delira Documentation

Release 0.2.0-beta.1

Justus Schock, Christoph Haarburger

Jan 15, 2019

Getting Started

1 Getting started **1**
- 1.1 Installation . . . 1

2 Delira Introduction **3**
- 2.1 Loading Data . . . 3
- 2.2 Models . . . 6
- 2.3 Abstract Networks for specific Backends . . . 7
- 2.4 Training . . . 10
- 2.5 Logging . . . 12
- 2.6 More Examples . . . 14

3 Classification with Delira - A very short introduction **15**
- 3.1 Logging and Visualization . . . 16
- 3.2 Data Preparation . . . 16
- 3.3 Training . . . 17
- 3.4 See Also . . . 18

4 Generative Adversarial Nets with Delira - A very short introduction **19**
- 4.1 HyperParameters . . . 19
- 4.2 Logging and Visualization . . . 20
- 4.3 Data Preparation . . . 20
- 4.4 Training . . . 21
- 4.5 See Also . . . 22

5 Segmentation in 2D using U-Nets with Delira - A very short introduction **23**
- 5.1 Logging and Visualization . . . 24
- 5.2 Data Praparation . . . 24
- 5.3 Training . . . 26
- 5.4 See Also . . . 27

6 Segmentation in 3D using U-Nets with Delira - A very short introduction **29**
- 6.1 Logging and Visualization . . . 30
- 6.2 Data Praparation . . . 30
- 6.3 Training . . . 32
- 6.4 See Also . . . 32

7 API Documentation **33**

7.1 Delira 33

8 Indices and tables **155**

Python Module Index **157**

CHAPTER 1

Getting started

1.1 Installation

1.1.1 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

- None: No Suffix

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: Currently the only other planned backend is TensorFlow (which is coming soon). 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).

1.1.2 From Source

To install `delira` you can simply run

- `pip install git+https://github.com/justusschock/delira.git[suffix]`

by replacing `[suffix]` with the suffix for your backend, i.e. installing `delira` with `torch` backend would become `pip install git+https://github.com/justusschock/delira.git[torch]` and installing without a backend at all would become `pip install git+https://github.com/justusschock/delira.git`

1.1.3 Prebuild Packages

Prebuild packages (via pip) will be available in the future, once all dependencies are registered on PyPi. This is a requirement because PyPi forbids packages that depend on non-pypi packages. Currently this requirement is not fullfilled by batchgenerators

CHAPTER 2

Delira Introduction

Authors: Justus Schock, Christoph Haarburger

2.1 Loading Data

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

2.1.1 The Dataset

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

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

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

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

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

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

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

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

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

Now, let's just initialize our trainset:

```
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.spatial_transforms import MirrorTransform
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. Assuming your network has three convolutional layers stored in `self.conv1`, `self.conv2` and `self.conv3` and a ReLU stored in `self.relu`, a simple `forward` function could look like this:

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

    return out_3
```

prepare_batch

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

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

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

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

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

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

    return return_dict
```

and can be customized by subclassing the `AbstractPyTorchNetwork`.

closure example

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

```
    @staticmethod
    def closure(model: AbstractPyTorchNetwork, data_dict: dict,
                optimizers: dict, criterions={}, metrics={},
                fold=0, **kwargs):
        """
        closure method to do a single backpropagation step

        Parameters
```

(continues on next page)

(continued from previous page)

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

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

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

        """

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

        loss_vals = {}
        metric_vals = {}
        total_loss = 0

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

        else:
            context_man = torch.no_grad

        with context_man():

            inputs = data_dict.pop("data")
            preds = model(inputs)

            if data_dict:

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

(continues on next page)

(continued from previous page)

```
                    loss_vals[key] = _loss_val.detach()
                    total_loss += _loss_val

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

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

        else:

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

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

            loss_vals = eval_loss_vals
            metric_vals = eval_metrics_vals

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

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

        return metric_vals, loss_vals, [preds]

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

2.3.2 Other examples

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

2.4 Training

2.4.1 Parameters

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

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

`lr_sched_cls` and `lr_sched_params` to define the learning rate scheduling and the parameter `metrics` to specify evaluation metrics.

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

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

```
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

**Note:** The following section is deprecated and is only contained
for legacy reasons. It is absolutely not recommended to use this
code ### ``ImgSaveHandler`` The ``ImgSaveHandler`` saves the images
to a specified directory. The logging message must either include an
image or a dictionary containing a key 'images' which should be
associated with a list or dict of images.
```

Types of VisdomHandlers

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

The provided handlers are:

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

2.6 More Examples

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

CHAPTER 3

Classification with Delira - A very short introduction

Author: Justus Schock

Date: 04.12.2018

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

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

```
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
        "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
    }
})
```

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([
    RandomCropTransform((200, 200)), # Perform Random Crops of Size 200 x 200 pixels
    ResizeTransform((224, 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

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)
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 4

Generative Adversarial Nets with Delira - A very short introduction

Author: Justus Schock

Date: 04.12.2018

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

4.1 HyperParameters

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

```
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
        "criterions": {"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
from batchgenerators.transforms.spatial_transforms import ResizeTransform
from batchgenerators.transforms.sample_normalization_transforms import␣
↪MeanStdNormalizationTransform

transforms = Compose([
    RandomCropTransform((200, 200)), # Perform Random Crops of Size 200 x 200 pixels
    ResizeTransform((224, 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

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 5

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

Author: Justus Schock, Alexander Moriz

Date: 17.12.2018

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

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

```
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
        "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
    }
})
```

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

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

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

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
```

(continues on next page)

(continued from previous page)

```
    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, 150), label_key="label"), # Perform Random Crops of␣
↪Size 150 x 150 pixels
    ResizeTransform((224, 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
```

(continues on next page)

(continued from previous page)

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

logger.info("Init Experiment")
experiment = PyTorchExperiment(params, UNet2dPyTorch,
                               name="Segmentation2dExample",
                               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)
```

5.4 See Also

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

CHAPTER 6

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

Author: Justus Schock, Alexander Moriz

Date: 17.12.2018

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

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

```
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
        "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
    }
})
```

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)
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 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)

```
    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"),
                                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

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 7

API Documentation

7.1 Delira

7.1.1 Data Loading

This module provides Utilities to load the Data

Arbitrary Data

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

Datasets

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

Datasets can be indexed by integers and return single samples.

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

AbstractDataset

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

Bases: `object`

Base Class for Dataset

_make_dataset(*path*)

Create dataset

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

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

Return type list

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*, *load_fn*, *img_extensions*, *gt_extensions*, ***load_kwargs*)
Bases: delira.data_loading.dataset.AbstractDataset

Dataset to load data in a lazy way

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

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

Returns is valid data sample

Return type bool

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

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

Returns list of sample paths

Return type list

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

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*, *load_fn*, *img_extensions*, *gt_extensions*, ***load_kwargs*)
Bases: `delira.data_loading.dataset.AbstractDataset`

Dataset to preload and cache data

Notes

data needs to fit completely into RAM!

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

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

Returns is valid data sample

Return type bool

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

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

Returns list of sample paths

Return type list

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

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

set_thread_id(*thread_id*)

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'>*, *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

get_batchgen(*seed=1*)

Create DataLoader and Batchgenerator

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

Returns Batchgenerator

Return type MultiThreadedAugmenter

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

n_batches

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

Returns Number of Batches

Return type int

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

n_samples
Number of Samples

Returns Number of Samples

Return type int

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

***args :** positional arguments for `sklearn.model_selection.train_test_split`

****kwargs :** keyword arguments for `sklearn.model_selection.train_test_split`

Utils

default_load_fn_2d

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

Parameters

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

Returns

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

Nii-Data

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

BaseLabelGenerator

class BaseLabelGenerator(*fpath*)
Bases: `object`

Base Class to load labels from json files

_load()
Private Helper function to load the file

Returns loaded values from file

Return type Any

get_labels()
Abstractmethod to get labels from class

Raises NotImplementedError – if not overwritten in subclass

load_sample_nii

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

Parameters

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

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

Return type dict

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

Sampler

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

AbstractSampler

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

Class to define an abstract Sampling API

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

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

Returns List with sampled indices

Return type list

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

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

Returns The initialzed sampler

Return type *AbstractSampler*

LambdaSampler

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

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

_get_indices(*n_indices*)
Actual Sampling

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

Returns list of sampled indices

Return type list

Raises StopIteration – Maximum number of indices sampled

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

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

Returns The initialzed sampler

Return type *AbstractSampler*

RandomSampler

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

Implements Random Sampling from whole Dataset

_get_indices(*n_indices*)
Actual Sampling

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

Returns list of sampled indices

Return type list

Raises StopIteration – If maximal number of samples is reached

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

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

Returns The initialzed sampler

Return type *AbstractSampler*

PrevalenceRandomSampler

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

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

_get_indices(*n_indices*)
Actual Sampling

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

Returns list of sampled indices

Return type list

Raises StopIteration – If maximal number of samples is reached

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

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

Returns The initialzed sampler

Return type *AbstractSampler*

StoppingPrevalenceRandomSampler

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

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

_get_indices(*n_indices*)
Actual Sampling

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

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

Returns **list**

Return type list of sampled indices

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

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

Returns The initialzed sampler

Return type *AbstractSampler*

SequentialSampler

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

Implements Sequential Sampling from whole Dataset

_get_indices(*n_indices*)
Actual Sampling

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

Raises StopIteration : If end of dataset reached

Returns list of sampled indices

Return type list

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

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

Returns The initialzed sampler

Return type *AbstractSampler*

PrevalenceSequentialSampler

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

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

_get_indices(*n_indices*)
Actual Sampling

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

Raises StopIteration : If end of class indices is reached

Returns list of sampled indices

Return type list

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

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

Returns The initialzed sampler

Return type *AbstractSampler*

StoppingPrevalenceSequentialSampler

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

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

_get_indices(*n_indices*)
Actual Sampling

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

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

Returns list of sampled indices

Return type list

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

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

Returns The initialzed sampler

Return type *AbstractSampler*

WeightedRandomSampler

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

Implements Weighted Random Sampling

_get_indices(*n_indices*)
Actual Sampling

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

Returns list of sampled indices

Return type list

Raises

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

classmethod from_dataset(*dataset: delira.data_loading.dataset.AbstractDataset*)
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

load_checkpoint

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

Parameters

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

Returns

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

save_checkpoint

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

Parameters

- **file** (*str*) – filepath the model should be saved to
- **model** (AbstractNetwork *or None*) – the model which should be saved if None: empty dict will be saved as state dict
- **optimizers** (*dict*) – dictionary containing all optimizers

- **epoch** (*int*) – current epoch (will also be pickled)
- **weights_only** (*bool*) – whether or not to save only the model's weights or also save additional information (for easy loading)

7.1.3 Logging

This module handles the embedding of `trixi`'s loggers into the python `logging` module. It also contains the deprecated logging handlers which were used previously in `delira`.

MultiStreamHandler

class MultiStreamHandler(**streams*, *level=0*)

Bases: `logging.Handler`

Logging Handler which accepts multiple streams and creates StreamHandlers

acquire()
Acquire the I/O thread lock.

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

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

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

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

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

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

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

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

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

flush()
Ensure all logging output has been flushed.

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

format(*record*)
Format the specified record.

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

get_name()

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

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

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

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

name

release()
Release the I/O thread lock.

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

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

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

set_name(*name*)

TrixiHandler

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

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

acquire()
Acquire the I/O thread lock.

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

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

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

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

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

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

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

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

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

flush()
Ensure all logging output has been flushed.

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

format(*record*)
Format the specified record.

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

get_name()

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

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

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

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

name

release()
Release the I/O thread lock.

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

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

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

set_name(*name*)

Deprecated

ImgSaveHandler

class ImgSaveHandler
Logging Handler which saves images to dir

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

Warning: *ImgSaveHandler* will be removed in next release

See also:

`TrixiHandler`

_save_image_batch (*batch*, *prefix*, *is_train=True*)
Saving image batch to save_dir

Parameters

- **batch** (*iterable*) – batch of images
- **prefix** (*str*) – file-prefix

_to_image
staticmethod(function) -> method

Convert a function to be a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

class C: @staticmethod def f(arg1, arg2, ...):

...

It can be called either on the class (e.g. C.f()) or on an instance (e.g. C().f()). The instance is ignored except for its class.

