thelper Documentation

Release 0.6.1

Pierre-Luc St-Charles

Jul 29, 2020

Contents

1 Overview **1**
- 1.1 Notes . . . 1

2 FAQ **3**
- 2.1 What it is. 3
- 2.2 What it is **NOT**. 3
- 2.3 What it supports. 4
- 2.4 How do I. 4

3 Installation **5**
- 3.1 Docker . . . 5
- 3.2 Installing from source . . . 5
- 3.3 Anaconda . . . 6
- 3.4 Testing the installation . . . 7
- 3.5 Documentation . . . 7

4 User Guide **9**
- 4.1 Command-Line Interface . . . 9
- 4.2 Configuration Files . . . 11
- 4.3 Session Directories . . . 17

5 Use Cases **21**
- 5.1 Image classification . . . 21
- 5.2 Image segmentation . . . 24
- 5.3 Object Detection . . . 24
- 5.4 Super-resolution . . . 24
- 5.5 Creating a new dataset interface . . . 24
- 5.6 Dataset/Loader visualization . . . 24
- 5.7 Dataset annotation . . . 25
- 5.8 Rebalancing a dataset . . . 25
- 5.9 Exporting a dataset . . . 25
- 5.10 Defining a data augmentation pipeline . . . 25
- 5.11 Supporting a custom trainer . . . 25
- 5.12 Supporting a custom task . . . 25
- 5.13 Supporting a custom model . . . 25
- 5.14 Visualizing metrics using `tensorboardX` . . . 25
- 5.15 Manually reloading a model . . . 25

5.16 Exporting a model . . . 26

6 thelper package 29
6.1 Subpackages . . . 29
6.2 Submodules . . . 33
6.3 thelper.cli module . . . 33
6.4 thelper.concepts module . . . 33
6.5 thelper.draw module . . . 33
6.6 thelper.ifaces module . . . 33
6.7 thelper.typedefs module . . . 33
6.8 thelper.utils module . . . 33

7 Contributing 35
7.1 Bug reports . . . 35
7.2 Documentation improvements . . . 35
7.3 Feature requests and feedback . . . 35
7.4 Development . . . 36

8 Authors 37

9 Maintainer Guide 39
9.1 Testing . . . 39
9.2 Releases . . . 40
9.3 Documentation . . . 40
9.4 Deployment . . . 40

10 Changelog 43
10.1 Unreleased (latest) . . . 43
10.2 0.6.1 (2020/07/29) . . . 43
10.3 0.6.0 (2020/07/28) . . . 43
10.4 0.5.0 (2020/07/21) . . . 43
10.5 0.5.0-rc2 (2020/07/08) . . . 44
10.6 0.5.0-rc1 (2020/07/07) . . . 44
10.7 0.5.0-rc0 (2020/04/25) . . . 44
10.8 0.4.7 (2019/11/20) . . . 44
10.9 0.4.6 (2019/11/20) . . . 44
10.10 0.4.5 (2019/11/18) . . . 44
10.11 0.4.4 (2019/11/18) . . . 45
10.12 0.4.3 (2019/11/06) . . . 45
10.13 0.4.2 (2019/11/06) . . . 45
10.14 0.4.1 (2019/10/15) . . . 45
10.15 0.4.0 (2019/10/11) . . . 46
10.16 0.3.14 (2019/09/30) . . . 46
10.17 0.3.13 (2019/09/26) . . . 46
10.18 0.3.12 (2019/09/13) . . . 46
10.19 0.3.11 (2019/09/09) . . . 46
10.20 0.3.10 (2019/09/05) . . . 47
10.21 0.3.9 (2019/08/20) . . . 47
10.22 0.3.8 (2019/08/08) . . . 47
10.23 0.3.7 (2019/07/31) . . . 47
10.24 0.3.6 (2019/07/26) . . . 48
10.25 0.3.5 (2019/07/23) . . . 48
10.26 0.3.4 (2019/07/12) . . . 48
10.27 0.3.3 (2019/07/09) . . . 48
10.28 0.3.2 (2019/07/05) . . . 48

10.29 0.3.1 (2019/06/17) 48
10.30 0.3.0 (2019/06/12) 49
10.31 0.2.8 (2019/03/17) 49
10.32 0.2.7 (2019/02/04) 49
10.33 0.2.6 (2019/01/31) 49
10.34 0.2.5 (2019/01/29) 49
10.35 0.2.4 (2019/01/29) 49
10.36 0.2.3 (2019/01/29) 50
10.37 0.2.2 (2019/01/29) 50
10.38 0.2.1 (2019/01/24) 50
10.39 0.2.0 (2019/01/15) 50
10.40 0.1.1 (2019/01/14) 51
10.41 0.1.0 (2018/11/28) 51
10.42 0.0.2 (2018/10/18) 51
10.43 0.0.1 (2018/10/03) 51

