delira Documentation

Release 0.3.0

Justus Schock, Oliver Rippel, Christoph Haarburger

Feb 07, 2019

Getting Started

CHAPTER 1

Getting started

1.1 Backends

Before installing `delira`, you have to choose a suitable backend. `delira` handles backends as optional dependencies and tries to escape all uses of a not-installed backend.

The currently supported backends are:

- torch (recommended, since it is the most tested backend): Suffix `torch`

  Note: `delira` supports mixed-precision training via apex, but `apex` must be installed separately

- tf (very experimental): Suffix `tf`

  Note: the `tf` backend is still very experimental and may be unstable.

- None: No Suffix
- All (installs all registered backends and their dependencies; not recommended, since this will install many large packages): Suffix `full`

Note: Depending on the backend, some functionalities may not be available for you. If you want to ensure, you can use each functionality, please use the `full` option, since it installs all backends

Note: If you want to add a backend like CNTK, Chainer, MXNET or something similar, please open an issue for that and we will guide you during that process (don't worry, it is not much effort at all).

1.2 Installation

Backend	Binary Installation	Source Installation	Notes
None	`pip install delira`	`pip install git+https://github.com/justusschock/delira.git`	Training not possible if backend is not installed separately
torch	`pip install delira[torch]`	`git clone https://github.com/justusschock/delira.git && cd delira && pip install .[torch]`	`delira` with `torch` backend supports mixed-precision training via NVIDIA/apex (must be installed separately).
tf	–	`git clone https://github.com/justusschock/delira.git && cd delira && pip install .[tf]`	The tensorflow backend is still very experimental and lacks some features
Full	`pip install delira[full]`	`git clone https://github.com/justusschock/delira.git && cd delira && pip install .[full]`	All backends are getting installed.

CHAPTER 2

Delira Introduction

Authors: Justus Schock, Christoph Haarburger

2.1 Loading Data

To train your network you first need to load your training data (and probably also your validation data). This chapter will therefore deal with `delira`'s capabilities to load your data (and apply some augmentation).

2.1.1 The Dataset

There are mainly two ways to load your data: Lazy or non-lazy. Loading in a lazy way means that you load the data just in time and keep the used memory to a bare minimum. This has, however, the disadvantage that your loading function could be a bottleneck since all postponed operations may have to wait until the needed data samples are loaded. In a no-lazy way, one would preload all data to RAM before starting any other operations. This has the advantage that there cannot be a loading bottleneck during latter operations. This advantage comes at cost of a higher memory usage and a (possibly) huge latency at the beginning of each experiment. Both ways to load your data are implemented in `delira` and they are named `BaseLazyDataset`and `BaseCacheDataset`. In the following steps you will only see the `BaseLazyDataset` since exchanging them is trivial. All Datasets (including the ones you might want to create yourself later) must be derived of `delira.data_loading.AbstractDataset` to ensure a minimum common API.

The dataset's `__init__` has the following signature:

```
def __init__(self, data_path, load_fn, img_extensions, gt_extensions,
                 **load_kwargs):
```

This means, you have to pass the path to the directory containing your data (`data_path`), a function to load a single sample of your data (`load_fn`), the file extensions for valid images (`img_extensions`) and the extensions for valid groundtruth files (`gt_files`). The defined extensions are used to index all data files in the given `data_path`. To get a single sample of your dataset after creating it, you can index it like this: `dataset[0]`.

The missing argument `**load_kwargs` accepts an arbitrary amount of additional keyword arguments which are directly passed to your loading function.

An example of how loading your data may look like is given below:

```
from delira.data_loading import BaseLazyDataset, default_load_fn_2d
dataset_train = BaseLazyDataset("/images/datasets/external/mnist/train",
                                default_load_fn_2d, img_extensions=[".png"],
                                gt_extensions=[".txt"], img_shape=(224, 224))
```

In this case all data lying in `/images/datasets/external/mnist/train` is loaded by `default_load_fn_2d`. The files containing the data must be PNG-files, while the groundtruth is defined in TXT-files. The `default_load_fn_2d` needs the additional argument `img_shape` which is passed as keyword argument via `**load_kwargs`.

Note: for reproducability we decided to use some wrapped PyTorch datasets for this introduction.

Now, let's just initialize our trainset:

Getting a single sample of your dataset with dataset_train[0] will produce:

which means, that our data is stored in a dictionary containing the keys `data` and `label`, each of them holding the corresponding numpy arrays. The dataloading works on `numpy` purely and is thus backend agnostic. It does not matter in which format or with which library you load/preprocess your data, but at the end it must be converted to numpy arrays For validation purposes another dataset could be created with the test data like this:

2.1.2 The Dataloader

The Dataloader wraps your dataset to privode the ability to load whole batches with an abstract interface. To create a dataloader, one would have to pass the following arguments to it's `__init__`: the previously created `dataset`.Additionally, it is possible to pass the `batch_size` defining the number of samples per batch, the total number of batches (`num_batches`), which will be the number of samples in your dataset devided by the batchsize per default, a random `seed`for always getting the same behaviour of random number generators and a `` `sampler <>`__ `` defining your sampling strategy. This would create a dataloader for your `dataset_train`:

Since the batch_size has been set to 32, the loader will load 32 samples as one batch.

Even though it would be possible to train your network with an instance of `BaseDataLoader`, `malira` also offers a different approach that covers multithreaded data loading and augmentation:

2.1.3 The Datamanager

The data manager is implemented as `delira.data_loading.BaseDataManager` and wraps a `DataLoader`. It also encapsulates augmentations. Having a view on the `BaseDataManager`'s signature, it becomes obvious that it accepts the same arguments as the `` `DataLoader <#The-Dataloader>`__ ``. You can either pass a `dataset` or a combination of path, dataset class and load function. Additionally, you can pass a custom dataloder class if necessary and a sampler class to choose a sampling algorithm.

The parameter `transforms` accepts augmentation transformations as implemented in `batchgenerators`. Augmentation is applied on the fly using `n_process_augmentation` threads.

All in all the DataManager is the recommended way to generate batches from your dataset.

The following example shows how to create a data manager instance:

The approach to initialize a DataManager from a datapath takes more arguments since, in opposite to initializaton from dataset, it needs all the arguments which are necessary to internally create a dataset.

Since we want to validate our model we have to create a second manager containing our `dataset_val`:

That's it - we just finished loading our data!

Iterating over a DataManager is possible in simple loops:

2.1.4 Sampler

In previous section samplers have been already mentioned but not yet explained. A sampler implements an algorithm how a batch should be assembled from single samples in a dataset. `delira` provides the following sampler classes in it's subpackage `delira.data_loading.sampler`:

- `AbstractSampler`
- `SequentialSampler`
- `PrevalenceSequentialSampler`
- `RandomSampler`
- `PrevalenceRandomSampler`
- `WeightedRandomSampler`
- `LambdaSampler`

The `AbstractSampler` implements no sampling algorithm but defines a sampling API and thus all custom samplers must inherit from this class. The `Sequential` sampler builds batches by just iterating over the samples' indices in a sequential way. Following this, the `RandomSampler` builds batches by randomly drawing the samples' indices with replacement. If the class each sample belongs to is known for each sample at the beginning, the `PrevalenceSequentialSampler` and the `PrevalenceRandomSampler` perform a per-class sequential or random sampling and building each batch with the exactly same number of samples from each class. The `WeightedRandomSampler`accepts custom weights to give specific samples a higher probability during random sampling than others.

The `LambdaSampler` is a wrapper for a custom sampling function, which can be passed to the wrapper during it's initialization, to ensure API conformity.

It can be passed to the DataLoader or DataManager as class (argument `sampler_cls`) or as instance (argument `sampler`).

2.2 Models

Since the purpose of this framework is to use machine learning algorithms, there has to be a way to define them. Defining models is straight forward. `delira` provides a class `delira.models.AbstractNetwork`. *All models must inherit from this class*.

To inherit this class four functions must be implemented in the subclass:

- `__init__`
- `closure`
- `prepare_batch`
- `__call__`

2.2.1 `__init__`

The `__init__`function is a classes constructor. In our case it builds the entire model (maybe using some helper functions). If writing your own custom model, you have to override this method.

Note: If you want the best experience for saving your model and completely recreating it during the loading process you need to take care of a few things: * if using `torchvision.models` to build your model, always import it with `from torchvision import models as t_models` * register all arguments in your custom `__init__` in the abstract class. A init_prototype could look like this:

```
def __init__(self, in_channels: int, n_outputs: int, **kwargs):
    """

    Parameters
    ----------
    in_channels: int
        number of input_channels
    n_outputs: int
        number of outputs (usually same as number of classes)
    """
    # register params by passing them as kwargs to parent class __init__
    # only params registered like this will be saved!
    super().__init__(in_channels=in_channels,
                     n_outputs=n_outputs,
                     **kwargs)
```

2.2.2 `closure`

The `closure`function defines one batch iteration to train the network. This function is needed for the framework to provide a generic trainer function which works with all kind of networks and loss functions.

The closure function must implement all steps from forwarding, over loss calculation, metric calculation, logging (for which `delira.logging_handlers` provides some extensions for pythons logging module), and the actual backpropagation.

It is called with an empty optimizer-dict to evaluate and should thus work with optional optimizers.

2.2.3 `prepare_batch`

The `prepare_batch`function defines the transformation from loaded data to match the networks input and output shape and pushes everything to the right device.

2.3 Abstract Networks for specific Backends

2.3.1 PyTorch

At the time of writing, PyTorch is the only backend which is supported, but other backends are planned. In PyTorch every network should be implemented as a subclass of `torch.nn.Module`, which also provides a `__call__` method.

This results in sloghtly different requirements for PyTorch networks: instead of implementing a `__call__` method, we simply call the `torch.nn.Module.__call__` and therefore have to implement the `forward` method, which defines the module's behaviour and is internally called by `torch.nn.Module.__call__` (among other stuff). To give a default behaviour suiting most cases and not have to care about internals, `delira` provides the `AbstractPyTorchNetwork` which is a more specific case of the `AbstractNetwork` for PyTorch modules.

forward

The `forward` function defines what has to be done to forward your input through your network. Assuming your network has three convolutional layers stored in `self.conv1`, `self.conv2` and `self.conv3` and a ReLU stored in `self.relu`, a simple `forward` function could look like this:

```
def forward(self, input_batch: torch.Tensor):
    out_1 = self.relu(self.conv1(input_batch))
    out_2 = self.relu(self.conv2(out_1))
    out_3 = self.conv3(out2)

    return out_3
```

prepare_batch

The default `prepare_batch` function for PyTorch networks looks like this:

```
@staticmethod
def prepare_batch(batch: dict, input_device, output_device):
    """
    Helper Function to prepare Network Inputs and Labels (convert them to
    correct type and shape and push them to correct devices)

    Parameters
    ----------
    batch : dict
        dictionary containing all the data
    input_device : torch.device
        device for network inputs
    output_device : torch.device
        device for network outputs

    Returns
    -------
    dict
        dictionary containing data in correct type and shape and on correct
        device

    """
    return_dict = {"data": torch.from_numpy(batch.pop("data")).to(
        input_device)}

    for key, vals in batch.items():
        return_dict[key] = torch.from_numpy(vals).to(output_device)

    return return_dict
```

and can be customized by subclassing the `AbstractPyTorchNetwork`.

closure example

A simple closure function for a PyTorch module could look like this:

```
    @staticmethod
    def closure(model: AbstractPyTorchNetwork, data_dict: dict,
```

(continues on next page)

(continued from previous page)

```
            optimizers: dict, criterions={}, metrics={},
            fold=0, **kwargs):
        """
        closure method to do a single backpropagation step

        Parameters
        ----------
        model : :class:`ClassificationNetworkBasePyTorch`
            trainable model
        data_dict : dict
            dictionary containing the data
        optimizers : dict
            dictionary of optimizers to optimize model's parameters
        criterions : dict
            dict holding the criterions to calculate errors
            (gradients from different criterions will be accumulated)
        metrics : dict
            dict holding the metrics to calculate
        fold : int
            Current Fold in Crossvalidation (default: 0)
        **kwargs:
            additional keyword arguments

        Returns
        -------
        dict
            Metric values (with same keys as input dict metrics)
        dict
            Loss values (with same keys as input dict criterions)
        list
            Arbitrary number of predictions as torch.Tensor

        Raises
        ------
        AssertionError
            if optimizers or criterions are empty or the optimizers are not
            specified

        """

        assert (optimizers and criterions) or not optimizers, \
            "Criterion dict cannot be emtpy, if optimizers are passed"

        loss_vals = {}
        metric_vals = {}
        total_loss = 0

        # choose suitable context manager:
        if optimizers:
            context_man = torch.enable_grad

        else:
            context_man = torch.no_grad

        with context_man():

            inputs = data_dict.pop("data")
```

(continues on next page)

(continued from previous page)

```
            preds = model(inputs)

            if data_dict:

                for key, crit_fn in criterions.items():
                    _loss_val = crit_fn(preds, *data_dict.values())
                    loss_vals[key] = _loss_val.detach()
                    total_loss += _loss_val

                with torch.no_grad():
                    for key, metric_fn in metrics.items():
                        metric_vals[key] = metric_fn(
                            preds, *data_dict.values())

        if optimizers:
            optimizers['default'].zero_grad()
            total_loss.backward()
            optimizers['default'].step()

        else:

            # add prefix "val" in validation mode
            eval_loss_vals, eval_metrics_vals = {}, {}
            for key in loss_vals.keys():
                eval_loss_vals["val_" + str(key)] = loss_vals[key]

            for key in metric_vals:
                eval_metrics_vals["val_" + str(key)] = metric_vals[key]

            loss_vals = eval_loss_vals
            metric_vals = eval_metrics_vals

        for key, val in {**metric_vals, **loss_vals}.items():
            logging.info({"value": {"value": val.item(), "name": key,
                                    "env_appendix": "_%02d" % fold
                                    }})

        logging.info({'image_grid': {"images": inputs, "name": "input_images",
                                     "env_appendix": "_%02d" % fold}})

        return metric_vals, loss_vals, [preds]

**Note:** This closure is taken from the
``delira.models.classification.ClassificationNetworkBasePyTorch``
```

2.3.2 Other examples

In `delira.models` you can find exemplaric implementations of generative adversarial networks, classification and regression approaches or segmentation networks.

2.4 Training

2.4.1 Parameters

Training-parameters (often called hyperparameters) can be defined in the `delira.training.Parameters` class.

The class accepts the parameters `batch_size` and `num_epochs` to define the batchsize and the number of epochs to train, the parameters `optimizer_cls` and `optimizer_params` to create an optimizer or training, the parameter `criterions` to specify the training criterions (whose gradients will be accumulated by defaut), the parameters `lr_sched_cls` and `lr_sched_params` to define the learning rate scheduling and the parameter `metrics` to specify evaluation metrics.

Additionally, it is possible to pass an aritrary number of keyword arguments to the class

It is good practice to create a `Parameters` object at the beginning and then use it for creating other objects which are needed for training, since you can use the classes attributes and changes in hyperparameters only have to be done once:

2.4.2 Trainer

The `delira.training.NetworkTrainer` class provides functions to train a single network by passing attributes from your parameter object, a `save_freq` to specify how often your model should be saved (`save_freq=1` indicates every epoch, `save_freq=2` every second epoch etc.) and `gpu_ids`. If you don't pass any ids at all, your network will be trained on CPU (and probably take a lot of time). If you specify 1 id, the network will be trained on the GPU with the corresponding index and if you pass multiple `gpu_ids` your network will be trained on multiple GPUs in parallel.

