delira Documentation

Release 0.4.0

Justus Schock, Oliver Rippel, Christoph Haarburger

Jun 16, 2019

GETTING STARTED

CHAPTER

ONE

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 `tensorflow`

  Note: the `tensorflow` 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	Bin ary Ins tal lat ion	Source Installation	Notes
None	‘‘p ip ins tal l d eli ra‘ ‘	`pip install git+ https:/ /github.com / justusschock/deli ra.git`	Training not possible if backend is not installed separately
torch _	‘‘p ip ins tal l d eli ra[tor ch] ‘‘	‘‘git clone https: //github.com/justu ss- chock/delira.git && cd delira && p ip install .[torch]‘‘	`delira` with `torch` backend supports mixed-precision training via NVIDIA/apex (must be installed separately).
‘tensor flow‘_	‘‘p ip ins tal l d eli ra[ten sor flo w]‘ ‘	‘‘git clone https: //github.com/justu ss- chock/delira.git && cd delira && p ip install .[tenso rflow]‘‘	the `tensorflow` backend is still very experimental and lacks some features
Full	‘‘p ip ins tal l d eli ra[ful l]‘ ‘	‘‘git clone https: //github.com/justu ss- chock/delira.git && cd delira && p ip install .[full] ‘‘	All backends will be installed.

CHAPTER

TWO

DELIRA INTRODUCTION

Last updated: 09.05.2019

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, **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`). To get a single sample of your dataset after creating it, you can index it like this: `dataset[0]`. Additionally you can iterate over your dataset just like over any other `python` iterator via

```
for sample in dataset:
    # do your stuff here
```

or enumerate it via

```
for idx, sample in enumerate(dataset):
    # do your stuff here
```

.

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_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:

```
from delira.data_loading import TorchvisionClassificationDataset
dataset_train = TorchvisionClassificationDataset("mnist", train=True,
                                                 img_shape=(224, 224))
```

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

```
dataset_train[0]
```

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:

```
dataset_val = TorchvisionClassificationDataset("mnist", train=False,
                                               img_shape=(224, 224))
```

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`:

```
from delira.data_loading import BaseDataLoader

batch_size = 32

loader_train = BaseDataLoader(dataset_train, batch_size)
```

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:

```
from delira.data_loading import BaseDataManager
from batchgenerators.transforms.abstract_transforms import Compose
from batchgenerators.transforms.sample_normalization_transforms import␣
↪MeanStdNormalizationTransform

batchsize = 64
transforms = Compose([MeanStdNormalizationTransform(mean=1*[0], std=1*[1])])

data_manager_train = BaseDataManager(dataset_train,  # dataset to use
                                     batchsize,  # batchsize
                                     n_process_augmentation=1,  # number of␣
↪augmentation processes
                                     transforms=transforms)  # augmentation transforms
```

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`:

```
data_manager_val = BaseDataManager(dataset_val,
                                   batchsize,
                                   n_process_augmentation=1,
                                   transforms=transforms)
```

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

Iterating over a DataManager is possible in simple loops:

```
from tqdm.auto import tqdm # utility for progress bars

# create actual batch generator from DataManager
batchgen = data_manager_val.get_batchgen()

for data in tqdm(batchgen):
    pass # here you can access the data of the current batch
```

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 WeightedRandomSampleraccepts 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
    ----------
```

(continues on next page)

(continued from previous page)

```
    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 and must return a dictionary. 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))
```

(continues on next page)

(continued from previous page)

```
    out_3 = self.conv3(out2)

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

        Parameters
        ----------
        model : :class:`ClassificationNetworkBasePyTorch`
```

(continues on next page)

(continued from previous page)

```
        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")
        # obtain outputs from network
        preds = model(inputs)["pred"]

        if data_dict:

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

(continues on next page)

(continued from previous page)

```
                    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:

```
import torch
from delira.training import Parameters
from delira.data_loading import RandomSampler, SequentialSampler

params = Parameters(fixed_params={
    "model": {},
    "training": {
        "batch_size": 64, # batchsize to use
        "num_epochs": 2, # number of epochs to train
        "optimizer_cls": torch.optim.Adam, # optimization algorithm to use
        "optimizer_params": {'lr': 1e-3}, # initialization parameters for this␣
↪algorithm
        "criterions": {"CE": torch.nn.CrossEntropyLoss()}, # the loss function
        "lr_sched_cls": None,  # the learning rate scheduling algorithm to use
        "lr_sched_params": {}, # the corresponding initialization parameters
        "metrics": {} # and some evaluation metrics
    }
})

# recreating the data managers with the batchsize of the params object
manager_train = BaseDataManager(dataset_train, params.nested_get("batch_size"), 1,
                                transforms=None, sampler_cls=RandomSampler,
                                n_process_loading=4)
manager_val = BaseDataManager(dataset_val, params.nested_get("batch_size"), 3,
                              transforms=None, sampler_cls=SequentialSampler,
                              n_process_loading=4)
```

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:

```
from delira.training import PyTorchNetworkTrainer
from delira.models.classification import ClassificationNetworkBasePyTorch

# path where checkpoints should be saved
```

(continues on next page)

(continued from previous page)

```
save_path = "./results/checkpoints"

model = ClassificationNetworkBasePyTorch(in_channels=1, n_outputs=10)

trainer = PyTorchNetworkTrainer(network=model,
                                save_path=save_path,
                                criterions=params.nested_get("criterions"),
                                optimizer_cls=params.nested_get("optimizer_cls"),
                                optimizer_params=params.nested_get("optimizer_params
↪"),
                                metrics=params.nested_get("metrics"),
                                lr_scheduler_cls=params.nested_get("lr_sched_cls"),
                                lr_scheduler_params=params.nested_get("lr_sched_params
↪"),
                                gpu_ids=[0]
                            )

#trainer.train(params.nested_get("num_epochs"), manager_train, manager_val)
```

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:

```
from delira.training import PyTorchExperiment
from delira.training.train_utils import create_optims_default_pytorch

# Add model parameters to Parameter class
params.fixed.model = {"in_channels": 1, "n_outputs": 10}

experiment = PyTorchExperiment(params=params,
                               model_cls=ClassificationNetworkBasePyTorch,
                               name="TestExperiment",
                               save_path="./results",
                               optim_builder=create_optims_default_pytorch,
                               gpu_ids=[0])

experiment.run(manager_train, manager_val)
```

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:

```
NUM_ITERS = 4

# import logging handler and logging module
from delira.logging import TrixiHandler
from trixi.logger import PytorchVisdomLogger
import logging

# configure logging module (and root logger)
logger_kwargs = {
    'name': 'test_env', # name of loggin environment
    'port': 9999 # visdom port to connect to
}
logger_cls = PytorchVisdomLogger

# configure logging module (and root logger)
logging.basicConfig(level=logging.INFO,
                    handlers=[TrixiHandler(logger_cls, **logger_kwargs)])
# derive logger from root logger
# (don't do `logger = logging.Logger("...")` since this will create a new
# logger which is unrelated to the root logger
logger = logging.getLogger("Test Logger")

# create dict containing the scalar numbers as torch.Tensor
scalars = {"one": torch.Tensor([1]),
           "two": torch.Tensor([2]),
           "three": torch.Tensor([3]),
           "four": torch.Tensor([4])}

# create dict containing the images as torch.Tensor
# pytorch awaits tensor dimensionality of
# batchsize x image channels x height x width
images = {"prediction": torch.rand(1, 3, 224, 224),
          "groundtruth": torch.rand(1, 3, 224, 224)}

# Simulate 4 Epochs
for i in range(4*NUM_ITERS):
```

(continues on next page)

(continued from previous page)

```
    logger.info({"image_grid": {"images": images["prediction"], "name": "predictions"}
↪})

    for key, val_tensor in scalars.items():
        logger.info({"value": {"value": val_tensor.item(), "name": key}})
        scalars[key] += 1
```

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

Dataset Guide (incl. Integration to new API) With Delira v0.3.2 a new dataset API was introduced to allow for more flexibility and add some features. This notebook shows the difference between the new and the old API and provides some examples for newly added features.

Overview Old API The old dataset API was based on the assumption that the underlying structure of the data can be described as followed: * root * sample1 * img1 * img2 * label * sample2 * img1 * img2 * label * ...

A single sample was constructed from multiple images which are all located in the same subdirectory. The corresponding signature of the `AbstractDataset` was given by `data_path, load_fn, img_extensions, gt_extensions`. While most datasets need a `load_fn` to load a single sample and a `data_path` to the root directory, `img_extensions`and `gt_exntensions` were often unsed. As a consequence a new dataset needed to be created which initialises the unused variables with arbitrary values.

Overview New API The new dataset API was refactored to a more general approach where only a `data_path` to the root directory and a `load_fn` for a single sample need to be provided. A simple loading function (`load_fn`) to generate random data independent from the given path might be realized as below.

```
import numpy as np


def load_random_data(path: str) -> dict:
    """Load random data

    Parameters
    ----------
    path : str
        path to sample (not used in this example)

    Returns
    -------
    dict
        return data inside a dict
    """
    return {
        'data': np.random.rand(3, 512, 512),
        'label': np.random.randint(0, 10),
        'path': path,
    }
```

When used with the provided BaseDatasets, the return value of the load function is not limited to dictionaries and might be of any type which can be added to a list with the `append` method.

New Datasets Some basic datasets are already implemented inside Delira and should be suitable for most cases. The `BaseCacheDataset` saves all samples inside the RAM and thus can only be used if everything fits inside the

memory. ´BaseLazyDataset´ loads the individual samples on time when they are needed, but might lead to slower training due to the additional loading time.

```
from delira.data_loading import BaseCacheDataset, BaseLazyDataset


# because `load_random_data` does not use the path argument, they can have
# arbitrary values in this example
paths = list(range(10))

# create case dataset
cached_set = BaseCacheDataset(paths, load_random_data)

# create lazy dataset
lazy_set = BaseLazyDataset(paths, load_random_data)

# print cached data
print(cached_set[0].keys())

# print lazy data
print(lazy_set[0].keys())
```

In the above example a list of multiple paths is used as the `data_path`. `load_fn` is called for every element inside the provided list (can be any iterator). If `data_path` is a single string, it is assumed to be the path to the root directory. In this case, `load_fn`is called for every element inside the root directory.

Sometimes, a single file/folder contains multiple samples. `BaseExtendCacheDataset` uses the `extend` function to add elements to the internal list. Thus it is assumed that `load_fn` provides an iterable object, where eacht item represents a single data sample.

`AbstractDataset` is now iterable and can be used directly in combination with for loops.

```
for cs in cached_set:
    print(cs["path"])
```

New Utility Function (Integration to new API) The behavior of the old API can be replicated with the `LoadSample`, `LoadSampleLabel`functions. `LoadSample` assumes that all needed images and the label (for a single sample) are located in a directory. Both functions return a dictionary containing the loaded data. `sample_ext` maps keys to iterables. Each iterable defines the names of the images which should be loaded from the directory. ´sample_fn´ is used to load the images which are than stacked inside a single array.

```
from delira.data_loading import LoadSample, LoadSampleLabel


def load_random_array(path: str):
    """Return random data

    Parameters
    ----------
    path : str
        path to image

    Returns
    -------
    np.ndarray
        loaded data
    """
    return np.random.rand(128, 128)
```

(continues on next page)

(continued from previous page)

```
# define the function to load a single sample from a directory
load_fn = LoadSample(
    sample_ext={
        # load 3 data channels
        'data': ['red.png', 'green.png', 'blue.png'],
        # load a singel segmentation channel
        'seg': ['seg.png']
    },
    sample_fn=load_random_array,
    # optionally: assign individual keys a datatype
    dtype={"data": "float", "seg": "uint8"},
    # optioanlly: normalize individual samples
    normalize=["data"])

# Note: in general the function should be called with the path of the
# directory where the imags are located
sample0 = load_fn(".")

print("data shape: {}".format(sample0["data"].shape))
print("segmentation shape: {}".format(sample0["seg"].shape))
print("data type: {}".format(sample0["data"].dtype))
print("segmentation type: {}".format(sample0["seg"].dtype))
print("data min value: {}".format(sample0["data"].min()))
print("data max value: {}".format(sample0["data"].max()))
```

By default the range is normalized to (-1, 1), but `norm_fn` can be changed to achieve other normalization schemes. Some examples are included in `delira.data_loading.load_utils`.

`LoadSampleLabel` takes an additional argument for the label and a function to load a label. This functions can be used in combination with the provided BaseDatasets to replicate (and extend) the old API.

CHAPTER

THREE

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:

```
logger = None
import torch
from delira.training import Parameters
params = Parameters(fixed_params={
    "model": {
        "in_channels": 1,
        "n_outputs": 10
    },
    "training": {
        "batch_size": 64, # batchsize to use
        "num_epochs": 10, # number of epochs to train
        "optimizer_cls": torch.optim.Adam, # optimization algorithm to use
        "optimizer_params": {'lr': 1e-3}, # initialization parameters for this␣
↪algorithm
        "losses": {"CE": torch.nn.CrossEntropyLoss()}, # the loss function
        "lr_sched_cls": None,  # the learning rate scheduling algorithm to use
        "lr_sched_params": {}, # the corresponding initialization parameters
        "metrics": {} # and some evaluation metrics
    }
})
```

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.

