width, height).
- **relative** – specifies whether the target crop size is relative to the image size or not.
- **bordertype** – argument forwarded to `cv2.copyMakeBorder`.
- **borderval** – argument forwarded to `cv2.copyMakeBorder`.

__call__(*sample*)
Extracts and returns a central crop from the provided image.

Parameters **sample** – the image to generate the crop from; should be a 2d or 3d numpy array.

Returns The center crop.

__init__(*size*, *bordertype=<sphinx.ext.autodoc.importer._MockObject object>*, *borderval=0*)
Validates and initializes center crop parameters.

Parameters

- **size** – size of the target crop, provided as tuple or list. If integer values are used, the size is assumed to be absolute. If floating point values are used, the size is assumed to be relative, and will be determined dynamically for each sample. If a tuple is used, it is assumed to be (width, height).
- **bordertype** – border copy type to use when the image is too small for the required crop size. See `cv2.copyMakeBorder` for more information.

- **borderval** – border value to use when the image is too small for the required crop size. See `cv2.copyMakeBorder` for more information.

invert(*sample*)
Specifies that this operation cannot be inverted, as data loss is incurred during image transformation.

class thelper.transforms.operations.**Duplicator**(*count*, *deepcopy=False*)
Bases: *thelper.transforms.operations.NoTransform*

Duplicates and returns a list of copies of the input sample.

This operation is used in data augmentation pipelines that rely on probabilistic or preset transformations. It can produce a fixed number of simple copies or deep copies of the input samples as required.

Warning: Since the duplicates will be given directly to the data loader as part of the same minibatch, using too many copies can adversely affect gradient descent for that minibatch. To simply increase the total size of the training set while still allowing a proper shuffling of samples and/or to keep the minibatch size intact, we instead recommend setting the `train_scale` configuration value in the data loader. See *thelper.data.utils.create_loaders()* for more information.

Variables

- **count** – number of copies to generate.
- ***deepcopy*** – specifies whether to deep-copy samples or not.

__call__(*sample*)
Generates and returns duplicates of the sample/object.

If a dictionary is provided, its values will be expanded into lists that will contain all duplicates. Otherwise, the duplicates will be returned directly as a list.

Parameters **sample** – the sample/object to duplicate.

Returns A list of duplicated samples, or a dictionary of duplicate lists.

__init__(*count*, *deepcopy=False*)
Validates and initializes duplication parameters.

Parameters

- **count** – number of copies to generate.
- **deepcopy** – specifies whether to deep-copy samples or not.

invert(*sample*)
Returns the first instance of the list of duplicates.

class thelper.transforms.operations.**NoTransform**
Bases: `object`

Used to flag some ops that should not be externally wrapped for sample/key handling.

__call__(*sample*)
Identity transform.

invert(*sample*)
Identity transform.

class thelper.transforms.operations.**NormalizeMinMax**(*min*, *max*, *out_type=<sphinx.ext.autodoc.importer._MockObject object>*)

Bases: object

Normalizes a given image using a set of minimum and maximum values.

The samples will be transformed such that `s = (s - min) / (max - min)`.

Note that this operation is also not restricted to images.

Variables

- **min** – an array of minimum values to subtract with.
- **max** – an array of maximum values to divide with.
- **out_type** – the output data type to cast the normalization result to.

__call__(*sample*)

Normalizes a given sample.

Parameters

- **sample** – the sample to normalize. If given as a PIL image, it
- **will be converted to a numpy array first.**

Returns The warped sample, in a numpy array of type `self.out_type`.

__init__(*min*, *max*, *out_type=<sphinx.ext.autodoc.importer._MockObject object>*)

Validates and initializes normalization parameters.

Parameters

- **min** – an array of minimum values to subtract with.
- **max** – an array of maximum values to divide with.
- **out_type** – the output data type to cast the normalization result to.

invert(*sample*)

Inverts the normalization.

class thelper.transforms.operations.**NormalizeZeroMeanUnitVar**(*mean*, *std*, *out_type=<sphinx.ext.autodoc.importer._M object>*)

Bases: object

Normalizes a given image using a set of mean and standard deviation parameters.

The samples will be transformed such that `s = (s - mean) / std`.

This can be used for whitening; see https://en.wikipedia.org/wiki/Whitening_transformation for more information. Note that this operation is also not restricted to images.

Variables

- **mean** – an array of mean values to subtract from data samples.
- **std** – an array of standard deviation values to divide with.
- **out_type** – the output data type to cast the normalization result to.

__call__(*sample*)

Normalizes a given sample.

Parameters

- **sample** – the sample to normalize. If given as a PIL image, it
- **will be converted to a numpy array first.**

Returns The warped sample, in a numpy array of type `self.out_type`.

__init__(*mean*, *std*, *out_type=<sphinx.ext.autodoc.importer._MockObject object>*)
Validates and initializes normalization parameters.

Parameters

- **mean** – an array of mean values to subtract from data samples.
- **std** – an array of standard deviation values to divide with.
- **out_type** – the output data type to cast the normalization result to.

invert(*sample*)
Inverts the normalization.

class thelper.transforms.operations.**RandomResizedCrop**(*output_size*, *input_size=(0.08, 1.0)*, *ratio=(0.75, 1.33)*, *probability=1.0*, *random_attempts=10*, *min_roi_iou=1.0*, *flags=<sphinx.ext.autodoc.importer._MockObject object>*)

Bases: `object`

Returns a resized crop of a randomly selected image region.

This operation is stochastic, and thus cannot be inverted. Each time the operation is called, a random check will determine whether a transformation is applied or not. The code relies on OpenCV, meaning the interpolation arguments must be compatible with `cv2.resize`.

Variables

- **output_size** – size of the output crop, provided as a single element (`edge_size`) or as a two-element tuple or list (`[width, height]`). If integer values are used, the size is assumed to be absolute. If floating point values are used (i.e. in [0,1]), the output size is assumed to be relative to the original image size, and will be determined at execution time for each sample. If set to `None`, the crop will not be resized.
- **input_size** – range of the input region sizes, provided as a pair of elements (`[min_edge_size, max_edge_size]`) or as a pair of tuples or lists (`[[min_width, min_height], [max_width, max_height]]`). If the pair-of-pairs format is used, the `ratio` argument cannot be used. If integer values are used, the ranges are assumed to be absolute. If floating point values are used (i.e. in [0,1]), the ranges are assumed to be relative to the original image size, and will be determined at execution time for each sample.
- **ratio** – range of minimum/maximum input region aspect ratios to use. This argument cannot be used if the pair-of-pairs format is used for the `input_size` argument.
- **probability** – the probability that the transformation will be applied when called; if not applied, the returned image will be the original.
- **random_attempts** – the number of random sampling attempts to try before reverting to center or most-probably-valid crop generation.
- **min_roi_iou** – minimum roi intersection over union (IoU) required for accepting a tile (in [0,1]).
- **flags** – interpolation flag forwarded to `cv2.resize`.

__call__(*image*, *roi=None*, *mask=None*, *bboxes=None*)
Extracts and returns a random (resized) crop from the provided image.

Parameters

- **image** – the image to generate the crop from. If given as a 2-element list, it is assumed to contain both the image and the roi (passed through a composer).
- **roi** – the roi to check tile intersections with (may be `None`).
- **mask** – a mask to crop simultaneously with the input image (may be `None`).
- **bboxes** – a list or array of bounding boxes to crop with the input image (may be `None`).

Returns The randomly selected and resized crop. If mask and/or bboxes is given, the output will be a dictionary containing the results under the `image`, `mask`, and `bboxes` keys.

__init__(*output_size*, *input_size=(0.08, 1.0)*, *ratio=(0.75, 1.33)*, *probability=1.0*, *random_attempts=10*, *min_roi_iou=1.0*, *flags=<sphinx.ext.autodoc.importer._MockObject object>*)
Validates and initializes center crop parameters.

Parameters

- **output_size** – size of the output crop, provided as a single element (`edge_size`) or as a two-element tuple or list (`[width, height]`). If integer values are used, the size is assumed to be absolute. If floating point values are used (i.e. in [0,1]), the output size is assumed to be relative to the original image size, and will be determined at execution time for each sample. If set to `None`, the crop will not be resized.
- **input_size** – range of the input region sizes, provided as a pair of elements (`[min_edge_size, max_edge_size]`) or as a pair of tuples or lists (`[[min_width, min_height], [max_width, max_height]]`). If the pair-of-pairs format is used, the `ratio` argument cannot be used. If integer values are used, the ranges are assumed to be absolute. If floating point values are used (i.e. in [0,1]), the ranges are assumed to be relative to the original image size, and will be determined at execution time for each sample.
- **ratio** – range of minimum/maximum input region aspect ratios to use. This argument cannot be used if the pair-of-pairs format is used for the `input_size` argument.
- **probability** – the probability that the transformation will be applied when called; if not applied, the returned image will be the original.
- **random_attempts** – the number of random sampling attempts to try before reverting to center or most-probably-valid crop generation.
- **min_roi_iou** – minimum roi intersection over union (IoU) required for producing a tile.
- **flags** – interpolation flag forwarded to `cv2.resize`.

invert(*image*)
Specifies that this operation cannot be inverted, as data loss is incurred during image transformation.

set_seed(*seed*)
Sets the internal seed to use for stochastic ops.

class thelper.transforms.operations.**RandomShift**(*min*, *max*, *probability=1.0*, *flags=None*, *border_mode=None*, *border_val=None*)
Bases: `object`

Randomly translates an image in a provided range via OpenCV and numpy.

This operation is stochastic, and thus cannot be inverted. Each time the operation is called, a random check will determine whether a transformation is applied or not. The code relies on OpenCV, meaning the border arguments must be compatible with `cv2.warpAffine`.

Variables

- **min** – the minimum pixel shift that can be applied stochastically.
- **max** – the maximum pixel shift that can be applied stochastically.
- **probability** – the probability that the transformation will be applied when called.
- **flags** – extra warp flags forwarded to `cv2.warpAffine`.
- **border_mode** – border extrapolation mode forwarded to `cv2.warpAffine`.
- **border_val** – border constant extrapolation value forwarded to `cv2.warpAffine`.

__call__(*sample*)

Translates a given image using a predetermined min/max range.

Parameters **sample** – the image to translate; should be a 2d or 3d numpy array.

Returns The translated image.

__init__(*min*, *max*, *probability=1.0*, *flags=None*, *border_mode=None*, *border_val=None*)

Validates and initializes shift parameters.