> **Note:** The GPU indices are refering to the devices listed in `CUDA_VISIBLE_DEVICES`. E.g if `CUDA_VISIBLE_DEVICES` lists GPUs 3, 4, 5 then gpu_id 0 will be the index for GPU 3 etc.
>
> **Note:** training on multiple GPUs is not recommended for easy and small networks, since for these networks the synchronization overhead is far greater than the parallelization benefit.

Training your network might look like this:

2.4.3 Experiment

The `delira.training.AbstractExperiment` class needs an experiment name, a path to save it's results to, a parameter object, a model class and the keyword arguments to create an instance of this class. It provides methods to perform a single training and also a method for running a kfold-cross validation. In order to create it, you must choose the `PyTorchExperiment`, which is basically just a subclass of the `AbstractExperiment` to provide a general setup for PyTorch modules. Running an experiment could look like this:

An `Experiment` is the most abstract (and recommended) way to define, train and validate your network.

2.5 Logging

Previous class and function definitions used pythons's `logging` library. As extensions for this library `delira` provides a package (`delira.logging`) containing handlers to realize different logging methods.

To use these handlers simply add them to your logger like this:

```
logger.addHandler(logging.StreamHandler())
```

Nowadays, delira mainly relies on trixi for logging and provides only a `MultiStreamHandler` and a `TrixiHandler`, which is a binding to `trixi`'s loggers and integrates them into the python `logging` module

2.5.1 `MultiStreamHandler`

The `MultiStreamHandler` accepts an arbitrary number of streams during initialization and writes the message to all of it's streams during logging.

2.5.2 Logging with `Visdom` - The `trixi` Loggers

`Visdom <https://github.com/facebookresearch/visdom>`__ is a tool designed to visualize your logs. To use this tool you need to open a port on the machine you want to train on via `visdom -port YOUR_PORTNUMBER` Afterwards just add the handler of your choice to the logger. For more detailed information and customization have a look at this website.

Logging the scalar tensors containing `1`, `2`, `3`, `4` (at the beginning; will increase to show epochwise logging) with the corresponding keys `"one"`, `"two"`, `"three"`, `"four"` and two random images with the keys `"prediction"` and `"groundtruth"` would look like this:

Types of VisdomHandlers

The abilities of a handler is simply derivable by it's name: A `VisdomImageHandler` is the pure visdom logger, whereas the `VisdomImageSaveHandler` combines the abilities of a `VisdomImageHandler`and a `ImgSaveHandler`. Together with a `StreamHandler` (in-built handler) you get the `VisdomImageStreamHandler` and if you also want to add the option to save images to disk, you should use the `VisdomImageSaveStreamHandler`

The provided handlers are:

- `ImgSaveHandler`
- `MultistreamHandler`
- `VisdomImageHandler`
- `VisdomImageSaveHandler`
- `VisdomImageSaveStreamHandler`
- `VisdomStreamHandler`

2.6 More Examples

More Examples can be found in * the classification example * the 2d segmentation example * the 3d segmentation example * the generative adversarial example

CHAPTER 3

Classification with Delira - A very short introduction

Author: Justus Schock

Date: 04.12.2018

This Example shows how to set up a basic classification PyTorch experiment and Visdom Logging Environment.

Let's first setup the essential hyperparameters. We will use `delira`'s `Parameters`-class for this:

Since we did not specify any metric, only the `CrossEntropyLoss` will be calculated for each batch. Since we have a classification task, this should be sufficient. We will train our network with a batchsize of 64 by using `Adam` as optimizer of choice.

3.1 Logging and Visualization

To get a visualization of our results, we should monitor them somehow. For logging we will use `Visdom`. To start a visdom server you need to execute the following command inside an environment which has visdom installed:

```
visdom -port=9999
```

This will start a visdom server on port 9999 of your machine and now we can start to configure our logging environment. To view your results you can open http://localhost:9999 in your browser.

Since a single visdom server can run multiple environments, we need to specify a (unique) name for our environment and need to tell the logger, on which port it can find the visdom server.

3.2 Data Preparation

3.2.1 Loading

Next we will create a small train and validation set (based on `torchvision` MNIST):

3.2.2 Augmentation

For Data-Augmentation we will apply a few transformations:

With these transformations we can now wrap our datasets into datamanagers:

3.3 Training

After we have done that, we can finally specify our experiment and run it. We will therfore use the already implemented `ClassificationNetworkBasePyTorch` which is basically a ResNet18:

Congratulations, you have now trained your first Classification Model using `delira`, we will now predict a few samples from the testset to show, that the networks predictions are valid:

3.4 See Also

For a more detailed explanation have a look at * the introduction tutorial * the 2d segmentation example * the 3d segmentation example * the generative adversarial example

CHAPTER 4

Generative Adversarial Nets with Delira - A very short introduction

Author: Justus Schock

Date: 04.12.2018

This Example shows how to set up a basic GAN PyTorch experiment and Visdom Logging Environment.

4.1 HyperParameters

Let's first setup the essential hyperparameters. We will use `delira`'s `Parameters`-class for this:

Since we specified `torch.nn.L1Loss` as criterion and `torch.nn.MSELoss` as metric, they will be both calculated for each batch, but only the criterion will be used for backpropagation. Since we have a simple generative task, this should be sufficient. We will train our network with a batchsize of 64 by using `Adam` as optimizer of choice.

4.2 Logging and Visualization

To get a visualization of our results, we should monitor them somehow. For logging we will use `Visdom`. To start a visdom server you need to execute the following command inside an environment which has visdom installed:

```
visdom -port=9999
```

This will start a visdom server on port 9999 of your machine and now we can start to configure our logging environment. To view your results you can open http://localhost:9999 in your browser.

Since a single visdom server can run multiple environments, we need to specify a (unique) name for our environment and need to tell the logger, on which port it can find the visdom server.

4.3 Data Preparation

4.3.1 Loading

Next we will create a small train and validation set (based on `torchvision` MNIST):

4.3.2 Augmentation

For Data-Augmentation we will apply a few transformations:

With these transformations we can now wrap our datasets into datamanagers:

4.4 Training

After we have done that, we can finally specify our experiment and run it. We will therfore use the already implemented `GenerativeAdversarialNetworkBasePyTorch` which is basically a vanilla DCGAN:

Congratulations, you have now trained your first Generative Adversarial Model using `delira`.

4.5 See Also

For a more detailed explanation have a look at * the introduction tutorial * the 2d segmentation example * the 3d segmentation example * the classification example

CHAPTER 5

Segmentation in 2D using U-Nets with Delira - A very short introduction

Author: Justus Schock, Alexander Moriz

Date: 17.12.2018

This Example shows how use the U-Net implementation in Delira with PyTorch.

Let's first setup the essential hyperparameters. We will use `delira`'s `Parameters`-class for this:

Since we did not specify any metric, only the `CrossEntropyLoss` will be calculated for each batch. Since we have a classification task, this should be sufficient. We will train our network with a batchsize of 64 by using `Adam` as optimizer of choice.

5.1 Logging and Visualization

To get a visualization of our results, we should monitor them somehow. For logging we will use `Visdom`. To start a visdom server you need to execute the following command inside an environment which has visdom installed:

```
visdom -port=9999
```

This will start a visdom server on port 9999 of your machine and now we can start to configure our logging environment. To view your results you can open http://localhost:9999 in your browser.

Since a single visdom server can run multiple environments, we need to specify a (unique) name for our environment and need to tell the logger, on which port it can find the visdom server.

5.2 Data Praparation

5.2.1 Loading

Next we will create a small train and validation set (in this case they will be the same to show the overfitting capability of the UNet).

Our data is a brain MR-image thankfully provided by the FSL in their introduction.

We first download the data and extract the T1 image and the corresponding segmentation:

Now, we load the image and the mask (they are both 3D), convert them to a 32-bit floating point numpy array and ensure, they have the same shape (i.e. that for each voxel in the image, there is a voxel in the mask):

```
(192, 192, 174)
```

By querying the unique values in the mask, we get the following:

This means, there are 4 classes (background and 3 types of tissue) in our sample.

Since we want to do a 2D segmentation, we extract a single slice out of the image and the mask (we choose slice 100 here) and plot it:

To load the data, we have to use a `Dataset`. The following defines a very simple dataset, accepting an image slice, a mask slice and the number of samples. It always returns the same sample until `num_samples` samples have been returned.

Now, we can finally instantiate our datasets:

5.2.2 Augmentation

For Data-Augmentation we will apply a few transformations:

With these transformations we can now wrap our datasets into datamanagers:

5.3 Training

After we have done that, we can finally specify our experiment and run it. We will therfore use the already implemented `UNet2dPytorch`:

5.4 See Also

For a more detailed explanation have a look at * the introduction tutorial * the classification example * the 3d segmentation example * the generative adversarial example

CHAPTER 6

Segmentation in 3D using U-Nets with Delira - A very short introduction

Author: Justus Schock, Alexander Moriz

Date: 17.12.2018

This Example shows how use the U-Net implementation in Delira with PyTorch.

Let's first setup the essential hyperparameters. We will use `delira`'s `Parameters`-class for this:

Since we did not specify any metric, only the `CrossEntropyLoss` will be calculated for each batch. Since we have a classification task, this should be sufficient. We will train our network with a batchsize of 64 by using `Adam` as optimizer of choice.

6.1 Logging and Visualization

To get a visualization of our results, we should monitor them somehow. For logging we will use `Visdom`. To start a visdom server you need to execute the following command inside an environment which has visdom installed:

```
visdom -port=9999
```

This will start a visdom server on port 9999 of your machine and now we can start to configure our logging environment. To view your results you can open http://localhost:9999 in your browser.

Since a single visdom server can run multiple environments, we need to specify a (unique) name for our environment and need to tell the logger, on which port it can find the visdom server.

6.2 Data Praparation

6.2.1 Loading

Next we will create a small train and validation set (in this case they will be the same to show the overfitting capability of the UNet).

Our data is a brain MR-image thankfully provided by the FSL in their introduction.

We first download the data and extract the T1 image and the corresponding segmentation:

Now, we load the image and the mask (they are both 3D), convert them to a 32-bit floating point numpy array and ensure, they have the same shape (i.e. that for each voxel in the image, there is a voxel in the mask):

By querying the unique values in the mask, we get the following:

This means, there are 4 classes (background and 3 types of tissue) in our sample.

To load the data, we have to use a `Dataset`. The following defines a very simple dataset, accepting an image slice, a mask slice and the number of samples. It always returns the same sample until `num_samples` samples have been returned.

Now, we can finally instantiate our datasets:

6.2.2 Augmentation

For Data-Augmentation we will apply a few transformations:

With these transformations we can now wrap our datasets into datamanagers:

6.3 Training

After we have done that, we can finally specify our experiment and run it. We will therfore use the already implemented `UNet3dPytorch`:

6.4 See Also

For a more detailed explanation have a look at * the introduction tutorial * the classification example * the 2d segmentation example * the generative adversarial example

CHAPTER 7

API Documentation

7.1 Delira

7.1.1 Data Loading

This module provides Utilities to load the Data

Arbitrary Data

The following classes are implemented to work with every kind of data. You can use every framework you want to load your data, but the returned samples should be a `dict` of `numpy ndarrays`

Datasets

The Dataset the most basic class and implements the loading of your dataset elements. You can either load your data in a lazy way e.g. loading them just at the moment they are needed or you could preload them and cache them.

Datasets can be indexed by integers and return single samples.

To implement custom datasets you should derive the *`AbstractDataset`*

AbstractDataset

class AbstractDataset(*data_path*, *load_fn*, *img_extensions*, *gt_extensions*)

Bases: `object`

Base Class for Dataset

_make_dataset(*path*)

Create dataset

Parameters **path** (*`str`*) – path to data samples

Returns data: List of sample paths if lazy; List of samples if not

Return type list

get_sample_from_index(*index*)
Returns the data sample for a given index (without any loading if it would be necessary) This implements the base case and can be subclassed for index mappings. The actual loading behaviour (lazy or cached) should be implemented in __getitem__

See also:

:method:ConcatDataset.get_sample_from_index :method:BaseLazyDataset.__getitem__ :method:BaseCacheDataset.__getitem__

Parameters **index** (*int*) – index corresponding to targeted sample

Returns sample corresponding to given index

Return type Any

get_subset(*indices*)
Returns a Subset of the current dataset based on given indices

Parameters **indices** (*iterable*) – valid indices to extract subset from current dataset

Returns the subset

Return type BlankDataset

train_test_split(**args*, ***kwargs*)
split dataset into train and test data

Deprecated since version 0.3: method will be removed in next major release

Parameters

- ***args** – positional arguments of train_test_split
- ****kwargs** – keyword arguments of train_test_split

Returns

- BlankDataset – train dataset
- BlankDataset – test dataset

See also:

sklearn.model_selection.train_test_split

BaseLazyDataset

class BaseLazyDataset(*data_path*, *load_fn*, *img_extensions*, *gt_extensions*, ***load_kwargs*)
Bases: delira.data_loading.dataset.AbstractDataset

Dataset to load data in a lazy way

_is_valid_image_file(*fname*)
Helper Function to check wheter file is image file and has at least one label file

Parameters **fname** (*str*) – filename of image path

Returns is valid data sample

Return type bool

_make_dataset(*path*)
Helper Function to make a dataset containing paths to all images in a certain directory

Parameters **path** (*str*) – path to data samples

Returns list of sample paths

Return type list

Raises AssertionError – if *path* is not a valid directory

get_sample_from_index(*index*)
Returns the data sample for a given index (without any loading if it would be necessary) This implements the base case and can be subclassed for index mappings. The actual loading behaviour (lazy or cached) should be implemented in __getitem__

See also:

:method:ConcatDataset.get_sample_from_index :method:BaseLazyDataset.__getitem__ :method:BaseCacheDataset.__getitem__

Parameters **index** (*int*) – index corresponding to targeted sample

Returns sample corresponding to given index

Return type Any

get_subset(*indices*)
Returns a Subset of the current dataset based on given indices

Parameters **indices** (*iterable*) – valid indices to extract subset from current dataset

Returns the subset

Return type BlankDataset

train_test_split(**args*, ***kwargs*)
split dataset into train and test data

Deprecated since version 0.3: method will be removed in next major release

Parameters

- ***args** – positional arguments of train_test_split
- ****kwargs** – keyword arguments of train_test_split

Returns

- BlankDataset – train dataset
- BlankDataset – test dataset

See also:

sklearn.model_selection.train_test_split

BaseCacheDataset

class BaseCacheDataset(*data_path*, *load_fn*, *img_extensions*, *gt_extensions*, ***load_kwargs*)
Bases: delira.data_loading.dataset.AbstractDataset

Dataset to preload and cache data

Notes

data needs to fit completely into RAM!