Static methods in Python are similar to those found in Java or C++. For a more advanced concept, see the classmethod builtin.

emit (*record*)
Logging record message

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

Returns if *record.msg* is not a dict

Return type None

VisdomImageHandler

class VisdomImageHandler
Logging Handler to show images and metric plots with visdom

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

Warning: *VisdomImageHandler* will be removed in next release

See also:

Visdom `TrixiHandler`

_to_image
staticmethod(function) -> method

Convert a function to be a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

class C: @staticmethod def f(arg1, arg2, ...):

...

It can be called either on the class (e.g. C.f()) or on an instance (e.g. C().f()). The instance is ignored except for its class.

Static methods in Python are similar to those found in Java or C++. For a more advanced concept, see the classmethod builtin.

_to_scalar
staticmethod(function) -> method

Convert a function to be a static method.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

class C: @staticmethod def f(arg1, arg2, ...):

...

It can be called either on the class (e.g. C.f()) or on an instance (e.g. C().f()). The instance is ignored except for its class.

Static methods in Python are similar to those found in Java or C++. For a more advanced concept, see the classmethod builtin.

emit(*record*)
shows images and metric plots in visdom

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

Returns

- if no connection to *visdom* could be found
- if *record.msg* is not a dict

Return type None

VisdomImageSaveHandler

class VisdomImageSaveHandler
Logs images to dir and to visdom

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

..warning:: *VisdomImageSaveHandler* will be removed in next release

See also:

Visdom *VisdomImageHandler* `TrixiHandler`

emit(*record*)
log images to visdom and dir

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

VisdomImageSaveStreamHandler

class VisdomImageSaveStreamHandler
Logs metrics to streams, metric plots and images to visdom

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

Warning: *VisdomImageSaveStreamHandler* will be removed in next release

See also:

Visdom *VisdomImageHandler* MultiStreamHandler TrixiHandler

emit(*record: dict*)
Logs metrics to streams, metric plots and images to visdom

Parameters **record** (*dict*) – entities to log

VisdomStreamHandler

class VisdomStreamHandler
Logs images and metric plots to visdom and scalar values to streams

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

Warning: *VisdomStreamHandler* will be removed in next release

See also:

Visdom *VisdomImageHandler* MultiStreamHandler TrixiHandler

emit(*record: dict*)
Logs images and metric plots to visdom and scalar values to streams

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

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(***kwargs*)
Bases: object

Abstract class all networks should be derived from

_init_kwargs = {}

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

Returns

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

Raises `NotImplementedError` – If not overwritten by subclass

init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

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

Parameters

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

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

Return type dict

Raises `NotImplementedError` – If not overwritten by subclass

AbstractPyTorchNetwork

class AbstractPyTorchNetwork(***kwargs*)
Bases: `delira.models.abstract_network.AbstractNetwork`, `torch.nn.modules.module.Module`

Abstract Class for PyTorch Networks

See also:

torch.nn.Module *AbstractNetwork*

_apply(*fn*)

_get_name()

_init_kwargs = {}

_load_from_state_dict(*state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*)
Copies parameters and buffers from *state_dict* into only this module, but not its descendants. This is called on every submodule in `load_state_dict()`. Metadata saved for this module in input *state_dict* is provided as :attr'local_metadata'. For state dicts without metadata, :attr'local_metadata' is empty. Subclasses can achieve class-specific backward compatible loading using the version number at *local_metadata.get("version", None)*.

Note: *state_dict* is not the same object as the input *state_dict* to `load_state_dict()`. So it can be modified.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **prefix** (*str*) – the prefix for parameters and buffers used in this module
- **local_metadata** (*dict*) – a dict containing the metadata for this moodule. See
- **strict** (*bool*) – whether to strictly enforce that the keys in *state_dict* with `prefix` match the names of parameters and buffers in this module
- **missing_keys** (*list of str*) – if `strict=False`, add missing keys to this list
- **unexpected_keys** (*list of str*) – if `strict=False`, add unexpected keys to this list
- **error_msgs** (*list of str*) – error messages should be added to this list, and will be reported together in `load_state_dict()`

_named_members(*get_members_fn*, *prefix=''*, *recurse=True*)
Helper method for yielding various names + members of modules.

_register_load_state_dict_pre_hook(*hook*)
These hooks will be called with arguments: *state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*, before loading *state_dict* into *self*. These arguments are exactly the same as those of *_load_from_state_dict*.

_register_state_dict_hook(*hook*)
These hooks will be called with arguments: *self*, *state_dict*, *prefix*, *local_metadata*, after the *state_dict* of *self* is set. Note that only parameters and buffers of *self* or its children are guaranteed to exist in *state_dict*. The hooks may modify *state_dict* inplace or return a new one.

_slow_forward(**input*, ***kwargs*)

_tracing_name(*tracing_state*)

_version = 1

add_module(*name*, *module*)
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*string*) – name of the child module. The child module can be accessed from this module using the given name
- **parameter** (*Module*) – child module to be added to the module.

apply(*fn*)
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also torch-nn-init).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> def init_weights(m):
        print(m)
        if type(m) == nn.Linear:
            m.weight.data.fill_(1.0)
            print(m.weight)

>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

buffers(*recurse=True*)

Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf.data), buf.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

children()

Returns an iterator over immediate children modules.

Yields *Module* – a child module

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

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

Returns

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

- *list* – Arbitrary number of predictions

Raises `NotImplementedError` – If not overwritten by subclass

cpu()
Moves all model parameters and buffers to the CPU.

Returns self

Return type Module

cuda(*device=None*)
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double()
Casts all floating point parameters and buffers to `double` datatype.

Returns self

Return type Module

dump_patches = False

eval()
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

extra_repr()
Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float()
Casts all floating point parameters and buffers to float datatype.

Returns self

Return type Module

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

half()
Casts all floating point parameters and buffers to `half` datatype.

Returns self

Return type Module

init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

load_state_dict(*state_dict*, *strict=True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

modules()
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)

0 -> Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
)
1 -> Linear (2 -> 2)
```

named_buffers(*prefix=''*, *recurse=True*)
Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(string, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children()
Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules (*memo=None*, *prefix=''*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
        print(idx, '->', m)

0 -> ('', Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
))
1 -> ('0', Linear (2 -> 2))
```

named_parameters (*prefix=''*, *recurse=True*)

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(string, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters (*recurse=True*)

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param.data), param.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

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

register_backward_hook(*hook*)

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

Warning: The current implementation will not have the presented behavior for complex `Module` that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such `Module`, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

register_buffer(*name*, *tensor*)

Adds a persistent buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the persistent state.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*string*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor*) – buffer to be registered.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook*)
Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None
```

The hook should not modify the input or output.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook*)
Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None
```

The hook should not modify the input.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_parameter(*name*, *param*)
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*string*) – name of the parameter. The parameter can be accessed from this module using the given name
- **parameter** (*Parameter*) – parameter to be added to the module.

share_memory()

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype` s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point type of the floating point parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

Returns self

Return type Module

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train(*mode=True*)

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Returns self

Return type Module

type(*dst_type*)
Casts all parameters and buffers to `dst_type`.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

zero_grad()
Sets gradients of all model parameters to zero.

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`

_apply(*fn*)

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

_get_name()

_init_kwargs = {}

_load_from_state_dict(*state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*)
Copies parameters and buffers from *state_dict* into only this module, but not its descendants. This is called on every submodule in `load_state_dict()`. Metadata saved for this module in input *state_dict* is provided as :attr‘local_metadata‘. For state dicts without metadata, :attr‘local_metadata‘ is empty. Subclasses can achieve class-specific backward compatible loading using the version number at *local_metadata.get(“version”, None)*.

Note: *state_dict* is not the same object as the input *state_dict* to `load_state_dict()`. So it can be modified.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **prefix** (*str*) – the prefix for parameters and buffers used in this module
- **local_metadata** (*dict*) – a dict containing the metadata for this moodule. See
- **strict** (*bool*) – whether to strictly enforce that the keys in *state_dict* with `prefix` match the names of parameters and buffers in this module
- **missing_keys** (*list of str*) – if `strict=False`, add missing keys to this list
- **unexpected_keys** (*list of str*) – if `strict=False`, add unexpected keys to this list
- **error_msgs** (*list of str*) – error messages should be added to this list, and will be reported together in `load_state_dict()`

_named_members(*get_members_fn*, *prefix=''*, *recurse=True*)
Helper method for yielding various names + members of modules.

_register_load_state_dict_pre_hook(*hook*)
These hooks will be called with arguments: *state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*, before loading *state_dict* into *self*. These arguments are exactly the same as those of *_load_from_state_dict*.

_register_state_dict_hook(*hook*)
These hooks will be called with arguments: *self*, *state_dict*, *prefix*, *local_metadata*, after the *state_dict* of *self* is set. Note that only parameters and buffers of *self* or its children are guaranteed to exist in *state_dict*. The hooks may modify *state_dict* inplace or return a new one.

_slow_forward(**input*, ***kwargs*)

_tracing_name(*tracing_state*)

_version = 1

add_module(*name*, *module*)
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*string*) – name of the child module. The child module can be accessed from this module using the given name
- **parameter** (*Module*) – child module to be added to the module.

apply(*fn*)
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also torch-nn-init).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> def init_weights(m):
        print(m)
        if type(m) == nn.Linear:
            m.weight.data.fill_(1.0)
            print(m.weight)

>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

buffers(*recurse=True*)

Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf.data), buf.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

children()

Returns an iterator over immediate children modules.

Yields *Module* – a child module

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

closure method to do a single backpropagation step

Parameters

- **model** (*ClassificationNetworkBasePyTorch*) – trainable model
- **data_dict** (*dict*) – dictionary containing the data
- **optimizers** (*dict*) – dictionary of optimizers to optimize model's parameters
- **criterions** (*dict*) – dict holding the criterions to calculate errors (gradients from different criterions will be accumulated)

- **metrics** (*dict*) – dict holding the metrics to calculate
- **fold** (*int*) – Current Fold in Crossvalidation (default: 0)
- ****kwargs** – additional keyword arguments

Returns

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

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

cpu()
Moves all model parameters and buffers to the CPU.

Returns self

Return type Module

cuda(*device=None*)
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double()
Casts all floating point parameters and buffers to `double` datatype.

Returns self

Return type Module

dump_patches = False

eval()
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

extra_repr()
Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float()
Casts all floating point parameters and buffers to float datatype.

Returns self

Return type Module

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

half()
Casts all floating point parameters and buffers to `half` datatype.

Returns self

Return type Module

init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

load_state_dict(*state_dict*, *strict=True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

modules()
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)

0 -> Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
)
1 -> Linear (2 -> 2)
```

named_buffers(*prefix=''*, *recurse=True*)
Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(string, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children()
Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo=None*, *prefix=''*)
Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
        print(idx, '->', m)

0 -> ('', Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
))
1 -> ('0', Linear (2 -> 2))
```

named_parameters(*prefix=''*, *recurse=True*)
Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(string, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse=True*)
Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param.data), param.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

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

register_backward_hook(*hook*)
Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

Warning: The current implementation will not have the presented behavior for complex `Module` that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such `Module`, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

register_buffer(*name*, *tensor*)
Adds a persistent buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the persistent state.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*string*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor*) – buffer to be registered.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook*)

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None
```

The hook should not modify the input or output.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook*)

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None
```

The hook should not modify the input.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_parameter(*name*, *param*)

Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*string*) – name of the parameter. The parameter can be accessed from this module using the given name
- **parameter** (*Parameter*) – parameter to be added to the module.

share_memory()

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)

Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)

Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype` s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point type of the floating point parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

Returns self

Return type Module

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
```

(continues on next page)

(continued from previous page)

```
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train(*mode=True*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Returns self

Return type Module

type(*dst_type*)
Casts all parameters and buffers to `dst_type`.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

zero_grad()
Sets gradients of all model parameters to zero.

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

_apply(*fn*)

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

_get_name()

_init_kwargs = {}

_load_from_state_dict(*state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*)

Copies parameters and buffers from *state_dict* into only this module, but not its descendants. This is called on every submodule in `load_state_dict()`. Metadata saved for this module in input *state_dict* is provided as :attr'local_metadata'. For state dicts without metadata, :attr'local_metadata' is empty. Subclasses can achieve class-specific backward compatible loading using the version number at *local_metadata.get("version", None)*.

Note: *state_dict* is not the same object as the input *state_dict* to `load_state_dict()`. So it can be modified.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **prefix** (*str*) – the prefix for parameters and buffers used in this module
- **local_metadata** (*dict*) – a dict containing the metadata for this moodule. See
- **strict** (*bool*) – whether to strictly enforce that the keys in *state_dict* with `prefix` match the names of parameters and buffers in this module
- **missing_keys** (*list of str*) – if `strict=False`, add missing keys to this list
- **unexpected_keys** (*list of str*) – if `strict=False`, add unexpected keys to this list
- **error_msgs** (*list of str*) – error messages should be added to this list, and will be reported together in `load_state_dict()`

_named_members(*get_members_fn*, *prefix=''*, *recurse=True*)

Helper method for yielding various names + members of modules.