```
from trixi.logger import PytorchVisdomLogger
from delira.logging import TrixiHandler
import logging

logger_kwargs = {
    'name': 'ClassificationExampleLogger', # name of our logging environment
    'port': 9999 # port on which our visdom server is alive
}

logger_cls = PytorchVisdomLogger

# configure logging module (and root logger)
logging.basicConfig(level=logging.INFO,
                    handlers=[TrixiHandler(logger_cls, **logger_kwargs)])


# derive logger from root logger
# (don't do `logger = logging.Logger("...")` since this will create a new
# logger which is unrelated to the root logger
logger = logging.getLogger("Test Logger")
```

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):

```
from delira.data_loading import TorchvisionClassificationDataset

dataset_train = TorchvisionClassificationDataset("mnist", # which dataset to use
                                                 train=True, # use trainset
                                                 img_shape=(224, 224) # resample to␣
↪224 x 224 pixels
                                                )
dataset_val = TorchvisionClassificationDataset("mnist",
                                               train=False,
                                               img_shape=(224, 224)
                                              )
```

3.2.2 Augmentation

For Data-Augmentation we will apply a few transformations:

```
from batchgenerators.transforms import RandomCropTransform, \
                                       ContrastAugmentationTransform, Compose
from batchgenerators.transforms.spatial_transforms import ResizeTransform
from batchgenerators.transforms.sample_normalization_transforms import␣
↪MeanStdNormalizationTransform

transforms = Compose([
```

(continues on next page)

(continued from previous page)

```
    RandomCropTransform(200), # Perform Random Crops of Size 200 x 200 pixels
    ResizeTransform(224), # Resample these crops back to 224 x 224 pixels
    ContrastAugmentationTransform(), # randomly adjust contrast
    MeanStdNormalizationTransform(mean=[0.5], std=[0.5])])
```

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

```
from delira.data_loading import BaseDataManager, SequentialSampler, RandomSampler

manager_train = BaseDataManager(dataset_train, params.nested_get("batch_size"),
                                transforms=transforms,
                                sampler_cls=RandomSampler,
                                n_process_augmentation=4)

manager_val = BaseDataManager(dataset_val, params.nested_get("batch_size"),
                              transforms=transforms,
                              sampler_cls=SequentialSampler,
                              n_process_augmentation=4)
```

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:

```
import warnings
warnings.simplefilter("ignore", UserWarning) # ignore UserWarnings raised by␣
↪dependency code
warnings.simplefilter("ignore", FutureWarning) # ignore FutureWarnings raised by␣
↪dependency code


from delira.training import PyTorchExperiment
from delira.training.train_utils import create_optims_default_pytorch
from delira.models.classification import ClassificationNetworkBasePyTorch

if logger is not None:
    logger.info("Init Experiment")
experiment = PyTorchExperiment(params, ClassificationNetworkBasePyTorch,
                               name="ClassificationExample",
                               save_path="./tmp/delira_Experiments",
                               optim_builder=create_optims_default_pytorch,
                               gpu_ids=[0])
experiment.save()

model = experiment.run(manager_train, manager_val)
```

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:

```
import numpy as np
from tqdm.auto import tqdm # utility for progress bars

device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # set device␣
↪(use GPU if available)
```

(continues on next page)

(continued from previous page)

```
model = model.to(device) # push model to device
preds, labels = [], []

with torch.no_grad():
    for i in tqdm(range(len(dataset_val))):
        img = dataset_val[i]["data"] # get image from current batch
        img_tensor = torch.from_numpy(img).unsqueeze(0).to(device).to(torch.float) #␣
↪create a tensor from image, push it to device and add batch dimension
        pred_tensor = model(img_tensor) # feed it through the network
        pred = pred_tensor.argmax(1).item() # get index with maximum class confidence
        label = np.asscalar(dataset_val[i]["label"]) # get label from batch
        if i % 1000 == 0:
            print("Prediction: %d \t label: %d" % (pred, label)) # print result
        preds.append(pred)
        labels.append(label)

# calculate accuracy
accuracy = (np.asarray(preds) == np.asarray(labels)).sum() / len(preds)
print("Accuracy: %.3f" % accuracy)
```

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

FOUR

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:

```
logger = None
import torch
from delira.training import Parameters
params = Parameters(fixed_params={
    "model": {
        "n_channels": 1,
        "noise_length": 10
    },
    "training": {
        "batch_size": 64, # batchsize to use
        "num_epochs": 10, # number of epochs to train
        "optimizer_cls": torch.optim.Adam, # optimization algorithm to use
        "optimizer_params": {'lr': 1e-3}, # initialization parameters for this␣
↪algorithm
        "losses": {"L1": torch.nn.L1Loss()}, # the loss function
        "lr_sched_cls": None,  # the learning rate scheduling algorithm to use
        "lr_sched_params": {}, # the corresponding initialization parameters
        "metrics": {} # and some evaluation metrics
    }
})
```

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.

```
from trixi.logger import PytorchVisdomLogger
from delira.logging import TrixiHandler
import logging

logger_kwargs = {
    'name': 'GANExampleLogger', # name of our logging environment
    'port': 9999 # port on which our visdom server is alive
}

logger_cls = PytorchVisdomLogger

# configure logging module (and root logger)
logging.basicConfig(level=logging.INFO,
                    handlers=[TrixiHandler(logger_cls, **logger_kwargs)])


# derive logger from root logger
# (don't do `logger = logging.Logger("...")` since this will create a new
# logger which is unrelated to the root logger
logger = logging.getLogger("Test Logger")
```

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):

```
from delira.data_loading import TorchvisionClassificationDataset

dataset_train = TorchvisionClassificationDataset("mnist", # which dataset to use
                                                 train=True, # use trainset
                                                 img_shape=(224, 224) # resample to␣
↪224 x 224 pixels
                                                )
dataset_val = TorchvisionClassificationDataset("mnist",
                                               train=False,
                                               img_shape=(224, 224)
                                              )
```

4.3.2 Augmentation

For Data-Augmentation we will apply a few transformations:

```
from batchgenerators.transforms import RandomCropTransform, \
                                       ContrastAugmentationTransform, Compose
```

(continues on next page)

(continued from previous page)

```
from batchgenerators.transforms.spatial_transforms import ResizeTransform
from batchgenerators.transforms.sample_normalization_transforms import␣
↪MeanStdNormalizationTransform

transforms = Compose([
    RandomCropTransform(200), # Perform Random Crops of Size 200 x 200 pixels
    ResizeTransform(224), # Resample these crops back to 224 x 224 pixels
    ContrastAugmentationTransform(), # randomly adjust contrast
    MeanStdNormalizationTransform(mean=[0.5], std=[0.5])])
```

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

```
from delira.data_loading import BaseDataManager, SequentialSampler, RandomSampler

manager_train = BaseDataManager(dataset_train, params.nested_get("batch_size"),
                                transforms=transforms,
                                sampler_cls=RandomSampler,
                                n_process_augmentation=4)

manager_val = BaseDataManager(dataset_val, params.nested_get("batch_size"),
                              transforms=transforms,
                              sampler_cls=SequentialSampler,
                              n_process_augmentation=4)
```

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:

```
import warnings
warnings.simplefilter("ignore", UserWarning) # ignore UserWarnings raised by␣
↪dependency code
warnings.simplefilter("ignore", FutureWarning) # ignore FutureWarnings raised by␣
↪dependency code


from delira.training import PyTorchExperiment
from delira.training.train_utils import create_optims_gan_default_pytorch
from delira.models.gan import GenerativeAdversarialNetworkBasePyTorch

if logger is not None:
    logger.info("Init Experiment")
experiment = PyTorchExperiment(params, GenerativeAdversarialNetworkBasePyTorch,
                               name="GANExample",
                               save_path="./tmp/delira_Experiments",
                               optim_builder=create_optims_gan_default_pytorch,
                               gpu_ids=[0])
experiment.save()

model = experiment.run(manager_train, manager_val)
```

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

FIVE

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:

```
logger = None
import torch
from delira.training import Parameters
params = Parameters(fixed_params={
    "model": {
        "in_channels": 1,
        "num_classes": 4
    },
    "training": {
        "batch_size": 64, # batchsize to use
        "num_epochs": 10, # number of epochs to train
        "optimizer_cls": torch.optim.Adam, # optimization algorithm to use
        "optimizer_params": {'lr': 1e-3}, # initialization parameters for this␣
↪algorithm
        "losses": {"CE": torch.nn.CrossEntropyLoss()}, # the loss function
        "lr_sched_cls": None,  # the learning rate scheduling algorithm to use
        "lr_sched_params": {}, # the corresponding initialization parameters
        "metrics": {} # and some evaluation metrics
    }
})
```

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.

```
from trixi.logger import PytorchVisdomLogger
from delira.logging import TrixiHandler
import logging

logger_kwargs = {
    'name': 'ClassificationExampleLogger', # name of our logging environment
    'port': 9999 # port on which our visdom server is alive
}

logger_cls = PytorchVisdomLogger

# configure logging module (and root logger)
logging.basicConfig(level=logging.INFO,
                    handlers=[TrixiHandler(logger_cls, **logger_kwargs)])


# derive logger from root logger
# (don't do `logger = logging.Logger("...")` since this will create a new
# logger which is unrelated to the root logger
logger = logging.getLogger("Test Logger")
```

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:

```
from io import BytesIO
from zipfile import ZipFile
from urllib.request import urlopen

resp = urlopen("http://www.fmrib.ox.ac.uk/primers/intro_primer/ExBox3/ExBox3.zip")
zipfile = ZipFile(BytesIO(resp.read()))
#zipfile_list = zipfile.namelist()
#print(zipfile_list)
img_file = zipfile.extract("ExBox3/T1_brain.nii.gz")
mask_file = zipfile.extract("ExBox3/T1_brain_seg.nii.gz")
```

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):

```
import SimpleITK as sitk
import numpy as np

# load image and mask
img = sitk.GetArrayFromImage(sitk.ReadImage(img_file))
img = img.astype(np.float32)
```

(continues on next page)

(continued from previous page)

```
mask = mask = sitk.GetArrayFromImage(sitk.ReadImage(mask_file))
mask = mask.astype(np.float32)

assert mask.shape == img.shape
print(img.shape)
```

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

```
np.unique(mask)
```

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:

```
import matplotlib.pyplot as plt

# load single slice
img_slice = img[:, :, 100]
mask_slice = mask[:, :, 100]

# plot slices
plt.figure(1, figsize=(15,10))
plt.subplot(121)
plt.imshow(img_slice, cmap="gray")
plt.colorbar(fraction=0.046, pad=0.04)
plt.subplot(122)
plt.imshow(mask_slice, cmap="gray")
plt.colorbar(fraction=0.046, pad=0.04)
plt.show()
```

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.