_is_valid_image_file(*fname*)
Helper Function to check wheter file is image file and has at least one label file

Parameters **fname** (*str*) – filename of image path

Returns is valid data sample

Return type bool

_make_dataset(*path*)
Helper Function to make a dataset containing all samples in a certain directory

Parameters **path** (*str*) – path to data samples

Returns list of sample paths

Return type list

Raises AssertionError – if *path* is not a valid directory

get_sample_from_index(*index*)
Returns the data sample for a given index (without any loading if it would be necessary) This implements the base case and can be subclassed for index mappings. The actual loading behaviour (lazy or cached) should be implemented in __getitem__

See also:

:method:ConcatDataset.get_sample_from_index :method:BaseLazyDataset.__getitem__ :method:BaseCacheDataset.__getitem__

Parameters **index** (*int*) – index corresponding to targeted sample

Returns sample corresponding to given index

Return type Any

get_subset(*indices*)
Returns a Subset of the current dataset based on given indices

Parameters **indices** (*iterable*) – valid indices to extract subset from current dataset

Returns the subset

Return type BlankDataset

train_test_split(**args*, ***kwargs*)
split dataset into train and test data

Deprecated since version 0.3: method will be removed in next major release

Parameters

- ***args** – positional arguments of train_test_split
- ****kwargs** – keyword arguments of train_test_split

Returns

- BlankDataset – train dataset
- BlankDataset – test dataset

See also:

`sklearn.model_selection.train_test_split`

ConcatDataset

class ConcatDataset(**datasets*)

Bases: `delira.data_loading.dataset.AbstractDataset`

_make_dataset(*path*)

Create dataset

Parameters **path** (*str*) – path to data samples

Returns data: List of sample paths if lazy; List of samples if not

Return type list

get_sample_from_index(*index*)

Returns the data sample for a given index (without any loading if it would be necessary) This method implements the index mapping of a global index to the subindices for each dataset. The actual loading behaviour (lazy or cached) should be implemented in __getitem__

See also:

:method:AbstractDataset.get_sample_from_index :method:BaseLazyDataset.__getitem__ :method:BaseCacheDataset.__getitem__

Parameters **index** (*int*) – index corresponding to targeted sample

Returns sample corresponding to given index

Return type Any

get_subset(*indices*)

Returns a Subset of the current dataset based on given indices

Parameters **indices** (*iterable*) – valid indices to extract subset from current dataset

Returns the subset

Return type `BlankDataset`

train_test_split(**args*, ***kwargs*)

split dataset into train and test data

Deprecated since version 0.3: method will be removed in next major release

Parameters

- ***args** – positional arguments of `train_test_split`
- ****kwargs** – keyword arguments of `train_test_split`

Returns

- `BlankDataset` – train dataset
- `BlankDataset` – test dataset

See also:

`sklearn.model_selection.train_test_split`

Dataloader

The Dataloader wraps the dataset and combines them with a sampler (see below) to combine single samples to whole batches.

ToDo: add flow chart diagramm

BaseDataLoader

class BaseDataLoader(*dataset: delira.data_loading.dataset.AbstractDataset*, *batch_size=1*, *num_batches=None*, *seed=1*, *sampler=None*)

Bases: sphinx.ext.autodoc.importer._MockObject

Class to create a data batch out of data samples

_get_sample(*index*)

Helper functions which returns an element of the dataset

Parameters **index** (*int*) – index specifying which sample to return

Returns Returned Data

Return type dict

generate_train_batch()

Generate Indices which behavior based on self.sampling gets data based on indices

Returns data and labels

Return type dict

Raises StopIteration – If the maximum number of batches has been generated

Datamanager

The datamanager wraps a dataloader and combines it with augmentations and multiprocessing.

BaseDataManager

class BaseDataManager(*data*, *batch_size*, *n_process_augmentation*, *transforms*, *sampler_cls=<class 'delira.data_loading.sampler.sequential_sampler.SequentialSampler'>*, *sampler_kwargs={}*, *data_loader_cls=None*, *dataset_cls=None*, *load_fn=<function default_load_fn_2d>*, *from_disc=True*, ***kwargs*)

Bases: object

Class to Handle Data Creates Dataset , Dataloader and BatchGenerator

batch_size

Property to access the batchsize

Returns the batchsize

Return type int

data_loader_cls

Property to access the current data loader class

Returns Subclass of SlimDataLoaderBase

Return type type

dataset
Property to access the current dataset

Returns the current dataset

Return type *AbstractDataset*

get_batchgen(*seed=1*)
Create DataLoader and Batchgenerator

Parameters **seed** (*int*) – seed for Random Number Generator

Returns Batchgenerator

Return type MultiThreadedAugmenter

Raises AssertionError – *BaseDataManager.n_batches* is smaller than or equal to zero

get_subset(*indices*)
Returns a Subset of the current datamanager based on given indices

Parameters **indices** (*iterable*) – valid indices to extract subset from current dataset

Returns manager containing the subset

Return type *BaseDataManager*

n_batches
Returns Number of Batches based on batchsize, number of samples and number of processes

Returns Number of Batches

Return type int

Raises AssertionError – *BaseDataManager.n_samples* is smaller than or equal to zero

n_process_augmentation
Property to access the number of augmentation processes

Returns number of augmentation processes

Return type int

n_samples
Number of Samples

Returns Number of Samples

Return type int

sampler
Property to access the current sampler

Returns the current sampler

Return type AbstractSampler

train_test_split(**args*, ***kwargs*)
Calls **:method:‘AbstractDataset.train_test_split‘** and returns a manager for each subset with same configuration as current manager

Parameters

- ***args** – positional arguments for sklearn.model_selection.train_test_split

- ****kwargs** – keyword arguments for `sklearn.model_selection.train_test_split`

transforms
Property to access the current data transforms

Returns The transformation, can either be None or an instance of `AbstractTransform`

Return type None, `AbstractTransform`

update_state_from_dict(*new_state: dict*)
Updates internal state and therfore the behavior from dict. If a key is not specified, the old attribute value will be used

Parameters **new_state** (*dict*) – The dict to update the state from. Valid keys are:

- `batch_size`
- `n_process_augmentation`
- `data_loader_cls`
- `sampler`
- `sampling_kwargs`
- `transforms`

If a key is not specified, the old value of the corresponding attribute will be used

Raises `KeyError` – Invalid keys are specified

Utils

default_load_fn_2d

default_load_fn_2d(*img_file*, **label_files*, *img_shape*, *n_channels=1*)
loading single 2d sample with arbitrary number of samples

Parameters

- **img_file** (*string*) – path to image file
- **label_files** (*list of strings*) – paths to label files
- **img_shape** (*iterable*) – shape of image
- **n_channels** (*int*) – number of image channels

Returns

- *numpy.ndarray* – image
- *Any* – labels

Nii-Data

Since `delira` aims to provide dataloading tools for medical data (which is often stored in Nii-Files), the following classes and functions provide a basic way to load data from nii-files:

BaseLabelGenerator

class BaseLabelGenerator(*fpath*)
Bases: object

Base Class to load labels from json files

_load()
Private Helper function to load the file

Returns loaded values from file

Return type Any

get_labels()
Abstractmethod to get labels from class

Raises NotImplementedError – if not overwritten in subclass

load_sample_nii

load_sample_nii(*files*, *label_load_cls*)
Load sample from multiple ITK files

Parameters

- **files** (dict with keys *img* and *label*) – filenames of nifti files and label file
- **label_load_cls** (*class*) – function to be used for label parsing

Returns sample: dict with keys *data* and *label* containing images and label

Return type dict

Raises AssertionError – if *img.max()* is greater than 511 or smaller than 1

Sampler

Sampler define the way of iterating over the dataset and returning samples.

AbstractSampler

class AbstractSampler(*indices=None*)
Bases: object

Class to define an abstract Sampling API

_get_indices(*n_indices*)
Function to return a specific number of indices. Implements the actual sampling strategy.

Parameters **n_indices** (*int*) – Number of indices to return

Returns List with sampled indices

Return type list

classmethod from_dataset(*dataset: delira.data_loading.dataset.AbstractDataset*, ***kwargs*)
Classmethod to initialize the sampler from a given dataset

Parameters **dataset** (AbstractDataset) – the given dataset

Returns The initialzed sampler

Return type *AbstractSampler*

LambdaSampler

class LambdaSampler(*indices*, *sampling_fn*)
Bases: delira.data_loading.sampler.abstract_sampler.AbstractSampler

Implements Arbitrary Sampling methods specified by a function which takes the index_list and the number of indices to return

_get_indices(*n_indices*)
Actual Sampling

Parameters **n_indices** (*int*) – number of indices to return

Returns list of sampled indices

Return type list

Raises StopIteration – Maximum number of indices sampled

classmethod from_dataset(*dataset: delira.data_loading.dataset.AbstractDataset*, ***kwargs*)
Classmethod to initialize the sampler from a given dataset

Parameters **dataset** (AbstractDataset) – the given dataset

Returns The initialzed sampler

Return type *AbstractSampler*

RandomSampler

class RandomSampler(*indices*)
Bases: delira.data_loading.sampler.abstract_sampler.AbstractSampler

Implements Random Sampling from whole Dataset

_get_indices(*n_indices*)
Actual Sampling

Parameters **n_indices** (*int*) – number of indices to return

Returns list of sampled indices

Return type list

Raises StopIteration – If maximal number of samples is reached

classmethod from_dataset(*dataset: delira.data_loading.dataset.AbstractDataset*, ***kwargs*)
Classmethod to initialize the sampler from a given dataset

Parameters **dataset** (AbstractDataset) – the given dataset

Returns The initialzed sampler

Return type *AbstractSampler*

PrevalenceRandomSampler

class PrevalenceRandomSampler(*indices*, *shuffle_batch=True*)
Bases: delira.data_loading.sampler.abstract_sampler.AbstractSampler

Implements random Per-Class Sampling and ensures same number of samplers per batch for each class

_get_indices(*n_indices*)
Actual Sampling

Parameters **n_indices** (*int*) – number of indices to return

Returns list of sampled indices

Return type list

Raises StopIteration – If maximal number of samples is reached

classmethod from_dataset(*dataset: delira.data_loading.dataset.AbstractDataset*, ***kwargs*)
Classmethod to initialize the sampler from a given dataset

Parameters **dataset** (AbstractDataset) – the given dataset

Returns The initialzed sampler

Return type *AbstractSampler*

StoppingPrevalenceRandomSampler

class StoppingPrevalenceRandomSampler(*indices*, *shuffle_batch=True*)
Bases: delira.data_loading.sampler.abstract_sampler.AbstractSampler

Implements random Per-Class Sampling and ensures same number of samplers per batch for each class; Stops if out of samples for smallest class

_get_indices(*n_indices*)
Actual Sampling

Parameters **n_indices** (*int*) – number of indices to return

Raises StopIteration: If end of class indices is reached for one class

Returns **list**

Return type list of sampled indices

classmethod from_dataset(*dataset: delira.data_loading.dataset.AbstractDataset*, ***kwargs*)
Classmethod to initialize the sampler from a given dataset

Parameters **dataset** (AbstractDataset) – the given dataset

Returns The initialzed sampler

Return type *AbstractSampler*

SequentialSampler

class SequentialSampler(*indices*)
Bases: delira.data_loading.sampler.abstract_sampler.AbstractSampler

Implements Sequential Sampling from whole Dataset

_get_indices(*n_indices*)
Actual Sampling

Parameters **n_indices** (*int*) – number of indices to return

Raises StopIteration : If end of dataset reached

Returns list of sampled indices

Return type list

classmethod from_dataset(*dataset: delira.data_loading.dataset.AbstractDataset*, ***kwargs*)
Classmethod to initialize the sampler from a given dataset

Parameters **dataset** (AbstractDataset) – the given dataset

Returns The initialzed sampler

Return type *AbstractSampler*

PrevalenceSequentialSampler

class PrevalenceSequentialSampler(*indices*, *shuffle_batch=True*)
Bases: delira.data_loading.sampler.abstract_sampler.AbstractSampler

Implements Per-Class Sequential sampling and ensures same number of samples per batch for each class; If out of samples for one class: restart at first sample

_get_indices(*n_indices*)
Actual Sampling

Parameters **n_indices** (*int*) – number of indices to return

Raises StopIteration : If end of class indices is reached

Returns list of sampled indices

Return type list

classmethod from_dataset(*dataset: delira.data_loading.dataset.AbstractDataset*, ***kwargs*)
Classmethod to initialize the sampler from a given dataset

Parameters **dataset** (AbstractDataset) – the given dataset

Returns The initialzed sampler

Return type *AbstractSampler*

StoppingPrevalenceSequentialSampler

class StoppingPrevalenceSequentialSampler(*indices*, *shuffle_batch=True*)
Bases: delira.data_loading.sampler.abstract_sampler.AbstractSampler

Implements Per-Class Sequential sampling and ensures same number of samples per batch for each class; Stops if all samples of first class have been sampled

_get_indices(*n_indices*)
Actual Sampling

Parameters **n_indices** (*int*) – number of indices to return

Raises StopIteration : If end of class indices is reached for one class

Returns list of sampled indices

Return type list

classmethod from_dataset(*dataset: delira.data_loading.dataset.AbstractDataset*)
Classmethod to initialize the sampler from a given dataset

Parameters **dataset** (AbstractDataset) – the given dataset

Returns The initialzed sampler

Return type *AbstractSampler*

WeightedRandomSampler

class WeightedRandomSampler(*indices*, *weights=None*, *cum_weights=None*)
Bases: delira.data_loading.sampler.abstract_sampler.AbstractSampler

Implements Weighted Random Sampling

_get_indices(*n_indices*)
Actual Sampling

Parameters **n_indices** (*int*) – number of indices to return

Returns list of sampled indices

Return type list

Raises

- StopIteration – If maximal number of samples is reached
- TypeError – if weights and cum_weights are specified at the same time
- ValueError – if weights or cum_weights don't match the population

classmethod from_dataset(*dataset: delira.data_loading.dataset.AbstractDataset*, ***kwargs*)
Classmethod to initialize the sampler from a given dataset

Parameters **dataset** (AbstractDataset) – the given dataset

Returns The initialzed sampler

Return type *AbstractSampler*

7.1.2 IO

torch_load_checkpoint

load_checkpoint(*file*, *weights_only=False*, ***kwargs*)
Loads a saved model

Parameters

- **file** (*str*) – filepath to a file containing a saved model
- **weights_only** (*bool*) – whether the file contains only weights / only weights should be returned
- ****kwargs** – Additional keyword arguments (passed to torch.load) Especially "map_location" is important to change the device the state_dict should be loaded to

Returns

- *OrderedDict* – checkpoint state_dict if *weights_only=True*
- *torch.nn.Module, OrderedDict, int* – Model, Optimizers, epoch with loaded state_dicts if *weights_only=False*

torch_save_checkpoint

save_checkpoint(*file: str*, *model=None*, *optimizers={}*, *epoch=None*, *weights_only=False*, ***kwargs*)
Save model's parameters

Parameters

- **file** (*str*) – filepath the model should be saved to
- **model** (*AbstractNetwork or None*) – the model which should be saved if None: empty dict will be saved as state dict
- **optimizers** (*dict*) – dictionary containing all optimizers
- **epoch** (*int*) – current epoch (will also be pickled)
- **weights_only** (*bool*) – whether or not to save only the model's weights or also save additional information (for easy loading)

tf_load_checkpoint

load_checkpoint(*file: str*, *model=None*)
Loads a saved model

Parameters

- **file** (*str*) – filepath to a file containing a saved model
- **model** (*TfNetwork*) – the model which should be loaded

tf_save_checkpoint

save_checkpoint(*file: str*, *model=None*)
Save model's parameters contained in it's graph

Parameters

- **file** (*str*) – filepath the model should be saved to
- **model** (*TfNetwork*) – the model which should be saved

7.1.3 Logging

This module handles the embedding of `trixi`'s loggers into the python `logging` module.