_register_load_state_dict_pre_hook(*hook*)

These hooks will be called with arguments: *state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*, before loading *state_dict* into *self*. These arguments are exactly the same as those of *_load_from_state_dict*.

_register_state_dict_hook(*hook*)

These hooks will be called with arguments: *self*, *state_dict*, *prefix*, *local_metadata*, after the *state_dict* of *self* is set. Note that only parameters and buffers of *self* or its children are guaranteed to exist in *state_dict*. The hooks may modify *state_dict* inplace or return a new one.

_slow_forward(**input*, ***kwargs*)

_tracing_name(*tracing_state*)

_version = 1

add_module(*name*, *module*)
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*string*) – name of the child module. The child module can be accessed from this module using the given name
- **parameter** (*Module*) – child module to be added to the module.

apply(*fn*)
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also torch-nn-init).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> def init_weights(m):
        print(m)
        if type(m) == nn.Linear:
            m.weight.data.fill_(1.0)
            print(m.weight)

>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

buffers(*recurse=True*)
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf.data), buf.size())
```

(continues on next page)

(continued from previous page)

```
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

children()
Returns an iterator over immediate children modules.

Yields *Module* – a child module

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

Parameters

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

Returns

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

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

cpu()
Moves all model parameters and buffers to the CPU.

Returns self

Return type Module

cuda(*device=None*)
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double()
Casts all floating point parameters and buffers to `double` datatype.

Returns self

Return type Module

dump_patches = False

eval()
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

extra_repr()
Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float()
Casts all floating point parameters and buffers to float datatype.

Returns self

Return type Module

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

half()
Casts all floating point parameters and buffers to `half` datatype.

Returns self

Return type Module

init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

load_state_dict(*state_dict*, *strict=True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

modules()
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)

0 -> Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
)
1 -> Linear (2 -> 2)
```

named_buffers(*prefix=''*, *recurse=True*)

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(string, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children()

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo=None*, *prefix=''*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
        print(idx, '->', m)

0 -> ('', Sequential (
```

(continues on next page)

(continued from previous page)

```
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
))
1 -> ('0', Linear (2 -> 2))
```

named_parameters (*prefix=''*, *recurse=True*)

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(string, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters (*recurse=True*)

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param.data), param.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

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

register_backward_hook (*hook*)

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

> **Warning:** The current implementation will not have the presented behavior for complex `Module` that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such `Module`, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

register_buffer(*name*, *tensor*)

Adds a persistent buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the persistent state.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*string*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor*) – buffer to be registered.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook*)

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None
```

The hook should not modify the input or output.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook*)

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None
```

The hook should not modify the input.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_parameter(*name*, *param*)
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*string*) – name of the parameter. The parameter can be accessed from this module using the given name
- **parameter** (*Parameter*) – parameter to be added to the module.

share_memory()

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype` s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point type of the floating point parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

Returns self

Return type Module

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train(*mode=True*)
Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Returns self

Return type Module

type(*dst_type*)
Casts all parameters and buffers to `dst_type`.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

zero_grad()
Sets gradients of all model parameters to zero.

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`

_apply(*fn*)

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

_get_name()

_init_kwargs = {}

_load_from_state_dict(*state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*)
Copies parameters and buffers from *state_dict* into only this module, but not its descendants. This is called on every submodule in `load_state_dict()`. Metadata saved for this module in input *state_dict* is provided as :attr'local_metadata'. For state dicts without metadata, :attr'local_metadata' is empty. Subclasses can achieve class-specific backward compatible loading using the version number at *local_metadata.get("version", None)*.

Note: *state_dict* is not the same object as the input *state_dict* to `load_state_dict()`. So it can be modified.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **prefix** (*str*) – the prefix for parameters and buffers used in this module
- **local_metadata** (*dict*) – a dict containing the metadata for this moodule. See
- **strict** (*bool*) – whether to strictly enforce that the keys in *state_dict* with `prefix` match the names of parameters and buffers in this module

- **missing_keys** (*list of str*) – if `strict=False`, add missing keys to this list
- **unexpected_keys** (*list of str*) – if `strict=False`, add unexpected keys to this list
- **error_msgs** (*list of str*) – error messages should be added to this list, and will be reported together in `load_state_dict()`

_named_members(*get_members_fn*, *prefix=''*, *recurse=True*)
Helper method for yielding various names + members of modules.

_register_load_state_dict_pre_hook(*hook*)
These hooks will be called with arguments: *state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*, before loading *state_dict* into *self*. These arguments are exactly the same as those of *_load_from_state_dict*.

_register_state_dict_hook(*hook*)
These hooks will be called with arguments: *self*, *state_dict*, *prefix*, *local_metadata*, after the *state_dict* of *self* is set. Note that only parameters and buffers of *self* or its children are guaranteed to exist in *state_dict*. The hooks may modify *state_dict* inplace or return a new one.

_slow_forward(**input*, ***kwargs*)

_tracing_name(*tracing_state*)

_version = 1

add_module(*name*, *module*)
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*string*) – name of the child module. The child module can be accessed from this module using the given name
- **parameter** (*Module*) – child module to be added to the module.

apply(*fn*)
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also torch-nn-init).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> def init_weights(m):
        print(m)
        if type(m) == nn.Linear:
            m.weight.data.fill_(1.0)
            print(m.weight)

>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
```

(continues on next page)

(continued from previous page)

```
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

buffers(*recurse=True*)

Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf.data), buf.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

children()

Returns an iterator over immediate children modules.

Yields *Module* – a child module

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

closure method to do a single backpropagation step

Parameters

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

Returns

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

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

cpu()
Moves all model parameters and buffers to the CPU.

Returns self

Return type Module

cuda(*device=None*)
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double()
Casts all floating point parameters and buffers to double datatype.

Returns self

Return type Module

dump_patches = False

eval()
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

extra_repr()
Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float()
Casts all floating point parameters and buffers to float datatype.

Returns self

Return type Module

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

half()
Casts all floating point parameters and buffers to half datatype.

Returns self

Return type Module

init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

load_state_dict(*state_dict*, *strict=True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

modules()
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)

0 -> Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
)
1 -> Linear (2 -> 2)
```

named_buffers(*prefix=''*, *recurse=True*)
Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(string, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children()
Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo=None*, *prefix=''*)
Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
        print(idx, '->', m)

0 -> ('', Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
))
1 -> ('0', Linear (2 -> 2))
```

named_parameters(*prefix=''*, *recurse=True*)
Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(string, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse=True*)
Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param.data), param.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

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

register_backward_hook(*hook*)

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

Warning: The current implementation will not have the presented behavior for complex `Module` that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such `Module`, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

register_buffer(*name*, *tensor*)

Adds a persistent buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the persistent state.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*string*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor*) – buffer to be registered.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook*)
Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None
```

The hook should not modify the input or output.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook*)
Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None
```

The hook should not modify the input.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_parameter(*name*, *param*)
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*string*) – name of the parameter. The parameter can be accessed from this module using the given name
- **parameter** (*Parameter*) – parameter to be added to the module.

share_memory()

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype` s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point type of the floating point parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

Returns self

Return type Module

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train(*mode=True*)

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Returns self

Return type Module

type(*dst_type*)
Casts all parameters and buffers to `dst_type`.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

zero_grad()
Sets gradients of all model parameters to zero.

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

_apply(*fn*)

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

Parameters

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

Notes

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

_get_name()

_init_kwargs = {}

_load_from_state_dict(*state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*)

Copies parameters and buffers from *state_dict* into only this module, but not its descendants. This is called on every submodule in `load_state_dict()`. Metadata saved for this module in input *state_dict* is provided as :attr‘local_metadata‘. For state dicts without metadata, :attr‘local_metadata‘ is empty. Subclasses can achieve class-specific backward compatible loading using the version number at *local_metadata.get(“version”, None)*.

Note: *state_dict* is not the same object as the input *state_dict* to `load_state_dict()`. So it can be modified.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **prefix** (*str*) – the prefix for parameters and buffers used in this module
- **local_metadata** (*dict*) – a dict containing the metadata for this moodule. See
- **strict** (*bool*) – whether to strictly enforce that the keys in *state_dict* with `prefix` match the names of parameters and buffers in this module
- **missing_keys** (*list of str*) – if `strict=False`, add missing keys to this list
- **unexpected_keys** (*list of str*) – if `strict=False`, add unexpected keys to this list
- **error_msgs** (*list of str*) – error messages should be added to this list, and will be reported together in `load_state_dict()`

_named_members(*get_members_fn*, *prefix=”*, *recurse=True*)

Helper method for yielding various names + members of modules.

_register_load_state_dict_pre_hook(*hook*)

These hooks will be called with arguments: *state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*, before loading *state_dict* into *self*. These arguments are exactly the same as those of *_load_from_state_dict*.

_register_state_dict_hook(*hook*)
These hooks will be called with arguments: *self*, *state_dict*, *prefix*, *local_metadata*, after the *state_dict* of *self* is set. Note that only parameters and buffers of *self* or its children are guaranteed to exist in *state_dict*. The hooks may modify *state_dict* inplace or return a new one.

_slow_forward(**input*, ***kwargs*)

_tracing_name(*tracing_state*)

_version = 1

add_module(*name*, *module*)
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*string*) – name of the child module. The child module can be accessed from this module using the given name
- **parameter** (*Module*) – child module to be added to the module.

apply(*fn*)
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also torch-nn-init).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> def init_weights(m):
        print(m)
        if type(m) == nn.Linear:
            m.weight.data.fill_(1.0)
            print(m.weight)

>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

buffers(*recurse=True*)
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf.data), buf.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

children()
Returns an iterator over immediate children modules.

Yields *Module* – a child module

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

Parameters

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

Returns

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

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

cpu()
Moves all model parameters and buffers to the CPU.

Returns self

Return type Module

cuda(*device=None*)
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double()
Casts all floating point parameters and buffers to `double` datatype.

Returns self

Return type Module

dump_patches = False

eval()
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

extra_repr()
Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float()
Casts all floating point parameters and buffers to float datatype.

Returns self

Return type Module

forward(*x*)
Feed tensor through network

Parameters **x** (*torch.Tensor*) –

Returns Prediction

Return type torch.Tensor

half()
Casts all floating point parameters and buffers to `half` datatype.

Returns self

Return type Module

init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

load_state_dict(*state_dict*, *strict=True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

modules()
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)

0 -> Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
)
1 -> Linear (2 -> 2)
```

named_buffers(*prefix=''*, *recurse=True*)
Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(string, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children()
Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo=None*, *prefix=''*)
Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
        print(idx, '->', m)

0 -> ('', Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
))
1 -> ('0', Linear (2 -> 2))
```

named_parameters (*prefix=''*, *recurse=True*)

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(string, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters (*recurse=True*)

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param.data), param.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

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

register_backward_hook(*hook*)

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

Warning: The current implementation will not have the presented behavior for complex `Module` that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such `Module`, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

register_buffer(*name*, *tensor*)

Adds a persistent buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the persistent state.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*string*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor*) – buffer to be registered.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook*)

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None
```

The hook should not modify the input or output.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook*)

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None
```

The hook should not modify the input.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_parameter(*name*, *param*)
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*string*) – name of the parameter. The parameter can be accessed from this module using the given name
- **parameter** (*Parameter*) – parameter to be added to the module.

reset_params()
Initialize all parameters

share_memory()

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype` s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (torch.device) – the desired device of the parameters and buffers in this module
- **dtype** (torch.dtype) – the desired floating point type of the floating point parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

Returns self

Return type Module

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train(*mode=True*)

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. Dropout, BatchNorm, etc.

Returns self

Return type Module

type(*dst_type*)

Casts all parameters and buffers to dst_type.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

static weight_init(*m*)

Initializes weights with xavier_normal and bias with zeros

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

zero_grad()
Sets gradients of all model parameters to zero.

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 up-mode='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

_apply(*fn*)

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

Parameters

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

Notes

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

_get_name()

_init_kwargs = {}

_load_from_state_dict(*state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*)

Copies parameters and buffers from *state_dict* into only this module, but not its descendants. This is called on every submodule in `load_state_dict()`. Metadata saved for this module in input *state_dict* is provided as :attr'local_metadata'. For state dicts without metadata, :attr'local_metadata' is empty. Subclasses can achieve class-specific backward compatible loading using the version number at *local_metadata.get("version", None)*.