Parameters

- **min** – the minimum pixel shift that can be applied stochastically.
- **max** – the maximum pixel shift that can be applied stochastically.
- **probability** – the probability that the transformation will be applied when called.
- **flags** – extra warp flags forwarded to `cv2.warpAffine`.
- **border_mode** – border extrapolation mode forwarded to `cv2.warpAffine`.
- **border_val** – border constant extrapolation value forwarded to `cv2.warpAffine`.

invert(*sample*)

Specifies that this operation cannot be inverted, as it is stochastic, and data loss occurs during transformation.

set_seed(*seed*)

Sets the internal seed to use for stochastic ops.

class thelper.transforms.operations.**Resize**(*dsize*, *fx=0*, *fy=0*, *interp=<sphinx.ext.autodoc.importer._MockObject object>*, *buffer=False*)

Bases: `object`

Resizes a given image using OpenCV and numpy.

This operation is deterministic. The code relies on OpenCV, meaning the interpolation arguments must be compatible with `cv2.resize`.

Variables

- **interp** – interpolation type to use (forwarded to `cv2.resize`)
- **buffer** – specifies whether a destination buffer should be used to avoid allocations
- **dsize** – target image size (tuple of width, height).
- **dst** – buffer used to avoid reallocations if `self.buffer == True`

- **fx** – argument forwarded to `cv2.resize`.
- **fy** – argument forwarded to `cv2.resize`.

__call__(*sample*)
Returns a resized copy of the provided image.

Parameters **sample** – the image to resize; should be a 2d or 3d numpy array.

Returns The resized image. May be allocated on the spot, or be a pointer to a local buffer.

__init__(*dsize*, *fx=0*, *fy=0*, *interp=<sphinx.ext.autodoc.importer._MockObject object>*, *buffer=False*)
Validates and initializes resize parameters.

Parameters

- **dsize** – size of the target image, forwarded to `cv2.resize`.
- **fx** – x-scaling factor, forwarded to `cv2.resize`.
- **fy** – y-scaling factor, forwarded to `cv2.resize`.
- **interp** – resize interpolation type, forwarded to `cv2.resize`.
- **buffer** – specifies whether a destination buffer should be used to avoid allocations

invert(*sample*)
Specifies that this operation cannot be inverted, as data loss is incurred during image transformation.

class thelper.transforms.operations.**Tile**(*tile_size*, *tile_overlap=0.0*, *min_mask_iou=1.0*, *offset_overlap=False*, *bordertype=<sphinx.ext.autodoc.importer._MockObject object>*, *borderval=0*)
Bases: `object`

Returns a list of tiles cut out from a given image.

This operation can perform tiling given an optional mask with a target intersection over union (IoU) score, and with an optional overlap between tiles. The tiling is deterministic and can thus be inverted, but only if a mask is not used, as some image regions may be lost otherwise.

If a mask is used, the first tile position is tested exhaustively by iterating over all input coordinates starting from the top-left corner of the image. Otherwise, the first tile position is set as (0,0). Then, all other tiles are found by offsetting frm these coordinates, and testing for IoU with the mask (if needed).

Variables

- **tile_size** – size of the output tiles, provided as a single element (`edge_size`) or as a two-element tuple or list (`[width, height]`). If integer values are used, the size is assumed to be absolute. If floating point values are used (i.e. in [0,1]), the output size is assumed to be relative to the original image size, and will be determined at execution time for each image.
- **tile_overlap** – overlap allowed between two neighboring tiles; should be a ratio in [0,1].
- **min_mask_iou** – minimum mask intersection over union (IoU) required for accepting a tile (in [0,1]).
- **offset_overlap** – specifies whether the overlap tiling should be offset outside the image or not.
- **bordertype** – border copy type to use when the image is too small for the required crop size. See `cv2.copyMakeBorder` for more information.

- **borderval** – border value to use when the image is too small for the required crop size. See `cv2.copyMakeBorder` for more information.

__call__(*image*, *mask=None*)
Extracts and returns a list of tiles cut out from the given image.

Parameters

- **image** – the image to cut into tiles. If given as a 2-element list, it is assumed to contain both the image and the mask (passed through a composer).
- **mask** – the mask to check tile intersections with (may be `None`).

Returns A list of tiles (numpy-compatible images).

__init__(*tile_size*, *tile_overlap=0.0*, *min_mask_iou=1.0*, *offset_overlap=False*, *bordertype=<sphinx.ext.autodoc.importer._MockObject object>*, *borderval=0*)
Validates and initializes tiling parameters.

Parameters

- **tile_size** – size of the output tiles, provided as a single element (`edge_size`) or as a two-element tuple or list (`[width, height]`). If integer values are used, the size is assumed to be absolute. If floating point values are used (i.e. in [0,1]), the output size is assumed to be relative to the original image size, and will be determined at execution time for each image.
- **tile_overlap** – overlap ratio between two consecutive (neighboring) tiles; should be in [0,1].
- **min_mask_iou** – minimum mask intersection over union (IoU) required for producing a tile.
- **offset_overlap** – specifies whether the overlap tiling should be offset outside the image or not.
- **bordertype** – border copy type to use when the image is too small for the required crop size. See `cv2.copyMakeBorder` for more information.
- **borderval** – border value to use when the image is too small for the required crop size. See `cv2.copyMakeBorder` for more information.

count_tiles(*image*, *mask=None*)
Returns the number of tiles that would be cut out from the given image.

Parameters

- **image** – the image to cut into tiles. If given as a 2-element list, it is assumed to contain both the image and the mask (passed through a composer).
- **mask** – the mask to check tile intersections with (may be `None`).

Returns The number of tiles that would be cut with *thelper.transforms.operations.Tile.__call__()*.

invert(*image*, *mask=None*)
Returns the reconstituted image from a list of tiles, or throws if a mask was used.

class thelper.transforms.operations.**ToColor**
Bases: `object`

Converts a single-channel image to color (RGB).

This operation is deterministic and reversible. It CANNOT be applied to images with more than one channel (HxWx1). The byte ordering (BGR or RGB) does not matter.

__call__(*sample*)
Call self as a function.

__init__()
Does nothing, there's no attribute to store for this operation.

invert(*sample*)
Inverts the operation by calling the 'ToGray' operation.

Note that this operation is probably lossy due to OpenCV's grayscale conversion code which uses "0.21 R + 0.72 G + 0.07 B" to compute the luminosity of a pixel.

class thelper.transforms.operations.**ToGray**
Bases: object

Converts a multi-channel image to grayscale.

This operation is deterministic, but not reversible. It can be applied to images with more than three channels (RGB) — in that case, it will compute their per-pixel mean value. Note that in any case, the last dimension (that corresponds to the channels) will remain and be of size 1.

__call__(*sample*)
Call self as a function.

__init__()
Does nothing, there's no attribute to store for this operation.

invert(*sample*)
Specifies that this operation cannot be inverted, as data loss occurs during transformation.

class thelper.transforms.operations.**ToNumpy**(*reorder_bgr=False*)
Bases: object

Converts and returns an image in numpy format from a `torch.Tensor` or `PIL.Image` format.

This operation is deterministic. The returns image will always be encoded as HxWxC, where if the input has three channels, the ordering might be optionally changed.

Variables **reorder_bgr** – specifies whether the channels should be reordered in OpenCV format.

__call__(*sample*)
Converts and returns an image in numpy format.

Parameters **sample** – the image to convert; should be a tensor, numpy array, or PIL image.

Returns The numpy-converted image.

__init__(*reorder_bgr=False*)
Initializes transformation parameters.

invert(*sample*)
Specifies that this operation cannot be inverted, as the original data type is unknown.

class thelper.transforms.operations.**Transpose**(*axes*)
Bases: object

Transposes an image via numpy.

This operation is deterministic.

Variables

- **axes** – the axes on which to apply the transpose; forwarded to `numpy.transpose`.
- **axes_inv** – used to invert the tranpose; also forwarded to `numpy.transpose`.

__call__(*sample*)
Transposes a given image.

Parameters **sample** – the image to transpose; should be a numpy array.

Returns The transposed image.

__init__(*axes*)
Validates and initializes tranpose parameters.

Parameters **axes** – the axes on which to apply the transpose; forwarded to `numpy.transpose`.

invert(*sample*)
Invert-transposes a given image.

Parameters **sample** – the image to invert-transpose; should be a numpy array.

Returns The invert-transposed image.

class thelper.transforms.operations.**Unsqueeze**(*axis*)
Bases: object

Expands a dimension in the input array via numpy/PyTorch.

This operation is deterministic.

Variables **axis** – the axis on which to apply the expansion.

__call__(*sample*)
Expands a dimension in the input array via numpy/PyTorch.

Parameters **sample** – the array to expand.

Returns The array with an extra dimension.

__init__(*axis*)
Validates and initializes tranpose parameters.

Parameters **axis** – the axis on which to apply the expansion.

invert(*sample*)
Squeezes a dimension in the input array via numpy/PyTorch.

Parameters **sample** – the array to squeeze.

Returns The array with one less dimension.

thelper.transforms.utils module

Transformations utilities module.

This module contains utility functions used to instantiate transformation/augmentation ops.

thelper.transforms.utils.**load_augments**(*config*)
Loads a data augmentation pipeline.

An augmentation pipeline is essentially a specialized transformation pipeline that can be appended or prefixed to the base transforms defined for all samples. Augmentations are typically used to diversify the samples within the training set in order to help model generalization. They can also be applied to validation and test samples in order to get multiple responses for the same input so that they can be averaged/concatenated into a single output.

Usage examples inside a session configuration file:

```
# ...
# the 'loaders' field can contain several augmentation pipelines
# (see 'thelper.data.utils.create_loaders' for more information on these␣
↪pipelines)
"loaders": {
    # ...
    # the 'train_augments' operations are applied to training samples only
    "train_augments": {
        # specifies whether to apply the augmentations before or after the base␣
↪transforms
        "append": false,
        "transforms": [
            {
                # here, we use a single stage, which is actually an augmentor sub-
↪pipeline
                # that is purely probabilistic (i.e. it does not increase input␣
↪sample count)
                "operation": "Augmentor.Pipeline",
                "params": {
                    # the augmentor pipeline defines two operations: rotations␣
↪and flips
                    "rotate_random_90": {"probability": 0.75},
                    "flip_random": {"probability": 0.75}
                }
            }
        ]
    },
    # ...
}
# ...
```

Parameters **config** – the configuration dictionary defining the meta parameters as well as the list of transformation operations of the augmentation pipeline.

Returns A tuple that consists of a pipeline compatible with the `torchvision.transforms` interfaces, and a bool specifying whether this pipeline should be appended or prefixed to the base transforms.

See also:

thelper.transforms.wrappers.AugmentorWrapper
thelper.transforms.utils.load_transforms()
thelper.data.utils.create_loaders()

`thelper.transforms.utils.`**`load_transforms`**(*stages*, *avoid_transform_wrapper=False*)

Loads a transformation pipeline from a list of stages.

Each entry in the provided list will be considered a stage in the pipeline. The ordering of the stages is important, as some transformations might not be compatible if taken out of order. The entries must each be dictionaries that define an operation, its parameters, and some meta-parameters (detailed below).