MultiStreamHandler

class MultiStreamHandler(**streams*, *level=0*)
Bases: `logging.Handler`

Logging Handler which accepts multiple streams and creates StreamHandlers

acquire()
Acquire the I/O thread lock.

addFilter(*filter*)
Add the specified filter to this handler.

close()
Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, _handlers, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden close() methods.

createLock()
Acquire a thread lock for serializing access to the underlying I/O.

emit(*record*)
logs the record entity to streams

Parameters **record** (*LogRecord*) – record to log

filter(*record*)
Determine if a record is loggable by consulting all the filters.

The default is to allow the record to be logged; any filter can veto this and the record is then dropped. Returns a zero value if a record is to be dropped, else non-zero.

Changed in version 3.2: Allow filters to be just callables.

flush()
Ensure all logging output has been flushed.

This version does nothing and is intended to be implemented by subclasses.

format(*record*)
Format the specified record.

If a formatter is set, use it. Otherwise, use the default formatter for the module.

get_name()

handle(*record*)
Conditionally emit the specified logging record.

Emission depends on filters which may have been added to the handler. Wrap the actual emission of the record with acquisition/release of the I/O thread lock. Returns whether the filter passed the record for emission.

handleError(*record*)
Handle errors which occur during an emit() call.

This method should be called from handlers when an exception is encountered during an emit() call. If raiseExceptions is false, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The record which was being processed is passed in to this method.

name

release()
Release the I/O thread lock.

removeFilter(*filter*)
Remove the specified filter from this handler.

setFormatter(*fmt*)
Set the formatter for this handler.

setLevel(*level*)
Set the logging level of this handler. level must be an int or a str.

set_name(*name*)

TrixiHandler

class TrixiHandler(*logging_cls*, *level=0*, **args*, ***kwargs*)
Bases: `logging.Handler`

Handler to integrate the `trixi` loggers into the `logging` module

acquire()
Acquire the I/O thread lock.

addFilter(*filter*)
Add the specified filter to this handler.

close()
Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, _handlers, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden close() methods.

createLock()
Acquire a thread lock for serializing access to the underlying I/O.

emit(*record*)
logs the record entity to *trixi* loggers

Parameters **record** (*LogRecord*) – record to log

filter(*record*)
Determine if a record is loggable by consulting all the filters.

The default is to allow the record to be logged; any filter can veto this and the record is then dropped. Returns a zero value if a record is to be dropped, else non-zero.

Changed in version 3.2: Allow filters to be just callables.

flush()
Ensure all logging output has been flushed.

This version does nothing and is intended to be implemented by subclasses.

format(*record*)
Format the specified record.

If a formatter is set, use it. Otherwise, use the default formatter for the module.

get_name()

handle(*record*)
Conditionally emit the specified logging record.

Emission depends on filters which may have been added to the handler. Wrap the actual emission of the record with acquisition/release of the I/O thread lock. Returns whether the filter passed the record for emission.

handleError(*record*)
Handle errors which occur during an emit() call.

This method should be called from handlers when an exception is encountered during an emit() call. If raiseExceptions is false, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The record which was being processed is passed in to this method.

name

release()
Release the I/O thread lock.

removeFilter(*filter*)
Remove the specified filter from this handler.

setFormatter(*fmt*)
Set the formatter for this handler.

setLevel(*level*)
Set the logging level of this handler. level must be an int or a str.

set_name(*name*)

7.1.4 Models

`delira` comes with it's own model-structure tree - with `AbstractNetwork` at it's root - and integrates PyTorch Models (`AbstractPyTorchNetwork`) deeply into the model structure. Tensorflow Integration is planned.

AbstractNetwork

class AbstractNetwork(*type*)
Bases: `object`

Abstract class all networks should be derived from

_init_kwargs = {}

static closure(*model*, *data_dict: dict*, *optimizers: dict*, *criterions={}*, *metrics={}*, *fold=0*, ***kwargs*)
Function which handles prediction from batch, logging, loss calculation and optimizer step :param model: model to forward data through :type model: *AbstractNetwork* :param data_dict: dictionary containing the data :type data_dict: dict :param optimizers: dictionary containing all optimizers to perform parameter update :type optimizers: dict :param criterions: Functions or classes to calculate criterions :type criterions: dict :param metrics: Functions or classes to calculate other metrics :type metrics: dict :param fold: Current Fold in Crossvalidation (default: 0) :type fold: int :param kwargs: additional keyword arguments :type kwargs: dict

Returns

- *dict* – Metric values (with same keys as input dict metrics)
- *dict* – Loss values (with same keys as input dict criterions)
- *list* – Arbitrary number of predictions

Raises `NotImplementedError` – If not overwritten by subclass

init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

static prepare_batch(*batch: dict*, *input_device*, *output_device*)
Converts a numpy batch of data and labels to suitable datatype and pushes them to correct devices

Parameters

- **batch** (*dict*) – dictionary containing the batch (must have keys 'data' and 'label'
- **input_device** – device for network inputs
- **output_device** – device for network outputs

Returns dictionary containing all necessary data in right format and type and on the correct device

Return type dict

Raises NotImplementedError – If not overwritten by subclass

AbstractPyTorchNetwork

class AbstractPyTorchNetwork(*type*)
Bases: delira.models.abstract_network.AbstractNetwork, sphinx.ext.autodoc.importer._MockObject

Abstract Class for PyTorch Networks

See also:

torch.nn.Module *AbstractNetwork*

_init_kwargs = {}

static closure(*model*, *data_dict: dict*, *optimizers: dict*, *criterions={}*, *metrics={}*, *fold=0*, ***kwargs*)
Function which handles prediction from batch, logging, loss calculation and optimizer step :param model: model to forward data through :type model: *AbstractNetwork* :param data_dict: dictionary containing the data :type data_dict: dict :param optimizers: dictionary containing all optimizers to perform parameter update :type optimizers: dict :param criterions: Functions or classes to calculate criterions :type criterions: dict :param metrics: Functions or classes to calculate other metrics :type metrics: dict :param fold: Current Fold in Crossvalidation (default: 0) :type fold: int :param kwargs: additional keyword arguments :type kwargs: dict

Returns

- *dict* – Metric values (with same keys as input dict metrics)
- *dict* – Loss values (with same keys as input dict criterions)
- *list* – Arbitrary number of predictions

Raises NotImplementedError – If not overwritten by subclass

forward(**inputs*)
Forward inputs through module (defines module behavior) :param inputs: inputs of arbitrary type and number :type inputs: list

Returns result: module results of arbitrary type and number

Return type Any

init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

static prepare_batch(*batch: dict*, *input_device*, *output_device*)
Helper Function to prepare Network Inputs and Labels (convert them to correct type and shape and push them to correct devices)

Parameters

- **batch** (*dict*) – dictionary containing all the data
- **input_device** (*torch.device*) – device for network inputs
- **output_device** (*torch.device*) – device for network outputs

Returns dictionary containing data in correct type and shape and on correct device

Return type dict

AbstractTfNetwork

class AbstractTfNetwork(*sess=<sphinx.ext.autodoc.importer._MockObject object>*, ***kwargs*)
Bases: delira.models.abstract_network.AbstractNetwork

Abstract Class for Tf Networks

See also:

AbstractNetwork

_add_losses(*losses: dict*)
Add losses to the model graph

Parameters **losses** (*dict*) – dictionary containing losses.

_add_optims(*optims: dict*)
Add optimizers to the model graph

Parameters **optims** (*dict*) – dictionary containing losses.

_init_kwargs = {}

static closure(*model*, *data_dict: dict*, *optimizers: dict*, *criterions={}*, *metrics={}*, *fold=0*, ***kwargs*)
Function which handles prediction from batch, logging, loss calculation and optimizer step :param model: model to forward data through :type model: *AbstractNetwork* :param data_dict: dictionary containing the data :type data_dict: dict :param optimizers: dictionary containing all optimizers to perform parameter update :type optimizers: dict :param criterions: Functions or classes to calculate criterions :type criterions: dict :param metrics: Functions or classes to calculate other metrics :type metrics: dict :param fold: Current Fold in Crossvalidation (default: 0) :type fold: int :param kwargs: additional keyword arguments :type kwargs: dict

Returns

- *dict* – Metric values (with same keys as input dict metrics)
- *dict* – Loss values (with same keys as input dict criterions)
- *list* – Arbitrary number of predictions

Raises NotImplementedError – If not overwritten by subclass

init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

static prepare_batch(*batch: dict*, *input_device*, *output_device*)
Converts a numpy batch of data and labels to suitable datatype and pushes them to correct devices

Parameters

- **batch** (*dict*) – dictionary containing the batch (must have keys 'data' and 'label'
- **input_device** – device for network inputs
- **output_device** – device for network outputs

Returns dictionary containing all necessary data in right format and type and on the correct device

Return type dict

Raises NotImplementedError – If not overwritten by subclass

run(**args*)
Evaluates *self.outputs_train* or *self.outputs_eval* based on *self.training*

Parameters ***args** – arguments to feed as *self.inputs*. Must have same length as *self.inputs*

Returns based on len(self.outputs*), returns either list or np.ndarray

Return type np.ndarray or list

Classification

ClassificationNetworkBasePyTorch

class ClassificationNetworkBasePyTorch(*in_channels: int*, *n_outputs: int*, ***kwargs*)
Bases: delira.models.abstract_network.AbstractPyTorchNetwork

Implements basic classification with ResNet18

References

https://arxiv.org/abs/1512.03385

See also:

AbstractPyTorchNetwork

static _build_model(*in_channels: int*, *n_outputs: int*, ***kwargs*)
builds actual model (resnet 18)

Parameters

- **in_channels** (*int*) – number of input channels
- **n_outputs** (*int*) – number of outputs (usually same as number of classes)
- ****kwargs** (*dict*) – additional keyword arguments

Returns created model

Return type torch.nn.Module

_init_kwargs = {}

static closure(*model: delira.models.abstract_network.AbstractPyTorchNetwork*, *data_dict: dict*, *optimizers: dict*, *criterions={}*, *metrics={}*, *fold=0*, ***kwargs*)

closure method to do a single backpropagation step

Parameters

- **model** (*ClassificationNetworkBasePyTorch*) – trainable model
- **data_dict** (*dict*) – dictionary containing the data
- **optimizers** (*dict*) – dictionary of optimizers to optimize model's parameters
- **criterions** (*dict*) – dict holding the criterions to calculate errors (gradients from different criterions will be accumulated)
- **metrics** (*dict*) – dict holding the metrics to calculate
- **fold** (*int*) – Current Fold in Crossvalidation (default: 0)
- ****kwargs** – additional keyword arguments

Returns

- *dict* – Metric values (with same keys as input dict metrics)
- *dict* – Loss values (with same keys as input dict criterions)
- *list* – Arbitrary number of predictions as torch.Tensor

Raises AssertionError – if optimizers or criterions are empty or the optimizers are not specified

forward(*input_batch: <sphinx.ext.autodoc.importer._MockObject object at 0x7f43be76bdd8>*)

Forward input_batch through network

Parameters **input_batch** (*torch.Tensor*) – batch to forward through network

Returns Classification Result

Return type torch.Tensor

init_kwargs

Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

static prepare_batch(*batch: dict*, *input_device*, *output_device*)

Helper Function to prepare Network Inputs and Labels (convert them to correct type and shape and push them to correct devices)

Parameters

- **batch** (*dict*) – dictionary containing all the data
- **input_device** (*torch.device*) – device for network inputs
- **output_device** (*torch.device*) – device for network outputs

Returns dictionary containing data in correct type and shape and on correct device

Return type dict

VGG3DClassificationNetworkPyTorch

class VGG3DClassificationNetworkPyTorch(*in_channels: int*, *n_outputs: int*, ***kwargs*)

Bases: delira.models.classification.classification_network.ClassificationNetworkBasePyTorch

Exemplaric VGG Network for 3D Classification

Notes

The original network has been adjusted to fit for 3D data

References

https://arxiv.org/abs/1409.1556

See also:

ClassificationNetworkBasePyTorch

static _build_model(*in_channels: int*, *n_outputs: int*, ***kwargs*)

Helper Function to build the actual model

Parameters

- **in_channels** (*int*) – number of input channels
- **n_outputs** (*int*) – number of outputs
- ****kwargs** – additional keyword arguments

Returns ensembeled model

Return type torch.nn.Module

_init_kwargs = {}

static closure(*model: delira.models.abstract_network.AbstractPyTorchNetwork*, *data_dict: dict*, *optimizers: dict*, *criterions={}*, *metrics={}*, *fold=0*, ***kwargs*)

closure method to do a single backpropagation step

Parameters

- **model** (*ClassificationNetworkBasePyTorch*) – trainable model
- **data_dict** (*dict*) – dictionary containing the data
- **optimizers** (*dict*) – dictionary of optimizers to optimize model's parameters
- **criterions** (*dict*) – dict holding the criterions to calculate errors (gradients from different criterions will be accumulated)
- **metrics** (*dict*) – dict holding the metrics to calculate
- **fold** (*int*) – Current Fold in Crossvalidation (default: 0)
- ****kwargs** – additional keyword arguments

Returns

- *dict* – Metric values (with same keys as input dict metrics)
- *dict* – Loss values (with same keys as input dict criterions)

- *list* – Arbitrary number of predictions as torch.Tensor

Raises `AssertionError` – if optimizers or criterions are empty or the optimizers are not specified

forward(*input_batch: <sphinx.ext.autodoc.importer._MockObject object at 0x7f43be76bdd8>*)
Forward input_batch through network

Parameters **input_batch** (*torch.Tensor*) – batch to forward through network

Returns Classification Result

Return type torch.Tensor

init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

static prepare_batch(*batch: dict*, *input_device*, *output_device*)
Helper Function to prepare Network Inputs and Labels (convert them to correct type and shape and push them to correct devices)

Parameters

- **batch** (*dict*) – dictionary containing all the data
- **input_device** (*torch.device*) – device for network inputs
- **output_device** (*torch.device*) – device for network outputs

Returns dictionary containing data in correct type and shape and on correct device

Return type dict

ClassificationNetworkBaseTf

class ClassificationNetworkBaseTf(*in_channels: int*, *n_outputs: int*, ***kwargs*)
Bases: `delira.models.abstract_network.AbstractTfNetwork`

Implements basic classification with ResNet18

See also:

`AbstractTfNetwork`

_add_losses(*losses: dict*)
Adds losses to model that are to be used by optimizers or during evaluation

Parameters **losses** (*dict*) – dictionary containing all losses. Individual losses are averaged

_add_optims(*optims: dict*)
Adds optims to model that are to be used by optimizers or during training

Parameters **optim** (*dict*) – dictionary containing all optimizers, optimizers should be of Type[tf.train.Optimizer]

static _build_model(*n_outputs: int*, ***kwargs*)
builds actual model (resnet 18)

Parameters

- **n_outputs** (*int*) – number of outputs (usually same as number of classes)

- ****kwargs** – additional keyword arguments

Returns created model

Return type tf.keras.Model

_init_kwargs = {}

static closure(*model: Type[delira.models.abstract_network.AbstractTfNetwork], data_dict: dict, metrics={}, fold=0, **kwargs*)

closure method to do a single prediction. This is followed by backpropagation or not based state of on model.train