Note: *state_dict* is not the same object as the input *state_dict* to `load_state_dict()`. So it can be modified.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **prefix** (*str*) – the prefix for parameters and buffers used in this module
- **local_metadata** (*dict*) – a dict containing the metadata for this moodule. See
- **strict** (*bool*) – whether to strictly enforce that the keys in *state_dict* with `prefix` match the names of parameters and buffers in this module
- **missing_keys** (*list of str*) – if `strict=False`, add missing keys to this list
- **unexpected_keys** (*list of str*) – if `strict=False`, add unexpected keys to this list
- **error_msgs** (*list of str*) – error messages should be added to this list, and will be reported together in `load_state_dict()`

_named_members(*get_members_fn*, *prefix=''*, *recurse=True*)

Helper method for yielding various names + members of modules.

_register_load_state_dict_pre_hook(*hook*)

These hooks will be called with arguments: *state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*, before loading *state_dict* into *self*. These arguments are exactly the same as those of *_load_from_state_dict*.

_register_state_dict_hook(*hook*)

These hooks will be called with arguments: *self*, *state_dict*, *prefix*, *local_metadata*, after the *state_dict* of *self* is set. Note that only parameters and buffers of *self* or its children are guaranteed to exist in *state_dict*. The hooks may modify *state_dict* inplace or return a new one.

_slow_forward(**input*, ***kwargs*)

_tracing_name(*tracing_state*)

_version = 1

add_module(*name*, *module*)
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*string*) – name of the child module. The child module can be accessed from this module using the given name
- **parameter** (*Module*) – child module to be added to the module.

apply(*fn*)
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also torch-nn-init).

Parameters **fn** (Module -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> def init_weights(m):
        print(m)
        if type(m) == nn.Linear:
            m.weight.data.fill_(1.0)
            print(m.weight)

>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

buffers(*recurse=True*)
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf.data), buf.size())
```

(continues on next page)

(continued from previous page)

```
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

children()
Returns an iterator over immediate children modules.

Yields *Module* – a child module

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

Parameters

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

Returns

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

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

cpu()
Moves all model parameters and buffers to the CPU.

Returns self

Return type Module

cuda(*device=None*)
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double()
Casts all floating point parameters and buffers to double datatype.

Returns self

Return type Module

dump_patches = False

eval()
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

extra_repr()
Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float()
Casts all floating point parameters and buffers to float datatype.

Returns self

Return type Module

forward(*x*)
Feed tensor through network

Parameters **x** (*torch.Tensor*) –

Returns Prediction

Return type torch.Tensor

half()
Casts all floating point parameters and buffers to `half` datatype.

Returns self

Return type Module

init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

load_state_dict(*state_dict*, *strict=True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

modules()
Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)

0 -> Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
)
1 -> Linear (2 -> 2)
```

named_buffers(*prefix=''*, *recurse=True*)

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(string, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children()

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo=None*, *prefix=''*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
        print(idx, '->', m)

0 -> ('', Sequential (
```

(continues on next page)

(continued from previous page)

```
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
))
1 -> ('0', Linear (2 -> 2))
```

named_parameters (*prefix=''*, *recurse=True*)

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(string, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters (*recurse=True*)

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param.data), param.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

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

register_backward_hook (*hook*)

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

> **Warning:** The current implementation will not have the presented behavior for complex `Module` that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such `Module`, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

register_buffer(*name*, *tensor*)

Adds a persistent buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the persistent state.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*string*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor*) – buffer to be registered.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook*)

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None
```

The hook should not modify the input or output.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook*)

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None
```

The hook should not modify the input.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_parameter(*name*, *param*)
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*string*) – name of the parameter. The parameter can be accessed from this module using the given name
- **parameter** (*Parameter*) – parameter to be added to the module.

reset_params()
Initialize all parameters

share_memory()

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype` s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point type of the floating point parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

Returns self

Return type Module

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train(*mode=True*)

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Returns self

Return type Module

type(*dst_type*)

Casts all parameters and buffers to `dst_type`.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

static weight_init(*m*)

Initializes weights with xavier_normal and bias with zeros

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

zero_grad()

Sets gradients of all model parameters to zero.

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`

clear() → None. Remove all items from D.

contains(*dict_like*)

Check whether all items in a dictionary-like object match the ones in this Config.

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

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

Return type bool

copy() → a shallow copy of D

deepcopy()

Get a deep copy of this Config.

Returns A deep copy of self.

Return type Config

deepupdate(*dict_like*, *ignore=None*, *allow_dict_overwrite=True*)

Identical to *update()* with *deep=True*.

Parameters

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

difference_config(**other_configs*)

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

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

Returns Difference of self and the other configs.

Return type Config

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

The resulting Config looks like this:

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

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

Parameters

- **configs** (*Config*) – Any number of Configs
- **only_set** (*bool*) – If only the set of different values hould be returned or for each config the
- **one** (*corresponding*) –

Returns Possibly empty Config

Return type Config

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

Parameters

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

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

Parameters

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

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

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

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

Becomes:

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

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

Parameters

- **keep_lists** – Keeps list along with unpacked values
- **max_split_size** – List longer than this will not be unpacked

Returns A flattened version of self

Return type dict

fromkeys()
Returns a new dict with keys from iterable and values equal to value.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

hasattr_not_none(*key*)

hierarchy
Returns the current hierarchy

Returns current hierarchy

Return type str

static init_objects(*config*)
Returns a new Config with types converted to instances.

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

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

becomes:

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

Note that additional entries can be lost as shown above.

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

Returns A new config with instances made from type entries.

Return type Config

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

load(*file_*, *raise_=True*, *decoder_cls_=<class 'trixi.util.util.ModuleMultiTypeDecoder'>*, ***kwargs*)
Load config from file using `json.load()`.

Parameters

- **file** (*str or File*) – Read from this location.
- **raise** (*bool*) – Raise errors.
- **decoder_cls** (*type*) – Class that is used to decode JSON string.
- ****kwargs** – Will be passed to `json.load()`.

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

Parameters

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

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*

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

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

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

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

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

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

Parameters

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

Examples

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

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

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

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

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

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

Returns Command line string

Return type str

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 Config.

Parameters

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

Examples

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

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

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

config1.update(config2, allow_dict_overwrite=True)

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

update_missing(*dict_like*, *deep=False*, *ignore=None*)
Recursively insert values that do not yet exist.

Parameters

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

values() → an object providing a view on D's values

variability_on_top
Return whether the variability is on top

Returns whether variability is on top

Return type bool

NetworkTrainer

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

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

AbstractNetworkTrainer

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

Defines an abstract API for Network Trainers

See also:

PyTorchNetworkTrainer

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

Parameters

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

Raises NotImplementedError – If not overwritten by subclass

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

Parameters

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

Raises NotImplementedError – If not overwritten by subclass

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

Parameters

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

Raises NotImplementedError – If not overwritten by subclass

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

Parameters

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

Raises NotImplementedError – If not overwritten by subclass

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

Parameters

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

Returns True if new score is better, False otherwise

Return type bool

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

Parameters

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

Raises NotImplementedError – If not overwritten by subclass

_train_single_epoch(*batchgen: batchgenerators.dataloading.multi_threaded_augmenter.MultiThreadedAugmenter*, *epoch*)
Defines a routine to train a single epoch

Parameters

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

Raises NotImplementedError – If not overwritten by subclass

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

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

Returns the trainer with a modified state

Return type *AbstractNetworkTrainer*

fold
Get current fold

Returns current fold

Return type int

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

Parameters

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

Returns new state

Return type dict

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

Parameters

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

Raises NotImplementedError – If not overwritten by subclass

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

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

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

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

Parameters

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

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

Parameters

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

Raises NotImplementedError – If not overwritten by subclass

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

Parameters

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

Returns the trainer with a modified state

Return type *AbstractNetworkTrainer*

PyTorchNetworkTrainer

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

Train and Validate a Network

See also:

`AbstractNetwork`

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

Parameters

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

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

Parameters

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

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

Parameters

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

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

Returns best network

Return type AbstractPyTorchNetwork

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

Parameters

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

Returns True if new score is better, False otherwise

Return type bool

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

Parameters

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

_train_single_epoch(*batchgen: batchgenerators.dataloading.multi_threaded_augmenter.MultiThreadedAugmenter*, *epoch*)
Trains the network a single epoch

Parameters

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

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

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

Returns the trainer with a modified state

Return type *AbstractNetworkTrainer*

fold
Get current fold

Returns current fold

Return type int

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

Parameters

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

Returns new state

Return type dict

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

Parameters

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

Returns

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

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

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

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

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

Parameters

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

- ****kwargs** – keyword arguments

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

Parameters

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

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

Return type AbstractPyTorchNetwork

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

Parameters

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

Returns the trainer with a modified state

Return type *AbstractNetworkTrainer*

Experiments

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

AbstractExperiment

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

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

See also:

PyTorchExperiment

kfold(*num_epochs: int, data: List[delira.data_loading.data_manager.BaseDataManager], num_splits=None, shuffle=False, random_seed=None, **kwargs*)
Runs K-Fold Crossvalidation

Parameters

- **num_epochs** (*int*) – number of epochs to train the model

- **data** (*list of BaseDataManager*) – list of datamanagers (will be split for cross-validation)
- **num_splits** (*None or int*) – number of splits for kfold if None: len(data) splits will be validated
- **shuffle** (*bool*) – whether or not to shuffle indices for kfold
- **random_seed** (*None or int*) – random seed used to seed the kfold (if shuffle is true), pytorch and numpy
- ****kwargs** – additional keyword arguments (completely passed to self.run())

static load(*file_name*)
Loads whole experiment

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

Raises NotImplementedError – if not overwritten in subclass

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

Parameters

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

Raises NotImplementedError – If not overwritten in subclass

save()
Saves the Whole experiments

Raises NotImplementedError – If not overwritten in subclass

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

Raises NotImplementedError – if not overwritten by subclass

PyTorchExperiment

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

Single Experiment for PyTorch Backend

See also:

AbstractExperiment

kfold(*num_epochs: int, data: List[delira.data_loading.data_manager.BaseDataManager], num_splits=None, shuffle=False, random_seed=None, **kwargs*)
Runs K-Fold Crossvalidation

Parameters

- **num_epochs** (*int*) – number of epochs to train the model
- **data** (*list of BaseDataManager*) – list of datamanagers (will be split for crossvalidation)
- **num_splits** (*None or int*) – number of splits for kfold if None: len(data) splits will be validated
- **shuffle** (*bool*) – whether or not to shuffle indices for kfold
- **random_seed** (*None or int*) – random seed used to seed the kfold (if shuffle is true), pytorch and numpy
- ****kwargs** – additional keyword arguments (completely passed to self.run())

static load(*file_name*)
Loads whole experiment

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

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

Parameters

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

Returns trainer of trained network

Return type *AbstractNetworkTrainer*

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

save()
Saves the Whole experiments

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

Parameters

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

Callbacks

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

AbstractCallback

class AbstractCallback(**args*, ***kwargs*)

Bases: `object`

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

See also:

`AbstractNetworkTrainer`

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

Function which will be executed at begin of each epoch

Parameters

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

Returns modified trainer

Return type `AbstractNetworkTrainer`

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

Function which will be executed at end of each epoch

Parameters

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

Returns modified trainer

Return type `AbstractNetworkTrainer`

EarlyStopping

class EarlyStopping(*monitor_key*, *min_delta=0*, *patience=0*, *mode='min'*)

Bases: `delira.training.callbacks.abstract_callback.AbstractCallback`

Implements Early Stopping as callback

See also:

AbstractCallback

_is_better(*metric*)

Helper function to decide whether the current metric is better than the best metric so far

Parameters **metric** – current metric value

Returns whether this metric is the new best metric or not

Return type bool

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

Parameters

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

Returns modified trainer

Return type AbstractNetworkTrainer

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

Parameters

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

Returns trainer with modified attributes

Return type AbstractNetworkTrainer

DefaultPyTorchSchedulerCallback

class DefaultPyTorchSchedulerCallback(**args*, ***kwargs*)
Bases: delira.training.callbacks.abstract_callback.AbstractCallback

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

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

Parameters

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

Returns modified trainer

Return type AbstractNetworkTrainer

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

Parameters

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

Returns modified trainer

Return type PyTorchNetworkTrainer

CosineAnnealingLRCallback

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

Bases: `delira.training.callbacks.pytorch_schedulers.DefaultPyTorchSchedulerCallback`