```
from delira.data_loading import AbstractDataset

class CustomDataset(AbstractDataset):
    def __init__(self, img, mask, num_samples=1000):
        super().__init__(None, None, None, None)
        self.data = {"data": img.reshape(1, *img.shape), "label": mask.reshape(1,␣
↪*mask.shape)}
        self.num_samples = num_samples

    def __getitem__(self, index):
        return self.data

    def __len__(self):
        return self.num_samples
```

Now, we can finally instantiate our datasets:

```
dataset_train = CustomDataset(img_slice, mask_slice, num_samples=10000)
dataset_val = CustomDataset(img_slice, mask_slice, num_samples=1)
```

5.2.2 Augmentation

For Data-Augmentation we will apply a few transformations:

```
from batchgenerators.transforms import RandomCropTransform, \
                                       ContrastAugmentationTransform, Compose
from batchgenerators.transforms.spatial_transforms import ResizeTransform
from batchgenerators.transforms.sample_normalization_transforms import␣
↪MeanStdNormalizationTransform

transforms = Compose([
    RandomCropTransform(150, label_key="label"), # Perform Random Crops of Size 150 x␣
↪150 pixels
    ResizeTransform(224, label_key="label"), # Resample these crops back to 224 x 224␣
↪pixels
    ContrastAugmentationTransform(), # randomly adjust contrast
    MeanStdNormalizationTransform(mean=[img_slice.mean()], std=[img_slice.std()])]) #␣
↪use concrete values since we only have one sample (have to estimate it over whole␣
↪dataset otherwise)
```

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

```
from delira.data_loading import BaseDataManager, SequentialSampler, RandomSampler

manager_train = BaseDataManager(dataset_train, params.nested_get("batch_size"),
                                transforms=transforms,
                                sampler_cls=RandomSampler,
                                n_process_augmentation=4)

manager_val = BaseDataManager(dataset_val, params.nested_get("batch_size"),
                              transforms=transforms,
                              sampler_cls=SequentialSampler,
                              n_process_augmentation=4)
```

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`:

```
import warnings
warnings.simplefilter("ignore", UserWarning) # ignore UserWarnings raised by␣
↪dependency code
warnings.simplefilter("ignore", FutureWarning) # ignore FutureWarnings raised by␣
↪dependency code


from delira.training import PyTorchExperiment
from delira.training.train_utils import create_optims_default_pytorch
from delira.models.segmentation import UNet2dPyTorch

if logger is not None:
    logger.info("Init Experiment")
experiment = PyTorchExperiment(params, UNet2dPyTorch,
                               name="Segmentation2dExample",
                               save_path="./tmp/delira_Experiments",
```

(continues on next page)

(continued from previous page)

```
                                optim_builder=create_optims_default_pytorch,
                                gpu_ids=[0], mixed_precision=True)
experiment.save()

model = experiment.run(manager_train, manager_val)
```

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

SIX

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:

```
logger = None
import torch
from delira.training import Parameters
params = Parameters(fixed_params={
    "model": {
        "in_channels": 1,
        "num_classes": 4
    },
    "training": {
        "batch_size": 64, # batchsize to use
        "num_epochs": 10, # number of epochs to train
        "optimizer_cls": torch.optim.Adam, # optimization algorithm to use
        "optimizer_params": {'lr': 1e-3}, # initialization parameters for this␣
↪algorithm
        "losses": {"CE": torch.nn.CrossEntropyLoss()}, # the loss function
        "lr_sched_cls": None,  # the learning rate scheduling algorithm to use
        "lr_sched_params": {}, # the corresponding initialization parameters
        "metrics": {} # and some evaluation metrics
    }
})
```

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.

```
from trixi.logger import PytorchVisdomLogger
from delira.logging import TrixiHandler
import logging

logger_kwargs = {
    'name': 'ClassificationExampleLogger', # name of our logging environment
    'port': 9999 # port on which our visdom server is alive
}

logger_cls = PytorchVisdomLogger

# configure logging module (and root logger)
logging.basicConfig(level=logging.INFO,
                    handlers=[TrixiHandler(logger_cls, **logger_kwargs)])


# derive logger from root logger
# (don't do `logger = logging.Logger("...")` since this will create a new
# logger which is unrelated to the root logger
logger = logging.getLogger("Test Logger")
```

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:

```
from io import BytesIO
from zipfile import ZipFile
from urllib.request import urlopen

resp = urlopen("http://www.fmrib.ox.ac.uk/primers/intro_primer/ExBox3/ExBox3.zip")
zipfile = ZipFile(BytesIO(resp.read()))
#zipfile_list = zipfile.namelist()
#print(zipfile_list)
img_file = zipfile.extract("ExBox3/T1_brain.nii.gz")
mask_file = zipfile.extract("ExBox3/T1_brain_seg.nii.gz")
```

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):

```
import SimpleITK as sitk
import numpy as np

# load image and mask
img = sitk.GetArrayFromImage(sitk.ReadImage(img_file))
img = img.astype(np.float32)
```

(continues on next page)

(continued from previous page)

```
mask = mask = sitk.GetArrayFromImage(sitk.ReadImage(mask_file))
mask = mask.astype(np.float32)

assert mask.shape == img.shape
print(img.shape)
```

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

```
np.unique(mask)
```

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.

```
from delira.data_loading import AbstractDataset

class CustomDataset(AbstractDataset):
    def __init__(self, img, mask, num_samples=1000):
        super().__init__(None, None, None, None)
        self.data = {"data": img.reshape(1, *img.shape), "label": mask.reshape(1,␣
↪*mask.shape)}
        self.num_samples = num_samples

    def __getitem__(self, index):
        return self.data

    def __len__(self):
        return self.num_samples
```

Now, we can finally instantiate our datasets:

```
dataset_train = CustomDataset(img, mask, num_samples=10000)
dataset_val = CustomDataset(img, mask, num_samples=1)
```

6.2.2 Augmentation

For Data-Augmentation we will apply a few transformations:

```
from batchgenerators.transforms import ContrastAugmentationTransform, Compose
from batchgenerators.transforms.sample_normalization_transforms import␣
↪MeanStdNormalizationTransform

transforms = Compose([
    ContrastAugmentationTransform(), # randomly adjust contrast
    MeanStdNormalizationTransform(mean=[img.mean()], std=[img.std()])]) # use␣
↪concrete values since we only have one sample (have to estimate it over whole␣
↪dataset otherwise)
```

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

```
from delira.data_loading import BaseDataManager, SequentialSampler, RandomSampler

manager_train = BaseDataManager(dataset_train, params.nested_get("batch_size"),
```

(continues on next page)

(continued from previous page)

```
                                transforms=transforms,
                                sampler_cls=RandomSampler,
                                n_process_augmentation=4)

manager_val = BaseDataManager(dataset_val, params.nested_get("batch_size"),
                              transforms=transforms,
                              sampler_cls=SequentialSampler,
                              n_process_augmentation=4)
```

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`:

```
import warnings
warnings.simplefilter("ignore", UserWarning) # ignore UserWarnings raised by␣
↪dependency code
warnings.simplefilter("ignore", FutureWarning) # ignore FutureWarnings raised by␣
↪dependency code


from delira.training import PyTorchExperiment
from delira.training.train_utils import create_optims_default_pytorch
from delira.models.segmentation import UNet3dPyTorch

if logger:
    logger.info("Init Experiment")
experiment = PyTorchExperiment(params, UNet3dPyTorch,
                               name="Segmentation3dExample",
                               save_path="./tmp/delira_Experiments",
                               optim_builder=create_optims_default_pytorch,
                               gpu_ids=[0], mixed_precision=True)
experiment.save()

model = experiment.run(manager_train, manager_val)
```

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

SEVEN

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: str*, *load_fn: Callable*)

Bases: `object`

Base Class for Dataset

abstract _make_dataset(*path: str*)

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

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: Union[str, list], load_fn: Callable, **load_kwargs*)
Bases: delira.data_loading.dataset.AbstractDataset

Dataset to load data in a lazy way

_make_dataset(*path: Union[str, list]*)
Helper Function to make a dataset containing paths to all images in a certain directory

Parameters **path** (*str or list*) – 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

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: Union[str, list], load_fn: Callable, **load_kwargs*)
Bases: delira.data_loading.dataset.AbstractDataset

Dataset to preload and cache data

Notes

data needs to fit completely into RAM!

_make_dataset(*path: Union[str, list]*)
Helper Function to make a dataset containing all samples in a certain directory

Parameters **path** (*str or list*) – if data_path is a string, _sample_fn is called for all items inside the specified directory if data_path is a list, _sample_fn is called for elements in the list

Returns list of items which where returned from _sample_fn (typically dict)

Return type list

Raises **AssertionError** – if *path* is not a list and 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

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

BaseExtendCacheDataset

class BaseExtendCacheDataset (*data_path: Union[str, list], load_fn: Callable, **load_kwargs*)
Bases: delira.data_loading.dataset.BaseCacheDataset

Dataset to preload and cache data. Function to load sample is expected to return an iterable which can contain multiple samples

Notes

data needs to fit completely into RAM!

_make_dataset (*path: Union[str, list]*)
Helper Function to make a dataset containing all samples in a certain directory

Parameters **path** (*str or iterable*) – if data_path is a string, _sample_fn is called for all items inside the specified directory if data_path is a list, _sample_fn is called for elements in the list

Returns list of items which where returned from _sample_fn (typically dict)

Return type list

Raises **AssertionError** – if *path* is not a list and 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

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

abstract _make_dataset(*path: str*)
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

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

BlankDataset

Nii3DLazyDataset

Nii3DCacheDataset

TorchvisionClassificationDataset:

class TorchvisionClassificationDataset(*dataset*, *root='/tmp/'*, *train=True*, *download=True*, *img_shape=(28, 28)*, *one_hot=False*, ***kwargs*)
Bases: delira.data_loading.dataset.AbstractDataset

Wrapper for torchvision classification datasets to provide consistent API

_make_dataset(*dataset*, ***kwargs*)
Create the actual dataset

Parameters

- **dataset** (*str*) – Defines the dataset to use. must be one of [‘mnist’, ‘emnist’, ‘fashion_mnist’, ‘cifar10’, ‘cifar100’]
- ****kwargs** – Additional keyword arguments passed to the torchvision dataset class for initialization

Returns actual Dataset

Return type torchvision.Dataset

Raises **KeyError** – Dataset string does not specify a valid dataset

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

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: batchgenerators.dataloading.data_loader.SlimDataLoaderBase

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

property batch_size
Property to access the batchsize

Returns the batchsize

Return type int

property data_loader_cls
Property to access the current data loader class

Returns Subclass of SlimDataLoaderBase

Return type type

property 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 Augmenter

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*

property n_batches
Returns Number of Batches based on batchsize and number of samples

Returns Number of Batches

Return type int

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

property n_process_augmentation
Property to access the number of augmentation processes

Returns number of augmentation processes

Return type int

property n_samples
Number of Samples

Returns Number of Samples

Return type int

property 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

property 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

norm_range

norm_range(*mode*)

Closure function for range normalization

Parameters **mode** (*str*) – '-1,1' normalizes data to range [-1, 1], while '0,1' normalizes data to range [0, 1]

Returns normalization function

Return type callable

norm_zero_mean_unit_std

norm_zero_mean_unit_std(*data*)

Return normalized data with mean 0, standard deviation 1

Parameters **data** (*np.nadarray*) –

Returns normalized data

Return type np.ndarray

is_valid_image_file

is_valid_image_file(*fname*, *img_extensions*, *gt_extensions*)

Helper Function to check wheter file is image file and has at least one label file

Parameters

- **fname** (*str*) – filename of image path Returns
- **-------** –
- **bool** – is valid data sample

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

LoadSample

class LoadSample(*sample_ext: dict*, *sample_fn: collections.abc.Callable*, *dtype={}*, *normalize=()*, *norm_fn=<function norm_range.<locals>.norm_fn>*, ***kwargs*)
Bases: object

Provides a callable to load a single sample from multiple files in a folder

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:

load_nii

load_nii(*path*)
Loads a single nii file :param path: path to nii file which should be loaded :type path: str

Returns numpy array containing the loaded data

Return type np.ndarray

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

abstract 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

_check_batchsize(*n_indices*)
Checks if the batchsize is valid (and truncates batches if necessary). Will also raise StopIteration if enough batches sampled

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

Returns number of indices to sample (truncated if necessary)

Return type int

Raises **StopIteration** – if enough batches sampled

abstract _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

_check_batchsize(*n_indices*)
Checks if the batchsize is valid (and truncates batches if necessary). Will also raise StopIteration if enough batches sampled

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

Returns number of indices to sample (truncated if necessary)

Return type int

Raises **StopIteration** – if enough batches sampled

_get_indices(*n_indices*)
Actual Sampling

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

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*

RandomSampler

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

Implements Random Sampling from whole Dataset

_check_batchsize(*n_indices*)
Checks if the batchsize is valid (and truncates batches if necessary). Will also raise StopIteration if enough batches sampled

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

Returns number of indices to sample (truncated if necessary)

Return type int

Raises **StopIteration** – if enough batches sampled

_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

_check_batchsize(*n_indices*)
Checks if the batchsize is valid (and truncates batches if necessary). Will also raise StopIteration if enough batches sampled

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

Returns number of indices to sample (truncated if necessary)

Return type int

Raises **StopIteration** – if enough batches sampled

_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 initialized 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

_check_batchsize(*n_indices*)
Checks if batchsize is valid for all classes

Parameters **n_indices** (*int*) – the number of samples to return

Returns number of samples per class to return

Return type dict

_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 initialized sampler

Return type *AbstractSampler*

SequentialSampler

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

Implements Sequential Sampling from whole Dataset

_check_batchsize(*n_indices*)
Checks if the batchsize is valid (and truncates batches if necessary). Will also raise StopIteration if enough batches sampled

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

Returns number of indices to sample (truncated if necessary)

Return type int

Raises **StopIteration** – if enough batches sampled

_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

_check_batchsize(*n_indices*)
Checks if the batchsize is valid (and truncates batches if necessary). Will also raise StopIteration if enough batches sampled

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

Returns number of indices to sample (truncated if necessary)

Return type int

Raises **StopIteration** – if enough batches 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:

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

_check_batchsize(*n_indices*)
Checks if batchsize is valid for all classes

Parameters **n_indices** (*int*) – the number of samples to return

Returns number of samples per class to return

Return type dict

_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*)
Bases: delira.data_loading.sampler.abstract_sampler.AbstractSampler

Implements Weighted Random Sampling

_check_batchsize(*n_indices*)
Checks if the batchsize is valid (and truncates batches if necessary). Will also raise StopIteration if enough batches sampled

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

Returns number of indices to sample (truncated if necessary)

Return type int

Raises **StopIteration** – if enough batches sampled

_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
- **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*, ***kwargs*)
Loads a saved model

Parameters

- **file** (*str*) – filepath to a file containing a saved model
- ****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 checkpoint state_dict

Return type OrderedDict

torch_save_checkpoint

save_checkpoint(*file: str*, *model=None*, *optimizers={}*, *epoch=None*, ***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)

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.

property 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.

property 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 = {}

abstract static closure(*model*, *data_dict: dict*, *optimizers: dict*, *losses={}*, *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 losses: Functions or classes to calculate losses :type losses: 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 losses)
- *dict* – Arbitrary number of predictions

Raises **NotImplementedError** – If not overwritten by subclass

property 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, torch.nn.Module

Abstract Class for PyTorch Networks

See also:

None, *AbstractNetwork*

_init_kwargs = {}

abstract static closure(*model*, *data_dict: dict*, *optimizers: dict*, *losses={}*, *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 losses: Functions or classes to calculate losses :type losses: 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 losses)
- *dict* – Arbitrary number of predictions

Raises **NotImplementedError** – If not overwritten by subclass

abstract 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

property 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=tensorflow.Session*, ***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 = {}

abstract static closure(*model*, *data_dict: dict*, *optimizers: dict*, *losses={}*, *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 losses: Functions or classes to calculate losses :type losses: 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 losses)
- *dict* – Arbitrary number of predictions

Raises **NotImplementedError** – If not overwritten by subclass

property 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*, ***kwargs*)
Evaluates *self.outputs_train* or *self.outputs_eval* based on *self.training*

Parameters

- ***args** – currently unused, exist for compatibility reasons
- ****kwargs** – kwargs used to feed as `self.inputs`. Same keys as for `self.inputs` must be used

Returns sames keys as outputs_train or outputs_eval, containing evaluated expressions as values

Return type dict

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*, *losses={}*, *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
- **losses** (*dict*) – dict holding the losses to calculate errors (gradients from different losses 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 losses)
- *list* – Arbitrary number of predictions as torch.Tensor

Raises **AssertionError** – if optimizers or losses are empty or the optimizers are not specified

forward(*input_batch: torch.Tensor*)
Forward input_batch through network

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

Returns Classification Result

Return type torch.Tensor

property 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*, *losses={}*, *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
- **losses** (*dict*) – dict holding the losses to calculate errors (gradients from different losses 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 losses)

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

Raises **AssertionError** – if optimizers or losses are empty or the optimizers are not specified

forward(*input_batch: torch.Tensor*)
Forward input_batch through network

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

Returns Classification Result

Return type torch.Tensor

property 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
- *dict* – outputs of *model.run*

property 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*, ***kwargs*)

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

Parameters

- ***args** – currently unused, exist for compatibility reasons
- ****kwargs** – kwargs used to feed as `self.inputs`. Same keys as for `self.inputs` must be used

Returns sames keys as outputs_train or outputs_eval, containing evaluated expressions as values

Return type dict

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*, *losses={}*, *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
- **losses** (*dict*) – dict holding the losses to calculate errors (gradients from different losses 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 losses)
- *list* – Arbitrary number of predictions as torch.Tensor

Raises **AssertionError** – if optimizers or losses 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

property 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` once offered 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 had to be saved too.

_init_kwargs = {}

static closure(*model*, *data_dict: dict*, *optimizers: dict*, *losses={}*, *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
- **losses** (*dict*) – dict holding the losses to calculate errors (gradients from different losses 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 losses)
- *list* – Arbitrary number of predictions as torch.Tensor

Raises **AssertionError** – if optimizers or losses are empty or the optimizers are not specified

forward(*x*)
Feed tensor through network

Parameters **x** (*torch.Tensor*) –

Returns Prediction

Return type torch.Tensor

property 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` once offered 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 had to be saved too.

_init_kwargs = {}

static closure(*model*, *data_dict: dict*, *optimizers: dict*, *losses={}*, *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
- **losses** (*dict*) – dict holding the losses to calculate errors (gradients from different losses 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 losses)
- *list* – Arbitrary number of predictions as torch.Tensor

Raises **AssertionError** – if optimizers or losses are empty or the optimizers are not specified

forward(*x*)
Feed tensor through network

Parameters **x** (*torch.Tensor*) –

Returns Prediction

Return type torch.Tensor

property 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`

property 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

property training_on_top
Return whether the training hierarchy is on top

Returns whether training is on top

Return type bool

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

Parameters

- **dict_like** (*dict*) – Update source
- **deep** (*bool*) – Make deep copies of all references in the source.
- **ignore** (*Iterable*) – Iterable of keys to ignore in update
- **allow_dict_overwrite** (*bool*) – Allow overwriting with dict. Regular dicts only update on the highest level while we recurse and merge Configs. This flag decides whether it is possible to overwrite a 'regular' value with a dict/Config at lower levels. See examples for an illustration of the difference
- **Examples** –
- **---------** –
- **following illustrates the update behaviour if** (*The*) –

:param : :type : obj:allow_dict_overwrite is active. If it isn't, an AttributeError :param would be raised, originating from trying to update "string"::

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

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

config1.update(config2, allow_dict_overwrite=True)

>>>config1
{
```