Parameters

- **model** (AbstractTfNetwork) – AbstractTfNetwork or its child-clases
- **data_dict** (*dict*) – dictionary containing the data
- **metrics** (*dict*) – dict holding the metrics to calculate
- **fold** (*int*) – Current Fold in Crossvalidation (default: 0)
- ****kwargs** – additional keyword arguments

Returns

- *dict* – Metric values (with same keys as input dict metrics)
- *dict* – Loss values (with same keys as those initially passed to model.init). Additionally, a total_loss key is added
- *list* – Arbitrary number of predictions as np.array

init_kwargs

Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

static prepare_batch(*batch: dict*, *input_device*, *output_device*)

Converts a numpy batch of data and labels to suitable datatype and pushes them to correct devices

Parameters

- **batch** (*dict*) – dictionary containing the batch (must have keys 'data' and 'label'
- **input_device** – device for network inputs
- **output_device** – device for network outputs

Returns dictionary containing all necessary data in right format and type and on the correct device

Return type dict

Raises NotImplementedError – If not overwritten by subclass

run(**args*)

Evaluates *self.outputs_train* or *self.outputs_eval* based on *self.training*

Parameters ***args** – arguments to feed as *self.inputs*. Must have same length as *self.inputs*

Returns based on len(self.outputs*), returns either list or np.ndarray

Return type np.ndarray or list

Generative Adversarial Networks

GenerativeAdversarialNetworkBasePyTorch

class GenerativeAdversarialNetworkBasePyTorch(*n_channels*, *noise_length*, ***kwargs*)

Bases: `delira.models.abstract_network.AbstractPyTorchNetwork`

Implementation of Vanilla DC-GAN to create 64x64 pixel images

Notes

The fully connected part in the discriminator has been replaced with an equivalent convolutional part

References

https://arxiv.org/abs/1511.06434

See also:

`AbstractPyTorchNetwork`

static _build_models(*in_channels*, *noise_length*, ***kwargs*)

Builds actual generator and discriminator models

Parameters

- **in_channels** (*int*) – number of channels for generated images by generator and inputs of discriminator
- **noise_length** (*int*) – length of noise vector (generator input)
- ****kwargs** – additional keyword arguments

Returns

- *torch.nn.Sequential* – generator
- *torch.nn.Sequential* – discriminator

_init_kwargs = {}

static closure(*model*, *data_dict: dict*, *optimizers: dict*, *criterions={}*, *metrics={}*, *fold=0*, ***kwargs*)

closure method to do a single backpropagation step

Parameters

- **model** (`ClassificationNetworkBase`) – trainable model
- **data_dict** (*dict*) – dictionary containing data
- **optimizers** (*dict*) – dictionary of optimizers to optimize model's parameters
- **criterions** (*dict*) – dict holding the criterions to calculate errors (gradients from different criterions will be accumulated)
- **metrics** (*dict*) – dict holding the metrics to calculate
- **fold** (*int*) – Current Fold in Crossvalidation (default: 0)
- **kwargs** (*dict*) – additional keyword arguments

Returns

- *dict* – Metric values (with same keys as input dict metrics)
- *dict* – Loss values (with same keys as input dict criterions)
- *list* – Arbitrary number of predictions as torch.Tensor

Raises `AssertionError` – if optimizers or criterions are empty or the optimizers are not specified

forward(*real_image_batch*)

Create fake images by feeding noise through generator and feed results and real images through discriminator

Parameters **real_image_batch** (*torch.Tensor*) – batch of real images

Returns

- *torch.Tensor* – Generated fake images
- *torch.Tensor* – Discriminator prediction of fake images
- *torch.Tensor* – Discriminator prediction of real images

init_kwargs

Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

static prepare_batch(*batch: dict*, *input_device*, *output_device*)

Helper Function to prepare Network Inputs and Labels (convert them to correct type and shape and push them to correct devices)

Parameters

- **batch** (*dict*) – dictionary containing all the data
- **input_device** (*torch.device*) – device for network inputs
- **output_device** (*torch.device*) – device for network outputs

Returns dictionary containing data in correct type and shape and on correct device

Return type dict

Segmentation

UNet2dPyTorch

class UNet2dPyTorch(*num_classes*, *in_channels=1*, *depth=5*, *start_filts=64*, *up_mode='transpose'*, *merge_mode='concat'*)

Bases: `delira.models.abstract_network.AbstractPyTorchNetwork`

The *UNet2dPyTorch* is a convolutional encoder-decoder neural network. Contextual spatial information (from the decoding, expansive pathway) about an input tensor is merged with information representing the localization of details (from the encoding, compressive pathway).

Notes

Differences to the original paper:

- padding is used in 3x3 convolutions to prevent loss of border pixels

- merging outputs does not require cropping due to (1)
- residual connections can be used by specifying `merge_mode='add'`
- **if non-parametric upsampling is used in the decoder pathway** (specified by upmode='upsample'), then an additional 1x1 2d convolution occurs after upsampling to reduce channel dimensionality by a factor of 2. This channel halving happens with the convolution in the tranpose convolution (specified by `upmode='transpose'`)

References

https://arxiv.org/abs/1505.04597

See also:

UNet3dPyTorch

_build_model(*num_classes*, *in_channels=3*, *depth=5*, *start_filts=64*)
Builds the actual model

Parameters

- **num_classes** (*int*) – number of output classes
- **in_channels** (*int*) – number of channels for the input tensor (default: 1)
- **depth** (*int*) – number of MaxPools in the U-Net (default: 5)
- **start_filts** (*int*) – number of convolutional filters for the first conv (affects all other conv-filter numbers too; default: 64)

Notes

The Helper functions and classes are defined within this function because `delira` offers a possibility to save the source code along the weights to completely recover the network without needing a manually created network instance and these helper functions have to be saved too.

_init_kwargs = {}

static closure(*model*, *data_dict: dict*, *optimizers: dict*, *criterions={}*, *metrics={}*, *fold=0*, ***kwargs*)
closure method to do a single backpropagation step

Parameters

- **model** (`ClassificationNetworkBasePyTorch`) – trainable model
- **data_dict** (*dict*) – dictionary containing the data
- **optimizers** (*dict*) – dictionary of optimizers to optimize model's parameters
- **criterions** (*dict*) – dict holding the criterions to calculate errors (gradients from different criterions will be accumulated)
- **metrics** (*dict*) – dict holding the metrics to calculate
- **fold** (*int*) – Current Fold in Crossvalidation (default: 0)
- ****kwargs** – additional keyword arguments

Returns

- *dict* – Metric values (with same keys as input dict metrics)

- *dict* – Loss values (with same keys as input dict criterions)
- *list* – Arbitrary number of predictions as torch.Tensor

Raises `AssertionError` – if optimizers or criterions are empty or the optimizers are not specified

forward(*x*)
Feed tensor through network

Parameters **x** (*torch.Tensor*) –

Returns Prediction

Return type torch.Tensor

init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

static prepare_batch(*batch: dict*, *input_device*, *output_device*)
Helper Function to prepare Network Inputs and Labels (convert them to correct type and shape and push them to correct devices)

Parameters

- **batch** (*dict*) – dictionary containing all the data
- **input_device** (*torch.device*) – device for network inputs
- **output_device** (*torch.device*) – device for network outputs

Returns dictionary containing data in correct type and shape and on correct device

Return type dict

reset_params()
Initialize all parameters

static weight_init(*m*)
Initializes weights with xavier_normal and bias with zeros

Parameters **m** (*torch.nn.Module*) – module to initialize

UNet3dPyTorch

class UNet3dPyTorch(*num_classes*, *in_channels=3*, *depth=5*, *start_filts=64*, *up_mode='transpose'*, *merge_mode='concat'*)
Bases: `delira.models.abstract_network.AbstractPyTorchNetwork`

The *UNet3dPyTorch* is a convolutional encoder-decoder neural network. Contextual spatial information (from the decoding, expansive pathway) about an input tensor is merged with information representing the localization of details (from the encoding, compressive pathway).

Notes

Differences to the original paper:

- Working on 3D data instead of 2D slices

- **padding is used in 3x3x3 convolutions to prevent loss of border** pixels
- merging outputs does not require cropping due to (1)
- residual connections can be used by specifying `merge_mode='add'`
- **if non-parametric upsampling is used in the decoder pathway (** specified by upmode='upsample'), then an additional 1x1x1 3d convolution occurs after upsampling to reduce channel dimensionality by a factor of 2. This channel halving happens with the convolution in the tranpose convolution (specified by `upmode='transpose'`)

References

https://arxiv.org/abs/1505.04597

See also:

UNet2dPyTorch

_build_model(*num_classes*, *in_channels=3*, *depth=5*, *start_filts=64*)
Builds the actual model

Parameters

- **num_classes** (*int*) – number of output classes
- **in_channels** (*int*) – number of channels for the input tensor (default: 1)
- **depth** (*int*) – number of MaxPools in the U-Net (default: 5)
- **start_filts** (*int*) – number of convolutional filters for the first conv (affects all other conv-filter numbers too; default: 64)

Notes

The Helper functions and classes are defined within this function because `delira` offers a possibility to save the source code along the weights to completely recover the network without needing a manually created network instance and these helper functions have to be saved too.

_init_kwargs = {}

static closure(*model*, *data_dict: dict*, *optimizers: dict*, *criterions={}*, *metrics={}*, *fold=0*, ***kwargs*)
closure method to do a single backpropagation step

Parameters

- **model** (`ClassificationNetworkBasePyTorch`) – trainable model
- **data_dict** (*dict*) – dictionary containing the data
- **optimizers** (*dict*) – dictionary of optimizers to optimize model's parameters
- **criterions** (*dict*) – dict holding the criterions to calculate errors (gradients from different criterions will be accumulated)
- **metrics** (*dict*) – dict holding the metrics to calculate
- **fold** (*int*) – Current Fold in Crossvalidation (default: 0)
- ****kwargs** – additional keyword arguments

Returns

- *dict* – Metric values (with same keys as input dict metrics)
- *dict* – Loss values (with same keys as input dict criterions)
- *list* – Arbitrary number of predictions as torch.Tensor

Raises `AssertionError` – if optimizers or criterions are empty or the optimizers are not specified

forward(*x*)
Feed tensor through network

Parameters **x** (*torch.Tensor*) –

Returns Prediction

Return type torch.Tensor

init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

static prepare_batch(*batch: dict*, *input_device*, *output_device*)
Helper Function to prepare Network Inputs and Labels (convert them to correct type and shape and push them to correct devices)

Parameters

- **batch** (*dict*) – dictionary containing all the data
- **input_device** (*torch.device*) – device for network inputs
- **output_device** (*torch.device*) – device for network outputs

Returns dictionary containing data in correct type and shape and on correct device

Return type dict

reset_params()
Initialize all parameters

static weight_init(*m*)
Initializes weights with xavier_normal and bias with zeros

Parameters **m** (*torch.nn.Module*) – module to initialize

7.1.5 Training

The training subpackage implements Callbacks, a class for Hyperparameters, training routines and wrapping experiments.

Parameters

Parameters

class Parameters(*fixed_params={'model': {}, 'training': {}}*, *variable_params={'model': {}, 'training': {}}*)
Bases: *delira.utils.config.LookupConfig*

Class Containing all variable and fixed parameters for training and model instantiation

See also:

`trixi.util.Config`

hierarchy
Returns the current hierarchy

Returns current hierarchy

Return type str

nested_get(*key*, **args*, ***kwargs*)
Returns all occurances of `key` in `self` and subdicts

Parameters

- **key** (*str*) – the key to search for
- ***args** – positional arguments to provide default value
- ****kwargs** – keyword arguments to provide default value

Raises `KeyError` – Multiple Values are found for key (unclear which value should be returned) OR No Value was found for key and no default value was given

Returns value corresponding to key (or default if value was not found)

Return type Any

permute_hierarchy()
switches hierarchy

Returns the class with a permuted hierarchy

Return type *Parameters*

Raises `AttributeError` – if no valid hierarchy is found

permute_to_hierarchy(*hierarchy: str*)
Permute hierarchy to match the specified hierarchy

Parameters **hierarchy** (*str*) – target hierarchy

Raises `ValueError` – Specified hierarchy is invalid

Returns parameters with proper hierarchy

Return type *Parameters*

permute_training_on_top()
permutes hierarchy in a way that the training-model hierarchy is on top

Returns Parameters with permuted hierarchy

Return type *Parameters*

permute_variability_on_top()
permutes hierarchy in a way that the training-model hierarchy is on top

Returns Parameters with permuted hierarchy

Return type *Parameters*

save(*filepath: str*)
Saves class to given filepath (YAML + Pickle)

Parameters **filepath** (*str*) – file to save data to

training_on_top
Return whether the training hierarchy is on top

Returns whether training is on top

Return type bool

variability_on_top
Return whether the variability is on top

Returns whether variability is on top

Return type bool

NetworkTrainer

The network trainer implements the actual training routine and can be subclassed for special routines.

Subclassing your trainer also means you have to subclass your experiment (to use the trainer).

AbstractNetworkTrainer

class AbstractNetworkTrainer(*fold=0*, *callbacks=[]*)
Bases: object

Defines an abstract API for Network Trainers

See also:

PyTorchNetworkTrainer

_at_epoch_begin(**args*, ***kwargs*)
Defines the behaviour at beginnig of each epoch

Parameters

- ***args** – positional arguments
- ****kwargs** – keyword arguments

Raises NotImplementedError – If not overwritten by subclass

_at_epoch_end(**args*, ***kwargs*)
Defines the behaviour at the end of each epoch

Parameters

- ***args** – positional arguments
- ****kwargs** – keyword arguments

Raises NotImplementedError – If not overwritten by subclass

_at_training_begin(**args*, ***kwargs*)
Defines the behaviour at beginnig of the training

Parameters

- ***args** – positional arguments
- ****kwargs** – keyword arguments

Raises NotImplementedError – If not overwritten by subclass

_at_training_end(**args*, ***kwargs*)
Defines the behaviour at the end of the training

Parameters

- ***args** – positional arguments
- ****kwargs** – keyword arguments

Raises NotImplementedError – If not overwritten by subclass

static _is_better_val_scores(*old_val_score*, *new_val_score*, *mode='highest'*)
Check whether the new val score is better than the old one with respect to the optimization goal

Parameters

- **old_val_score** – old validation score
- **new_val_score** – new validation score
- **mode** (*str*) – String to specify whether a higher or lower validation score is optimal; must be in ['highest', 'lowest']

Returns True if new score is better, False otherwise

Return type bool

_setup(**args*, ***kwargs*)
Defines the actual Trainer Setup

Parameters

- ***args** – positional arguments
- ****kwargs** – keyword arguments

Raises NotImplementedError – If not overwritten by subclass

_train_single_epoch(*batchgen: <sphinx.ext.autodoc.importer._MockObject object at 0x7f43be1b9710>*, *epoch*)
Defines a routine to train a single epoch

Parameters

- **batchgen** (*MultiThreadedAugmenter*) – generator holding the batches
- **epoch** (*int*) – current epoch

Raises NotImplementedError – If not overwritten by subclass

_update_state(*new_state*)
Update the state from a given new state

Parameters **new_state** (*dict*) – new state to update internal state from

Returns the trainer with a modified state

Return type *AbstractNetworkTrainer*

fold
Get current fold

Returns current fold

Return type int

static load_state(*file_name*, **args*, ***kwargs*)
Loads the new state from file

Parameters

- **file_name** (*str*) – the file to load the state from
- ***args** – positional arguments
- ****kwargs** (*keyword arguments*) –