Wraps PyTorch's *CosineAnnealingLR* Scheduler as callback

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

Function which will be executed at begin of each epoch

Parameters

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

Returns modified trainer

Return type `AbstractNetworkTrainer`

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

Executes a single scheduling step

Parameters

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

Returns modified trainer

Return type `PyTorchNetworkTrainer`

ExponentialLRCallback

class ExponentialLRCallback(*optimizer*, *gamma*, *last_epoch=-1*)

Bases: `delira.training.callbacks.pytorch_schedulers.DefaultPyTorchSchedulerCallback`

Wraps PyTorch's *ExponentialLR* Scheduler as callback

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

Function which will be executed at begin of each epoch

Parameters

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

Returns modified trainer

Return type `AbstractNetworkTrainer`

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

Executes a single scheduling step

Parameters

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

Returns modified trainer

Return type `PyTorchNetworkTrainer`

LambdaLRCallback

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

Wraps PyTorch's *LambdaLR* Scheduler as callback

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

Parameters

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

Returns modified trainer

Return type AbstractNetworkTrainer

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

Parameters

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

Returns modified trainer

Return type PyTorchNetworkTrainer

MultiStepLRCallback

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

Wraps PyTorch's *MultiStepLR* Scheduler as callback

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

Parameters

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

Returns modified trainer

Return type AbstractNetworkTrainer

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

Parameters

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

Returns modified trainer

Return type PyTorchNetworkTrainer

ReduceLROnPlateauCallback

class ReduceLROnPlateauCallback(*optimizer*, *mode='min'*, *factor=0.1*, *patience=10*, *verbose=False*, *threshold=0.0001*, *threshold_mode='rel'*, *cooldown=0*, *min_lr=0*, *eps=1e-08*)

Bases: `delira.training.callbacks.pytorch_schedulers.DefaultPyTorchSchedulerCallback`

Wraps PyTorch's *ReduceLROnPlateau* Scheduler as Callback

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

Function which will be executed at begin of each epoch

Parameters

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

Returns modified trainer

Return type `AbstractNetworkTrainer`

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

Executes a single scheduling step

Parameters

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

Returns modified trainer

Return type `PyTorchNetworkTrainer`

StepLRCallback

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

Bases: `delira.training.callbacks.pytorch_schedulers.DefaultPyTorchSchedulerCallback`

Wraps PyTorch's *StepLR* Scheduler as callback

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

Function which will be executed at begin of each epoch

Parameters

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

Returns modified trainer

Return type `AbstractNetworkTrainer`

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

Executes a single scheduling step

Parameters

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

Returns modified trainer

Return type PyTorchNetworkTrainer

CosineAnnealingLRCallbackPyTorch

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

ExponentialLRCallbackPyTorch

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

LambdaLRCallbackPyTorch

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

MultiStepLRCallbackPyTorch

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

ReduceLROnPlateauCallbackPyTorch

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

StepLRCallbackPyTorch

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

Custom Loss Functions

BCEFocalLossPyTorch

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

Focal loss for binary case without(!) logit

_apply(*fn*)

_get_name()

_load_from_state_dict(*state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*)

Copies parameters and buffers from *state_dict* into only this module, but not its descendants. This is called on every submodule in `load_state_dict()`. Metadata saved for this module in input *state_dict* is provided as :attr'local_metadata'. For state dicts without metadata, :attr'local_metadata' is empty. Subclasses can achieve class-specific backward compatible loading using the version number at *local_metadata.get("version", None)*.

Note: *state_dict* is not the same object as the input *state_dict* to `load_state_dict()`. So it can be modified.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **prefix** (*str*) – the prefix for parameters and buffers used in this module
- **local_metadata** (*dict*) – a dict containing the metadata for this moodule. See
- **strict** (*bool*) – whether to strictly enforce that the keys in *state_dict* with `prefix` match the names of parameters and buffers in this module
- **missing_keys** (*list of str*) – if `strict=False`, add missing keys to this list
- **unexpected_keys** (*list of str*) – if `strict=False`, add unexpected keys to this list
- **error_msgs** (*list of str*) – error messages should be added to this list, and will be reported together in `load_state_dict()`

_named_members(*get_members_fn*, *prefix=''*, *recurse=True*)

Helper method for yielding various names + members of modules.

_register_load_state_dict_pre_hook(*hook*)

These hooks will be called with arguments: *state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*, before loading *state_dict* into *self*. These arguments are exactly the same as those of *_load_from_state_dict*.

_register_state_dict_hook(*hook*)

These hooks will be called with arguments: *self*, *state_dict*, *prefix*, *local_metadata*, after the *state_dict* of *self* is set. Note that only parameters and buffers of *self* or its children are guaranteed to exist in *state_dict*. The hooks may modify *state_dict* inplace or return a new one.

_slow_forward(**input*, ***kwargs*)

_tracing_name(*tracing_state*)

_version = 1

add_module(*name*, *module*)

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*string*) – name of the child module. The child module can be accessed from this module using the given name
- **parameter** (*Module*) – child module to be added to the module.

apply(*fn*)

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also torch-nn-init).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> def init_weights(m):
        print(m)
        if type(m) == nn.Linear:
            m.weight.data.fill_(1.0)
            print(m.weight)

>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

buffers(*recurse=True*)

Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf.data), buf.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

children()

Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu()

Moves all model parameters and buffers to the CPU.

Returns self

Return type Module

cuda(*device=None*)
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double()
Casts all floating point parameters and buffers to `double` datatype.

Returns self

Return type Module

dump_patches = False

eval()
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

extra_repr()
Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float()
Casts all floating point parameters and buffers to float datatype.

Returns self

Return type Module

forward(*p*, *t*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

half()
Casts all floating point parameters and buffers to `half` datatype.

Returns self

Return type Module

load_state_dict(*state_dict*, *strict=True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

modules()

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)

0 -> Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
)
1 -> Linear (2 -> 2)
```

named_buffers(*prefix=''*, *recurse=True*)

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(string, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children()

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo=None*, *prefix=''*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
        print(idx, '->', m)

0 -> ('', Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
))
1 -> ('0', Linear (2 -> 2))
```

named_parameters(*prefix=''*, *recurse=True*)

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(string, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse=True*)

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param.data), param.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook*)

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

> **Warning:** The current implementation will not have the presented behavior for complex `Module` that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such `Module`, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

register_buffer(*name*, *tensor*)

Adds a persistent buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the persistent state.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*string*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor*) – buffer to be registered.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook*)

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None
```

The hook should not modify the input or output.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook*)

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None
```

The hook should not modify the input.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_parameter(*name*, *param*)
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*string*) – name of the parameter. The parameter can be accessed from this module using the given name
- **parameter** (*Parameter*) – parameter to be added to the module.

share_memory()

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype` s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point type of the floating point parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

Returns self

Return type Module

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train(*mode=True*)

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Returns self

Return type Module

type(*dst_type*)

Casts all parameters and buffers to `dst_type`.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

zero_grad()

Sets gradients of all model parameters to zero.

BCEFocalLossLogitPyTorch

class BCEFocalLossLogitPyTorch(*alpha=None*, *gamma=2*, *reduction='elementwise_mean'*)

Bases: `torch.nn.modules.module.Module`

Focal loss for binary case WITH logit

_apply(*fn*)

_get_name()

_load_from_state_dict(*state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*)

Copies parameters and buffers from *state_dict* into only this module, but not its descendants. This is called on every submodule in load_state_dict(). Metadata saved for this module in input *state_dict* is provided as :attr'local_metadata'. For state dicts without metadata, :attr'local_metadata' is empty. Subclasses can achieve class-specific backward compatible loading using the version number at *local_metadata.get("version", None)*.

Note: *state_dict* is not the same object as the input *state_dict* to load_state_dict(). So it can be modified.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **prefix** (*str*) – the prefix for parameters and buffers used in this module
- **local_metadata** (*dict*) – a dict containing the metadata for this moodule. See
- **strict** (*bool*) – whether to strictly enforce that the keys in *state_dict* with prefix match the names of parameters and buffers in this module
- **missing_keys** (*list of str*) – if strict=False, add missing keys to this list
- **unexpected_keys** (*list of str*) – if strict=False, add unexpected keys to this list
- **error_msgs** (*list of str*) – error messages should be added to this list, and will be reported together in load_state_dict()

_named_members(*get_members_fn*, *prefix=''*, *recurse=True*)

Helper method for yielding various names + members of modules.

_register_load_state_dict_pre_hook(*hook*)

These hooks will be called with arguments: *state_dict*, *prefix*, *local_metadata*, *strict*, *missing_keys*, *unexpected_keys*, *error_msgs*, before loading *state_dict* into *self*. These arguments are exactly the same as those of *_load_from_state_dict*.

_register_state_dict_hook(*hook*)

These hooks will be called with arguments: *self*, *state_dict*, *prefix*, *local_metadata*, after the *state_dict* of *self* is set. Note that only parameters and buffers of *self* or its children are guaranteed to exist in *state_dict*. The hooks may modify *state_dict* inplace or return a new one.

_slow_forward(**input*, ***kwargs*)

_tracing_name(*tracing_state*)

_version = 1

add_module(*name*, *module*)

Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

Parameters

- **name** (*string*) – name of the child module. The child module can be accessed from this module using the given name
- **parameter** (*Module*) – child module to be added to the module.

apply(*fn*)

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also torch-nn-init).

Parameters **fn** (`Module` -> None) – function to be applied to each submodule

Returns self

Return type Module

Example:

```
>>> def init_weights(m):
        print(m)
        if type(m) == nn.Linear:
            m.weight.data.fill_(1.0)
            print(m.weight)

>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

buffers(*recurse=True*)

Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf.data), buf.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

children()

Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu()

Moves all model parameters and buffers to the CPU.

Returns self

Return type Module

cuda(*device=None*)
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type Module

double()
Casts all floating point parameters and buffers to `double` datatype.

Returns self

Return type Module

dump_patches = False

eval()
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

extra_repr()
Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float()
Casts all floating point parameters and buffers to float datatype.

Returns self

Return type Module

forward(*p*, *t*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

half()
Casts all floating point parameters and buffers to `half` datatype.

Returns self

Return type Module

load_state_dict(*state_dict*, *strict=True*)
Copies parameters and buffers from *state_dict* into this module and its descendants. If `strict` is `True`, then the keys of *state_dict* must exactly match the keys returned by this module's `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in *state_dict* match the keys returned by this module's `state_dict()` function. Default: `True`

modules()

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)

0 -> Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
)
1 -> Linear (2 -> 2)
```

named_buffers(*prefix=''*, *recurse=True*)

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.
- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(string, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>    if name in ['running_var']:
>>>        print(buf.size())
```

named_children()

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple containing a name and child module

Example:

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo=None*, *prefix=''*)

Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
        print(idx, '->', m)

0 -> ('', Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
))
1 -> ('0', Linear (2 -> 2))
```

named_parameters(*prefix=''*, *recurse=True*)

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(string, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>    if name in ['bias']:
>>>        print(param.size())
```

parameters(*recurse=True*)

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

Example:

```
>>> for param in model.parameters():
>>>     print(type(param.data), param.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook*)

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

> **Warning:** The current implementation will not have the presented behavior for complex `Module` that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such `Module`, you should use `torch.Tensor.register_hook()` directly on a specific input or output to get the required gradients.

register_buffer(*name*, *tensor*)

Adds a persistent buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the persistent state.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*string*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (*Tensor*) – buffer to be registered.

Example:

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook*)

Registers a forward hook on the module.

The hook will be called every time after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None
```

The hook should not modify the input or output.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook*)

Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None
```

The hook should not modify the input.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_parameter(*name*, *param*)
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*string*) – name of the parameter. The parameter can be accessed from this module using the given name
- **parameter** (*Parameter*) – parameter to be added to the module.

share_memory()

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)
Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

Its signature is similar to `torch.Tensor.to()`, but only accepts floating point desired `dtype` s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point type of the floating point parameters and buffers in this module
- **tensor** (*torch.Tensor*) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

Returns self

Return type Module

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train(*mode=True*)

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Returns self

Return type Module

type(*dst_type*)

Casts all parameters and buffers to `dst_type`.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type Module

zero_grad()

Sets gradients of all model parameters to zero.