The `operation` field of each stage will be used to dynamically import a specific type of operation to apply. The `params` field of each stage will then be used to pass parameters to the constructor of this operation.

If an operation is identified as `"Augmentor.Pipeline"` or `"albumentations.Compose"`, it will be specially handled. In both case, the `params` field becomes mandatory in the stage dictionary, and it must specify the Augmentor or albumentations pipeline operation names and parameters (as a dictionary). Two additional optional config fields can then be set for Augmentor pipelines: `input_tensor` (bool) which specifies whether the previous stage provides a `torch.Tensor` to the pipeline (default=False); and `output_tensor` (bool) which specifies whether the output of the pipeline should be converted into a tensor (default=False). For albumentations pipelines, two additional fields are also available, namely `bbox_params` (dict) and `keypoint_params` (dict). For more information on these, refer to the documentation of `albumentations.core.composition.Compose`. Finally, when unpacking dictionaries for albumentations pipelines, the keys associated to bounding boxes/masks/keypoints that must be forwarded to the composer can be specified via the `bboxes_key`, `mask_key`, and `keypoints_key` fields.

All operations can also specify which sample components they should be applied to via the `target_key` field. This field can contain a single key (typically a string), or a list of keys. The operation will be applied at runtime to all values which are found in the samples with one of those keys. If no key is provided for an operation, it will be applied to all array-like components of the sample. Finally, all operations can specify a `linked_fate` field (bool) to specify whether the samples provided in lists should all have the same fate or not (default=True).

Usage examples inside a session configuration file:

```
# ...
# the 'loaders' field may contain several transformation pipelines
# (see 'thelper.data.utils.create_loaders' for more information on these␣
↪pipelines)
"loaders": {
    # ...
    # the 'base_transforms' operations are applied to all loaded samples
    "base_transforms": [
        {
            "operation": "...",
            "params": {
                ...
            },
            "target_key": [ ... ],
            "linked_fate": ...
        },
        {
            "operation": "...",
            "params": {
                ...
            },
            "target_key": [ ... ],
            "linked_fate": ...
        },
        ...
    ],
# ...
```

Parameters **stages** – a list defining a series of transformations to apply as a single pipeline.

Returns A transformation pipeline object compatible with the `torchvision.transforms` interface.

See also:

thelper.transforms.wrappers.AlbumentationsWrapper

thelper.transforms.wrappers.AugmentorWrapper
thelper.transforms.wrappers.TransformWrapper
thelper.transforms.utils.load_augments()
thelper.data.utils.create_loaders()

thelper.transforms.wrappers module

Transformations wrappers module.

The wrapper classes herein are used to either support inline operations on odd sample types (e.g. lists of images) or for external libraries (e.g. Augmentor).

class thelper.transforms.wrappers.**AlbumentationsWrapper**(*transforms*, *bbox_params=None*, *add_targets=None*, *image_key='image'*, *bboxes_key='bboxes'*, *mask_key='mask'*, *keypoints_key='keypoints'*, *probability=1.0*, *cvt_kpts_to_bboxes=False*, *linked_fate=False*)

Bases: object

Albumentations pipeline wrapper that allows dictionary unpacking.

See https://github.com/albu/albumentations for more information.

Variables

- **pipeline** – the augmentor pipeline instance to apply to images.
- **image_key** – the key to fetch images from (when dictionaries are passed in).
- **bboxes_key** – the key to fetch bounding boxes from (when dictionaries are passed in).
- **mask_key** – the key to fetch masks from (when dictionaries are passed in).
- **keypoints_key** – the key to fetch keypoints from (when dictionaries are passed in).
- **cvt_kpts_to_bboxes** – specifies whether keypoints should be converted to bboxes for compatbility.
- **linked_fate** – specifies whether input list samples should all have the same fate or not.

See also:

thelper.transforms.utils.load_transforms()

__call__(*sample*, *force_linked_fate=False*, *op_seed=None*)

Transforms a (dict) sample, a single image, or a list of images using the augmentor pipeline.

Parameters

- **sample** – the sample or image(s) to transform (can also contain embedded lists/tuples of images).
- **force_linked_fate** – override flag for recursive use allowing forced linking of arrays.

- **op_seed** – seed to set before calling the wrapped operation.

Returns The transformed image(s), with the same list/tuple formatting as the input.

__init__(*transforms*, *bbox_params=None*, *add_targets=None*, *image_key='image'*, *bboxes_key='bboxes'*, *mask_key='mask'*, *keypoints_key='keypoints'*, *probability=1.0*, *cvt_kpts_to_bboxes=False*, *linked_fate=False*)
Receives and stores an augmentor pipeline for later use.

The pipeline itself is instantiated in *thelper.transforms.utils.load_transforms()*.

set_epoch(*epoch=0*)
Sets the current epoch number in order to change the behavior of some suboperations.

set_seed(*seed*)
Sets the internal seed to use for stochastic ops.

class thelper.transforms.wrappers.**AugmentorWrapper**(*pipeline*, *target_keys=None*, *linked_fate=True*)

Bases: object

Augmentor pipeline wrapper that allows pickling and multi-threading.

See https://github.com/mdbloice/Augmentor for more information. This wrapper was last updated to work with version 0.2.2 — more recent versions introduced yet unfixed (as of 2018/08) issues on some platforms.

All original transforms are supported here. This wrapper also fixes the list output bug for single-image samples when using operations individually.

Variables

- **pipeline** – the augmentor pipeline instance to apply to images.
- **target_keys** – the sample keys to apply the pipeline to (when dictionaries are passed in).
- **linked_fate** – specifies whether input list samples should all have the same fate or not.

See also:

thelper.transforms.utils.load_transforms()

__call__(*sample*, *force_linked_fate=False*, *op_seed=None*, *in_cvts=None*)
Transforms a (dict) sample, a single image, or a list of images using the augmentor pipeline.

Parameters

- **sample** – the sample or image(s) to transform (can also contain embedded lists/tuples of images).
- **force_linked_fate** – override flag for recursive use allowing forced linking of arrays.
- **op_seed** – seed to set before calling the wrapped operation.
- **in_cvts** – holds the input conversion flag array (for recursive usage).

Returns The transformed image(s), with the same list/tuple formatting as the input.

__init__(*pipeline*, *target_keys=None*, *linked_fate=True*)
Receives and stores an augmentor pipeline for later use.

The pipeline itself is instantiated in *thelper.transforms.utils.load_transforms()*.

set_epoch(*epoch=0*)
Sets the current epoch number in order to change the behavior of some suboperations.

set_seed(*seed*)
Sets the internal seed to use for stochastic ops.

class thelper.transforms.wrappers.**TransformWrapper**(*operation*, *params=None*, *probability=1*, *convert_pil=False*, *target_keys=None*, *linked_fate=True*)

Bases: object

Transform wrapper that allows operations on samples, lists, tuples, and single elements.

Can be used to wrap the operations in thelper.transforms or in torchvision.transforms that only accept array-like objects as input. Will optionally force-convert content to PIL images.

Can also be used to transform a list/tuple of images uniformly based on a shared dice roll, or to ensure that each image is transformed independently.

Warning: Stochastic transforms (e.g. torchvision.transforms.RandomCrop) will always treat each image in a list differently. If the same operations are to be applied to all images, you should consider using a series non-stochastic operations wrapped inside an instance of torchvision.transforms.RandomApply, or simply provide the probability of applying the transforms to this wrapper's constructor.

Variables

- **operation** – the wrapped operation (callable object or class name string to import).
- **params** – the parameters that are passed to the operation when init'd or called.
- **probability** – the probability that the wrapped operation will be applied.
- **convert_pil** – specifies whether images should be converted into PIL format or not.
- **target_keys** – the sample keys to apply the transform to (when dictionaries are passed in).
- **linked_fate** – specifies whether images given in a list/tuple should have the same fate or not.

__call__(*sample*, *force_linked_fate=False*, *op_seed=None*, *in_cvts=None*)
Transforms a (dict) sample, a single image, or a list of images using a wrapped operation.

Parameters

- **sample** – the sample or image(s) to transform (can also contain embedded lists/tuples of images).
- **force_linked_fate** – override flag for recursive use allowing forced linking of arrays.
- **op_seed** – seed to set before calling the wrapped operation.
- **in_cvts** – holds the input conversion flag array (for recursive usage).

Returns The transformed image(s), with the same list/tuple formatting as the input.

__init__(*operation*, *params=None*, *probability=1*, *convert_pil=False*, *target_keys=None*, *linked_fate=True*)
Receives and stores a torchvision transform operation for later use.

If the operation is given as a string, it is assumed to be a class name and it will be imported. The parameters (if any) will then be given to the constructor of that class. Otherwise, the operation is assumed to be a callable object, and its parameters (if any) will be provided at call-time.

Parameters

- **operation** – the wrapped operation (callable object or class name string to import).
- **params** – the parameters that are passed to the operation when init'd or called.
- **probability** – the probability that the wrapped operation will be applied.
- **convert_pil** – specifies whether images should be forced into PIL format or not.
- **target_keys** – the sample keys to apply the pipeline to (when dictionaries are passed in).
- **linked_fate** – specifies whether images given in a list/tuple should have the same fate or not.

set_epoch(*epoch=0*)
Sets the current epoch number in order to change the behavior of some suboperations.

set_seed(*seed*)
Sets the internal seed to use for stochastic ops.

6.1.10 thelper.viz package

Visualization package.

In contrast with *thelper.draw*, this package regroups utilities and tools used to create visualizations that can be used to debug and understand the behavior of models. For example, it contains a t-SNE module that can create a projection of high-dimensional embeddings created by a model, and different modules to visualize the image regions that cause certain activations inside a model. All these techniques are used to create logs and/or images which in turn can be displayed using the *thelper.draw* module.

thelper.viz.**visualize**(*model*, *task*, *loader*, *viz_type*, ***kwargs*)
Dispatches a visualization call to the proper package module.

Submodules

thelper.viz.tsne module

Tools related to the t-Distributed Stochastic Neighbor Embedding (t-SNE, or TSNE).

For more information on t-SNE, see https://lvdmaaten.github.io/tsne/ for the original author's repository, or https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html for the `scikit-learn` tools.

thelper.viz.tsne.**plot**(*projs*, *targets*, *preds*, *color_map=None*, *task=None*, ***kwargs*)
Returns a matplotlib figure of a set of projected embeddings with optional target/predicted labels.

thelper.viz.tsne.**visualize**(*model*, *task*, *loader*, *draw=False*, *color_map=None*, *max_samples=None*, *return_meta=False*, ***kwargs*)
Creates (and optionally displays) a 2D t-SNE visualization of sample embeddings.

By default, all samples from the data loader will be projected using the model and used for the visualization. If the task is related to classification, the prediction and groundtruth labels will be highlighting using various colors.

If the model does not possess a `get_embedding` attribute, its raw output will be used for projections. Otherwise, `get_embedding` will be called.