Returns new state

Return type dict

predict (*batchgen*, *batchsize=None*)
Defines a rotine to predict data obtained from a batchgenerator

Parameters

- **batchgen** (*MultiThreadedAugmenter*) – Generator Holding the Batches
- **batchsize** (*Artificial batchsize (sampling will be done with batchsize*) – 1 and sampled data will be stacked to match the artificial batchsize)(default: None)

Raises NotImplementedError – If not overwritten by subclass

register_callback (*callback: delira.training.callbacks.abstract_callback.AbstractCallback*)
Register Callback to Trainer

Parameters **callback** (AbstractCallback) – the callback to register

Raises AssertionError – *callback* is not an instance of AbstractCallback and has not both methods [‘at_epoch_begin’, ‘at_epoch_end’]

save_state (*file_name*, **args*, ***kwargs*)
saves the current state

Parameters

- **file_name** (*str*) – filename to save the state to
- ***args** – positional arguments
- ****kwargs** – keyword arguments

train (*num_epochs*, *datamgr_train*, *datamgr_valid=None*, *val_score_key=None*, *val_score_mode='highest'*)
Defines a routine to train a specified number of epochs

Parameters

- **num_epochs** (*int*) – number of epochs to train
- **datamgr_train** (*DataManager*) – the datamanager holding the train data
- **datamgr_valid** (*DataManager*) – the datamanager holding the validation data (default: None)
- **val_score_key** (*str*) – the key specifying which metric to use for validation (default: None)
- **val_score_mode** (*str*) – key specifying what kind of validation score is best

Raises NotImplementedError – If not overwritten by subclass

update_state (*file_name*, **args*, ***kwargs*)
Update internal state from a loaded state

Parameters

- **file_name** (*str*) – file containing the new state to load
- ***args** – positional arguments
- ****kwargs** – keyword arguments

Returns the trainer with a modified state

Return type *AbstractNetworkTrainer*

PyTorchNetworkTrainer

class PyTorchNetworkTrainer(*network*, *save_path*, *criterions: dict*, *optimizer_cls*, *optimizer_params={}*, *metrics={}*, *lr_scheduler_cls=None*, *lr_scheduler_params={}*, *gpu_ids=[]*, *save_freq=1*, *optim_fn=<function create_optims_default_pytorch>*, *fold=0*, *callbacks=[]*, *start_epoch=1*, *mixed_precision=False*, *mixed_precision_kwargs={'allow_banned': False, 'enable_caching': True, 'verbose': False}*, ***kwargs*)

Bases: `delira.training.abstract_trainer.AbstractNetworkTrainer`

Train and Validate a Network

See also:

`AbstractNetwork`

_at_epoch_begin(*metrics_val*, *val_score_key*, *epoch*, *num_epochs*, ***kwargs*)

Defines behaviour at beginning of each epoch: Executes all callbacks's *at_epoch_begin* method

Parameters

- **metrics_val** (*dict*) – validation metrics
- **val_score_key** (*str*) – validation score key
- **epoch** (*int*) – current epoch
- **num_epochs** (*int*) – total number of epochs
- ****kwargs** – keyword arguments

_at_epoch_end(*metrics_val*, *val_score_key*, *epoch*, *is_best*, ***kwargs*)

Defines behaviour at beginning of each epoch: Executes all callbacks's *at_epoch_end* method and saves current state if necessary

Parameters

- **metrics_val** (*dict*) – validation metrics
- **val_score_key** (*str*) – validation score key
- **epoch** (*int*) – current epoch
- **num_epochs** (*int*) – total number of epochs
- **is_best** (*bool*) – whether current model is best one so far
- ****kwargs** – keyword arguments

_at_training_begin(**args*, ***kwargs*)

Defines behaviour at beginning of training

Parameters

- ***args** – positional arguments

- ****kwargs** – keyword arguments

_at_training_end()
Defines Behaviour at end of training: Loads best model if available

Returns best network

Return type AbstractPyTorchNetwork

static _is_better_val_scores(*old_val_score*, *new_val_score*, *mode='highest'*)
Check whether the new val score is better than the old one with respect to the optimization goal

Parameters

- **old_val_score** – old validation score
- **new_val_score** – new validation score
- **mode** (*str*) – String to specify whether a higher or lower validation score is optimal; must be in ['highest', 'lowest']

Returns True if new score is better, False otherwise

Return type bool

_setup(*network*, *optim_fn*, *optimizer_cls*, *optimizer_params*, *lr_scheduler_cls*, *lr_scheduler_params*, *gpu_ids*, *mixed_precision*, *mixed_precision_kwargs*)
Defines the Trainers Setup

Parameters

- **network** (AbstractPyTorchNetwork) – the network to train
- **optim_fn** (*function*) – creates a dictionary containing all necessary optimizers
- **optimizer_cls** (*subclass of torch.optim.Optimizer*) – optimizer class implementing the optimization algorithm of choice
- **optimizer_params** (*dict*) –
- **lr_scheduler_cls** (*Any*) – learning rate schedule class: must implement step() method
- **lr_scheduler_params** (*dict*) – keyword arguments passed to lr scheduler during construction
- **gpu_ids** (*list*) – list containing ids of GPUs to use; if empty: use cpu instead
- **mixed_precision** (*bool*) – whether to use mixed precision or not (False per default)
- **mixed_precision_kwargs** (*dict*) – additional keyword arguments for mixed precision

_train_single_epoch(*batchgen: <sphinx.ext.autodoc.importer._MockObject object at 0x7f43be7956d8>*, *epoch*)
Trains the network a single epoch

Parameters

- **batchgen** (*MultiThreadedAugmenter*) – Generator yielding the training batches
- **epoch** (*int*) – current epoch

_update_state(*new_state*)
Update the state from a given new state

Parameters **new_state** (*dict*) – new state to update internal state from

Returns the trainer with a modified state

Return type *AbstractNetworkTrainer*

fold
Get current fold

Returns current fold

Return type int

static load_state(*file_name*, *weights_only=True*, ***kwargs*)
Loads the new state from file via *delira.io.torch.load_checkpoint()*

Parameters

- **file_name** (*str*) – the file to load the state from
- **weights_only** (*bool*) – whether file contains stored weights only (default: False)
- ****kwargs** (*keyword arguments*) –

Returns new state

Return type dict

predict(*batchgen*, *batch_size=None*)
Returns predictions from network for batches from batchgen

Parameters

- **batchgen** (*MultiThreadedAugmenter*) – Generator yielding the batches to predict
- **batch_size** (*None or int*) – if int: collect batches until batch_size is reached and forward them together

Returns

- *np.ndarray* – predictions from batches
- *list of np.ndarray* – labels from batches
- *dict* – dictionary containing the mean validation metrics and the mean loss values

register_callback(*callback: delira.training.callbacks.abstract_callback.AbstractCallback*)
Register Callback to Trainer

Parameters **callback** (AbstractCallback) – the callback to register

Raises AssertionError – *callback* is not an instance of AbstractCallback and has not both methods [‘at_epoch_begin’, ‘at_epoch_end’]

save_state(*file_name*, *epoch*, *weights_only=False*, ***kwargs*)
saves the current state via *delira.io.torch.save_checkpoint()*

Parameters

- **file_name** (*str*) – filename to save the state to
- **epoch** (*int*) – current epoch (will be saved for mapping back)
- **weights_only** (*bool*) – whether to store only weights (default: False)
- ***args** – positional arguments
- ****kwargs** – keyword arguments

train(*num_epochs*, *datamgr_train*, *datamgr_valid=None*, *val_score_key=None*, *val_score_mode='highest'*)
train network

Parameters

- **num_epochs** (*int*) – number of epochs to train
- **datamgr_train** (BaseDataManager) – Data Manager to create Batch Generator for training
- **datamgr_valid** (BaseDataManager) – Data Manager to create Batch Generator for validation
- **val_score_key** (*str*) – Key of validation metric; must be key in self.metrics
- **val_score_mode** (*str*) – String to specify whether a higher or lower validation score is optimal; must be in ['highest', 'lowest']

Returns Best model (if *val_score_key* is not a valid key the model of the last epoch will be returned)

Return type AbstractPyTorchNetwork

update_state(*file_name*, **args*, ***kwargs*)
Update internal state from a loaded state

Parameters

- **file_name** (*str*) – file containing the new state to load
- ***args** – positional arguments
- ****kwargs** – keyword arguments

Returns the trainer with a modified state

Return type *AbstractNetworkTrainer*

TfNetworkTrainer

class TfNetworkTrainer(*network*, *save_path*, *losses: dict*, *optimizer_cls*, *optimizer_params={}*, *metrics={}*, *lr_scheduler_cls=None*, *lr_scheduler_params={}*, *gpu_ids=[]*, *save_freq=1*, *optim_fn=<function create_optims_default_tf>*, *fold=0*, *callbacks=[]*, *start_epoch=1*, ***kwargs*)
Bases: delira.training.abstract_trainer.AbstractNetworkTrainer

Train and Validate a Network

See also:

AbstractNetwork

_at_epoch_begin(*metrics_val*, *val_score_key*, *epoch*, *num_epochs*, ***kwargs*)
Defines behaviour at beginning of each epoch: Executes all callbacks's *at_epoch_begin* method

Parameters

- **metrics_val** (*dict*) – validation metrics
- **val_score_key** (*str*) – validation score key
- **epoch** (*int*) – current epoch
- **num_epochs** (*int*) – total number of epochs

- ****kwargs** – keyword arguments

_at_epoch_end(*metrics_val*, *val_score_key*, *epoch*, *is_best*, ***kwargs*)
Defines behaviour at beginning of each epoch: Executes all callbacks's *at_epoch_end* method and saves current state if necessary

Parameters

- **metrics_val** (*dict*) – validation metrics
- **val_score_key** (*str*) – validation score key
- **epoch** (*int*) – current epoch
- **num_epochs** (*int*) – total number of epochs
- ****kwargs** – keyword arguments

_at_training_begin(**args*, ***kwargs*)
Defines behaviour at beginning of training

Parameters

- ***args** – positional arguments
- ****kwargs** – keyword arguments

_at_training_end()
Defines Behaviour at end of training: Loads best model if available

Returns best network

Return type AbstractTfNetwork

static _is_better_val_scores(*old_val_score*, *new_val_score*, *mode='highest'*)
Check whether the new val score is better than the old one with respect to the optimization goal

Parameters

- **old_val_score** – old validation score
- **new_val_score** – new validation score
- **mode** (*str*) – String to specify whether a higher or lower validation score is optimal; must be in ['highest', 'lowest']

Returns True if new score is better, False otherwise

Return type bool

_setup(*network*, *optim_fn*, *optimizer_cls*, *optimizer_params*, *lr_scheduler_cls*, *lr_scheduler_params*, *gpu_ids*)
Defines the Trainers Setup

Parameters

- **network** (instance of :class: *AbstractTfNetwork*) – the network to train
- **optim_fn** (*function*) – creates a dictionary containing all necessary optimizers
- **optimizer_cls** (*subclass of tf.train.Optimizer*) – optimizer class implementing the optimization algorithm of choice
- **optimizer_params** (*dict*) –
- **lr_scheduler_cls** (*Any*) – learning rate schedule class: must implement step() method

- **lr_scheduler_params** (*dict*) – keyword arguments passed to lr scheduler during construction
- **gpu_ids** (*list*) – list containing ids of GPUs to use; if empty: use cpu instead

_train_single_epoch(*batchgen: <sphinx.ext.autodoc.importer._MockObject object at 0x7f43be795978>*, *epoch*)
Trains the network a single epoch

Parameters

- **batchgen** (*MultiThreadedAugmenter*) – Generator yielding the training batches
- **epoch** (*int*) – current epoch

_update_state(*new_state*)
Update the state from a given new state

Parameters **new_state** (*dict*) – new state to update internal state from

Returns the trainer with a modified state

Return type *AbstractNetworkTrainer*

fold
Get current fold

Returns current fold

Return type int

load_state(*file_name*)
Loads the new state from file via *delira.io.tf.load_checkpoint()*

Parameters **file_name** (*str*) – the file to load the state from

predict(*batchgen*, *batch_size=None*)
Returns predictions from network for batches from batchgen

Parameters

- **batchgen** (*MultiThreadedAugmenter*) – Generator yielding the batches to predict
- **batch_size** (*None or int*) – if int: collect batches until batch_size is reached and forward them together

Returns

- *np.ndarray* – predictions from batches
- *list of np.ndarray* – labels from batches
- *dict* – dictionary containing the mean validation metrics and the mean loss values

register_callback(*callback: delira.training.callbacks.abstract_callback.AbstractCallback*)
Register Callback to Trainer

Parameters **callback** (AbstractCallback) – the callback to register

Raises AssertionError – *callback* is not an instance of AbstractCallback and has not both methods [‘at_epoch_begin’, ‘at_epoch_end’]

save_state(*file_name*)
saves the current state via *delira.io.tf.save_checkpoint()*

Parameters **file_name** (*str*) – filename to save the state to

train(*num_epochs*, *datamgr_train*, *datamgr_valid=None*, *val_score_key=None*, *val_score_mode='highest'*)
train network

Parameters

- **num_epochs** (*int*) – number of epochs to train
- **datamgr_train** (BaseDataManager) – Data Manager to create Batch Generator for training
- **datamgr_valid** (BaseDataManager) – Data Manager to create Batch Generator for validation
- **val_score_key** (*str*) – Key of validation metric; must be key in self.metrics
- **val_score_mode** (*str*) – String to specify whether a higher or lower validation score is optimal; must be in ['highest', 'lowest']

Returns Best model (if *val_score_key* is not a valid key the model of the last epoch will be returned)

Return type AbstractTfNetwork

update_state(*file_name*, **args*, ***kwargs*)
Update internal state from a loaded state

Parameters

- **file_name** (*str*) – file containing the new state to load
- ***args** – positional arguments
- ****kwargs** – keyword arguments

Returns the trainer with a modified state

Return type *AbstractNetworkTrainer*

Experiments

Experiments are the outermost class to control your training, it wraps your NetworkTrainer and provides utilities for cross-validation.

AbstractExperiment

class AbstractExperiment(*n_epochs*, **args*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

Abstract Class Representing a single Experiment (must be subclassed for each Backend)

See also:

PyTorchExperiment

kfold(*num_epochs: int*, *data: delira.data_loading.data_manager.BaseDataManager*, *num_splits=None*, *shuffle=False*, *random_seed=None*, *train_kwargs={}*, *test_kwargs={}*, ***kwargs*)
Runs K-Fold Crossvalidation The supported scenario is:

- passing a single datamanager: the data within the single manager

will be split and multiple datamanagers will be created holding the subsets.