AurocMetricPyTorch

class AurocMetricPyTorch

Metric to Calculate AuROC

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

Warning: *AurocMetricPyTorch* will be removed in next release

forward(*outputs: torch.Tensor*, *targets: torch.Tensor*)
Actual AuROC calculation

Parameters

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

Returns auroc value

Return type torch.Tensor

AccuracyMetricPyTorch

class AccuracyMetricPyTorch
Metric to Calculate Accuracy

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

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

forward(*outputs: torch.Tensor*, *targets: torch.Tensor*)
Actual accuracy calcuation

Parameters

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

Returns accuracy value

Return type torch.Tensor

pytorch_batch_to_numpy

pytorch_batch_to_numpy(*tensor: torch.Tensor*)
Utility Function to cast a whole PyTorch batch to numpy

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

Returns the converted batch

Return type np.ndarray

pytorch_tensor_to_numpy

pytorch_tensor_to_numpy(*tensor: torch.Tensor*)
Utility Function to cast a single PyTorch Tensor to numpy

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

Returns the converted tensor

Return type np.ndarray

float_to_pytorch_tensor

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

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

Returns converted float

Return type torch.Tensor

create_optims_default_pytorch

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

Parameters

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

Returns dictionary containing all created optimizers

Return type dict

create_optims_gan_default_pytorch

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

Parameters

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

Returns dictionary containing all created optimizers

Return type dict

7.1.6 Utils

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

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

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

Returns

Return type Wrapped Function

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

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

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

Returns

Return type Wrapped Function

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

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

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

Returns

Return type Wrapped Function

Raises Deprecation Warning

numpy_array_func(*func*)

torch_module_func(*func*)

torch_tensor_func(*func*)

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

Parameters

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

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

Return type tuple

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

Parameters

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

Returns origin offset

Return type np.ndarray

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

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

Returns slice index

Return type int

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

Parameters

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

Returns Target image with copied metadata

Return type SimpleITK.Image

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

Parameters

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

Returns Blank image with given properties

Return type SimpleITK.Image

sitk_resample_to_image(*image*, *reference_image*, *default_value=0.0*, *interpolator=2*, *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=2*, *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(*file_=None*, *config=None*, *update_from_argv=False*, *deep=False*, ***kwargs*)
Bases: trixi.util.config.Config

Helper class to have nested lookups in all subdicts of Config

clear() → None. Remove all items from D.

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

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

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

Return type bool

copy() → a shallow copy of D

deepcopy()
Get a deep copy of this Config.

Returns A deep copy of self.

Return type Config

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

Parameters

- **dict_like** (*dict or derivative thereof*) – Update source.
- **ignore** (*iterable*) – Iterable of keys to ignore in update.

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

difference_config(**other_configs*)

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

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

Returns Difference of self and the other configs.

Return type Config

static difference_config_static(**configs*, *only_set=False*)

Make a Config of all elements that differ between N configs.

The resulting Config looks like this:

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

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

Parameters

- **configs** (*Config*) – Any number of Configs
- **only_set** (*bool*) – If only the set of different values hould be returned or for each config the
- **one** (*corresponding*) –

Returns Possibly empty Config

Return type Config

dump(*file_*, *indent=4*, *separators=(', ', ': ')*, ***kwargs*)

Write config to file using `json.dump()`.

Parameters

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

dumps(*indent=4*, *separators=(', ', ': ')*, ***kwargs*)

Get string representation using `json.dumps()`.

Parameters

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

flat(*keep_lists=True*, *max_split_size=10*)

Returns a flattened version of the Config as dict.

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

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

Becomes:

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

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

Parameters

- **keep_lists** – Keeps list along with unpacked values
- **max_split_size** – List longer than this will not be unpacked

Returns A flattened version of self

Return type dict

fromkeys()
Returns a new dict with keys from iterable and values equal to value.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

hasattr_not_none(*key*)

static init_objects(*config*)
Returns a new Config with types converted to instances.

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

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

becomes:

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

Note that additional entries can be lost as shown above.

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

Returns A new config with instances made from type entries.

Return type Config

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

load(*file_*, *raise_=True*, *decoder_cls_=<class 'trixi.util.util.ModuleMultiTypeDecoder'>*, ***kwargs*)
Load config from file using `json.load()`.

Parameters

- **file** (*str or File*) – Read from this location.
- **raise** (*bool*) – Raise errors.
- **decoder_cls** (*type*) – Class that is used to decode JSON string.
- ****kwargs** – Will be passed to `json.load()`.

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

Parameters

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

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

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

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

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

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

Parameters

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

Examples

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

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

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

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

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

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

Returns Command line string

Return type str

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

Parameters

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

Examples

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

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

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

config1.update(config2, allow_dict_overwrite=True)

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

update_missing(*dict_like*, *deep=False*, *ignore=None*)
Recursively insert values that do not yet exist.

Parameters

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

values() → an object providing a view on D's values

7.1.7 Class Hierarchy Diagrams

Contents

- *Class Hierarchy Diagrams*
  - *Data Loading*
    - *Sampler*
  - *Logging*
  - *Models*
  - *Training*
    - *Hyperparameters*

Data Loading

- Coarse
- Fine

Sampler

- Coarse
- Fine

Logging

- Coarse
- Fine

Models

- Coarse
- Fine

Training

- Coarse
- Fine

Hyperparameters

- Coarse
- Fine

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

delira.utils.config, 147
delira.utils.decorators, 144
delira.utils.imageops, 145
delira.utils.path, 147
delira.utils.time, 147

Index

Symbols

_apply() (AbstractPyTorchNetwork method), 49
_apply() (BCEFocalLossLogitPyTorch method), 134
_apply() (BCEFocalLossPyTorch method), 126
_apply() (ClassificationNetworkBasePyTorch method), 58
_apply() (GenerativeAdversarialNetworkBasePyTorch method), 77
_apply() (UNet2dPyTorch method), 86
_apply() (UNet3dPyTorch method), 96
_apply() (VGG3DClassificationNetworkPyTorch method), 67
_at_epoch_begin() (AbstractNetworkTrainer method), 112
_at_epoch_begin() (PyTorchNetworkTrainer method), 115
_at_epoch_end() (AbstractNetworkTrainer method), 112
_at_epoch_end() (PyTorchNetworkTrainer method), 115
_at_training_begin() (AbstractNetworkTrainer method), 112
_at_training_begin() (PyTorchNetworkTrainer method), 115
_at_training_end() (AbstractNetworkTrainer method), 113
_at_training_end() (PyTorchNetworkTrainer method), 116
_build_model() (ClassificationNetworkBasePyTorch static method), 58
_build_model() (UNet2dPyTorch method), 86
_build_model() (UNet3dPyTorch method), 96
_build_model() (VGG3DClassificationNetworkPyTorch static method), 67
_build_models() (GenerativeAdversarialNetworkBasePyTorch static method), 77
_get_indices() (AbstractSampler method), 38
_get_indices() (LambdaSampler method), 38
_get_indices() (PrevalenceRandomSampler method), 39
_get_indices() (PrevalenceSequentialSampler method), 41
_get_indices() (RandomSampler method), 39
_get_indices() (SequentialSampler method), 40
_get_indices() (StoppingPrevalenceRandomSampler method), 40
_get_indices() (StoppingPrevalenceSequentialSampler method), 41
_get_indices() (WeightedRandomSampler method), 41
_get_name() (AbstractPyTorchNetwork method), 49
_get_name() (BCEFocalLossLogitPyTorch method), 134
_get_name() (BCEFocalLossPyTorch method), 126
_get_name() (ClassificationNetworkBasePyTorch method), 58
_get_name() (GenerativeAdversarialNetworkBasePyTorch method), 77
_get_name() (UNet2dPyTorch method), 87
_get_name() (UNet3dPyTorch method), 97
_get_name() (VGG3DClassificationNetworkPyTorch method), 68
_get_sample() (BaseDataLoader method), 36
_init_kwargs (AbstractNetwork attribute), 48
_init_kwargs (AbstractPyTorchNetwork attribute), 49
_init_kwargs (ClassificationNetworkBasePyTorch attribute), 58
_init_kwargs (GenerativeAdversarialNetworkBasePyTorch attribute), 77
_init_kwargs (UNet2dPyTorch attribute), 87
_init_kwargs (UNet3dPyTorch attribute), 97
_init_kwargs (VGG3DClassificationNetworkPyTorch attribute), 68
_is_better() (EarlyStopping method), 121
_is_better_val_scores() (AbstractNetworkTrainer static method), 113
_is_better_val_scores() (PyTorchNetworkTrainer static method), 116
_is_valid_image_file() (BaseCacheDataset method), 35
_is_valid_image_file() (BaseLazyDataset method), 34
_load() (BaseLabelGenerator method), 37
_load_from_state_dict() (AbstractPyTorchNetwork method), 49
_load_from_state_dict() (BCEFocalLossLogitPyTorch

method), 135
_load_from_state_dict() (BCEFocalLossPyTorch method), 126
_load_from_state_dict() (ClassificationNetworkBasePyTorch method), 58
_load_from_state_dict() (GenerativeAdversarialNetworkBasePyTorch method), 77
_load_from_state_dict() (UNet2dPyTorch method), 87
_load_from_state_dict() (UNet3dPyTorch method), 97
_load_from_state_dict() (VGG3DClassificationNetworkPyTorch method), 68
_make_dataset() (AbstractDataset method), 33
_make_dataset() (BaseCacheDataset method), 35
_make_dataset() (BaseLazyDataset method), 34
_named_members() (AbstractPyTorchNetwork method), 50
_named_members() (BCEFocalLossLogitPyTorch method), 135
_named_members() (BCEFocalLossPyTorch method), 127
_named_members() (ClassificationNetworkBasePyTorch method), 59
_named_members() (GenerativeAdversarialNetworkBasePyTorch method), 78
_named_members() (UNet2dPyTorch method), 87
_named_members() (UNet3dPyTorch method), 97
_named_members() (VGG3DClassificationNetworkPyTorch method), 68
_register_load_state_dict_pre_hook() (AbstractPyTorchNetwork method), 50
_register_load_state_dict_pre_hook() (BCEFocalLossLogitPyTorch method), 135
_register_load_state_dict_pre_hook() (BCEFocalLossPyTorch method), 127
_register_load_state_dict_pre_hook() (ClassificationNetworkBasePyTorch method), 59
_register_load_state_dict_pre_hook() (GenerativeAdversarialNetworkBasePyTorch method), 78
_register_load_state_dict_pre_hook() (UNet2dPyTorch method), 87
_register_load_state_dict_pre_hook() (UNet3dPyTorch method), 97
_register_load_state_dict_pre_hook() (VGG3DClassificationNetworkPyTorch method), 68
_register_state_dict_hook() (AbstractPyTorchNetwork method), 50
_register_state_dict_hook() (BCEFocalLossLogitPyTorch method), 135
_register_state_dict_hook() (BCEFocalLossPyTorch method), 127
_register_state_dict_hook() (ClassificationNetworkBasePyTorch method), 59
_register_state_dict_hook() (GenerativeAdversarialNetworkBasePyTorch method), 78
_register_state_dict_hook() (UNet2dPyTorch method), 87
_register_state_dict_hook() (UNet3dPyTorch method), 97
_register_state_dict_hook() (VGG3DClassificationNetworkPyTorch method), 68
_save_image_batch() (ImgSaveHandler method), 45
_setup() (AbstractNetworkTrainer method), 113
_setup() (PyTorchNetworkTrainer method), 116
_slow_forward() (AbstractPyTorchNetwork method), 50
_slow_forward() (BCEFocalLossLogitPyTorch method), 135
_slow_forward() (BCEFocalLossPyTorch method), 127
_slow_forward() (ClassificationNetworkBasePyTorch method), 59
_slow_forward() (GenerativeAdversarialNetworkBasePyTorch method), 78
_slow_forward() (UNet2dPyTorch method), 88
_slow_forward() (UNet3dPyTorch method), 97
_slow_forward() (VGG3DClassificationNetworkPyTorch method), 68
_to_image (ImgSaveHandler attribute), 46
_to_image (VisdomImageHandler attribute), 46
_to_scalar (VisdomImageHandler attribute), 47
_tracing_name() (AbstractPyTorchNetwork method), 50
_tracing_name() (BCEFocalLossLogitPyTorch method), 135
_tracing_name() (BCEFocalLossPyTorch method), 127
_tracing_name() (ClassificationNetworkBasePyTorch method), 59
_tracing_name() (GenerativeAdversarialNetworkBasePyTorch method), 78
_tracing_name() (UNet2dPyTorch method), 88
_tracing_name() (UNet3dPyTorch method), 97
_tracing_name() (VGG3DClassificationNetworkPyTorch method), 68
_train_single_epoch() (AbstractNetworkTrainer method), 113
_train_single_epoch() (PyTorchNetworkTrainer method), 116
_update_state() (AbstractNetworkTrainer method), 113
_update_state() (PyTorchNetworkTrainer method), 116
_version (AbstractPyTorchNetwork attribute), 50
_version (BCEFocalLossLogitPyTorch attribute), 135
_version (BCEFocalLossPyTorch attribute), 127
_version (ClassificationNetworkBasePyTorch attribute), 59
_version (GenerativeAdversarialNetworkBasePyTorch attribute), 78
_version (UNet2dPyTorch attribute), 88
_version (UNet3dPyTorch attribute), 97