Parameters

- **model** – the model which will be used to produce embeddings.
- **task** – the task object used to decode predictions and color samples (if possible).
- **loader** – the data loader used to get data samples to project.
- **draw** – boolean flag used to toggle internal display call on or off.
- **color_map** – map of RGB triplets used to color predictions (for classification only).
- **max_samples** – maximum number of samples to draw from the data loader.
- **return_meta** – toggles whether sample metadata should be provided as output or not.

Returns A dictionary of the visualization result (an RGB image in numpy format), a list of projected embedding coordinates, the labels of the samples, and the predictions of the samples.

thelper.viz.umap module

Tools related to the Uniform Manifold Approximation and Projection (UMAP).

For more information on UMAP, see https://github.com/lmcinnes/umap for the original author's repository.

`thelper.viz.umap.`**`visualize`**(*model*, *task*, *loader*, *draw=False*, *color_map=None*, *max_samples=None*, *return_meta=False*, ***kwargs*)

Creates (and optionally displays) a 2D UMAP visualization of sample embeddings.

By default, all samples from the data loader will be projected using the model and used for the visualization. If the task is related to classification, the prediction and groundtruth labels will be highlighting using various colors.

If the model does not possess a `get_embedding` attribute, its raw output will be used for projections. Otherwise, `get_embedding` will be called.

Parameters

- **model** – the model which will be used to produce embeddings.
- **task** – the task object used to decode predictions and color samples (if possible).
- **loader** – the data loader used to get data samples to project.
- **draw** – boolean flag used to toggle internal display call on or off.
- **color_map** – map of RGB triplets used to color predictions (for classification only).
- **max_samples** – maximum number of samples to draw from the data loader.
- **return_meta** – toggles whether sample metadata should be provided as output or not.

Returns A dictionary of the visualization result (an RGB image in numpy format), a list of projected embedding coordinates, the labels of the samples, and the predictions of the samples.

6.2 Submodules

6.3 thelper.cli module

Command-line module, for use with a `__main__` entrypoint.

This module contains the primary functions used to create or resume a training session, to start a visualization session, or to start an annotation session. The basic argument that needs to be provided by the user to create any kind of session is a configuration dictionary. For sessions that produce outputs, the path to a directory where to save the data is also needed.

thelper.cli.**annotate_data**(*config*, *save_dir*)

Launches an annotation session for a dataset using a specialized GUI tool.

Note that the annotation type must be supported by the GUI tool. The annotations created by the user during the session will be saved in the session directory.

Parameters

- **config** – a dictionary that provides all required dataset and GUI tool configuration parameters; see *thelper.data.utils.create_parsers()* and *thelper.gui.utils.create_annotator()* for more information.
- **save_dir** – the path to the root directory where the session directory should be saved. Note that this is not the path to the session directory itself, but its parent, which may also contain other session directories.

See also:

thelper.gui.annotators.Annotator()
thelper.gui.annotators.ImageSegmentAnnotator()

thelper.cli.**create_session**(*config*, *save_dir*)

Creates a session to train a model.

All generated outputs (model checkpoints and logs) will be saved in a directory named after the session (the name itself is specified in `config`), and located in `save_dir`.

Parameters

- **config** – a dictionary that provides all required data configuration and trainer parameters; see *thelper.train.base.Trainer* and *thelper.data.utils.create_loaders()* for more information. Here, it is only expected to contain a `name` field that specifies the name of the session.
- **save_dir** – the path to the root directory where the session directory should be saved. Note that this is not the path to the session directory itself, but its parent, which may also contain other session directories.

See also:

thelper.train.base.Trainer

thelper.cli.**export_model**(*config*, *save_dir*)

Launches a model exportation session.

This function will export a model defined via a configuration file into a new checkpoint that can be loaded elsewhere. The model can be built using the framework, or provided via its type, construction parameters, and weights. Its exported format will be compatible with the framework, and may also be an optimized/compiled version obtained using PyTorch's JIT tracer.

The configuration dictionary must minimally contain a 'model' section that provides details on the model to be exported. A section named 'export' can be used to provide settings regarding the exportation approaches to use, and the task interface to save with the model. If a task is not explicitly defined in the 'export' section, the session configuration will be parsed for a 'datasets' section that can be used to define it. Otherwise, it must be provided through the model.

The exported checkpoint containing the model will be saved in the session's output directory.

Parameters

- **config** – a dictionary that provides all required data configuration parameters; see *thelper.nn.utils.create_model()* for more information.
- **save_dir** – the path to the root directory where the session directory should be saved. Note that this is not the path to the session directory itself, but its parent, which may also contain other session directories.

See also:

thelper.nn.utils.create_model()

thelper.cli.**inference_session**(*config*, *save_dir=None*, *ckpt_path=None*)

Executes an inference session on samples with a trained model checkpoint.

In order to run inference, a model is mandatory and therefore expected to be provided in the configuration. Similarly, a list of input sample file paths are expected for which to run inference on. These inputs can provide additional data according to how they are being parsed by lower level operations.

The session will save the current configuration as *config-infer.json* and the employed model's training configuration as *config-train.json*. Other outputs depend on the specific implementation of the session runner.

Parameters

- **config** – a dictionary that provides all required data configuration.
- **save_dir** – the path to the root directory where the session directory should be saved. Note that this is not the path to the session directory itself, but its parent, which may also contain other session directories. If not provided, will use the best value extracted from either the configuration path or the configuration dictionary itself.
- **ckpt_path** – explicit checkpoint path to use for loading a model to execute inference. Otherwise look for a model definition in the configuration.

See also:

thelper.data.geo.utils.prepare_raster_metadata()
thelper.data.geo.utils.sliding_window_inference()

thelper.cli.**main**(*args=None*, *argparser=None*)

Main entrypoint to use with console applications.

This function parses command line arguments and dispatches the execution based on the selected operating mode. Run with `--help` for information on the available arguments.

Warning: If you are trying to resume a session that was previously executed using a now unavailable GPU, you will have to force the checkpoint data to be loaded on CPU using `--map-location=cpu` (or using `-m=cpu`).

See also:

thelper.cli.create_session()
thelper.cli.resume_session()
thelper.cli.visualize_data()
thelper.cli.annotate_data()
thelper.cli.split_data()
thelper.cli.inference_session()

thelper.cli.**make_argparser**()

Creates the (default) argument parser to use for the main entrypoint.

The argument parser will contain different "operating modes" that dictate the high-level behavior of the CLI. This function may be modified in branches of the framework to add project-specific features.

thelper.cli.**resume_session**(*ckptdata*, *save_dir*, *config=None*, *eval_only=False*, *task_compat=None*)

Resumes a previously created training session.

Since the saved checkpoints contain the original session's configuration, the `config` argument can be set to `None` if the session should simply pick up where it was interrupted. Otherwise, the `config` argument can be set to a new configuration that will override the older one. This is useful when fine-tuning a model, or when testing on a new dataset.

Warning: If a session is resumed with an overriding configuration, the user must make sure that the inputs/outputs of the older model are compatible with the new parameters. For example, with classifiers, this means that the number of classes parsed by the dataset (and thus to be predicted by the model) should remain the same. This is a limitation of the framework that should be addressed in a future update.

Warning: A resumed session will not be compatible with its original RNG states if the number of workers used is changed. To get 100% reproducible results, make sure you run with the same worker count.

Parameters

- **ckptdata** – raw checkpoint data loaded via `torch.load()`; it will be parsed by the various parts of the framework that need to reload their previous state.
- **save_dir** – the path to the root directory where the session directory should be saved. Note that this is not the path to the session directory itself, but its parent, which may also contain other session directories.
- **config** – a dictionary that provides all required data configuration and trainer parameters; see *thelper.train.base.Trainer* and *thelper.data.utils.create_loaders()* for more information. Here, it is only expected to contain a `name` field that specifies the name of the session.

- **eval_only** – specifies whether training should be resumed or the model should only be evaluated.
- **task_compat** – specifies how to handle discrepancy between old task from checkpoint and new task from config

See also:

thelper.train.base.Trainer

thelper.cli.**setup**(*args=None*, *argparser=None*)

Sets up the argument parser (if not already done externally) and parses the input CLI arguments.

This function may return an error code (integer) if the program should exit immediately. Otherwise, it will return the parsed arguments to use in order to redirect the execution flow of the entrypoint.

thelper.cli.**split_data**(*config*, *save_dir*)

Launches a dataset splitting session.

This mode will generate an HDF5 archive that contains the split datasets defined in the session configuration file. This archive can then be reused in a new training session to guarantee a fixed distribution of training, validation, and testing samples. It can also be used outside the framework in order to reproduce an experiment.

The configuration dictionary must minimally contain two sections: 'datasets' and 'loaders'. A third section, 'split', can be used to provide settings regarding the archive packing and compression approaches to use.

The HDF5 archive will be saved in the session's output directory.

Parameters

- **config** – a dictionary that provides all required data configuration parameters; see *thelper.data.utils.create_loaders()* for more information.
- **save_dir** – the path to the root directory where the session directory should be saved. Note that this is not the path to the session directory itself, but its parent, which may also contain other session directories.

See also:

thelper.data.utils.create_loaders()
thelper.data.utils.create_hdf5()
thelper.data.parsers.HDF5Dataset

thelper.cli.**visualize_data**(*config*)

Displays the images used in a training session.

This mode does not generate any output, and is only used to visualize the (transformed) images used in a training session. This is useful to debug the data augmentation and base transformation pipelines and make sure the modified images are valid. It does not attempt to load a model or instantiate a trainer, meaning the related fields are not required inside `config`.

If the configuration dictionary includes a 'loaders' field, it will be parsed and used. Otherwise, if only a 'datasets' field is available, basic loaders will be instantiated to load the data. The 'loaders' field can also be ignored if 'ignore_loaders' is found within the 'viz' section of the config and set to `True`. Each minibatch will be displayed via pyplot or OpenCV. The display will block and wait for user input, unless 'block' is set within the 'viz' section's 'kwargs' config as `False`.

Parameters config – a dictionary that provides all required data configuration parameters; see *thelper.data.utils.create_loaders()* for more information.

See also:

thelper.data.utils.create_loaders()
thelper.data.utils.create_parsers()

6.4 thelper.concepts module

Framework concepts module for high-level interface documenting.

The decorators and type checkers defined in this module help highlight the purpose of classes and functions with respect to high-level ML tasks.

thelper.concepts.**SUPPORT_PREFIX = 'supports_'**
Prefix that is applied before any 'concept' decorator.

thelper.concepts.**apply_support**(*func_or_cls=None*, *concept=None*)
Utility decorator that allows marking a function or a class as *supporting* a certain `concept`.

Notes

`concept` support by the function or class is marked only as documentation reference, no strict validation is accomplished to ensure that further underlying requirements are met for the *concept*.