Parameters

- **num_epochs** (*int*) – number of epochs to train the model
- **data** (single BaseDataManager) – single datamanager (will be split for crossvalidation)
- **num_splits** (*None or int*) – number of splits for kfold if None: 10 splits will be validated per default
- **shuffle** (*bool*) – whether or not to shuffle indices for kfold
- **random_seed** (*None or int*) – random seed used to seed the kfold (if shuffle is true), pytorch and numpy
- **train_kwargs** (*dict*) – keyword arguments to specify training behavior
- **test_kwargs** (*dict*) – keyword arguments to specify testing behavior
- ****kwargs** – additional keyword arguments (completely passed to self.run())

See also:

:method:'BaseDataManager.update_state_from_dict' train_kwargs and test_kwargs

static load(*file_name*)
Loads whole experiment

Parameters **file_name** (*str*) – file_name to load the experiment from

Raises NotImplementedError – if not overwritten in subclass

run(*train_data: delira.data_loading.data_manager.BaseDataManager, val_data: Optional[delira.data_loading.data_manager.BaseDataManager] = None, params: Optional[delira.training.parameters.Parameters] = None, **kwargs*)
trains single model

Parameters

- **train_data** (BaseDataManager) – data manager containing the training data
- **val_data** (BaseDataManager) – data manager containing the validation data
- **parameters** (*Parameters*, optional) – Class containing all parameters (defaults to None). If not specified, the parameters fall back to the ones given during class initialization

Raises NotImplementedError – If not overwritten in subclass

save()
Saves the Whole experiments

Raises NotImplementedError – If not overwritten in subclass

setup(**args*, ***kwargs*)
Abstract Method to setup a *AbstractNetworkTrainer*

Raises NotImplementedError – if not overwritten by subclass

stratified_kfold(*num_epochs: int, data: delira.data_loading.data_manager.BaseDataManager, num_splits=None, shuffle=False, random_seed=None, label_key='label', train_kwargs={}, test_kwargs={}, **kwargs*)
Runs stratified K-Fold Crossvalidation The supported supported scenario is:

- passing a single datamanager: the data within the single manager will be split and multiple datamanagers will be created holding the subsets.

Parameters

- **num_epochs** (*int*) – number of epochs to train the model
- **data** (BaseDataManager) – single datamanager (will be split for crossvalidation)
- **num_splits** (*None or int*) – number of splits for kfold if None: 10 splits will be validated
- **shuffle** (*bool*) – whether or not to shuffle indices for kfold
- **random_seed** (*None or int*) – random seed used to seed the kfold (if shuffle is true), pytorch and numpy
- **label_key** (*str (default: "label")*) – the key to extract the label for stratification from each data sample
- **train_kwargs** (*dict*) – keyword arguments to specify training behavior
- **test_kwargs** (*dict*) – keyword arguments to specify testing behavior
- ****kwargs** – additional keyword arguments (completely passed to self.run())

See also:

:method:'BaseDataManager.update_state_from_dict' train_kwargs and test_kwargs

test(*params: delira.training.parameters.Parameters*, *network: delira.models.abstract_network.AbstractNetwork*, *datamgr_test: delira.data_loading.data_manager.BaseDataManager*, *trainer_cls=<class 'delira.training.abstract_trainer.AbstractNetworkTrainer'>*, ***kwargs*)

Executes prediction for all items in datamgr_test with network

Parameters

- **params** (*Parameters*) – the parameters to construct a model
- **network** (*:class:'AbstractNetwork'*) – the network to train
- **datamgr_test** (*:class:'BaseDataManager'*) – holds the test data
- **trainer_cls** – class defining the actual trainer, defaults to *AbstractNetworkTrainer*, which should be suitable for most cases, but can easily be overwritten and exchanged if necessary
- ****kwargs** – holds additional keyword arguments (which are completly passed to the trainers init)

Returns

- *np.ndarray* – predictions from batches
- *list of np.ndarray* – labels from batches
- *dict* – dictionary containing the mean validation metrics and the mean loss values

PyTorchExperiment

class PyTorchExperiment(*params: delira.training.parameters.Parameters*, *model_cls: delira.models.abstract_network.AbstractPyTorchNetwork*, *name=None*, *save_path=None*, *val_score_key=None*, *optim_builder=<function create_optims_default_pytorch>*, *checkpoint_freq=1*, *trainer_cls=<class 'delira.training.pytorch_trainer.PyTorchNetworkTrainer'>*, ***kwargs*)

Bases: delira.training.experiment.AbstractExperiment

Single Experiment for PyTorch Backend

See also:

AbstractExperiment

kfold(*num_epochs: int, data: Union[List[delira.data_loading.data_manager.BaseDataManager], delira.data_loading.data_manager.BaseDataManager], num_splits=None, shuffle=False, random_seed=None, train_kwargs={}, test_kwargs={}, **kwargs*)

Runs K-Fold Crossvalidation The supported scenario is:

- passing a single datamanager: the data within the single manager

will be split and multiple datamanagers will be created holding the subsets.

Parameters

- **num_epochs** (*int*) – number of epochs to train the model
- **data** (single BaseDataManager) – single datamanager (will be split for crossvalidation)
- **num_splits** (*None or int*) – number of splits for kfold if None: 10 splits will be validated per default
- **shuffle** (*bool*) – whether or not to shuffle indices for kfold
- **random_seed** (*None or int*) – random seed used to seed the kfold (if shuffle is true), pytorch and numpy
- **train_kwargs** (*dict*) – keyword arguments to specify training behavior
- **test_kwargs** (*dict*) – keyword arguments to specify testing behavior
- ****kwargs** – additional keyword arguments (completely passed to self.run())

See also:

:method:‘BaseDataManager.update_state_from_dict‘ train_kwargs and test_kwargs

static load(*file_name*)

Loads whole experiment

Parameters **file_name** (*str*) – file_name to load the experiment from

run(*train_data: delira.data_loading.data_manager.BaseDataManager, val_data: Optional[delira.data_loading.data_manager.BaseDataManager], params: Optional[delira.training.parameters.Parameters] = None, **kwargs*)

trains single model

Parameters

- **train_data** (BaseDataManager) – holds the trainset
- **val_data** (BaseDataManager *or None*) – holds the validation set (if None: Model will not be validated)
- **params** (*Parameters*) – the parameters to construct a model and network trainer
- ****kwargs** – holds additional keyword arguments (which are completly passed to the trainers init)

Returns trainer of trained network

Return type *AbstractNetworkTrainer*

Raises `ValueError` – Class has no Attribute `params` and no parameters were given as function argument

save()
Saves the Whole experiments

setup(*params: delira.training.parameters.Parameters*, ***kwargs*)
Perform setup of Network Trainer

Parameters

- **params** (*Parameters*) – the parameters to construct a model and network trainer
- ****kwargs** – keyword arguments

stratified_kfold(*num_epochs: int*, *data: delira.data_loading.data_manager.BaseDataManager*, *num_splits=None*, *shuffle=False*, *random_seed=None*, *label_key='label'*, *train_kwargs={}*, *test_kwargs={}*, ***kwargs*)
Runs stratified K-Fold Crossvalidation The supported supported scenario is:

- passing a single datamanager: the data within the single manager will be split and multiple datamanagers will be created holding the subsets.

Parameters

- **num_epochs** (*int*) – number of epochs to train the model
- **data** (`BaseDataManager`) – single datamanager (will be split for crossvalidation)
- **num_splits** (*None or int*) – number of splits for kfold if None: 10 splits will be validated
- **shuffle** (*bool*) – whether or not to shuffle indices for kfold
- **random_seed** (*None or int*) – random seed used to seed the kfold (if shuffle is true), pytorch and numpy
- **label_key** (*str (default: "label")*) – the key to extract the label for stratification from each data sample
- **train_kwargs** (*dict*) – keyword arguments to specify training behavior
- **test_kwargs** (*dict*) – keyword arguments to specify testing behavior
- ****kwargs** – additional keyword arguments (completely passed to self.run())

See also:

:method:'BaseDataManager.update_state_from_dict' `train_kwargs` and `test_kwargs`

test(*params: delira.training.parameters.Parameters*, *network: delira.models.abstract_network.AbstractPyTorchNetwork*, *datamgr_test: delira.data_loading.data_manager.BaseDataManager*, ***kwargs*)
Executes prediction for all items in datamgr_test with network

Parameters

- **params** (*Parameters*) – the parameters to construct a model
- **network** (*:class:'AbstractPyTorchNetwork'*) – the network to train
- **datamgr_test** (*:class:'BaseDataManager'*) – holds the test data
- ****kwargs** – holds additional keyword arguments (which are completly passed to the trainers init)

Returns

- *np.ndarray* – predictions from batches
- *list of np.ndarray* – labels from batches
- *dict* – dictionary containing the mean validation metrics and the mean loss values

TfExperiment

class TfExperiment(*params: Union[delira.training.parameters.Parameters, str], model_cls: delira.models.abstract_network.AbstractTfNetwork, name=None, save_path=None, val_score_key=None, optim_builder=<function create_optims_default_tf>, checkpoint_freq=1, trainer_cls=<class 'delira.training.tf_trainer.TfNetworkTrainer'>, **kwargs*)

Bases: `delira.training.experiment.AbstractExperiment`

Single Experiment for Tf Backend

See also:

AbstractExperiment

kfold(*num_epochs: int, data: Union[List[delira.data_loading.data_manager.BaseDataManager], delira.data_loading.data_manager.BaseDataManager], num_splits=None, shuffle=False, random_seed=None, train_kwargs={}, test_kwargs={}, **kwargs*)

Runs K-Fold Crossvalidation The supported scenario is:

- passing a single datamanager: the data within the single manager

will be split and multiple datamanagers will be created holding the subsets.

Parameters

- **num_epochs** (*int*) – number of epochs to train the model
- **data** (single `BaseDataManager`) – single datamanager (will be split for crossvalidation)
- **num_splits** (*None or int*) – number of splits for kfold if None: 10 splits will be validated per default
- **shuffle** (*bool*) – whether or not to shuffle indices for kfold
- **random_seed** (*None or int*) – random seed used to seed the kfold (if shuffle is true), pytorch and numpy
- **train_kwargs** (*dict*) – keyword arguments to specify training behavior
- **test_kwargs** (*dict*) – keyword arguments to specify testing behavior
- ****kwargs** – additional keyword arguments (completely passed to self.run())

See also:

:method:‘BaseDataManager.update_state_from_dict‘ `train_kwargs` and `test_kwargs`

static load(*file_name*)

Loads whole experiment

Parameters **file_name** (*str*) – file_name to load the experiment from

run(*train_data: delira.data_loading.data_manager.BaseDataManager, val_data: Optional[delira.data_loading.data_manager.BaseDataManager], params: Optional[delira.training.parameters.Parameters] = None, **kwargs*)
trains single model

Parameters

- **train_data** (BaseDataManager) – holds the trainset
- **val_data** (BaseDataManager *or* None) – holds the validation set (if None: Model will not be validated)
- **params** (*Parameters*) – the parameters to construct a model and network trainer
- ****kwargs** – holds additional keyword arguments (which are completly passed to the trainers init)

Returns trainer of trained network

Return type *AbstractNetworkTrainer*

Raises ValueError – Class has no Attribute params and no parameters were given as function argument

save()
Saves the Whole experiments

setup(*params: delira.training.parameters.Parameters*, ***kwargs*)
Perform setup of Network Trainer

Parameters

- **params** (*Parameters*) – the parameters to construct a model and network trainer
- ****kwargs** – keyword arguments

stratified_kfold(*num_epochs: int*, *data: delira.data_loading.data_manager.BaseDataManager*, *num_splits=None*, *shuffle=False*, *random_seed=None*, *label_key='label'*, *train_kwargs={}*, *test_kwargs={}*, ***kwargs*)
Runs stratified K-Fold Crossvalidation The supported supported scenario is:

- passing a single datamanager: the data within the single manager will be split and multiple datamanagers will be created holding the subsets.

Parameters

- **num_epochs** (*int*) – number of epochs to train the model
- **data** (BaseDataManager) – single datamanager (will be split for crossvalidation)
- **num_splits** (*None or int*) – number of splits for kfold if None: 10 splits will be validated
- **shuffle** (*bool*) – whether or not to shuffle indices for kfold
- **random_seed** (*None or int*) – random seed used to seed the kfold (if shuffle is true), pytorch and numpy
- **label_key** (*str (default: "label")*) – the key to extract the label for stratification from each data sample
- **train_kwargs** (*dict*) – keyword arguments to specify training behavior
- **test_kwargs** (*dict*) – keyword arguments to specify testing behavior
- ****kwargs** – additional keyword arguments (completely passed to self.run())

See also:

:method:‘BaseDataManager.update_state_from_dict‘ `train_kwargs` and `test_kwargs`

test (*params: delira.training.parameters.Parameters*, *network: delira.models.abstract_network.AbstractNetwork*, *datamgr_test: delira.data_loading.data_manager.BaseDataManager*, *trainer_cls=<class 'delira.training.abstract_trainer.AbstractNetworkTrainer'>*, ***kwargs*)
Executes prediction for all items in datamgr_test with network

Parameters

- **params** (*Parameters*) – the parameters to construct a model
- **network** (*:class:'AbstractNetwork'*) – the network to train
- **datamgr_test** (*:class:'BaseDataManager'*) – holds the test data
- **trainer_cls** – class defining the actual trainer, defaults to *AbstractNetworkTrainer*, which should be suitable for most cases, but can easily be overwritten and exchanged if necessary
- ****kwargs** – holds additional keyword arguments (which are completly passed to the trainers init)

Returns

- *np.ndarray* – predictions from batches
- *list of np.ndarray* – labels from batches
- *dict* – dictionary containing the mean validation metrics and the mean loss values

Callbacks

Callbacks are essential to provide a uniform API for tasks like early stopping etc. The PyTorch learning rate schedulers are also implemented as callbacks. Every callback should ber derived from *AbstractCallback* and must provide the methods `at_epoch_begin` and `at_epoch_end`.