_version (VGG3DClassificationNetworkPyTorch attribute), 68

A

AbstractCallback (class in delira.training.callbacks), 121
AbstractDataset (class in delira.data_loading), 33
AbstractExperiment (class in delira.training), 118
AbstractNetwork (class in delira.models), 48
AbstractNetworkTrainer (class in delira.training), 112
AbstractPyTorchNetwork (class in delira.models), 49
AbstractPyTorchNetwork.to() (in module delira.models), 56
AbstractSampler (class in delira.data_loading.sampler), 38
AccuracyMetricPyTorch (class in delira.training), 143
acquire() (MultiStreamHandler method), 43
acquire() (TrixiHandler method), 44
add_module() (AbstractPyTorchNetwork method), 50
add_module() (BCEFocalLossLogitPyTorch method), 135
add_module() (BCEFocalLossPyTorch method), 127
add_module() (ClassificationNetworkBasePyTorch method), 59
add_module() (GenerativeAdversarialNetworkBasePyTorch method), 78
add_module() (UNet2dPyTorch method), 88
add_module() (UNet3dPyTorch method), 97
add_module() (VGG3DClassificationNetworkPyTorch method), 68
addFilter() (MultiStreamHandler method), 43
addFilter() (TrixiHandler method), 44
apply() (AbstractPyTorchNetwork method), 50
apply() (BCEFocalLossLogitPyTorch method), 136
apply() (BCEFocalLossPyTorch method), 127
apply() (ClassificationNetworkBasePyTorch method), 59
apply() (GenerativeAdversarialNetworkBasePyTorch method), 78
apply() (UNet2dPyTorch method), 88
apply() (UNet3dPyTorch method), 98
apply() (VGG3DClassificationNetworkPyTorch method), 69
at_epoch_begin() (AbstractCallback method), 121
at_epoch_begin() (CosineAnnealingLRCallback method), 123
at_epoch_begin() (DefaultPyTorchSchedulerCallback method), 122
at_epoch_begin() (EarlyStopping method), 121
at_epoch_begin() (ExponentialLRCallback method), 123
at_epoch_begin() (LambdaLRCallback method), 124
at_epoch_begin() (MultiStepLRCallback method), 124
at_epoch_begin() (ReduceLROnPlateauCallback method), 125
at_epoch_begin() (StepLRCallback method), 125
at_epoch_end() (AbstractCallback method), 121
at_epoch_end() (CosineAnnealingLRCallback method), 123
at_epoch_end() (DefaultPyTorchSchedulerCallback method), 122
at_epoch_end() (EarlyStopping method), 122
at_epoch_end() (ExponentialLRCallback method), 123
at_epoch_end() (LambdaLRCallback method), 124
at_epoch_end() (MultiStepLRCallback method), 124
at_epoch_end() (ReduceLROnPlateauCallback method), 125
at_epoch_end() (StepLRCallback method), 125
AurocMetricPyTorch (class in delira.training), 142

B

BaseCacheDataset (class in delira.data_loading), 35
BaseDataLoader (class in delira.data_loading), 36
BaseDataManager (class in delira.data_loading), 36
BaseLabelGenerator (class in delira.data_loading.nii), 37
BaseLazyDataset (class in delira.data_loading), 34
BCEFocalLossLogitPyTorch (class in delira.training.losses), 134
BCEFocalLossLogitPyTorch.to() (in module delira.training.losses), 141
BCEFocalLossPyTorch (class in delira.training.losses), 126
BCEFocalLossPyTorch.to() (in module delira.training.losses), 133
bounding_box() (in module delira.utils.imageops), 145
buffers() (AbstractPyTorchNetwork method), 51
buffers() (BCEFocalLossLogitPyTorch method), 136
buffers() (BCEFocalLossPyTorch method), 128
buffers() (ClassificationNetworkBasePyTorch method), 60
buffers() (GenerativeAdversarialNetworkBasePyTorch method), 79
buffers() (UNet2dPyTorch method), 88
buffers() (UNet3dPyTorch method), 98
buffers() (VGG3DClassificationNetworkPyTorch method), 69

C

calculate_origin_offset() (in module delira.utils.imageops), 145
children() (AbstractPyTorchNetwork method), 51
children() (BCEFocalLossLogitPyTorch method), 136
children() (BCEFocalLossPyTorch method), 128
children() (ClassificationNetworkBasePyTorch method), 60
children() (GenerativeAdversarialNetworkBasePyTorch method), 79
children() (UNet2dPyTorch method), 89
children() (UNet3dPyTorch method), 99
children() (VGG3DClassificationNetworkPyTorch method), 70

ClassificationNetworkBasePyTorch (class in delira.models.classification), 58
ClassificationNetworkBasePyTorch.to() (in module delira.models.classification), 66
classtype_func() (in module delira.utils.decorators), 144
clear() (LookupConfig method), 147
clear() (Parameters method), 106
close() (MultiStreamHandler method), 43
close() (TrixiHandler method), 44
closure() (AbstractNetwork static method), 48
closure() (AbstractPyTorchNetwork static method), 51
closure() (ClassificationNetworkBasePyTorch static method), 60
closure() (GenerativeAdversarialNetworkBasePyTorch static method), 79
closure() (UNet2dPyTorch static method), 89
closure() (UNet3dPyTorch static method), 99
closure() (VGG3DClassificationNetworkPyTorch static method), 70
contains() (LookupConfig method), 147
contains() (Parameters method), 106
copy() (LookupConfig method), 147
copy() (Parameters method), 106
CosineAnnealingLRCallback (class in delira.training.callbacks.pytorch_schedulers), 123
CosineAnnealingLRCallbackPyTorch (in module delira.training.callbacks), 126
cpu() (AbstractPyTorchNetwork method), 52
cpu() (BCEFocalLossLogitPyTorch method), 136
cpu() (BCEFocalLossPyTorch method), 128
cpu() (ClassificationNetworkBasePyTorch method), 61
cpu() (GenerativeAdversarialNetworkBasePyTorch method), 80
cpu() (UNet2dPyTorch method), 89
cpu() (UNet3dPyTorch method), 99
cpu() (VGG3DClassificationNetworkPyTorch method), 70
create_optims_default_pytorch() (in module delira.training.train_utils), 144
create_optims_gan_default_pytorch() (in module delira.training.train_utils), 144
createLock() (MultiStreamHandler method), 43
createLock() (TrixiHandler method), 44
cuda() (AbstractPyTorchNetwork method), 52
cuda() (BCEFocalLossLogitPyTorch method), 137
cuda() (BCEFocalLossPyTorch method), 129
cuda() (ClassificationNetworkBasePyTorch method), 61
cuda() (GenerativeAdversarialNetworkBasePyTorch method), 80
cuda() (UNet2dPyTorch method), 89
cuda() (UNet3dPyTorch method), 99
cuda() (VGG3DClassificationNetworkPyTorch method), 70

D

deepcopy() (LookupConfig method), 147
deepcopy() (Parameters method), 106
deepupdate() (LookupConfig method), 147
deepupdate() (Parameters method), 106
default_load_fn_2d() (in module delira.data_loading), 37
DefaultPyTorchSchedulerCallback (class in delira.training.callbacks), 122
delira.utils.config (module), 147
delira.utils.decorators (module), 144
delira.utils.imageops (module), 145
delira.utils.path (module), 147
delira.utils.time (module), 147
difference_config() (LookupConfig method), 148
difference_config() (Parameters method), 106
difference_config_static() (LookupConfig static method), 148
difference_config_static() (Parameters static method), 106
double() (AbstractPyTorchNetwork method), 52
double() (BCEFocalLossLogitPyTorch method), 137
double() (BCEFocalLossPyTorch method), 129
double() (ClassificationNetworkBasePyTorch method), 61
double() (GenerativeAdversarialNetworkBasePyTorch method), 80
double() (UNet2dPyTorch method), 90
double() (UNet3dPyTorch method), 99
double() (VGG3DClassificationNetworkPyTorch method), 70
dtype_func() (in module delira.utils.decorators), 145
dump() (LookupConfig method), 148
dump() (Parameters method), 107
dump_patches (AbstractPyTorchNetwork attribute), 52
dump_patches (BCEFocalLossLogitPyTorch attribute), 137
dump_patches (BCEFocalLossPyTorch attribute), 129
dump_patches (ClassificationNetworkBasePyTorch attribute), 61
dump_patches (GenerativeAdversarialNetworkBasePyTorch attribute), 80
dump_patches (UNet2dPyTorch attribute), 90
dump_patches (UNet3dPyTorch attribute), 99
dump_patches (VGG3DClassificationNetworkPyTorch attribute), 70
dumps() (LookupConfig method), 148
dumps() (Parameters method), 107

E

EarlyStopping (class in delira.training.callbacks), 121
emit() (ImgSaveHandler method), 46
emit() (MultiStreamHandler method), 43
emit() (TrixiHandler method), 44
emit() (VisdomImageHandler method), 47

emit() (VisdomImageSaveHandler method), 47
emit() (VisdomImageSaveStreamHandler method), 48
emit() (VisdomStreamHandler method), 48
eval() (AbstractPyTorchNetwork method), 52
eval() (BCEFocalLossLogitPyTorch method), 137
eval() (BCEFocalLossPyTorch method), 129
eval() (ClassificationNetworkBasePyTorch method), 61
eval() (GenerativeAdversarialNetworkBasePyTorch method), 80
eval() (UNet2dPyTorch method), 90
eval() (UNet3dPyTorch method), 100
eval() (VGG3DClassificationNetworkPyTorch method), 71
ExponentialLRCallback (class in delira.training.callbacks.pytorch_schedulers), 123
ExponentialLRCallbackPyTorch (in module delira.training.callbacks), 126
extra_repr() (AbstractPyTorchNetwork method), 52
extra_repr() (BCEFocalLossLogitPyTorch method), 137
extra_repr() (BCEFocalLossPyTorch method), 129
extra_repr() (ClassificationNetworkBasePyTorch method), 61
extra_repr() (GenerativeAdversarialNetworkBasePyTorch method), 80
extra_repr() (UNet2dPyTorch method), 90
extra_repr() (UNet3dPyTorch method), 100
extra_repr() (VGG3DClassificationNetworkPyTorch method), 71

F

filter() (MultiStreamHandler method), 43
filter() (TrixiHandler method), 44
flat() (LookupConfig method), 148
flat() (Parameters method), 107
float() (AbstractPyTorchNetwork method), 52
float() (BCEFocalLossLogitPyTorch method), 137
float() (BCEFocalLossPyTorch method), 129
float() (ClassificationNetworkBasePyTorch method), 61
float() (GenerativeAdversarialNetworkBasePyTorch method), 80
float() (UNet2dPyTorch method), 90
float() (UNet3dPyTorch method), 100
float() (VGG3DClassificationNetworkPyTorch method), 71
float_to_pytorch_tensor() (in module delira.training.train_utils), 144
flush() (MultiStreamHandler method), 43
flush() (TrixiHandler method), 45
fold (AbstractNetworkTrainer attribute), 113
fold (PyTorchNetworkTrainer attribute), 117
format() (MultiStreamHandler method), 43
format() (TrixiHandler method), 45
forward() (AbstractPyTorchNetwork method), 52
forward() (AccuracyMetricPyTorch method), 143
forward() (AurocMetricPyTorch method), 143
forward() (BCEFocalLossLogitPyTorch method), 137
forward() (BCEFocalLossPyTorch method), 129
forward() (ClassificationNetworkBasePyTorch method), 61
forward() (GenerativeAdversarialNetworkBasePyTorch method), 80
forward() (UNet2dPyTorch method), 90
forward() (UNet3dPyTorch method), 100
forward() (VGG3DClassificationNetworkPyTorch method), 71
from_dataset() (delira.data_loading.sampler.AbstractSampler class method), 38
from_dataset() (delira.data_loading.sampler.LambdaSampler class method), 39
from_dataset() (delira.data_loading.sampler.PrevalenceRandomSampler class method), 39
from_dataset() (delira.data_loading.sampler.PrevalenceSequentialSampler class method), 41
from_dataset() (delira.data_loading.sampler.RandomSampler class method), 39
from_dataset() (delira.data_loading.sampler.SequentialSampler class method), 40
from_dataset() (delira.data_loading.sampler.StoppingPrevalenceRandomSa class method), 40
from_dataset() (delira.data_loading.sampler.StoppingPrevalenceSequentialS class method), 41
from_dataset() (delira.data_loading.sampler.WeightedRandomSampler class method), 42
fromkeys() (LookupConfig method), 149
fromkeys() (Parameters method), 108