(continues on next page)

(continued from previous page)

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

property 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.

BaseNetworkTrainer

class BaseNetworkTrainer(*network: delira.models.abstract_network.AbstractNetwork, save_path: str, losses: dict, optimizer_cls: type, optimizer_params: dict, train_metrics: dict, val_metrics: dict, lr_scheduler_cls: type, lr_scheduler_params: dict, gpu_ids: List[int], save_freq: int, optim_fn, key_mapping: dict, logging_type: str, logging_kwargs: dict, fold: int, callbacks: List[delira.training.callbacks.abstract_callback.AbstractCallback], start_epoch=1, metric_keys=None, convert_batch_to_npy_fn=<function BaseNetworkTrainer.<lambda>>, val_freq=1, **kwargs*)
Bases: delira.training.predictor.Predictor

Defines a Base API and basic functions for Network Trainers

See also:

PyTorchNetworkTrainer, *TfNetworkTrainer*

_BaseNetworkTrainer__KEYS_TO_GUARD = ['use_gpu', 'input_device', 'output_device', '_ca

_Predictor__KEYS_TO_GUARD = []

static _Predictor__concatenate_dict_items(*dict_like: dict*)
Function to recursively concatenate dict-items

Parameters **dict_like** (*dict*) – the (nested) dict, whoose items should be concatenated

static _Predictor__convert_dict(*old_dict*, *new_dict*)
Function to recursively convert dicts

Parameters

- **old_dict** (*dict*) – the old nested dict
- **new_dict** (*dict*) – the new nested dict

Returns the updated new nested dict

Return type dict

_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 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

_reinitialize_logging(*logging_type*, *logging_kwargs: dict*)

static _search_for_prev_state(*path*, *extensions=[]*)
Helper function to search in a given path for previous epoch states (indicated by extensions)

Parameters

- **path** (*str*) – the path to search in
- **extensions** (*list*) – list of strings containing valid file extensions for checkpoint files

Returns

- *str* – the file containing the latest checkpoint (if available)
- *None* – if no latst checkpoint was found
- *int* – the latest epoch (1 if no checkpoint was found)

_setup(*network*, *lr_scheduler_cls*, *lr_scheduler_params*, *gpu_ids*, *key_mapping*, *convert_batch_to_npy_fn*, *prepare_batch_fn*)

Parameters

- **network** (AbstractNetwork) – the network to predict from
- **key_mapping** (*dict*) – a dictionary containing the mapping from the data_dict to the actual model's inputs. E.g. if a model accepts one input named 'x' and the data_dict contains one entry named 'data' this argument would have to be {'x': 'data'}
- **convert_batch_to_npy_fn** (*type*) – a callable function to convert tensors in positional and keyword arguments to numpy
- **prepare_batch_fn** (*type*) – function converting a batch-tensor to the framework specific tensor-type and pushing it to correct device, default: identity function

_train_single_epoch(*batchgen: delira.data_loading.data_manager.Augmenter*, *epoch*, *verbose=False*)
Trains the network a single epoch

Parameters

- **batchgen** (Augmenter) – 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 *BaseNetworkTrainer*

static calc_metrics(*batch: delira.utils.config.LookupConfig*, *metrics={}*, *metric_keys=None*)
Compute metrics

Parameters

- **batch** (LookupConfig) – dictionary containing the whole batch (including predictions)
- **metrics** (*dict*) – dict with metrics
- **metric_keys** (*dict*) – dict of tuples which contains hashables for specifying the items to use for calculating the respective metric. If not specified for a metric, the keys "pred" and "label" are used per default

Returns dict with metric results

Return type dict

property 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(*data: dict*, ***kwargs*)
Predict single batch Returns the predictions corresponding to the given data obtained by the model

Parameters

- **data** (*dict*) – batch dictionary
- ****kwargs** – keyword arguments(directly passed to `prepare_batch`)

Returns predicted data

Return type dict

predict_data_mgr(*datamgr*, *batchsize=None*, *metrics={}*, *metric_keys=None*, *verbose=False*, ***kwargs*)
Defines a routine to predict data obtained from a batchgenerator without explicitly caching anything

Parameters

- **datamgr** (`BaseDataManager`) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the `batch_dict` items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- **kwargs** – keyword arguments passed to `prepare_batch_fn()`

Yields

- *dict* – a dictionary containing all predictions of the current batch
- *dict* – a dictionary containing all metrics of the current batch

predict_data_mgr_cache(*datamgr*, *batchsize=None*, *metrics={}*, *metric_keys=None*, *verbose=False*, *cache_preds=False*, ***kwargs*)
Defines a routine to predict data obtained from a batchgenerator and caches all predictions and metrics (yields them in dicts)

Parameters

- **datamgr** (BaseDataManager) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the batch_dict items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- **kwargs** – keyword arguments passed to prepare_batch_fn()

Yields

- *dict* – a dictionary containing all validation metrics (maybe empty)
- *dict* – a dictionary containing all predictions; If cache_preds=True

Warning: Since this function caches all metrics and may additionally cache all predictions (based on the argument cache_preds), this may result in huge memory consumption. If you are running out of memory, please have a look at *Predictor.predict_data_mgr_cache_metrics_only()* or *Predictor.predict_data_mgr()* or consider setting cache_preds to False (if not done already)

predict_data_mgr_cache_all(*datamgr*, *batchsize=None*, *metrics={}*, *metric_keys=None*, *verbose=False*, ***kwargs*)

Defines a routine to predict data obtained from a batchgenerator and caches all predictions and metrics (yields them in dicts)

Parameters

- **datamgr** (BaseDataManager) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the batch_dict items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- **kwargs** – keyword arguments passed to prepare_batch_fn()

Yields

- *dict* – a dictionary containing all predictions;
- *dict* – a dictionary containing all validation metrics (maybe empty)

Warning: Since this function caches all predictions and metrics, this may result in huge memory consumption. If you are running out of memory, please have a look at *Predictor.predict_data_mgr_cache_metrics_only()* or *Predictor.predict_data_mgr()*

predict_data_mgr_cache_metrics_only(*datamgr*, *batchsize=None*, *metrics={}*, *metric_keys=None*, *verbose=False*, ***kwargs*)

Defines a routine to predict data obtained from a batchgenerator and caches the metrics

Parameters

- **datamgr** (BaseDataManager) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the batch_dict items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- **kwargs** – keyword arguments passed to prepare_batch_fn()

Yields *dict* – a dictionary containing all validation metrics (maybe empty)

Notes

This function stores each prediction temporarily for metric calculation; This results in a (typically) way lower memory consumption than *Predictor.predict_data_mgr_cache_all()*, but still caches the metrics. If this is not desired, it is recommended to use *Predictor.predict_data_mgr()* and iterate over the generator as this only produces per-batch metrics and predictions and does not cache anything by default