See also:

thelper.concepts.classification()
thelper.concepts.detection()
thelper.concepts.segmentation()
thelper.concepts.regression()

thelper.concepts.**classification**(*func_or_cls=None*)
Decorator that allows marking a function or class as *supporting* the image classification task.

Example:

```
@thelper.concepts.classification
class ClassifObject():
    pass

c = ClassifObject()
c.supports_classification
> True

thelper.concepts.supports(c, "classification")
> True
```

thelper.concepts.**detection**(*func_or_cls=None*)
Decorator that allows marking a function or class as *supporting* the object detection task.

Example:

```
@thelper.concepts.detection
class DetectObject():
    pass

d = DetectObject()
d.supports_detection
> True

thelper.concepts.supports(d, "detection")
> True
```

thelper.concepts.**regression**(*func_or_cls=None*)

Decorator that allows marking a function or class as *supporting* the generic regression task.

Example:

```
@thelper.concepts.regression
class RegrObject():
    pass

r = RegrObject()
r.supports_regression
> True

thelper.concepts.supports(r, "regression")
> True
```

thelper.concepts.**segmentation**(*func_or_cls=None*)

Decorator that allows marking a function or class as *supporting* the image segmentation task.

Example:

```
@thelper.concepts.segmentation
class SegmentObject():
    pass

s = SegmentObject()
s.supports_segmentation
> True

thelper.concepts.supports(s, "segmentation")
> True
```

thelper.concepts.**supports**(*thing*, *concept*)

Utility method to evaluate if `thing` *supports* a given `concept` as defined by decorators.

Parameters

- **thing** – any type, function, method, class or object instance to evaluate if it is marked by the concept
- **concept** – concept to check

See also:

thelper.concepts.classification()
thelper.concepts.detection()

thelper.concepts.segmentation()
thelper.concepts.regression()

6.5 thelper.draw module

Drawing/display utilities module.

These functions currently rely on OpenCV and/or matplotlib.

thelper.draw.**apply_color_map**(*image*, *colormap*, *dst=None*)
Applies a color map to an image of 8-bit color indices; works similarly to cv2.applyColorMap (v3.3.1).

thelper.draw.**draw**(*task*, *input*, *pred=None*, *target=None*, *block=False*, *ch_transpose=True*, *flip_bgr=False*, *redraw=None*, ***kwargs*)
Draws and returns a figure of a model input/predictions/targets using pyplot or OpenCV.

thelper.draw.**draw_bbox**(*image*, *tl*, *br*, *text*, *color*, *box_thickness=2*, *font_thickness=1*, *font_scale=0.4*, *show=False*, *block=False*, *win_name='bbox'*)
Draws a single bounding box on a given image (used in *thelper.draw.draw_bboxes()*).

thelper.draw.**draw_bboxes**(*images*, *preds=None*, *bboxes=None*, *color_map=None*, *redraw=None*, *block=False*, *min_confidence=0.5*, *class_map=None*, ***kwargs*)
Draws a set of bounding box prediction results on images.

Parameters

- **images** – images with first dimension as list index, and other dimensions are each image's content
- **preds** – predicted bounding boxes per image to be displayed, must match images count if provided
- **bboxes** – ground truth (targets) bounding boxes per image to be displayed, must match images count if provided
- **color_map** – mapping of class-id to color to be applied to drawn bounding boxes on the image
- **redraw** – existing figure and axes to reuse for drawing the new images and bounding boxes
- **block** – indicate whether to block execution until all figures have been closed or not
- **min_confidence** – ignore display of bounding boxes that have a confidence below this value, if available
- **class_map** – alternative class-id to class-name mapping to employ for display. This overrides the default class names retrieved from each bounding box's attributed task. Useful for displaying generic bounding boxes obtained from raw input values without a specific task.
- **kwargs** – other arguments to be passed down to further drawing functions or drawing settings (amongst other settings, box_thickness, font_thickness and font_scale can be provided)

thelper.draw.**draw_classifs**(*images*, *preds=None*, *labels=None*, *is_multi_label=False*, *class_names_map=None*, *redraw=None*, *block=False*, ***kwargs*)
Draws and returns a set of classification results.

thelper.draw.**draw_confmat**(*confmat*, *class_list*, *size_inch=(5, 5)*, *dpi=160*, *normalize=False*, *keep_unset=False*, *show=False*, *block=False*)
Draws and returns an a confusion matrix figure using pyplot.

thelper.draw.**draw_errbars**(*labels*, *min_values*, *max_values*, *stddev_values*, *mean_values*, *xlabel=''*, *ylabel='Raw Value'*, *show=False*, *block=False*)
Draws and returns an error bar histogram figure using pyplot.

thelper.draw.**draw_histogram**(*data*, *bins=50*, *xlabel=''*, *ylabel='Proportion'*, *show=False*, *block=False*)
Draws and returns a histogram figure using pyplot.

thelper.draw.**draw_images**(*images*, *captions=None*, *redraw=None*, *show=True*, *block=False*, *use_cv2=True*, *cv2_flip_bgr=True*, *img_shape=None*, *max_img_size=None*, *grid_size_x=None*, *grid_size_y=None*, *caption_opts=None*, *window_name=None*)
Draws a set of images with optional captions.

thelper.draw.**draw_pascalvoc_curve**(*metrics*, *size_inch=(5, 5)*, *dpi=160*, *show=False*, *block=False*)
Draws and returns a precision-recall curve according to pascalvoc metrics.

thelper.draw.**draw_popbars**(*labels*, *counts*, *xlabel=''*, *ylabel='Pop. Count'*, *show=False*, *block=False*)
Draws and returns a bar histogram figure using pyplot.

thelper.draw.**draw_predicts**(*images*, *preds=None*, *targets=None*, *swap_channels=False*, *redraw=None*, *block=False*, ***kwargs*)
Draws and returns a set of generic prediction results.

thelper.draw.**draw_roc_curve**(*fpr*, *tpr*, *labels=None*, *size_inch=(5, 5)*, *dpi=160*, *show=False*, *block=False*)
Draws and returns an ROC curve figure using pyplot.

thelper.draw.**draw_segments**(*images*, *preds=None*, *masks=None*, *color_map=None*, *redraw=None*, *block=False*, *segm_threshold=None*, *target_class=None*, *target_threshold=None*, ***kwargs*)
Draws and returns a set of segmentation results.

thelper.draw.**fig2array**(*fig*)
Transforms a pyplot figure into a numpy-compatible RGB array.

thelper.draw.**get_bgr_from_hsl**(*hue*, *sat*, *light*)
Converts a single HSL triplet (0-360 hue, 0-1 sat & lightness) into an 8-bit RGB triplet.

thelper.draw.**get_displayable_heatmap**(*array*, *convert_rgb=True*)
Returns a 'displayable' array that has been min-maxed and mapped to color triplets.

thelper.draw.**get_displayable_image**(*image*, *grayscale=False*)
Returns a 'displayable' image that has been normalized and padded to three channels.

thelper.draw.**get_label_color_mapping**(*idx*)
Returns the PASCAL VOC color triplet for a given label index.

thelper.draw.**get_label_html_color_code**(*idx*)
Returns the PASCAL VOC HTML color code for a given label index.

thelper.draw.**safe_crop**(*image*, *tl*, *br*, *bordertype=<sphinx.ext.autodoc.importer._MockObject object>*, *borderval=0*, *force_copy=False*)
Safely crops a region from within an image, padding borders if needed.

Parameters

- **image** – the image to crop (provided as a numpy array).
- **tl** – a tuple or list specifying the (x,y) coordinates of the top-left crop corner.
- **br** – a tuple or list specifying the (x,y) coordinates of the bottom-right crop corner.

- **bordertype** – border copy type to use when the image is too small for the required crop size. See `cv2.copyMakeBorder` for more information.
- **borderval** – border value to use when the image is too small for the required crop size. See `cv2.copyMakeBorder` for more information.
- **force_copy** – defines whether to force a copy of the target image region even when it can be avoided.

Returns The cropped image.

6.6 thelper.ifaces module

Common object interfaces module.

The interfaces defined here are fairly generic and used to eliminate issues related to circular module importation.

class thelper.ifaces.**ClassNamesHandler**(*class_names: Optional[Iterable[AnyStr]] = None*, **args*, ***kwargs*)

Bases: abc.ABC

Generic interface to handle class names operations for inheriting classes.

Variables

- ***class_names*** – holds the list of class label names.
- ***class_indices*** – holds a mapping (dict) of class-names-to-label-indices.

__init__(*class_names: Optional[Iterable[AnyStr]] = None*, **args*, ***kwargs*) → None
Initializes the class names array, if an object is provided.

class_indices
Returns the class-name-to-index map used for encoding labels as integers.

class_names
Returns the list of class names considered "of interest" by the derived class.

class thelper.ifaces.**FormatHandler**(*format: Optional[AnyStr] = 'text'*, **args*, ***kwargs*)

Bases: abc.ABC

Generic interface to handle format output operations for inheriting classes.

If `format` is specified and matches a supported one (with a matching `report_<format>` method), this method is used to generate the output. Defaults to `"text"` if not specified or provided value is not found within supported formatting methods.

Variables

- **format** – format to be used for producing the report (default: "text")
- **ext** – extension associated with generated format (default: "txt")

__init__(*format: Optional[AnyStr] = 'text'*, **args*, ***kwargs*) → None
Initialize self. See help(type(self)) for accurate signature.

report(*format: Optional[AnyStr] = None*) → Optional[AnyStr]
Returns the report as a print-friendly string, matching the specified format if specified in configuration.

Parameters **format** – format to be used for producing the report (default: initialization attribute or "text" if invalid)

report_text() → Optional[AnyStr]
Must be implemented by inheriting classes. Default report text representation.

solve_format(*format: Optional[AnyStr] = None*) → None

class thelper.ifaces.**PredictionConsumer**
Bases: abc.ABC

Abstract model prediction consumer class.

This interface defines basic functions required so that *thelper.train.base.Trainer* can figure out how to instantiate and update a model prediction consumer. The most notable class derived from this interface is *thelper.optim.metrics.Metric* which is used to monitor the improvement of a model during a training session. Other prediction consumers defined in *thelper.train.utils* will instead log predictions to local files, create graphs, etc.

reset() → None
Resets the internal state of the consumer.

May be called for example by the trainer between two evaluation epochs. The default implementation does nothing, and if a reset behavior is needed, it should be implemented by the derived class.

update(*task: thelper.tasks.utils.Task, input: <sphinx.ext.autodoc.importer._MockObject object at 0x7f4250534a58>, pred: Union[<sphinx.ext.autodoc.importer._MockObject object at 0x7f42505349b0>, <sphinx.ext.autodoc.importer._MockObject object at 0x7f4250534ac8>, List[List[thelper.tasks.detect.BoundingBox]], <sphinx.ext.autodoc.importer._MockObject object at 0x7f4250534cc0>], target: Union[<sphinx.ext.autodoc.importer._MockObject object at 0x7f42505348d0>, <sphinx.ext.autodoc.importer._MockObject object at 0x7f4250534b00>, List[List[thelper.tasks.detect.BoundingBox]], <sphinx.ext.autodoc.importer._MockObject object at 0x7f4250534ba8>], sample: Dict[Union[AnyStr, int], Any], loss: Optional[float], iter_idx: int, max_iters: int, epoch_idx: int, max_epochs: int, output_path: AnyStr, **kwargs*) → None
Receives the latest prediction and groundtruth tensors from the training session.