11 Indices and tables **53**

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

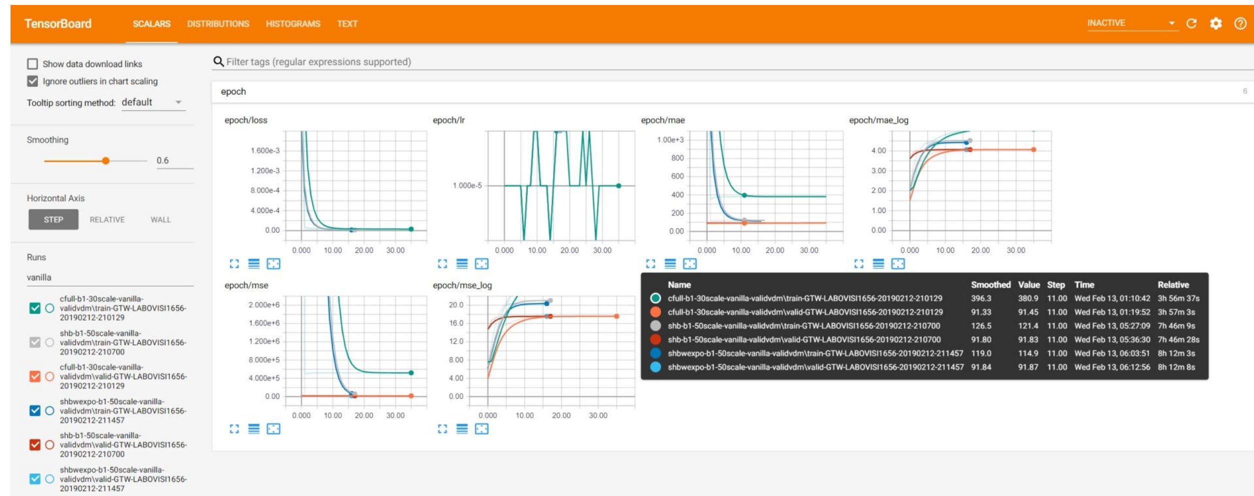
All information printed to the terminal during a session will also automatically be printed to files located in the `logs` folder of the session directory. Moreover, useful information about the training environment and datasets will be printed in other files in the same location. A brief description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

6.1 Subpackages

6.1.1 thelper.data package

Subpackages

thelper.data.geo package

Submodules

thelper.data.geo.agrivis module

thelper.data.geo.bigearthnet module

thelper.data.geo.gdl module

thelper.data.geo.infer module

thelper.data.geo.ogc module

thelper.data.geo.parsers module

thelper.data.geo.utils module

Submodules

thelper.data.loaders module

thelper.data.parsers module

thelper.data.pascalvoc module

thelper.data.samplers module

thelper.data.utils module

6.1.2 thelper.gui package

Submodules

thelper.gui.annotators module

thelper.gui.utils module

6.1.3 thelper.infer package

Submodules

thelper.infer.base module

thelper.infer.impl module

thelper.infer.utils module

6.1.4 thelper.nn package

Subpackages

thelper.nn.segmentation package

Submodules

thelper.nn.segmentation.base module

thelper.nn.segmentation.deeplabv3 module

thelper.nn.segmentation.fcn module

thelper.nn.sr package

Submodules

thelper.nn.sr.srcnn module

thelper.nn.sr.vdsr module

Submodules

thelper.nn.common module

thelper.nn.coordconv module

thelper.nn.densenet module

thelper.nn.efficientnet module

thelper.nn.fcn module

thelper.nn.inceptionresnetv2 module

thelper.nn.lenet module

thelper.nn.mobilenet module

thelper.nn.resnet module

thelper.nn.srm module

thelper.nn.unet module

thelper.nn.utils module

6.1.5 thelper.optim package

Submodules

thelper.optim.eval module

thelper.optim.losses module

thelper.optim.metrics module

thelper.optim.schedulers module

thelper.optim.utils module

6.1.6 thelper.session package

Submodules

thelper.session.base module

6.1.7 thelper.tasks package

Submodules

thelper.tasks.classif module

thelper.tasks.detect module

thelper.tasks.regr module

thelper.tasks.segm module

thelper.tasks.utils module

6.1.8 thelper.train package

Submodules

thelper.train.ae module

thelper.train.base module

thelper.train.classif module

thelper.train.detect module

thelper.train.regr module

thelper.train.segm module

thelper.train.utils module

6.1.9 thelper.transforms package

Submodules

thelper.transforms.composers module

thelper.transforms.operations module

thelper.transforms.utils module

thelper.transforms.wrappers module

6.1.10 thelper.viz package

Submodules

thelper.viz.tsne module

thelper.viz.umap module

6.2 Submodules

6.3 thelper.cli module

6.4 thelper.concepts module

6.5 thelper.draw module

6.6 thelper.ifaces module

6.7 thelper.typedefs module

6.8 thelper.utils module

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