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'*, *reduce_mode='mean'*, *verbose=True*)
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
- **reduce_mode** (*str*) – ‘mean’,’sum’,’first_only’
- **verbose** (*bool*) – whether to show progress bars or not

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 *BaseNetworkTrainer*

PyTorchNetworkTrainer

class PyTorchNetworkTrainer (*network: delira.models.abstract_network.AbstractPyTorchNetwork*, *save_path: str*, *key_mapping*, *losses=None*, *optimizer_cls=None*, *optimizer_params={}*, *train_metrics={}*, *val_metrics={}*, *lr_scheduler_cls=None*, *lr_scheduler_params={}*, *gpu_ids=[]*, *save_freq=1*, *optim_fn=<function create_optims_default_pytorch>*, *logging_type='tensorboardx'*, *logging_kwargs={}*, *fold=0*, *callbacks=[]*, *start_epoch=1*, *metric_keys=None*, *convert_batch_to_npy_fn=<function convert_torch_tensor_to_npy>*, *mixed_precision=False*, *mixed_precision_kwargs={'allow_banned': False, 'enable_caching': True, 'verbose': False}*, *criterions=None*, *val_freq=1*, ***kwargs*)

Bases: delira.training.base_trainer.BaseNetworkTrainer

Train and Validate a Network

See also:

AbstractNetwork

_BaseNetworkTrainer__KEYS_TO_GUARD = ['use_gpu', 'input_device', 'output_device', '_ca

_Predictor__KEYS_TO_GUARD = []

static _Predictor__concatenate_dict_items (*dict_like: dict*)
Function to recursively concatenate dict-items

Parameters **dict_like** (*dict*) – the (nested) dict, whoose items should be concatenated

static _Predictor__convert_dict (*old_dict*, *new_dict*)
Function to recursively convert dicts

Parameters

- **old_dict** (*dict*) – the old nested dict
- **new_dict** (*dict*) – the new nested dict

Returns the updated new nested dict

Return type dict

_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

_reinitialize_logging(*logging_type*, *logging_kwargs: dict*)

static _search_for_prev_state(*path*, *extensions=[]*)
Helper function to search in a given path for previous epoch states (indicated by extensions)

Parameters

- **path** (*str*) – the path to search in
- **extensions** (*list*) – list of strings containing valid file extensions for checkpoint files

Returns

- *str* – the file containing the latest checkpoint (if available)
- *None* – if no latst checkpoint was found
- *int* – the latest epoch (1 if no checkpoint was found)

_setup(*network*, *optim_fn*, *optimizer_cls*, *optimizer_params*, *lr_scheduler_cls*, *lr_scheduler_params*, *gpu_ids*, *key_mapping*, *convert_batch_to_npy_fn*, *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
- **convert_batch_to_npy_fn** (*type*) – function converting a batch-tensor to numpy
- **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: batchgenerators.dataloading.MultiThreadedAugmenter*, *epoch*, *verbose=False*)

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 *PyTorchNetworkTrainer*

static calc_metrics(*batch: delira.utils.config.LookupConfig*, *metrics={}*, *metric_keys=None*)

Compute metrics

Parameters

- **batch** (*LookupConfig*) – dictionary containing the whole batch (including predictions)
- **metrics** (*dict*) – dict with metrics

- **metric_keys** (*dict*) – dict of tuples which contains hashables for specifying the items to use for calculating the respective metric. If not specified for a metric, the keys "pred" and "label" are used per default

Returns dict with metric results

Return type dict

property fold
Get current fold

Returns current fold

Return type int

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

Parameters

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

Returns new state

Return type dict

predict(*data: dict*, ***kwargs*)
Predict single batch Returns the predictions corresponding to the given data obtained by the model

Parameters

- **data** (*dict*) – batch dictionary
- ****kwargs** – keyword arguments(directly passed to `prepare_batch`)

Returns predicted data

Return type dict

predict_data_mgr(*datamgr*, *batchsize=None*, *metrics={}*, *metric_keys={}*, *verbose=False*, ***kwargs*)
Defines a routine to predict data obtained from a batchgenerator

Parameters

- **datamgr** (`BaseDataManager`) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the `batch_dict` items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- ****kwargs** – additional keyword arguments

Returns

- *dict* – predictions
- *dict* – calculated metrics

predict_data_mgr_cache(*datamgr*, *batchsize=None*, *metrics={}*, *metric_keys=None*, *verbose=False*, *cache_preds=False*, ***kwargs*)

Defines a routine to predict data obtained from a batchgenerator and caches all predictions and metrics (yields them in dicts)

Parameters

- **datamgr** (`BaseDataManager`) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the `batch_dict` items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- **kwargs** – keyword arguments passed to `prepare_batch_fn()`

Yields

- *dict* – a dictionary containing all validation metrics (maybe empty)
- *dict* – a dictionary containing all predictions; If `cache_preds=True`

Warning: Since this function caches all metrics and may additionally cache all predictions (based on the argument `cache_preds`), this may result in huge memory consumption. If you are running out of memory, please have a look at *Predictor.predict_data_mgr_cache_metrics_only()* or *Predictor.predict_data_mgr()* or consider setting `cache_preds` to `False` (if not done already)

predict_data_mgr_cache_all(*datamgr*, *batchsize=None*, *metrics={}*, *metric_keys=None*, *verbose=False*, ***kwargs*)

Defines a routine to predict data obtained from a batchgenerator and caches all predictions and metrics (yields them in dicts)

Parameters

- **datamgr** (`BaseDataManager`) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the `batch_dict` items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- **kwargs** – keyword arguments passed to `prepare_batch_fn()`

Yields

- *dict* – a dictionary containing all predictions;
- *dict* – a dictionary containing all validation metrics (maybe empty)

Warning: Since this function caches all predictions and metrics, this may result in huge memory consumption. If you are running out of memory, please have a look at *Predictor.predict_data_mgr_cache_metrics_only()* or *Predictor.predict_data_mgr()*

predict_data_mgr_cache_metrics_only(*datamgr*, *batchsize=None*, *metrics={}*, *metric_keys=None*, *verbose=False*, ***kwargs*)
Defines a routine to predict data obtained from a batchgenerator and caches the metrics

Parameters

- **datamgr** (BaseDataManager) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the batch_dict items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- **kwargs** – keyword arguments passed to prepare_batch_fn()

Yields *dict* – a dictionary containing all validation metrics (maybe empty)

Notes

This function stores each prediction temporarily for metric calculation; This results in a (typically) way lower memory consumption than *Predictor.predict_data_mgr_cache_all()*, but still caches the metrics. If this is not desired, it is recommended to use *Predictor.predict_data_mgr()* and iterate over the generator as this only produces per-batch metrics and predictions and does not cache anything by default

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

train(*num_epochs*, *datamgr_train*, *datamgr_valid=None*, *val_score_key=None*, *val_score_mode='highest'*, *reduce_mode='mean'*, *verbose=True*)
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
- **reduce_mode** (*str*) – 'mean','sum','first_only'
- **verbose** (*bool*) – whether to show progress bars or not

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 *BaseNetworkTrainer*

TfNetworkTrainer

class TfNetworkTrainer(*network*, *save_path*, *key_mapping*, *losses: dict*, *optimizer_cls*, *optimizer_params={}*, *train_metrics={}*, *val_metrics={}*, *lr_scheduler_cls=None*, *lr_scheduler_params={}*, *gpu_ids=[]*, *save_freq=1*, *optim_fn=<function create_optims_default_tf>*, *logging_type='tensorboardx'*, *logging_kwargs={}*, *fold=0*, *callbacks=[]*, *start_epoch=1*, *metric_keys=None*, *convert_batch_to_npy_fn=<function convert_tf_tensor_to_npy>*, *val_freq=1*, ***kwargs*)
Bases: delira.training.base_trainer.BaseNetworkTrainer

Train and Validate a Network

See also:

AbstractNetwork

_BaseNetworkTrainer__KEYS_TO_GUARD = ['use_gpu', 'input_device', 'output_device', '_ca

_Predictor__KEYS_TO_GUARD = []

static _Predictor__concatenate_dict_items(*dict_like: dict*)
Function to recursively concatenate dict-items

Parameters **dict_like** (*dict*) – the (nested) dict, whoose items should be concatenated

static _Predictor__convert_dict(*old_dict*, *new_dict*)
Function to recursively convert dicts

Parameters

- **old_dict** (*dict*) – the old nested dict
- **new_dict** (*dict*) – the new nested dict

Returns the updated new nested dict

Return type dict

_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 the behaviour at beginnig of the training

Parameters

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

Raises **NotImplementedError** – If not overwritten by subclass

_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

_reinitialize_logging(*logging_type*, *logging_kwargs: dict*)

static _search_for_prev_state(*path*, *extensions=[]*)
Helper function to search in a given path for previous epoch states (indicated by extensions)

Parameters

- **path** (*str*) – the path to search in

- **extensions** (*list*) – list of strings containing valid file extensions for checkpoint files

Returns

- *str* – the file containing the latest checkpoint (if available)
- *None* – if no latst checkpoint was found
- *int* – the latest epoch (1 if no checkpoint was found)

_setup(*network*, *optim_fn*, *optimizer_cls*, *optimizer_params*, *lr_scheduler_cls*, *lr_scheduler_params*, *key_mapping*, *convert_batch_to_npy_fn*, *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
- **convert_batch_to_npy_fn** (*type, optional*) – function converting a batch-tensor to numpy, per default this is the identity function
- **gpu_ids** (*list*) – list containing ids of GPUs to use; if empty: use cpu instead

_train_single_epoch(*batchgen: batchgenerators.dataloading.MultiThreadedAugmenter*, *epoch*, *verbose=False*)

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 *BaseNetworkTrainer*

static calc_metrics(*batch: delira.utils.config.LookupConfig*, *metrics={}*, *metric_keys=None*)

Compute metrics

Parameters

- **batch** (LookupConfig) – dictionary containing the whole batch (including predictions)
- **metrics** (*dict*) – dict with metrics
- **metric_keys** (*dict*) – dict of tuples which contains hashables for specifying the items to use for calculating the respective metric. If not specified for a metric, the keys "pred" and "label" are used per default

Returns dict with metric results

Return type dict

property fold
Get current fold

Returns current fold

Return type int

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

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

predict(*data: dict*, ***kwargs*)
Predict single batch Returns the predictions corresponding to the given data obtained by the model

Parameters

- **data** (*dict*) – batch dictionary
- ****kwargs** – keyword arguments(directly passed to `prepare_batch`)

Returns predicted data

Return type dict

predict_data_mgr(*datamgr*, *batch_size=None*, *metrics={}*, *metric_keys={}*, *verbose=False*, ***kwargs*)
Defines a routine to predict data obtained from a batchgenerator

Parameters

- **datamgr** (`BaseDataManager`) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the `batch_dict` items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- ****kwargs** – additional keword arguments

predict_data_mgr_cache(*datamgr*, *batchsize=None*, *metrics={}*, *metric_keys=None*, *verbose=False*, *cache_preds=False*, ***kwargs*)
Defines a routine to predict data obtained from a batchgenerator and caches all predictions and metrics (yields them in dicts)

Parameters

- **datamgr** (`BaseDataManager`) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the `batch_dict` items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- **kwargs** – keyword arguments passed to `prepare_batch_fn()`

Yields

- *dict* – a dictionary containing all validation metrics (maybe empty)
- *dict* – a dictionary containing all predictions; If `cache_preds=True`

Warning: Since this function caches all metrics and may additionally cache all predictions (based on the argument `cache_preds`), this may result in huge memory consumption. If you are running out of memory, please have a look at *Predictor.predict_data_mgr_cache_metrics_only()* or *Predictor.predict_data_mgr()* or consider setting `cache_preds` to `False` (if not done already)

predict_data_mgr_cache_all(*datamgr*, *batchsize=None*, *metrics={}*, *metric_keys=None*, *verbose=False*, ***kwargs*)

Defines a routine to predict data obtained from a batchgenerator and caches all predictions and metrics (yields them in dicts)

Parameters

- **datamgr** (`BaseDataManager`) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the `batch_dict` items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- **kwargs** – keyword arguments passed to `prepare_batch_fn()`

Yields

- *dict* – a dictionary containing all predictions;
- *dict* – a dictionary containing all validation metrics (maybe empty)

Warning: Since this function caches all predictions and metrics, this may result in huge memory consumption. If you are running out of memory, please have a look at *Predictor.predict_data_mgr_cache_metrics_only()* or *Predictor.predict_data_mgr()*

predict_data_mgr_cache_metrics_only(*datamgr*, *batchsize=None*, *metrics={}*, *metric_keys=None*, *verbose=False*, ***kwargs*)

Defines a routine to predict data obtained from a batchgenerator and caches the metrics

Parameters

- **datamgr** (`BaseDataManager`) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the `batch_dict` items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- **kwargs** – keyword arguments passed to `prepare_batch_fn()`

Yields *dict* – a dictionary containing all validation metrics (maybe empty)

Notes

This function stores each prediction temporarily for metric calculation; This results in a (typically) way lower memory consumption than *Predictor.predict_data_mgr_cache_all()*, but still caches the metrics. If this is not desired, it is recommended to use *Predictor.predict_data_mgr()* and iterate over the generator as this only produces per-batch metrics and predictions and does not cache anything by default

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 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'*, *reduce_mode='mean'*, *verbose=True*)
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
- **reduce_mode** (*str*) – 'mean','sum','first_only'
- **verbose** (*bool*) – whether to show progress bars or not

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 *BaseNetworkTrainer*

Predictor

The predictor implements the basic prediction and metric calculation routines and can be subclassed for special routines.