The data given here will be "consumed" internally, but it should NOT be modified. For example, a classification accuracy metric would accumulate the correct number of predictions in comparison to groundtruth labels, while a plotting logger would add new corresponding dots to a curve.

Remember that input, prediction, and target tensors received here will all have a batch dimension!

The exact signature of this function should match the one of the callbacks defined in *thelper.train.base.Trainer* and specified by thelper.typedefs.IterCallbackParams.

6.7 thelper.typedefs module

Typing definitions for thelper.

6.8 thelper.utils module

General utilities module.

This module only contains non-ML specific functions, i/o helpers, and matplotlib/pyplot drawing calls.

class thelper.utils.**Struct**(***kwargs*)
Bases: object

Generic runtime-defined C-like data structure (maps constructor elements to fields).

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

thelper.utils.**check_func_signature**(*func*, *params*)
Checks whether the signature of a function matches the expected parameter list.

thelper.utils.**check_installed**(*package_name*)
Attempts to import a specified package by name, returning a boolean indicating success.

thelper.utils.**check_version**(*version_check*, *version_required*)
Verifies that the checked version is not greater than the required one (ie: not a future version).

Version format is `MAJOR[.MINOR[.PATCH[[-]<RELEASE>]]]`.

Note that for `RELEASE` part, comparison depends on alphabetical order if all other previous parts were equal (i.e.: `alpha` will be lower than `beta`, which in turn is lower than `rc` and so on). The `-` is optional and will be removed for comparison (i.e.: `0.5.0-rc` is exactly the same as `0.5.0rc` and the additional `-` will not result in evaluating `0.5.0a0` as a greater version because of `-` being lower ascii than `a`).

Parameters

- **version_check** – the version string that needs to be verified and compared for lower than the required version.
- **version_required** – the control version against which the check is done.

Returns Tuple of the validated check, and lists of both parsed version parts as `[MAJOR, MINOR, PATCH, 'RELEASE']`. The returned lists are *guaranteed* to be formed of 4 elements, adding 0 or '' as applicable for missing parts.

thelper.utils.**clipstr**(*s*, *size*, *fill=' '*)
Clips a string to a specific length, with an optional fill character.

thelper.utils.**create_hdf5_dataset**(*fd*, *name*, *max_len*, *batch_like*, *compression='chunk_lz4'*, *chunk_size=None*, *flatten=True*)
Creates an HDF5 dataset inside the provided HDF5.File object descriptor.

thelper.utils.**decode_data**(*data*, *approach='lz4'*, ***kwargs*)
Decodes a binary array using a given coding approach.

Parameters

- **data** – the binary array to decode.
- **approach** – the encoding; supports *none*, *lz4*, *jpg*, *png*.

See also:

thelper.utils.encode_data()

thelper.utils.**download_file**(*url*, *root*, *filename*, *md5=None*)
Downloads a file from a given URL to a local destination.

Parameters

- **url** – path to query for the file (query will be based on urllib).
- **root** – destination folder where the file should be saved.
- **filename** – destination name for the file.
- **md5** – optional, for md5 integrity check.

Returns The path to the downloaded file.

thelper.utils.**encode_data**(*data*, *approach='lz4'*, ***kwargs*)
Encodes a numpy array using a given coding approach.

Parameters

- **data** – the numpy array to encode.
- **approach** – the encoding; supports *none*, *lz4*, *jpg*, *png*.

See also:

thelper.utils.decode_data()

thelper.utils.**extract_tar**(*filepath*, *root*, *flags='r:gz'*)
Extracts the content of a tar file to a specific location.

Parameters

- **filepath** – location of the tar archive.
- **root** – where to extract the archive's content.
- **flags** – extra flags passed to `tarfile.open`.

thelper.utils.**fetch_hdf5_sample**(*dset*, *idx*, *dtype='auto'*, *shape='auto'*, *compression='auto'*, ***decompr_kwargs*)
Returns a sample from the specified HDF5 dataset object.

thelper.utils.**fill_hdf5_sample**(*dset*, *dset_idx*, *array_idx*, *array*, *compression='chunk_lz4'*, ***compr_kwargs*)
Fills a sample inside the specified HDF5 dataset object.

thelper.utils.**get_available_cuda_devices**(*attempts_per_device=5*)
Tests all visible cuda devices and returns a list of available ones.

Returns List of available cuda device IDs (integers). An empty list means no cuda device is available, and the app should fallback to cpu.

thelper.utils.**get_caller_name**(*skip=2*, *base_class=False*)
Returns the name of a caller in the format module.class.method.

Parameters

- **skip** – specifies how many levels of stack to skip while getting the caller.
- **base_class** – specified if the base class should be returned or the top-most class in case of inheritance If the caller is not a class, this doesn't do anything.

Returns An empty string is returned if skipped levels exceed stack height; otherwise, returns the requested caller name.

thelper.utils.**get_checkpoint_session_root**(*ckpt_path*)
Returns the session root directory associated with a checkpoint path.

The given path can point to a checkpoint file or to a directory that contains checkpoints. The returned output directory will be the top-level of the session that created the checkpoint, or None if it cannot be deduced.

Parameters **ckpt_path** – the path to a checkpoint or to an exisiting directory that contains checkpoints.

Returns The path to the session root directory. Will always point to an existing directory, or be None.

thelper.utils.**get_class_logger**(*skip=0*, *base=False*)
Shorthand to get logger for current class frame.

thelper.utils.**get_config_output_root**(*config*)
Returns the output root directory as defined inside a configuration dictionary.

The current implementation will scan for multiple keywords and return the first value found. If no keyword is matched, the function will return None.

Parameters **config** – the configuration dictionary to parse for a root output directory.

Returns The path to the output root directory. Can point to a non-existing directory, or be None.

thelper.utils.**get_config_session_name**(*config*)
Returns the 'name' of a session as defined inside a configuration dictionary.

The current implementation will scan for multiple keywords and return the first value found. If no keyword is matched, the function will return None.

Parameters **config** – the configuration dictionary to parse for a name.

Returns The name that should be given to the session (or 'None' if unknown/unavailable).

thelper.utils.**get_env_list**()
Returns a list of all packages installed in the current environment.

If the required packages cannot be imported, the returned list will be empty. Note that some packages may not be properly detected by this approach, and it is pretty hacky, so use it with a grain of salt (i.e. logging is fine).

thelper.utils.**get_file_paths**(*input_path*, *data_root*, *allow_glob=False*, *can_be_dir=False*)
Parse a wildcard-compatible file name pattern at a given root level for valid file paths.

thelper.utils.**get_func_logger**(*skip=0*)
Shorthand to get logger for current function frame.

thelper.utils.**get_git_stamp**()
Returns a print-friendly SHA signature for the framework's underlying git repository (if found).

thelper.utils.**get_glob_paths**(*input_glob_pattern*, *can_be_dir=False*)
Parse a wildcard-compatible file name pattern for valid file paths.

thelper.utils.**get_key**(*key*, *config*, *msg=None*, *delete=False*)
Returns a value given a dictionary key, throwing if not available.

thelper.utils.**get_key_def**(*key*, *config*, *default=None*, *msg=None*, *delete=False*)
Returns a value given a dictionary key, or the default value if it cannot be found.

thelper.utils.**get_log_stamp**()
Returns a print-friendly and filename-friendly identification string containing platform and time.

thelper.utils.**get_params_hash**(**args*, ***kwargs*)
Returns a sha1 hash for the given list of parameters (useful for caching).

thelper.utils.**get_save_dir**(*out_root*, *dir_name*, *config=None*, *resume=False*, *backup_ext='.json'*)
Returns a directory path in which the app can save its data.

If a folder with name `dir_name` already exists in the directory `out_root`, then the user will be asked to pick a new name. If the user refuses, `sys.exit(1)` is called. If config is not `None`, it will be saved to the output directory as a json file. Finally, a `logs` directory will also be created in the output directory for writing logger files.

Parameters

- **out_root** – path to the directory root where the save directory should be created.
- **dir_name** – name of the save directory to create. If it already exists, a new one will be requested.
- **config** – dictionary of app configuration parameters. Used to overwrite i/o queries, and will be written to the save directory in json format to test writing. Default is `None`.
- **resume** – specifies whether this session is new, or resumed from an older one (in the latter case, overwriting is allowed, and the user will never have to choose a new folder)
- **backup_ext** – extension to use when creating configuration file backups.

Returns The path to the created save directory for this session.

`thelper.utils.`**`get_slurm_tmpdir`**`()` → str
Returns the local SLURM_TMPDIR path if available, or `None`.

`thelper.utils.`**`import_class`**(*fullname*)
General-purpose runtime class importer.

Supported syntax:

1. `module.package.Class` will import the fully qualified `Class` located in `package` from the *installed* `module`
2. `/some/path/mod.pkg.Cls` will import `Cls` as fully qualified `mod.pkg.Cls` from `/some/path` directory

Parameters **fullname** – the fully qualified class name to be imported.

Returns The imported class.

`thelper.utils.`**`import_function`**(*func*, *params=None*)
General-purpose runtime function importer, with support for parameter binding.

Parameters

- **func** – the fully qualified function name to be imported, or a dictionary with two members (a `type` and optional `params`), or a list of any of these.
- **params** – optional params dictionary to bind to the function call via functools. If a dictionary of parameters is also provided in `func`, both will be merged.

Returns The imported function, with optionally bound parameters.

`thelper.utils.`**`init_logger`**(*log_level=0*, *filename=None*, *force_stdout=False*)
Initializes the framework logger with a specific filter level, and optional file output.

`thelper.utils.`**`is_scalar`**(*val*)
Returns whether the input value is a scalar according to numpy and PyTorch.

`thelper.utils.`**`load_checkpoint`**(*ckpt*, *map_location=None*, *always_load_latest=False*, *check_version=True*)
Loads a session checkpoint via PyTorch, check its compatibility, and returns its data.

If the `ckpt` parameter is a path to a valid directory, then that directly will be searched for a checkpoint. If multiple checkpoints are found, the latest will be returned (based on the epoch index in its name). iF `always_load_latest` is set to False and if a checkpoint named `ckpt.best.pth` is found, it will be returned instead.