AbstractCallback

class AbstractCallback (**args*, ***kwargs*)
Bases: `object`

Implements abstract callback interface. All callbacks should be derived from this class

See also:

`AbstractNetworkTrainer`

at_epoch_begin (*trainer*, ***kwargs*)
Function which will be executed at begin of each epoch

Parameters

- **trainer** (`AbstractNetworkTrainer`) –
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type `AbstractNetworkTrainer`

at_epoch_end(*trainer*, ***kwargs*)
Function which will be executed at end of each epoch

Parameters

- **trainer** (AbstractNetworkTrainer) –
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type AbstractNetworkTrainer

EarlyStopping

class EarlyStopping(*monitor_key*, *min_delta=0*, *patience=0*, *mode='min'*)
Bases: delira.training.callbacks.abstract_callback.AbstractCallback

Implements Early Stopping as callback

See also:

AbstractCallback

_is_better(*metric*)
Helper function to decide whether the current metric is better than the best metric so far

Parameters **metric** – current metric value

Returns whether this metric is the new best metric or not

Return type bool

at_epoch_begin(*trainer*, ***kwargs*)
Function which will be executed at begin of each epoch

Parameters

- **trainer** (AbstractNetworkTrainer) –
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type AbstractNetworkTrainer

at_epoch_end(*trainer*, ***kwargs*)
Actual early stopping: Checks at end of each epoch if monitored metric is new best and if it hasn't improved over *self.patience* epochs, the training will be stopped

Parameters

- **trainer** (AbstractNetworkTrainer) – the trainer whose arguments can be modified
- ****kwargs** – additional keyword arguments

Returns trainer with modified attributes

Return type AbstractNetworkTrainer

DefaultPyTorchSchedulerCallback

class DefaultPyTorchSchedulerCallback(**args*, ***kwargs*)

Bases: delira.training.callbacks.abstract_callback.AbstractCallback

Implements a Callback, which *at_epoch_end* function is suitable for most schedulers

at_epoch_begin(*trainer*, ***kwargs*)

Function which will be executed at begin of each epoch

Parameters

- **trainer** (AbstractNetworkTrainer) –
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type AbstractNetworkTrainer

at_epoch_end(*trainer*, ***kwargs*)

Executes a single scheduling step

Parameters

- **trainer** (PyTorchNetworkTrainer) – the trainer class, which can be changed
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type PyTorchNetworkTrainer

CosineAnnealingLRCallback

class CosineAnnealingLRCallback(*optimizer*, *T_max*, *eta_min=0*, *last_epoch=-1*)

Bases: delira.training.callbacks.pytorch_schedulers.DefaultPyTorchSchedulerCallback

Wraps PyTorch's *CosineAnnealingLR* Scheduler as callback

at_epoch_begin(*trainer*, ***kwargs*)

Function which will be executed at begin of each epoch

Parameters

- **trainer** (AbstractNetworkTrainer) –
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type AbstractNetworkTrainer

at_epoch_end(*trainer*, ***kwargs*)

Executes a single scheduling step

Parameters

- **trainer** (PyTorchNetworkTrainer) – the trainer class, which can be changed
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type PyTorchNetworkTrainer

ExponentialLRCallback

class ExponentialLRCallback(*optimizer*, *gamma*, *last_epoch=-1*)
Bases: `delira.training.callbacks.pytorch_schedulers.DefaultPyTorchSchedulerCallback`

Wraps PyTorch's *ExponentialLR* Scheduler as callback

at_epoch_begin(*trainer*, ***kwargs*)
Function which will be executed at begin of each epoch

Parameters

- **trainer** (`AbstractNetworkTrainer`) –
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type `AbstractNetworkTrainer`

at_epoch_end(*trainer*, ***kwargs*)
Executes a single scheduling step

Parameters

- **trainer** (`PyTorchNetworkTrainer`) – the trainer class, which can be changed
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type `PyTorchNetworkTrainer`

LambdaLRCallback

class LambdaLRCallback(*optimizer*, *lr_lambda*, *last_epoch=-1*)
Bases: `delira.training.callbacks.pytorch_schedulers.DefaultPyTorchSchedulerCallback`

Wraps PyTorch's *LambdaLR* Scheduler as callback

at_epoch_begin(*trainer*, ***kwargs*)
Function which will be executed at begin of each epoch

Parameters

- **trainer** (`AbstractNetworkTrainer`) –
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type `AbstractNetworkTrainer`

at_epoch_end(*trainer*, ***kwargs*)
Executes a single scheduling step

Parameters

- **trainer** (`PyTorchNetworkTrainer`) – the trainer class, which can be changed
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type `PyTorchNetworkTrainer`

MultiStepLRCallback

class MultiStepLRCallback(*optimizer*, *milestones*, *gamma=0.1*, *last_epoch=-1*)
Bases: delira.training.callbacks.pytorch_schedulers.DefaultPyTorchSchedulerCallback

Wraps PyTorch's *MultiStepLR* Scheduler as callback

at_epoch_begin(*trainer*, ***kwargs*)
Function which will be executed at begin of each epoch

Parameters

- **trainer** (AbstractNetworkTrainer) –
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type AbstractNetworkTrainer

at_epoch_end(*trainer*, ***kwargs*)
Executes a single scheduling step

Parameters

- **trainer** (PyTorchNetworkTrainer) – the trainer class, which can be changed
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type PyTorchNetworkTrainer

ReduceLROnPlateauCallback

class ReduceLROnPlateauCallback(*optimizer*, *mode='min'*, *factor=0.1*, *patience=10*, *verbose=False*, *threshold=0.0001*, *threshold_mode='rel'*, *cooldown=0*, *min_lr=0*, *eps=1e-08*)
Bases: delira.training.callbacks.pytorch_schedulers.DefaultPyTorchSchedulerCallback

Wraps PyTorch's *ReduceLROnPlateau* Scheduler as Callback

at_epoch_begin(*trainer*, ***kwargs*)
Function which will be executed at begin of each epoch

Parameters

- **trainer** (AbstractNetworkTrainer) –
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type AbstractNetworkTrainer

at_epoch_end(*trainer*, ***kwargs*)
Executes a single scheduling step

Parameters

- **trainer** (PyTorchNetworkTrainer) – the trainer class, which can be changed
- **kwargs** – additional keyword arguments

Returns modified trainer

Return type PyTorchNetworkTrainer

StepLRCallback

class StepLRCallback(*optimizer*, *step_size*, *gamma=0.1*, *last_epoch=-1*)

Bases: delira.training.callbacks.pytorch_schedulers.DefaultPyTorchSchedulerCallback

Wraps PyTorch's *StepLR* Scheduler as callback

at_epoch_begin(*trainer*, ***kwargs*)

Function which will be executed at begin of each epoch

Parameters

- **trainer** (AbstractNetworkTrainer) –
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type AbstractNetworkTrainer

at_epoch_end(*trainer*, ***kwargs*)

Executes a single scheduling step

Parameters

- **trainer** (PyTorchNetworkTrainer) – the trainer class, which can be changed
- ****kwargs** – additional keyword arguments

Returns modified trainer

Return type PyTorchNetworkTrainer

CosineAnnealingLRCallbackPyTorch

CosineAnnealingLRCallbackPyTorch

alias of *delira.training.callbacks.pytorch_schedulers.CosineAnnealingLRCallback*

ExponentialLRCallbackPyTorch

ExponentialLRCallbackPyTorch

alias of *delira.training.callbacks.pytorch_schedulers.ExponentialLRCallback*

LambdaLRCallbackPyTorch

LambdaLRCallbackPyTorch

alias of *delira.training.callbacks.pytorch_schedulers.LambdaLRCallback*

MultiStepLRCallbackPyTorch

MultiStepLRCallbackPyTorch

alias of *delira.training.callbacks.pytorch_schedulers.MultiStepLRCallback*

ReduceLROnPlateauCallbackPyTorch

ReduceLROnPlateauCallbackPyTorch
alias of *delira.training.callbacks.pytorch_schedulers.ReduceLROnPlateauCallback*

StepLRCallbackPyTorch

StepLRCallbackPyTorch
alias of *delira.training.callbacks.pytorch_schedulers.StepLRCallback*

Custom Loss Functions

BCEFocalLossPyTorch

class BCEFocalLossPyTorch(*alpha=None*, *gamma=2*, *reduction='elementwise_mean'*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

Focal loss for binary case without(!) logit

forward(*p*, *t*)

BCEFocalLossLogitPyTorch

class BCEFocalLossLogitPyTorch(*alpha=None*, *gamma=2*, *reduction='elementwise_mean'*)
Bases: `sphinx.ext.autodoc.importer._MockObject`

Focal loss for binary case WITH logit

forward(*p*, *t*)

AurocMetricPyTorch

class AurocMetricPyTorch
Metric to Calculate AuROC

Deprecated since version 0.1: *AurocMetricPyTorch* will be removed in next release and is deprecated in favor of `trixi.logging` Modules

> **Warning:** *AurocMetricPyTorch* will be removed in next release

forward(*outputs: <sphinx.ext.autodoc.importer._MockObject object at 0x7f43be4fefd0>*, *targets: <sphinx.ext.autodoc.importer._MockObject object at 0x7f43be4fe668>*)
Actual AuROC calculation

Parameters

- **outputs** (*torch.Tensor*) – predictions from network
- **targets** (*torch.Tensor*) – training targets

Returns auroc value

Return type torch.Tensor

AccuracyMetricPyTorch

class AccuracyMetricPyTorch
Metric to Calculate Accuracy

Deprecated since version 0.1: *AccuracyMetricPyTorch* will be removed in next release and is deprecated in favor of `trixi.logging` Modules

> **Warning:** class:*AccuracyMetricPyTorch* will be removed in next release

forward(*outputs: <sphinx.ext.autodoc.importer._MockObject object at 0x7f43be500c88>, targets: <sphinx.ext.autodoc.importer._MockObject object at 0x7f43be7952e8>*)
Actual accuracy calcuation

Parameters

- **outputs** (*torch.Tensor*) – predictions from network
- **targets** (*torch.Tensor*) – training targets

Returns accuracy value

Return type torch.Tensor

pytorch_batch_to_numpy

pytorch_batch_to_numpy(*tensor: <sphinx.ext.autodoc.importer._MockObject object at 0x7f43be1b92b0>*)
Utility Function to cast a whole PyTorch batch to numpy

Parameters **tensor** (*torch.Tensor*) – the batch to convert

Returns the converted batch

Return type np.ndarray

pytorch_tensor_to_numpy

pytorch_tensor_to_numpy(*tensor: <sphinx.ext.autodoc.importer._MockObject object at 0x7f43be1b9588>*)
Utility Function to cast a single PyTorch Tensor to numpy

Parameters **tensor** (*torch.Tensor*) – the tensor to convert

Returns the converted tensor

Return type np.ndarray

float_to_pytorch_tensor

float_to_pytorch_tensor(*f: float*)
Converts a single float to a PyTorch Float-Tensor

Parameters **f** (*float*) – float to convert

Returns converted float

Return type torch.Tensor

create_optims_default_pytorch

create_optims_default_pytorch(*model*, *optim_cls*, ***optim_params*)
Function to create a optimizer dictionary (in this case only one optimizer for the whole network)

Parameters

- **model** (AbstractPyTorchNetwork) – model whose parameters should be updated by the optimizer
- **optim_cls** – Class implementing an optimization algorithm
- ****optim_params** – Additional keyword arguments (passed to the optimizer class

Returns dictionary containing all created optimizers

Return type dict

create_optims_gan_default_pytorch

create_optims_gan_default_pytorch(*model*, *optim_cls*, ***optim_params*)
Function to create a optimizer dictionary (in this case two optimizers: One for the generator, one for the discriminator)

Parameters

- **model** (AbstractPyTorchNetwork) – model whose parameters should be updated by the optimizer
- **optim_cls** – Class implementing an optimization algorithm
- **optim_params** – Additional keyword arguments (passed to the optimizer class

Returns dictionary containing all created optimizers

Return type dict

create_optims_default_tf

create_optims_default_tf(*optim_cls*, ***optim_params*)
Function to create a optimizer dictionary (in this case only one optimizer)

Parameters

- **optim_cls** – Class implementing an optimization algorithm
- ****optim_params** – Additional keyword arguments (passed to the optimizer class)

Returns dictionary containing all created optimizers

Return type dict

7.1.6 Utils

This package provides utility functions as image operations, various decorators, path operations and time operations.

classtype_func(*class_object*)
Decorator to Check whether the first argument of the decorated function is a subclass of a certain type

Parameters **class_object** (*Any*) – type the first function argument should be subclassed from

Returns

Return type Wrapped Function

Raises `AssertionError` – First argument of decorated function is not a subclass of given type

dtype_func(*class_object*)
Decorator to Check whether the first argument of the decorated function is of a certain type

Parameters **class_object** (*Any*) – type the first function argument should have

Returns

Return type Wrapped Function

Raises `AssertionError` – First argument of decorated function is not of given type

make_deprecated(*new_func*)
Decorator which raises a DeprecationWarning for the decorated object

Parameters **new_func** (*Any*) – new function which should be used instead of the decorated one

Returns

Return type Wrapped Function

Raises Deprecation Warning

numpy_array_func(*func*)

torch_module_func(*func*)

torch_tensor_func(*func*)

bounding_box(*mask*, *margin=None*)
Calculate bounding box coordinates of binary mask

Parameters

- **mask** (*SimpleITK.Image*) – Binary mask
- **margin** (*int, default: None*) – margin to be added to min/max on each dimension

Returns bounding box coordinates of the form (xmin, xmax, ymin, ymax, zmin, zmax)

Return type tuple

calculate_origin_offset(*new_spacing*, *old_spacing*)
Calculates the origin offset of two spacings

Parameters

- **new_spacing** (*list or np.ndarray or tuple*) – new spacing
- **old_spacing** (*list or np.ndarray or tuple*) – old spacing

Returns origin offset

Return type np.ndarray

max_energy_slice(*img*)
Determine the axial slice in which the image energy is max

Parameters **img** (*SimpleITK.Image*) – given image

Returns slice index

Return type int

sitk_copy_metadata(*img_source*, *img_target*)
Copy metadata (=DICOM Tags) from one image to another

Parameters

- **img_source** (*SimpleITK.Image*) – Source image
- **img_target** (*SimpleITK.Image*) – Source image

Returns Target image with copied metadata

Return type SimpleITK.Image

sitk_new_blank_image(*size*, *spacing*, *direction*, *origin*, *default_value=0.0*)
Create a new blank image with given properties

Parameters

- **size** (*list or np.ndarray or tuple*) – new image size
- **spacing** (*list or np.ndarray or tuple*) – spacing of new image
- **direction** – new image's direction
- **origin** – new image's origin
- **default_value** (*float*) – new image's default value

Returns Blank image with given properties

Return type SimpleITK.Image

sitk_resample_to_image(*image*, *reference_image*, *default_value=0.0*, *interpolator=<sphinx.ext.autodoc.importer._MockObject object>*, *transform=None*, *output_pixel_type=None*)
Resamples Image to reference image

Parameters

- **image** (*SimpleITK.Image*) – the image which should be resampled
- **reference_image** (*SimpleITK.Image*) – the resampling target
- **default_value** (*float*) – default value
- **interpolator** (*Any*) – implements the actual interpolation
- **transform** (*Any (default: None)*) – transformation
- **output_pixel_type** (*Any (default:None)*) – type of output pixels

Returns resampled image

Return type SimpleITK.Image

sitk_resample_to_shape(*img*, *x*, *y*, *z*, *order=1*)
Resamples Image to given shape

Parameters

- **img** (*SimpleITK.Image*) –
- **x** (*int*) – shape in x-direction
- **y** (*int*) – shape in y-direction
- **z** (*int*) – shape in z-direction
- **order** (*int*) – interpolation order

Returns Resampled Image

Return type SimpleITK.Image

sitk_resample_to_spacing(*image*, *new_spacing=(1.0, 1.0, 1.0)*, *interpolator=<sphinx.ext.autodoc.importer._MockObject object>*, *default_value=0.0*)

Resamples SITK Image to a given spacing

Parameters

- **image** (*SimpleITK.Image*) – image which should be resampled
- **new_spacing** (*list or np.ndarray or tuple*) – target spacing
- **interpolator** (*Any*) – implements the actual interpolation
- **default_value** (*float*) – default value

Returns resampled Image with target spacing

Return type SimpleITK.Image

subdirs(*d*)

For a given directory, return a list of all subdirectories (full paths)

Parameters **d** (*string*) – given root directory

Returns list of strings of all subdirectories

Return type list

now()

Return current time as YYYY-MM-DD_HH-MM-SS

Returns current time

Return type string

class LookupConfig(**args*, ***kwargs*)

Bases: sphinx.ext.autodoc.importer._MockObject

Helper class to have nested lookups in all subdicts of Config

nested_get(*key*, **args*, ***kwargs*)

Returns all occurances of `key` in `self` and subdicts

Parameters

- **key** (*str*) – the key to search for
- ***args** – positional arguments to provide default value
- ****kwargs** – keyword arguments to provide default value

Raises KeyError – Multiple Values are found for key (unclear which value should be returned) OR No Value was found for key and no default value was given

Returns value corresponding to key (or default if value was not found)

Return type Any

7.1.7 Class Hierarchy Diagrams

Contents

Data Loading

- Coarse
- Fine

Sampler

- Coarse
- Fine

Logging

- Coarse
- Fine

Models

- Coarse
- Fine

Training

- Coarse
- Fine

Hyperparameters

- Coarse
- Fine

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

P

R

S

T

U

V

W