It is also the baseclass of all the trainers which extend it's functionality by training routines

Predictor

class Predictor(*model*, *key_mapping: dict*, *convert_batch_to_npy_fn=<function convert_batch_to_numpy_identity>*, *prepare_batch_fn=<function Predictor.<lambda>>*, ***kwargs*)

Bases: `object`

Defines an API for Predictions from a Network

See also:

PyTorchNetworkTrainer

_Predictor__KEYS_TO_GUARD = []

static _Predictor__concatenate_dict_items(*dict_like: dict*)

Function to recursively concatenate dict-items

Parameters **dict_like** (*dict*) – the (nested) dict, whoose items should be concatenated

static _Predictor__convert_dict(*old_dict*, *new_dict*)

Function to recursively convert dicts

Parameters

- **old_dict** (*dict*) – the old nested dict
- **new_dict** (*dict*) – the new nested dict

Returns the updated new nested dict

Return type dict

_setup(*network*, *key_mapping*, *convert_batch_args_kwargs_to_npy_fn*, *prepare_batch_fn*, ***kwargs*)

Parameters

- **network** (`AbstractNetwork`) – the network to predict from
- **key_mapping** (*dict*) – a dictionary containing the mapping from the `data_dict` to the actual model's inputs. E.g. if a model accepts one input named 'x' and the data_dict contains one entry named 'data' this argument would have to be `{'x': 'data'}`
- **convert_batch_to_npy_fn** (*type*) – a callable function to convert tensors in positional and keyword arguments to numpy
- **prepare_batch_fn** (*type*) – function converting a batch-tensor to the framework specific tensor-type and pushing it to correct device, default: identity function

static calc_metrics(*batch: delira.utils.config.LookupConfig*, *metrics={}*, *metric_keys=None*)

Compute metrics

Parameters

- **batch** (`LookupConfig`) – dictionary containing the whole batch (including predictions)
- **metrics** (*dict*) – dict with metrics
- **metric_keys** (*dict*) – dict of tuples which contains hashables for specifying the items to use for calculating the respective metric. If not specified for a metric, the keys "pred" and "label" are used per default

Returns dict with metric results

Return type dict

predict(*data: dict*, ***kwargs*)

Predict single batch Returns the predictions corresponding to the given data obtained by the model

Parameters

- **data** (*dict*) – batch dictionary
- ****kwargs** – keyword arguments(directly passed to `prepare_batch`)

Returns predicted data

Return type dict

predict_data_mgr(*datamgr*, *batchsize=None*, *metrics={}*, *metric_keys=None*, *verbose=False*, ***kwargs*)

Defines a routine to predict data obtained from a batchgenerator without explicitly caching anything

Parameters

- **datamgr** (`BaseDataManager`) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the `batch_dict` items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- **kwargs** – keyword arguments passed to `prepare_batch_fn()`

Yields

- *dict* – a dictionary containing all predictions of the current batch
- *dict* – a dictionary containing all metrics of the current batch

predict_data_mgr_cache(*datamgr*, *batchsize=None*, *metrics={}*, *metric_keys=None*, *verbose=False*, *cache_preds=False*, ***kwargs*)

Defines a routine to predict data obtained from a batchgenerator and caches all predictions and metrics (yields them in dicts)

Parameters

- **datamgr** (`BaseDataManager`) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the `batch_dict` items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- **kwargs** – keyword arguments passed to `prepare_batch_fn()`

Yields

- *dict* – a dictionary containing all validation metrics (maybe empty)
- *dict* – a dictionary containing all predictions; If `cache_preds=True`

Warning: Since this function caches all metrics and may additionally cache all predictions (based on the argument `cache_preds`), this may result in huge memory consumption. If you are running out of memory, please have a look at *Predictor.predict_data_mgr_cache_metrics_only()* or *Predictor.predict_data_mgr()* or consider setting `cache_preds` to `False` (if not done already)

predict_data_mgr_cache_all(*datamgr*, *batchsize=None*, *metrics={}*, *metric_keys=None*, *verbose=False*, ***kwargs*)

Defines a routine to predict data obtained from a batchgenerator and caches all predictions and metrics (yields them in dicts)

Parameters

- **datamgr** (`BaseDataManager`) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the `batch_dict` items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- **kwargs** – keyword arguments passed to `prepare_batch_fn()`

Yields

- *dict* – a dictionary containing all predictions;
- *dict* – a dictionary containing all validation metrics (maybe empty)

Warning: Since this function caches all predictions and metrics, this may result in huge memory consumption. If you are running out of memory, please have a look at *Predictor.predict_data_mgr_cache_metrics_only()* or *Predictor.predict_data_mgr()*

predict_data_mgr_cache_metrics_only(*datamgr*, *batchsize=None*, *metrics={}*, *metric_keys=None*, *verbose=False*, ***kwargs*)

Defines a routine to predict data obtained from a batchgenerator and caches the metrics

Parameters

- **datamgr** (`BaseDataManager`) – Manager producing a generator holding the batches
- **batchsize** (*int*) – Artificial batchsize (sampling will be done with batchsize 1 and sampled data will be stacked to match the artificial batchsize)(default: None)
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict*) – the `batch_dict` items to use for metric calculation
- **verbose** (*bool*) – whether to show a progress-bar or not, default: False
- **kwargs** – keyword arguments passed to `prepare_batch_fn()`

Yields *dict* – a dictionary containing all validation metrics (maybe empty)

Notes

This function stores each prediction temporarily for metric calculation; This results in a (typically) way lower memory consumption than *Predictor.predict_data_mgr_cache_all()*, but still caches the metrics. If this is not desired, it is recommended to use *Predictor.predict_data_mgr()* and iterate over the generator as this only produces per-batch metrics and predictions and does not cache anything by default

Experiments

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

BaseExperiment

class BaseExperiment(*params: Union[str, delira.training.parameters.Parameters], model_cls: delira.models.abstract_network.AbstractNetwork, n_epochs=None, name=None, save_path=None, key_mapping=None, val_score_key=None, optim_builder=None, checkpoint_freq=1, trainer_cls=<class 'delira.training.base_trainer.BaseNetworkTrainer'>, predictor_cls=<class 'delira.training.predictor.Predictor'>, **kwargs*)

Bases: object

Baseclass for Experiments.

Implements:

- **Setup-Behavior for Models, Trainers and Predictors (depending on train** and test case)
- The K-Fold logic (including stratified and random splitting)
- Argument Handling

_resolve_kwargs(*kwargs: Optional[dict]*)

Merges given kwargs with `self.kwargs` If same argument is present in both kwargs, the one from the given kwargs will be used here

Parameters **kwargs** (*dict*) – the given kwargs to merge with self.kwargs

Returns merged kwargs

Return type dict

_resolve_params(*params: Optional[delira.training.parameters.Parameters]*)

Merges the given params with `self.params`. If the same argument is given in both params, the one from the currently given parameters is used here

Parameters **params** (*Parameters* or None) – the parameters to merge with `self.params`

Returns the merged parameter instance

Return type *Parameters*

_setup_test(*params*, *model*, *convert_batch_to_npy_fn*, *prepare_batch_fn*, ***kwargs*)

Parameters

- **params** (*Parameters*) – the parameters containing the model and training kwargs (ignored here, just passed for subclassing and unified API)
- **model** (`AbstractNetwork`) – the model to test

- **convert_batch_to_npy_fn** (*function*) – function to convert a batch of tensors to numpy
- **prepare_batch_fn** (*function*) – function to convert a batch-dict to a format accepted by the model. This conversion typically includes dtype-conversion, reshaping, wrapping to backend-specific tensors and pushing to correct devices
- ****kwargs** – additional keyword arguments

Returns the created predictor

Return type *Predictor*

_setup_training(*params*, ***kwargs*)
Handles the setup for training case

Parameters

- **params** (*Parameters*) – the parameters containing the model and training kwargs
- ****kwargs** – additional keyword arguments

Returns the created trainer

Return type *BaseNetworkTrainer*

kfold(*data: delira.data_loading.data_manager.BaseDataManager*, *metrics: dict*, *num_epochs=None*, *num_splits=None*, *shuffle=False*, *random_seed=None*, *split_type='random'*, *val_split=0.2*, *label_key='label'*, *train_kwargs: dict = None*, *metric_keys: dict = None*, *test_kwargs: dict = None*, *params=None*, *verbose=False*, ***kwargs*)
Performs a k-Fold cross-validation

Parameters

- **data** (BaseDataManager) – the data to use for training(, validation) and testing. Will be split based on `split_type` and `val_split`
- **metrics** (*dict*) – dictionary containing the metrics to evaluate during k-fold
- **num_epochs** (*int or None*) – number of epochs to train (if not given, will either be extracted from `params`, `self.parms` or `self.n_epochs`)
- **num_splits** (*int or None*) – the number of splits to extract from `data`. If None: uses a default of 10
- **shuffle** (*bool*) – whether to shuffle the data before splitting or not (implemented by index-shuffling rather than actual data-shuffling to retain potentially lazy-behavior of datasets)
- **random_seed** (*None*) – seed to seed numpy, the splitting functions and the used backend-framework
- **split_type** (*str*) – must be one of ['random', 'stratified'] if 'random': uses random data splitting if 'stratified': uses stratified data splitting. Stratification will be based on `label_key`
- **val_split** (*float or None*) – the fraction of the train data to use as validation set. If None: No validation will be done during training; only testing for each fold after the training is complete
- **label_key** (*str*) – the label to use for stratification. Will be ignored unless `split_type` is 'stratified'. Default: 'label'
- **train_kwargs** (*dict or None*) – kwargs to update the behavior of the `BaseDataManager` containing the train data. If None: empty dict will be passed

- **metric_keys** (*dict of tuples*) – the batch_dict keys to use for each metric to calculate. Should contain a value for each key in `metrics`. If no values are given for a key, per default `pred` and `label` will be used for metric calculation
- **test_kwargs** (*dict or None*) – kwargs to update the behavior of the `BaseDataManager` containing the test and validation data. If None: empty dict will be passed
- **params** (:class:'Parameters'or None) – the training and model parameters (will be merged with `self.params`)
- **verbose** (*bool*) – verbosity
- ****kwargs** – additional keyword arguments

Returns

- *dict* – all predictions from all folds
- *dict* – all metric values from all folds

Raises **ValueError** – if `split_type` is neither 'random', nor 'stratified'

See also:

- `sklearn.model_selection.KFold`

and `sklearn.model_selection.ShuffleSplit` for random data-splitting

- `sklearn.model_selection.StratifiedKFold`

and `sklearn.model_selection.StratifiedShuffleSplit` for stratified data-splitting

- `BaseDataManager.update_from_state_dict()` for updating the

data managers by kwargs

- *BaseExperiment.run()* for the training
- *BaseExperiment.test()* for the testing

Notes

using stratified splits may be slow during split-calculation, since each item must be loaded once to obtain the labels necessary for stratification.

static load(*file_name*)
Loads whole experiment

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

resume(*save_path: str, train_data: delira.data_loading.data_manager.BaseDataManager, val_data: delira.data_loading.data_manager.BaseDataManager = None, params: delira.training.parameters.Parameters = None, **kwargs*)
Resumes a previous training by passing an explicit `save_path` instead of generating a new one

Parameters

- **save_path** (*str*) – path to previous training
- **train_data** (`BaseDataManager`) – the data to use for training
- **val_data** (`BaseDataManager` or None) – the data to use for validation (no validation is done if passing None); default: None

- **params** (*Parameters* or None) – the parameters to use for training and model instantiation (will be merged with `self.params`)
- ****kwargs** – additional keyword arguments

Returns The trained network returned by the trainer (usually best network)