Parameters

- **ckpt** – a file-like object or a path to the checkpoint file or session directory.
- **map_location** – a function, string or a dict specifying how to remap storage locations. See `torch.load` for more information.
- **always_load_latest** – toggles whether to always try to load the latest checkpoint if a session directory is provided (instead of loading the 'best' checkpoint).
- **check_version** – toggles whether the checkpoint's version should be checked for compatibility issues, and query the user for how to proceed.

Returns Content of the checkpoint (a dictionary).

`thelper.utils.`**`load_config`**(*path*, *as_json=False*, *add_name_if_missing=True*, ***kwargs*)

Loads the configuration dictionary from the provided path.

The type of file that is loaded is based on the extension in the path.

If the loaded configuration dictionary does not contain a 'name' field, the name of the file itself will be inserted as a value.

Parameters

- **path** – the path specifying which configuration to be loaded. only supported types are loaded unless *as_json* is *True*.
- **as_json** – specifies if an alternate extension should be considered as JSON format.
- **add_name_if_missing** – specifies whether the file name should be added to the config dictionary if it is missing a 'name' field.
- **kwargs** – forwarded to the extension-specific importer.

`thelper.utils.`**`lreplace`**(*string*, *old_prefix*, *new_prefix*)

Replaces a single occurrence of *old_prefix* in the given string by *new_prefix*.

`thelper.utils.`**`migrate_checkpoint`**(*ckptdata*)

Migrates the content of an incompatible or outdated checkpoint to the current version of the framework.

This function might not be able to fix all backward compatibility issues (e.g. it cannot fix class interfaces that were changed). Perfect reproductibility of tests cannot be guaranteed either if this migration tool is used.

Parameters **ckptdata** – checkpoint data in dictionary form obtained via `thelper.utils.load_checkpoint`. Note that the data contained in this dictionary will be modified in-place.

Returns An updated checkpoint dictionary that should be compatible with the current version of the framework.

`thelper.utils.`**`migrate_config`**(*config*, *cfg_ver_str*)

Migrates the content of an incompatible or outdated configuration to the current version of the framework.

This function might not be able to fix all backward compatibility issues (e.g. it cannot fix class interfaces that were changed). Perfect reproducibility of tests cannot be guaranteed either if this migration tool is used.

Parameters

- **config** – session configuration dictionary obtained e.g. by parsing a JSON file. Note that the data contained in this dictionary will be modified in-place.
- **cfg_ver_str** – string representing the version for which the configuration was created (e.g. "0.2.0").

Returns An updated configuration dictionary that should be compatible with the current version of the framework.

thelper.utils.**query_string**(*question*, *choices=None*, *default=None*, *allow_empty=False*, *bypass=None*)
Asks the user a question and returns the answer (a generic string).

Parameters

- **question** – the string that is presented to the user.
- **choices** – a list of predefined choices that the user can pick from. If `None`, then whatever the user types will be accepted.
- **default** – the presumed answer if the user just hits `<Enter>`. If `None`, then an answer is required to continue.
- **allow_empty** – defines whether an empty answer should be accepted.
- **bypass** – the returned value if the `bypass_queries` global variable is set to `True`. Can be `None`, in which case the function will throw an exception.

Returns The string entered by the user.

thelper.utils.**query_yes_no**(*question*, *default=None*, *bypass=None*)
Asks the user a yes/no question and returns the answer.

Parameters

- **question** – the string that is presented to the user.
- **default** – the presumed answer if the user just hits `<Enter>`. It must be 'yes', 'no', or `None` (meaning an answer is required).
- **bypass** – the option to select if the `bypass_queries` global variable is set to `True`. Can be `None`, in which case the function will throw an exception.

Returns `True` for 'yes', or `False` for 'no' (or their respective variations).

thelper.utils.**report_orion_results**(*session_runner: SessionRunner*) → None
Reports the results of a session runner, but only if the config allows it (true by default).

thelper.utils.**reporthook**(*count*, *block_size*, *total_size*)
Report hook used to display a download progression bar when using urllib requests.

thelper.utils.**resolve_import**(*fullname*)
Class name resolver.

Takes a string corresponding to a module and class fullname to be imported with *thelper.utils.import_class()* and resolves any back compatibility issues related to renamed or moved classes.

Parameters **fullname** – the fully qualified class name to be resolved.

Returns The resolved class fullname.

thelper.utils.**save_config**(*config*, *path*, *force_convert=True*, *as_json=False*, ***kwargs*)
Saves the given session/object configuration dictionary to the provided path.

The type of file that is created is based on the extension specified in the path. If the file cannot hold some of the objects within the configuration, they will be converted to strings before serialization, unless *force_convert* is set to *False* (in which case the function will raise an exception).

Parameters

- **config** – the session/object configuration dictionary to save.
- **path** – the path specifying where to create the output file. The extension used will determine what type of backup to create (e.g. Pickle = .pkl, JSON = .json, YAML = .yml/.yaml). if *as_json* is *True*, then any specified extension will be preserved bump dumped as JSON.

- **force_convert** – specifies whether non-serializable types should be converted if necessary.
- **as_json** – specifies if an alternate extension should be considered as JSON format.
- **kwargs** – forwarded to the extension-specific exporter.

thelper.utils.**save_env_list**(*path*)
Saves a list of all packages installed in the current environment to a log file.

Parameters **path** – the path where the log file should be created.

thelper.utils.**set_matplotlib_agg**()
Sets the matplotlib backend to Agg.

thelper.utils.**setup_cudnn**(*config*)
Parses the provided config for CUDNN flags and sets up PyTorch accordingly.

thelper.utils.**setup_cv2**(*config*)
Parses the provided config for OpenCV flags and sets up its global state accordingly.

thelper.utils.**setup_gdal**(*config*)
Parses the provided config for GDAL flags and sets up its global state accordingly.

thelper.utils.**setup_globals**(*config*)
Parses the provided config for global flags and sets up the global state accordingly.

thelper.utils.**setup_plt**(*config*)
Parses the provided config for matplotlib flags and sets up its global state accordingly.

thelper.utils.**setup_sys**(*config*)
Parses the provided config for PYTHON sys paths and sets up its global state accordingly.

thelper.utils.**str2bool**(*s*)
Converts a string to a boolean.

If the lower case version of the provided string matches any of 'true', '1', or 'yes', then the function returns `True`.

thelper.utils.**str2size**(*input_str*)
Returns a (WIDTH, HEIGHT) integer size tuple from a string formatted as 'WxH'.

thelper.utils.**stringify_confmat**(*confmat*, *class_list*, *hide_zeroes=False*, *hide_diagonal=False*, *hide_threshold=None*)
Transforms a confusion matrix array obtained in list or numpy format into a printable string.

thelper.utils.**test_cuda_device_availability**(*device_idx*)
Tests the availability of a single cuda device and returns its status.

thelper.utils.**to_numpy**(*array*)
Converts a list or PyTorch tensor to numpy. Does nothing if already a numpy array.

CHAPTER 7

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

7.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

7.2 Documentation improvements

The framework could always use more documentation, whether as part of the official docs, in docstrings, or even on the web in blog posts, articles, and such.

7.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/plstcharles/thelper/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

7.4 Development

To set up *thelper* for local development:

1. Fork thelper (look for the "Fork" button).
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/thelper.git
```

3. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the tests via:

```
$ make test-all
$ make docs
```

5. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

7.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Make sure all tests are passing (run `make test-all`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 8

Authors

- Pierre-Luc St-Charles - stcharpl<at>crim.ca
- Francis Charette Migneault - francis.charette-migneault<at>crim.ca
- Mario Beaulieu - mario.beaulieu<at>crim.ca
- Justine Boulent - justine.boulent<at>usherbrooke.ca
- Cyril Pecoraro - cyril.pecoraro<at>crim.ca

CHAPTER 9

Maintainer Guide

This guide provides an overview of the basic steps required for testing, creating new releases, and deploying new builds of the framework. For installation instructions, refer to the installation guide *[here]*.

As of November 2019 (v0.4.3), the framework's Continuous Integration (CI) pipeline is based primarily on Travis CI and Docker Hub. Automated conda builds have slowly been getting harder and harder to fix on Travis, and have gradually been phased out in favor of installation from source.

Maintainers are expected to be using an installation from source and running on a platform that supports Makefiles. All commands below are also provided under the assumption that they will be executed from the root directory of the framework.

9.1 Testing

The simplest way to test whether local changes have broken something in the framework is to use make:

```
$ make test-all
```

This command will start by running linters (flake8, isort, twine, check-manifest), and then execute all tests and produce a coverage report. To only run the linters, you can use:

```
$ make check
```

The framework tests are based on pytest, and can be executed indenpendently via:

```
$ make test
```

Since Travis CI runs on GPU-less platforms, unit tests and integration tests with mocked device-aware components are preferred. Future regression tests should also keep this limitation in consideration.

Tests can leave some logs and artifacts in the working directory, especially if they are cancelled in the middle of a run. To get rid of these, you can use:

```
$ make clean-test
```

9.2 Releases

To tag a commit for a release, you should use `bumpversion`. It is pre-configured to update all framework version references everywhere and automatically create a new commit with the required tag. Bumping the version works by incrementing the patch (v0.0.X), minor (v0.X.0) or major (vX.0.0) integer:

```
$ bumpversion patch
    or
$ bumpversion minor
    or
$ bumpversion major
```

Creating and pushing any tag on GitHub will trigger the deployment phase on Travis CI.

9.3 Documentation

Building the documentation can be accomplished by simply calling:

```
$ make docs
```

This will create the documentation pages in HTML format and display them in your browser. The same documentation will be built and deployed on readthedocs.io.

9.4 Deployment

Travis CI will automatically attempt to deploy the framework after successfully testing a tagged version. The deployment will target PyPI, Docker Hub, and Anaconda. If any of these steps fail, the deployment can be completed manually as specified below.

9.4.1 Python Package Index (PyPI)

A source distribution (sdist) and be prepared and uploaded using the following commands:

```
$ python setup.py sdist
$ twine upload dist/* --skip-existing
```

This will allow you to upload to your own project page, or to the origin (if you have collaborator access).

9.4.2 Docker Hub

A docker image can be prepared and uploaded using the following commands:

```
$ docker build -t ${DOCKER_REPO}:${TAG} -t ${DOCKER_REPO}:latest .
$ docker push ${DOCKER_REPO}
```

Again, uploading to the original project page will require collaborator access, but you can also upload the image to your own private repository.

9.4.3 Anaconda

The Anaconda package build process is very long, and requires a lot of disk space. It tends to fail on Travis CI, and must often be completed manually. To do so, you must first configure your conda environment to use custom channels to find proper project dependencies:

```
$ conda config --prepend channels conda-forge
$ conda config --prepend channels albumentations
$ conda config --prepend channels pytorch
```

Then, updating conda itself is never a bad idea:

```
$ conda update -q conda
```

To build and upload a package, you must also install the correct CLI tools:

```
$ conda install conda-build conda-verify anaconda-client
```

Finally, using the meta-config already available inside the project, you can build the package via:

```
$ conda build ci/
```

Instructions will be printed in the terminal regarding how to upload the built packages.

CHAPTER 10

Changelog

10.1 Unreleased (latest)

10.2 0.6.1 (2020/07/29)

- Fixed conda package builds for tagged deployments

10.3 0.6.0 (2020/07/28)

- Refactored and cleaned up HDF5 data extraction/parsing classes
- Added dataset interfaces for BigEarthNet, Agri-Vis challenge
- Update classification task to allow multi-label classification
- Added activation layer customization for in-framework ResNet archs
- Updated default move_tensor behavior to be non-blocking
- Added trainer implementation for auto-encoder-type models
- Added Orion reporting support for hyperparameter explorations
- Added SLURM cluster utilities (tmpdir getter, launch scripts)

10.4 0.5.0 (2020/07/21)

- Skip image save call during metric rendering if the provided value is `None` as employed by basic logger/reporter.
- Add JSON implementation for `thelper.train.utils.ClassifLogger`.
- Fix `concepts` to handle any variation of upper/lower concept name.

10.5 0.5.0-rc2 (2020/07/08)

- Employ `requirements.txt` within `conda-env.yml` to kept dependencies in sync.
- Fixes built docker image not using appropriate dependencies enforced through `requirements.txt`.

10.6 0.5.0-rc1 (2020/07/07)

- Fix version comparison check when validating configuration and/or checkpoint against package version. Version can now have a release part which was not considered.
- Fix incorrect calculation of sample coordinates in `thelper.data.geo.parsers.SlidingWindowDataset`.
- Remove `not_skip = __init__.py` config option for `isort` since `__init__.py` is included since `4.3.5`. Also force `isort<5` since many import checks break suddenly (e.g.: direct import with *as* alias break).

10.7 0.5.0-rc0 (2020/04/25)

- Update this changelog to use rst links (renders on github and readthedocs)
- Add `infer` mode for classification of geo-referenced rasters
- Add `Dockerfile-geo` to build thelper with pre-installed geo packages
- Add geo-related build instructions to travis-ci build steps
- Add auto-documentation of makefile targets and docker related targets

10.8 0.4.7 (2019/11/20)

- Removed optional dependencies from conda build env

10.9 0.4.6 (2019/11/20)

- Travis deploy test w/ split conda/docker stages

10.10 0.4.5 (2019/11/18)

- Split travis deploy stage into two phases
- Fixed draw_segment threshold usage & params lookup
- Fixed FCResNet embedding getter wrt latest pooling update
- Update all matplotlib plots to use 160 dpi by default
- Refactor trainer data/metric writer to save all viz data

10.11 0.4.4 (2019/11/18)

- Added viz pkg w/ t-SNE & UMAP support for in-trainer usage
- Fixed geo pkg documentation build issue related to mocking
- Fixed type and output format checks in numerous metrics
- Updated all callback readers to rely on new utility function
- Cleaned and optimize coordconv implementation
- Added U-Net architecture implementation to nn package
- Added IoU metric implementation
- Added support for SRM kernels and SRM convolutions
- Updated documentation (install, faq, maintenance)
- Added fixed weight sampler to data package
- Added lots of extra unit tests
- Added efficientnet 3rd-party module wrapper
- Fixed potential conflicts in task class names ordering

10.12 0.4.3 (2019/11/06)

- Fixed pytest-mock scope usage in metrics utests

10.13 0.4.2 (2019/11/06)

- Updated common resnet impl to support segmentation heads
- Fixed samples usage for auto-weighting of loss functions
- Cleaned up samples usage in loader factory data splitter
- Add GDL compatibility module to geo package
- Fix segmentation task dontcare default color mapping
- Cleaned up and simplified coordconv implementation
- Update segmentation trainer to use long-typed label maps
- Cleaned up augmentor/albumentations demo configurations

10.14 0.4.1 (2019/10/15)

- Removed travis check in deploy stage for master branch

10.15 0.4.0 (2019/10/11)

- Added geo subpackage
- Added geo vector/raster parsing classes
- Added ogc module for testbed15-specific utilities
- Added testbed15 train/viz configuration files
- Cleaned up makefile targets & coverage usage
- Replaced tox build system with makefile completely
- Merged 3rdparty configs into setup.cfg
- Updated travis to rely on makefile directly

10.16 0.3.14 (2019/09/30)

- Added extra logging calls in trainer and framework utils
- Cleaned up data configuration parsing logger calls
- Bypassed full device check when specific one is requested

10.17 0.3.13 (2019/09/26)

- Moved drawing utilities to new module
- Cleaned up output root/save directory parsing
- Cleaned up potential circular imports
- Moved optional dependency imports inside relevant functions
- Added support for root directory specification via config
- Updated config load/save to make naming optional

10.18 0.3.12 (2019/09/13)

- Fixed potential issue when reinstantiating custom ResNet
- Fixed ClassifLogger prediction logger w/o groundtruth

10.19 0.3.11 (2019/09/09)

- Add cli/config override for task compatibility mode setting

10.20 0.3.10 (2019/09/05)

- Cleaned up dependency lists, docstrings
- Fixed bbox iou computation with mixed int/float
- Fixed dontcare label deletion in segmentation task
- Cleaned up training session output directory localization
- Fixed object detection trainer empty bbox lists
- Fixed exponential parsing with pyyaml
- Fixed bbox display when using integer coords values

10.21 0.3.9 (2019/08/20)

- Fixed collate issues for pytorch >= 1.2
- Fixed null-size batch issues
- Cleaned up params#kwargs parsing in trainer
- Added pickled hashed param support utils
- Added support for yaml-based session configuration
- Added concept decorators for metrics/consumer classes
- Cleaned up shared interfaces to fix circular dependencies
- Added detection (bbox) logger class

10.22 0.3.8 (2019/08/08)

- Fixed nn modules constructor args forwarding
- Updated class importer to allow parsing of non-package dirs
- Fixed file-based logging from submodules (e.g. for all data)
- Cleaned and API-fied the CLI entrypoints for external use

10.23 0.3.7 (2019/07/31)

- Fixed travis timeouts on long deploy operations
- Added output path to trainer callback impls
- Added new draw-and-save display callback
- Added togray/tocolor transformation operations
- Cleaned up matplotlib use and show/block across draw functions
- Fixed various dependency and logging issues

10.24 0.3.6 (2019/07/26)

- Fixed torch version checks in custom default collate impl
- Fixed bbox predictions forwarding and evaluation in objdetect
- Refactored metrics/callbacks to clean up trainer impls
- Added pretrained opt to default resnet impl
- Fixed objdetect trainer display and prediction callbacks

10.25 0.3.5 (2019/07/23)

- Refactored metrics/consumers into separate interfaces
- Added unit tests for all metrics/prediction consumers
- Updated trainer callback signatures to include more data
- Updated install doc with links to anaconda/docker hubs
- Cleaned drawing functions args wrt callback refactoring
- Added eval module to optim w/ pascalvoc evaluation funcs

10.26 0.3.4 (2019/07/12)

- Fixed issues when reloading objdet model checkpoints
- Fixed issues when trying to use missing color maps
- Fixed backward compat issues when reloading old tasks
- Cleaned up object detection drawing utilities

10.27 0.3.3 (2019/07/09)

- Fixed travis conda build dependencies & channels

10.28 0.3.2 (2019/07/05)

- Update documentation use cases (model export) & faq
- Cleanup module base class config backup
- Fixed docker build and automated it via travis

10.29 0.3.1 (2019/06/17)

- Fix metrics RawPredictions not returning predictions during eval
- Fix parsing of checkpoint base path

10.30 0.3.0 (2019/06/12)

- Added dockerfile for containerized builds
- Added object detection task & trainer implementations
- Added CLI model/checkpoint export support
- Added CLI dataset splitting/HDF5 support
- Added baseline superresolution implementations
- Added lots of new unit tests & docstrings
- Cleaned up transform & display operations

10.31 0.2.8 (2019/03/17)

- Cleaned up build tools & docstrings throughout api
- Added user guide in documentation build
- Update tasks to allow dataset interface override
- Cleaned up trainer output logs
- Added fully convolutional resnet implementation
- Fixup various issues related to fine-tuning via 'resume'

10.32 0.2.7 (2019/02/04)

- Updated conda build recipe for python variants w/ auto upload

10.33 0.2.6 (2019/01/31)

- Added framework checkpoint/configuration migration utilities
- Fixed minor config parsing backward compatibility issues
- Fixed minor bugs related to query & drawing utilities

10.34 0.2.5 (2019/01/29)

- Fix travis-ci conda build/env path

10.35 0.2.4 (2019/01/29)

- Fix travis-ci conda channel setup

10.36 0.2.3 (2019/01/29)

- Fix `openssl` dependency

10.37 0.2.2 (2019/01/29)

- Fixed travis-ci matrix configuration
- Added travis-ci deployment step for pypi
- Fixed readthedocs documentation building
- Updated readme shields & front page look
- Cleaned up cli module entrypoint
- Fixed openssl dependency issues for travis tox check jobs
- Updated travis post-deploy to try to fix conda packaging (wip)

10.38 0.2.1 (2019/01/24)

- Added typedef module & cleaned up parameter inspections
- Cleaned up all drawing utils & added callback support to trainers
- Added support for albumentation pipelines via wrapper
- Updated all trainers/schedulers to rely on 0-based indexing
- Updated travis/rtd configs for auto-deploy & 3.6 support

10.39 0.2.0 (2019/01/15)

- Added regression/segmentation tasks and trainers
- Added interface for pascalvoc dataset
- Refactored data loaders/parsers and cleaned up data package
- Added lots of new utilities in base trainer implementation
- Added new unit tests for transformations
- Refactored transformations to use wrappers for augments/lists
- Added new samplers with dataset scaling support
- Added baseline implementation for FCN32s
- Added mae/mse metrics implementations
- Added trainer support for loss computation via external members
- Added utils to download/verify/extract files

10.40 0.1.1 (2019/01/14)

- Minor fixups and updates for CCFB02 compatibility
- Added RawPredictions metric to fetch data from trainers

10.41 0.1.0 (2018/11/28)

- Fixed readthedocs sphinx auto-build w/ mocking.
- Refactored package structure to avoid env issues.
- Rewrote seeding to allow 100% reproducible sessions.
- Cleaned up config file parameter lists.
- Cleaned up session output vars/logs/images.
- Add support for eval-time augmentation.
- Update transform wrappers for multi-channels & lists.
- Add gui module w/ basic segmentation annotation tool.
- Refactored task interfaces to allow merging.
- Simplified model fine-tuning via checkpoints.

10.42 0.0.2 (2018/10/18)

- Completed first documentation pass.
- Fixed travis/rtfd builds.
- Fixed device mapping/loading issues.

10.43 0.0.1 (2018/10/03)

- Initial release (work in progress).

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

S

T

U

V

W