Return type `AbstractNetwork`

See also:

BaseNetworkTrainer

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

Parameters

- **train_data** (`BaseDataManager`) – the data to use for training
- **val_data** (`BaseDataManager` or None) – the data to use for validation (no validation is done if passing None); default: None
- **params** (*Parameters* or None) – the parameters to use for training and model instantiation (will be merged with `self.params`)
- ****kwargs** – additional keyword arguments

Returns The trained network returned by the trainer (usually best network)

Return type `AbstractNetwork`

See also:

BaseNetworkTrainer

save()
Saves the Whole experiments

setup(*params*, *training=True*, ***kwargs*)
Defines the setup behavior (model, trainer etc.) for training and testing case

Parameters

- **params** (*Parameters*) – the parameters to use for setup
- **training** (*bool*) – whether to setup for training case or for testing case
- ****kwargs** – additional keyword arguments

Returns

- *BaseNetworkTrainer* – the created trainer (if `training=True`)
- *Predictor* – the created predictor (if `training=False`)

See also:

- *BaseExperiment._setup_training()* for training setup
- *BaseExperiment._setup_test()* for test setup

test(*network*, *test_data: delira.data_loading.data_manager.BaseDataManager*, *metrics: dict*, *metric_keys=None*, *verbose=False*, *prepare_batch=<function BaseExperiment.<lambda>>*, *convert_fn=<function BaseExperiment.<lambda>>*, ***kwargs*)
Setup and run testing on a given network

Parameters

- **network** (`AbstractNetwork`) – the (trained) network to test
- **test_data** (`BaseDataManager`) – the data to use for testing
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict of tuples*) – the batch_dict keys to use for each metric to calculate. Should contain a value for each key in `metrics`. If no values are given for a key, per default `pred` and `label` will be used for metric calculation
- **verbose** (*bool*) – verbosity of the test process
- **prepare_batch** (*function*) – function to convert a batch-dict to a format accepted by the model. This conversion typically includes dtype-conversion, reshaping, wrapping to backend-specific tensors and pushing to correct devices
- **convert_fn** (*function*) – function to convert a batch of tensors to numpy
- ****kwargs** – additional keyword arguments

Returns

- *dict* – all predictions obtained by feeding the `test_data` through the `network`
- *dict* – all metrics calculated upon the `test_data` and the obtained predictions

PyTorchExperiment

class PyTorchExperiment(*params: Union[str, delira.training.parameters.Parameters]*, *model_cls: delira.models.abstract_network.AbstractPyTorchNetwork*, *n_epochs=None*, *name=None*, *save_path=None*, *key_mapping=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.BaseExperiment`

_resolve_kwargs(*kwargs: Optional[dict]*)
Merges given kwargs with `self.kwargs` If same argument is present in both kwargs, the one from the given kwargs will be used here

Parameters **kwargs** (*dict*) – the given kwargs to merge with self.kwargs

Returns merged kwargs

Return type dict

_resolve_params(*params: Optional[delira.training.parameters.Parameters]*)
Merges the given params with `self.params`. If the same argument is given in both params, the one from the currently given parameters is used here

Parameters **params** (*Parameters* or None) – the parameters to merge with `self.params`

Returns the merged parameter instance

Return type *Parameters*

_setup_test(*params*, *model*, *convert_batch_to_npy_fn*, *prepare_batch_fn*, ***kwargs*)

Parameters

- **params** (*Parameters*) – the parameters containing the model and training kwargs (ignored here, just passed for subclassing and unified API)
- **model** (AbstractNetwork) – the model to test
- **convert_batch_to_npy_fn** (*function*) – function to convert a batch of tensors to numpy
- **prepare_batch_fn** (*function*) – function to convert a batch-dict to a format accepted by the model. This conversion typically includes dtype-conversion, reshaping, wrapping to backend-specific tensors and pushing to correct devices
- ****kwargs** – additional keyword arguments

Returns the created predictor

Return type *Predictor*

_setup_training(*params*, ***kwargs*)

Handles the setup for training case

Parameters

- **params** (*Parameters*) – the parameters containing the model and training kwargs
- ****kwargs** – additional keyword arguments

Returns the created trainer

Return type *BaseNetworkTrainer*

kfold(*data: delira.data_loading.data_manager.BaseDataManager*, *metrics: dict*, *num_epochs=None*, *num_splits=None*, *shuffle=False*, *random_seed=None*, *split_type='random'*, *val_split=0.2*, *label_key='label'*, *train_kwargs: dict = None*, *test_kwargs: dict = None*, *metric_keys: dict = None*, *params=None*, *verbose=False*, ***kwargs*)

Performs a k-Fold cross-validation

Parameters

- **data** (BaseDataManager) – the data to use for training(, validation) and testing. Will be split based on split_type and val_split
- **metrics** (*dict*) – dictionary containing the metrics to evaluate during k-fold
- **num_epochs** (*int or None*) – number of epochs to train (if not given, will either be extracted from params, self.parms or self.n_epochs)
- **num_splits** (*int or None*) – the number of splits to extract from data. If None: uses a default of 10
- **shuffle** (*bool*) – whether to shuffle the data before splitting or not (implemented by index-shuffling rather than actual data-shuffling to retain potentially lazy-behavior of datasets)
- **random_seed** (*None*) – seed to seed numpy, the splitting functions and the used backend-framework
- **split_type** (*str*) – must be one of ['random', 'stratified'] if 'random': uses random data splitting if 'stratified': uses stratified data splitting. Stratification will be based on label_key
- **val_split** (*float or None*) – the fraction of the train data to use as validation set. If None: No validation will be done during training; only testing for each fold after the training is complete

- **label_key** (*str*) – the label to use for stratification. Will be ignored unless `split_type` is 'stratified'. Default: 'label'
- **train_kwargs** (*dict or None*) – kwargs to update the behavior of the `BaseDataManager` containing the train data. If None: empty dict will be passed
- **metric_keys** (*dict of tuples*) – the batch_dict keys to use for each metric to calculate. Should contain a value for each key in `metrics`. If no values are given for a key, per default `pred` and `label` will be used for metric calculation
- **test_kwargs** (*dict or None*) – kwargs to update the behavior of the `BaseDataManager` containing the test and validation data. If None: empty dict will be passed
- **params** (:class:'Parameters'or None) – the training and model parameters (will be merged with `self.params`)
- **verbose** (*bool*) – verbosity
- ****kwargs** – additional keyword arguments

Returns

- *dict* – all predictions from all folds
- *dict* – all metric values from all folds

Raises **ValueError** – if `split_type` is neither 'random', nor 'stratified'

See also:

- `sklearn.model_selection.KFold`

and `sklearn.model_selection.ShuffleSplit` for random data-splitting

- `sklearn.model_selection.StratifiedKFold`

and `sklearn.model_selection.StratifiedShuffleSplit` for stratified data-splitting

- `BaseDataManager.update_from_state_dict()` for updating the

data managers by kwargs

- *BaseExperiment.run()* for the training
- *BaseExperiment.test()* for the testing

Notes

using stratified splits may be slow during split-calculation, since each item must be loaded once to obtain the labels necessary for stratification.

static load(*file_name*)
Loads whole experiment

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

resume(*save_path: str, train_data: delira.data_loading.data_manager.BaseDataManager, val_data: delira.data_loading.data_manager.BaseDataManager = None, params: delira.training.parameters.Parameters = None, **kwargs*)
Resumes a previous training by passing an explicit `save_path` instead of generating a new one

Parameters

- **save_path** (*str*) – path to previous training
- **train_data** (BaseDataManager) – the data to use for training
- **val_data** (BaseDataManager or None) – the data to use for validation (no validation is done if passing None); default: None
- **params** (*Parameters* or None) – the parameters to use for training and model instantiation (will be merged with `self.params`)
- ****kwargs** – additional keyword arguments

Returns The trained network returned by the trainer (usually best network)

Return type AbstractNetwork

See also:

BaseNetworkTrainer

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

Parameters

- **train_data** (BaseDataManager) – the data to use for training
- **val_data** (BaseDataManager or None) – the data to use for validation (no validation is done if passing None); default: None
- **params** (*Parameters* or None) – the parameters to use for training and model instantiation (will be merged with `self.params`)
- ****kwargs** – additional keyword arguments

Returns The trained network returned by the trainer (usually best network)

Return type AbstractNetwork

See also:

BaseNetworkTrainer

save()
Saves the Whole experiments

setup(*params*, *training=True*, ***kwargs*)
Defines the setup behavior (model, trainer etc.) for training and testing case

Parameters

- **params** (*Parameters*) – the parameters to use for setup
- **training** (*bool*) – whether to setup for training case or for testing case
- ****kwargs** – additional keyword arguments

Returns

- *BaseNetworkTrainer* – the created trainer (if `training=True`)
- *Predictor* – the created predictor (if `training=False`)

See also:

- *BaseExperiment._setup_training()* for training setup

- *BaseExperiment._setup_test()* for test setup

test(*network*, *test_data: delira.data_loading.data_manager.BaseDataManager*, *metrics: dict*, *metric_keys=None*, *verbose=False*, *prepare_batch=None*, *convert_fn=None*, ***kwargs*)
Setup and run testing on a given network

Parameters

- **network** (AbstractNetwork) – the (trained) network to test
- **test_data** (BaseDataManager) – the data to use for testing
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict of tuples*) – the batch_dict keys to use for each metric to calculate. Should contain a value for each key in metrics. If no values are given for a key, per default pred and label

  will be used for metric calculation
- **verbose** (*bool*) – verbosity of the test process
- **prepare_batch** (*function*) – function to convert a batch-dict to a format accepted by the model. This conversion typically includes dtype-conversion, reshaping, wrapping to backend-specific tensors and pushing to correct devices. If not further specified uses the network's prepare_batch with CPU devices
- **convert_fn** (*function*) – function to convert a batch of tensors to numpy if not specified defaults to convert_torch_tensor_to_npy()
- ****kwargs** – additional keyword arguments

Returns

- *dict* – all predictions obtained by feeding the test_data through the network
- *dict* – all metrics calculated upon the test_data and the obtained predictions

TfExperiment

class TfExperiment(*params: Union[str, delira.training.parameters.Parameters]*, *model_cls: delira.models.abstract_network.AbstractTfNetwork*, *n_epochs=None*, *name=None*, *save_path=None*, *key_mapping=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.BaseExperiment

_resolve_kwargs(*kwargs: Optional[dict]*)
Merges given kwargs with self.kwargs If same argument is present in both kwargs, the one from the given kwargs will be used here

Parameters **kwargs** (*dict*) – the given kwargs to merge with self.kwargs

Returns merged kwargs

Return type dict

_resolve_params(*params: Optional[delira.training.parameters.Parameters]*)
Merges the given params with self.params. If the same argument is given in both params, the one from the currently given parameters is used here

Parameters **params** (*Parameters* or None) – the parameters to merge with self.params

Returns the merged parameter instance

Return type *Parameters*

_setup_test(*params*, *model*, *convert_batch_to_npy_fn*, *prepare_batch_fn*, ***kwargs*)

Parameters

- **params** (*Parameters*) – the parameters containing the model and training kwargs (ignored here, just passed for subclassing and unified API)
- **model** (AbstractNetwork) – the model to test
- **convert_batch_to_npy_fn** (*function*) – function to convert a batch of tensors to numpy
- **prepare_batch_fn** (*function*) – function to convert a batch-dict to a format accepted by the model. This conversion typically includes dtype-conversion, reshaping, wrapping to backend-specific tensors and pushing to correct devices
- ****kwargs** – additional keyword arguments

Returns the created predictor

Return type *Predictor*

_setup_training(*params*, ***kwargs*)

Handles the setup for training case

Parameters

- **params** (*Parameters*) – the parameters containing the model and training kwargs
- ****kwargs** – additional keyword arguments

Returns the created trainer

Return type *BaseNetworkTrainer*

kfold(*data: delira.data_loading.data_manager.BaseDataManager*, *metrics: dict*, *num_epochs=None*, *num_splits=None*, *shuffle=False*, *random_seed=None*, *split_type='random'*, *val_split=0.2*, *label_key='label'*, *train_kwargs: dict = None*, *test_kwargs: dict = None*, *metric_keys: dict = None*, *params=None*, *verbose=False*, ***kwargs*)

Performs a k-Fold cross-validation

Parameters

- **data** (BaseDataManager) – the data to use for training(, validation) and testing. Will be split based on split_type and val_split
- **metrics** (*dict*) – dictionary containing the metrics to evaluate during k-fold
- **num_epochs** (*int or None*) – number of epochs to train (if not given, will either be extracted from params, self.parms or self.n_epochs)
- **num_splits** (*int or None*) – the number of splits to extract from data. If None: uses a default of 10
- **shuffle** (*bool*) – whether to shuffle the data before splitting or not (implemented by index-shuffling rather than actual data-shuffling to retain potentially lazy-behavior of datasets)
- **random_seed** (*None*) – seed to seed numpy, the splitting functions and the used backend-framework

- **split_type** (*str*) – must be one of ['random', 'stratified'] if 'random': uses random data splitting if 'stratified': uses stratified data splitting. Stratification will be based on `label_key`
- **val_split** (*float or None*) – the fraction of the train data to use as validation set. If None: No validation will be done during training; only testing for each fold after the training is complete
- **label_key** (*str*) – the label to use for stratification. Will be ignored unless `split_type` is 'stratified'. Default: 'label'
- **train_kwargs** (*dict or None*) – kwargs to update the behavior of the `BaseDataManager` containing the train data. If None: empty dict will be passed
- **metric_keys** (*dict of tuples*) – the batch_dict keys to use for each metric to calculate. Should contain a value for each key in `metrics`. If no values are given for a key, per default `pred` and `label` will be used for metric calculation
- **test_kwargs** (*dict or None*) – kwargs to update the behavior of the `BaseDataManager` containing the test and validation data. If None: empty dict will be passed
- **params** (:class:'Parameters'or None) – the training and model parameters (will be merged with `self.params`)
- **verbose** (*bool*) – verbosity
- ****kwargs** – additional keyword arguments

Returns

- *dict* – all predictions from all folds
- *dict* – all metric values from all folds

Raises **ValueError** – if `split_type` is neither 'random', nor 'stratified'

See also:

- `sklearn.model_selection.KFold`

and `sklearn.model_selection.ShuffleSplit` for random data-splitting

- `sklearn.model_selection.StratifiedKFold`

and `sklearn.model_selection.StratifiedShuffleSplit` for stratified data-splitting

- `BaseDataManager.update_from_state_dict()` for updating the

data managers by kwargs

- *BaseExperiment.run()* for the training
- *BaseExperiment.test()* for the testing

Notes

using stratified splits may be slow during split-calculation, since each item must be loaded once to obtain the labels necessary for stratification.

static load(*file_name*)

Loads whole experiment

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

resume(*save_path: str*, *train_data: delira.data_loading.data_manager.BaseDataManager*, *val_data: delira.data_loading.data_manager.BaseDataManager = None*, *params: delira.training.parameters.Parameters = None*, ***kwargs*)
Resumes a previous training by passing an explicit `save_path` instead of generating a new one

Parameters

- **save_path** (*str*) – path to previous training
- **train_data** (`BaseDataManager`) – the data to use for training
- **val_data** (`BaseDataManager` or None) – the data to use for validation (no validation is done if passing None); default: None
- **params** (*Parameters* or None) – the parameters to use for training and model instantiation (will be merged with `self.params`)
- ****kwargs** – additional keyword arguments

Returns The trained network returned by the trainer (usually best network)

Return type `AbstractNetwork`

See also:

BaseNetworkTrainer

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

Parameters

- **train_data** (`BaseDataManager`) – the data to use for training
- **val_data** (`BaseDataManager` or None) – the data to use for validation (no validation is done if passing None); default: None
- **params** (*Parameters* or None) – the parameters to use for training and model instantiation (will be merged with `self.params`)
- ****kwargs** – additional keyword arguments

Returns The trained network returned by the trainer (usually best network)

Return type `AbstractNetwork`

See also:

BaseNetworkTrainer

save()
Saves the Whole experiments

setup(*params*, *training=True*, ***kwargs*)
Defines the setup behavior (model, trainer etc.) for training and testing case

Parameters

- **params** (*Parameters*) – the parameters to use for setup
- **training** (*bool*) – whether to setup for training case or for testing case
- ****kwargs** – additional keyword arguments

Returns

- *BaseNetworkTrainer* – the created trainer (if `training=True`)
- *Predictor* – the created predictor (if `training=False`)

See also:

- *BaseExperiment._setup_training()* for training setup
- *BaseExperiment._setup_test()* for test setup

test(*network*, *test_data: delira.data_loading.data_manager.BaseDataManager*, *metrics: dict*, *metric_keys=None*, *verbose=False*, *prepare_batch=<function TfExperiment.<lambda>>*, *convert_fn=None*, ***kwargs*)

Setup and run testing on a given network

Parameters

- **network** (`AbstractNetwork`) – the (trained) network to test
- **test_data** (`BaseDataManager`) – the data to use for testing
- **metrics** (*dict*) – the metrics to calculate
- **metric_keys** (*dict of tuples*) – the batch_dict keys to use for each metric to calculate. Should contain a value for each key in `metrics`. If no values are given for a key, per default `pred` and `label`

  will be used for metric calculation
- **verbose** (*bool*) – verbosity of the test process
- **prepare_batch** (*function*) – function to convert a batch-dict to a format accepted by the model. This conversion typically includes dtype-conversion, reshaping, wrapping to backend-specific tensors and pushing to correct devices. If not further specified uses the `network`'s `prepare_batch` with CPU devices
- **convert_fn** (*function*) – function to convert a batch of tensors to numpy if not specified defaults to `convert_torch_tensor_to_npy()`
- ****kwargs** – additional keyword arguments

Returns

- *dict* – all predictions obtained by feeding the `test_data` through the `network`
- *dict* – all metrics calculated upon the `test_data` and the obtained predictions

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 attributes, where the name must correspond to the trainer's attribute name

Return type dict

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 attributes, where the name must correspond to the trainer's attribute name

Return type dict

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 attributes, where the name must correspond to the trainer's attribute name

Return type dict

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 attributes, where the name must correspond to the trainer's attribute name

Return type dict

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 attributes, where the name must correspond to the trainer's attribute name

Return type dict

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 attributes, where the name must correspond to the trainer's attribute name

Return type dict

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 attributes, where the name must correspond to the trainer's attribute name

Return type dict

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 attributes, where the name must correspond to the trainer's attribute name

Return type dict

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 attributes, where the name must correspond to the trainer's attribute name

Return type dict

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 attributes, where the name must correspond to the trainer's attribute name

Return type dict

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: `torch.nn.Module`

Focal loss for binary case without(!) logit

forward(*p*, *t*)

BCEFocalLossLogitPyTorch

class BCEFocalLossLogitPyTorch(*alpha=None*, *gamma=2*, *reduction='elementwise_mean'*)
Bases: torch.nn.Module

Focal loss for binary case WITH logit

forward(*p*, *t*)

Metrics

SklearnClassificationMetric

class SklearnClassificationMetric(*score_fn*, *gt_logits=False*, *pred_logits=True*, ***kwargs*)
Bases: object

SklearnAccuracyScore

class SklearnAccuracyScore(*gt_logits=False*, *pred_logits=True*, ***kwargs*)
Bases: *delira.training.metrics.SklearnClassificationMetric*

Accuracy Metric

SklearnBalancedAccuracyScore

class SklearnBalancedAccuracyScore(*gt_logits=False*, *pred_logits=True*, ***kwargs*)
Bases: *delira.training.metrics.SklearnClassificationMetric*

Balanced Accuracy Metric

SklearnF1Score

class SklearnF1Score(*gt_logits=False*, *pred_logits=True*, ***kwargs*)
Bases: *delira.training.metrics.SklearnClassificationMetric*

F1 Score

SklearnFBetaScore

class SklearnFBetaScore(*gt_logits=False*, *pred_logits=True*, ***kwargs*)
Bases: *delira.training.metrics.SklearnClassificationMetric*

F-Beta Score (Generalized F1)

SklearnHammingLoss

class SklearnHammingLoss(*gt_logits=False*, *pred_logits=True*, ***kwargs*)
Bases: *delira.training.metrics.SklearnClassificationMetric*

Hamming Loss

SklearnJaccardSimilarityScore

class SklearnJaccardSimilarityScore(*gt_logits=False*, *pred_logits=True*, ***kwargs*)
Bases: *delira.training.metrics.SklearnClassificationMetric*

Jaccard Similarity Score

SklearnLogLoss

class SklearnLogLoss(*gt_logits=False*, *pred_logits=True*, ***kwargs*)
Bases: *delira.training.metrics.SklearnClassificationMetric*

Log Loss (NLL)

SklearnMatthewsCorrCoeff

class SklearnMatthewsCorrCoeff(*gt_logits=False*, *pred_logits=True*, ***kwargs*)
Bases: *delira.training.metrics.SklearnClassificationMetric*

Matthews Correlation Coefficient

SklearnPrecisionScore

class SklearnPrecisionScore(*gt_logits=False*, *pred_logits=True*, ***kwargs*)
Bases: *delira.training.metrics.SklearnClassificationMetric*

Precision Score

SklearnRecallScore

class SklearnRecallScore(*gt_logits=False*, *pred_logits=True*, ***kwargs*)
Bases: *delira.training.metrics.SklearnClassificationMetric*

Recall Score

SklearnZeroOneLoss

class SklearnZeroOneLoss(*gt_logits=False*, *pred_logits=True*, ***kwargs*)
Bases: *delira.training.metrics.SklearnClassificationMetric*

Zero One Loss

AurocMetric

class AurocMetric(*classes=(0, 1)*, ***kwargs*)
Bases: object

convert_batch_to_numpy_identity

convert_batch_to_numpy_identity(**args*, ***kwargs*)
Corrects the shape of all zero-sized numpy arrays to be at least 1d

Parameters

- ***args** – positional arguments of potential arrays to be corrected
- ****kwargs** – keyword arguments of potential arrays to be corrected

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_pytorch

convert_torch_tensor_to_npy

convert_torch_tensor_to_npy(**args*, ***kwargs*)
Function to convert all torch Tensors to numpy arrays and reshape zero-size tensors

Parameters

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

Returns

- *Iterable* – all given positional arguments (converted if necessary)
- *dict* – all given keyword arguments (converted if necessary)

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

initialize_uninitialized

initialize_uninitialized(*sess*)
Function to initialize only uninitialized variables in a session graph

Parameters **sess** (*tf.Session()*) –

convert_tf_tensor_to_npy

convert_tf_tensor_to_npy(**args*, ***kwargs*)
Function to convert all tf Tensors to numpy arrays and reshape zero-size tensors

Parameters

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

Returns

- *Iterable* – all given positional arguments (converted if necessary)
- *dict* – all given keyword arguments (converted if necessary)

7.1.6 Utils

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

class DebugDisabled
Bases: *delira.utils.context_managers.DebugMode*

Context Manager to disable the debug mode for the wrapped context

_switch_to_new_mode()
helper function to switch to the new debug mode (and saving the previous one in `self._mode`)

class DebugEnabled
Bases: *delira.utils.context_managers.DebugMode*

Context Manager to enable the debug mode for the wrapped context

_switch_to_new_mode()
helper function to switch to the new debug mode (and saving the previous one in `self._mode`)

class DebugMode(*mode*)
Bases: object

Context Manager to set a specific debug mode for the code inside the defined context (and reverting to previous mode afterwards)

_switch_to_new_mode()
helper function to switch to the new debug mode (and saving the previous one in self._mode)

class DefaultOptimWrapperTorch(*optimizer: torch.optim.Optimizer*, **args*, ***kwargs*)
Bases: object

Class wrapping a torch optimizer to mirror the behavior of apex without depending on it

add_param_group(*param_group*)

load_state_dict(*state_dict*)

scale_loss(*loss*)
Function which scales the loss in apex and yields the unscaled loss here to mirror the API

Parameters **loss** (*torch.Tensor*) – the unscaled loss

state_dict()

step(*closure=None*)
Wraps the step method of the optimizer and calls the original step method

Parameters **closure** (*callable*) – A closure that reevaluates the model and returns the loss. Optional for most optimizers.

zero_grad()

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_img_func(*func*)

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=SimpleITK.sitkLinear*, *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=SimpleITK.sitkLinear*, *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: `trixi.util.Config`

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 Backend Resolution

These functions are used to determine the installed backends and update the created config file. They also need to be used, to guard backend specific code,

when writing code with several backends in one file like this:

```
if "YOUR_BACKEND" in delira.get_backends():
```

get_backends

get_backends()
Return List of currently available backends

Returns list of strings containing the currently installed backends

Return type list

7.1.8 Class Hierarchy Diagrams

Contents

Data Loading

- Coarse
- Fine

Sampler

- Coarse
- Fine

Logging

- Coarse
- Fine

Models

- Coarse
- Fine

Training

Experiment

- Coarse
- Fine

Trainer

- Coarse
- Fine

Hyperparameters

- Coarse
- Fine

Callbacks

- Coarse
- Fine

CHAPTER

EIGHT

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

Z