G

generate_train_batch() (BaseDataLoader method), 36
GenerativeAdversarialNetworkBasePyTorch (class in delira.models.gan), 76
GenerativeAdversarialNetworkBasePyTorch.to() (in module delira.models.gan), 84, 85
get() (LookupConfig method), 149
get() (Parameters method), 108
get_batchgen() (BaseDataManager method), 36
get_labels() (BaseLabelGenerator method), 37
get_name() (MultiStreamHandler method), 43
get_name() (TrixiHandler method), 45

H

half() (AbstractPyTorchNetwork method), 52
half() (BCEFocalLossLogitPyTorch method), 137
half() (BCEFocalLossPyTorch method), 129
half() (ClassificationNetworkBasePyTorch method), 62
half() (GenerativeAdversarialNetworkBasePyTorch method), 80
half() (UNet2dPyTorch method), 90

half() (UNet3dPyTorch method), 100
half() (VGG3DClassificationNetworkPyTorch method), 71
handle() (MultiStreamHandler method), 43
handle() (TrixiHandler method), 45
handleError() (MultiStreamHandler method), 44
handleError() (TrixiHandler method), 45
hasattr_not_none() (LookupConfig method), 149
hasattr_not_none() (Parameters method), 108
hierarchy (Parameters attribute), 108

I

ImgSaveHandler (class in delira.logging.deprecated), 45
init_kwargs (AbstractNetwork attribute), 49
init_kwargs (AbstractPyTorchNetwork attribute), 52
init_kwargs (ClassificationNetworkBasePyTorch attribute), 62
init_kwargs (GenerativeAdversarialNetworkBasePyTorch attribute), 81
init_kwargs (UNet2dPyTorch attribute), 90
init_kwargs (UNet3dPyTorch attribute), 100
init_kwargs (VGG3DClassificationNetworkPyTorch attribute), 71
init_objects() (LookupConfig static method), 149
init_objects() (Parameters static method), 108
items() (LookupConfig method), 150
items() (Parameters method), 109

K

keys() (LookupConfig method), 150
keys() (Parameters method), 109
kfold() (AbstractExperiment method), 118
kfold() (PyTorchExperiment method), 119

L

LambdaLRCallback (class in delira.training.callbacks.pytorch_schedulers), 124
LambdaLRCallbackPyTorch (in module delira.training.callbacks), 126
LambdaSampler (class in delira.data_loading.sampler), 38
load() (AbstractExperiment static method), 119
load() (LookupConfig method), 150
load() (Parameters method), 109
load() (PyTorchExperiment static method), 120
load_checkpoint() (in module delira.io.torch), 42
load_sample_nii() (in module delira.data_loading.nii), 38
load_state() (AbstractNetworkTrainer static method), 113
load_state() (PyTorchNetworkTrainer static method), 117
load_state_dict() (AbstractPyTorchNetwork method), 53
load_state_dict() (BCEFocalLossLogitPyTorch method), 137
load_state_dict() (BCEFocalLossPyTorch method), 129
load_state_dict() (ClassificationNetworkBasePyTorch method), 62
load_state_dict() (GenerativeAdversarialNetworkBasePyTorch method), 81
load_state_dict() (UNet2dPyTorch method), 90
load_state_dict() (UNet3dPyTorch method), 100
load_state_dict() (VGG3DClassificationNetworkPyTorch method), 71
loads() (LookupConfig method), 150
loads() (Parameters method), 109
LookupConfig (class in delira.utils.config), 147

M

make_deprecated() (in module delira.utils.decorators), 145
max_energy_slice() (in module delira.utils.imageops), 145
modules() (AbstractPyTorchNetwork method), 53
modules() (BCEFocalLossLogitPyTorch method), 138
modules() (BCEFocalLossPyTorch method), 130
modules() (ClassificationNetworkBasePyTorch method), 62
modules() (GenerativeAdversarialNetworkBasePyTorch method), 81
modules() (UNet2dPyTorch method), 90
modules() (UNet3dPyTorch method), 100
modules() (VGG3DClassificationNetworkPyTorch method), 71
MultiStepLRCallback (class in delira.training.callbacks.pytorch_schedulers), 124
MultiStepLRCallbackPyTorch (in module delira.training.callbacks), 126
MultiStreamHandler (class in delira.logging), 43

N

n_batches (BaseDataManager attribute), 36
n_samples (BaseDataManager attribute), 37
name (MultiStreamHandler attribute), 44
name (TrixiHandler attribute), 45
named_buffers() (AbstractPyTorchNetwork method), 53
named_buffers() (BCEFocalLossLogitPyTorch method), 138
named_buffers() (BCEFocalLossPyTorch method), 130
named_buffers() (ClassificationNetworkBasePyTorch method), 62
named_buffers() (GenerativeAdversarialNetworkBasePyTorch method), 81
named_buffers() (UNet2dPyTorch method), 91
named_buffers() (UNet3dPyTorch method), 101
named_buffers() (VGG3DClassificationNetworkPyTorch method), 72
named_children() (AbstractPyTorchNetwork method), 53

named_children() (BCEFocalLossLogitPyTorch method), 138
named_children() (BCEFocalLossPyTorch method), 130
named_children() (ClassificationNetworkBasePyTorch method), 63
named_children() (GenerativeAdversarialNetworkBasePyTorch method), 81
named_children() (UNet2dPyTorch method), 91
named_children() (UNet3dPyTorch method), 101
named_children() (VGG3DClassificationNetworkPyTorch method), 72
named_modules() (AbstractPyTorchNetwork method), 54
named_modules() (BCEFocalLossLogitPyTorch method), 138
named_modules() (BCEFocalLossPyTorch method), 130
named_modules() (ClassificationNetworkBasePyTorch method), 63
named_modules() (GenerativeAdversarialNetworkBasePyTorch method), 82
named_modules() (UNet2dPyTorch method), 91
named_modules() (UNet3dPyTorch method), 101
named_modules() (VGG3DClassificationNetworkPyTorch method), 72
named_parameters() (AbstractPyTorchNetwork method), 54
named_parameters() (BCEFocalLossLogitPyTorch method), 139
named_parameters() (BCEFocalLossPyTorch method), 131
named_parameters() (ClassificationNetworkBasePyTorch method), 63
named_parameters() (GenerativeAdversarialNetworkBasePyTorch method), 82
named_parameters() (UNet2dPyTorch method), 92
named_parameters() (UNet3dPyTorch method), 102
named_parameters() (VGG3DClassificationNetworkPyTorch method), 73
nested_get() (LookupConfig method), 150
nested_get() (Parameters method), 109
now() (in module delira.utils.time), 147
numpy_array_func() (in module delira.utils.decorators), 145

P

Parameters (class in delira.training), 106
parameters() (AbstractPyTorchNetwork method), 54
parameters() (BCEFocalLossLogitPyTorch method), 139
parameters() (BCEFocalLossPyTorch method), 131
parameters() (ClassificationNetworkBasePyTorch method), 63
parameters() (GenerativeAdversarialNetworkBasePyTorch method), 82
parameters() (UNet2dPyTorch method), 92
parameters() (UNet3dPyTorch method), 102
parameters() (VGG3DClassificationNetworkPyTorch method), 73
permute_hierarchy() (Parameters method), 109
permute_to_hierarchy() (Parameters method), 109
permute_training_on_top() (Parameters method), 110
permute_variability_on_top() (Parameters method), 110
pop() (LookupConfig method), 150
pop() (Parameters method), 110
popitem() (LookupConfig method), 150
popitem() (Parameters method), 110
predict() (AbstractNetworkTrainer method), 114
predict() (PyTorchNetworkTrainer method), 117
prepare_batch() (AbstractNetwork static method), 49
prepare_batch() (AbstractPyTorchNetwork static method), 55
prepare_batch() (ClassificationNetworkBasePyTorch static method), 64
prepare_batch() (GenerativeAdversarialNetworkBasePyTorch static method), 83
prepare_batch() (UNet2dPyTorch static method), 92
prepare_batch() (UNet3dPyTorch static method), 102
prepare_batch() (VGG3DClassificationNetworkPyTorch static method), 73
PrevalenceRandomSampler (class in delira.data_loading.sampler), 39
PrevalenceSequentialSampler (class in delira.data_loading.sampler), 41
pytorch_batch_to_numpy() (in module delira.training.train_utils), 143
pytorch_tensor_to_numpy() (in module delira.training.train_utils), 143
PyTorchExperiment (class in delira.training), 119
PyTorchNetworkTrainer (class in delira.training), 115

R

RandomSampler (class in delira.data_loading.sampler), 39
ReduceLROnPlateauCallback (class in delira.training.callbacks.pytorch_schedulers), 125
ReduceLROnPlateauCallbackPyTorch (in module delira.training.callbacks), 126
register_backward_hook() (AbstractPyTorchNetwork method), 55
register_backward_hook() (BCEFocalLossLogitPyTorch method), 139
register_backward_hook() (BCEFocalLossPyTorch method), 131
register_backward_hook() (ClassificationNetworkBasePyTorch method), 64
register_backward_hook() (GenerativeAdversarialNetworkBasePyTorch method), 83
register_backward_hook() (UNet2dPyTorch method), 93

register_backward_hook() (UNet3dPyTorch method), 102
register_backward_hook() (VGG3DClassificationNetworkPyTorch method), 73
register_buffer() (AbstractPyTorchNetwork method), 55
register_buffer() (BCEFocalLossLogitPyTorch method), 140
register_buffer() (BCEFocalLossPyTorch method), 132
register_buffer() (ClassificationNetworkBasePyTorch method), 64
register_buffer() (GenerativeAdversarialNetworkBasePyTorch method), 83
register_buffer() (UNet2dPyTorch method), 93
register_buffer() (UNet3dPyTorch method), 103
register_buffer() (VGG3DClassificationNetworkPyTorch method), 74
register_callback() (AbstractNetworkTrainer method), 114
register_callback() (PyTorchNetworkTrainer method), 117
register_forward_hook() (AbstractPyTorchNetwork method), 55
register_forward_hook() (BCEFocalLossLogitPyTorch method), 140
register_forward_hook() (BCEFocalLossPyTorch method), 132
register_forward_hook() (ClassificationNetworkBasePyTorch method), 65
register_forward_hook() (GenerativeAdversarialNetworkBasePyTorch method), 84
register_forward_hook() (UNet2dPyTorch method), 93
register_forward_hook() (UNet3dPyTorch method), 103
register_forward_hook() (VGG3DClassificationNetworkPyTorch method), 74
register_forward_pre_hook() (AbstractPyTorchNetwork method), 56
register_forward_pre_hook() (BCEFocalLossLogitPyTorch method), 140
register_forward_pre_hook() (BCEFocalLossPyTorch method), 132
register_forward_pre_hook() (ClassificationNetworkBasePyTorch method), 65
register_forward_pre_hook() (GenerativeAdversarialNetworkBasePyTorch method), 84
register_forward_pre_hook() (UNet2dPyTorch method), 93
register_forward_pre_hook() (UNet3dPyTorch method), 103
register_forward_pre_hook() (VGG3DClassificationNetworkPyTorch method), 74
register_parameter() (AbstractPyTorchNetwork method), 56
register_parameter() (BCEFocalLossLogitPyTorch method), 140
register_parameter() (BCEFocalLossPyTorch method), 132
register_parameter() (ClassificationNetworkBasePyTorch method), 65
register_parameter() (GenerativeAdversarialNetworkBasePyTorch method), 84
register_parameter() (UNet2dPyTorch method), 94
register_parameter() (UNet3dPyTorch method), 103
register_parameter() (VGG3DClassificationNetworkPyTorch method), 74
release() (MultiStreamHandler method), 44
release() (TrixiHandler method), 45
removeFilter() (MultiStreamHandler method), 44
removeFilter() (TrixiHandler method), 45
reset_params() (UNet2dPyTorch method), 94
reset_params() (UNet3dPyTorch method), 104
run() (AbstractExperiment method), 119
run() (PyTorchExperiment method), 120

S

save() (AbstractExperiment method), 119
save() (Parameters method), 110
save() (PyTorchExperiment method), 120
save_checkpoint() (in module delira.io.torch), 42
save_state() (AbstractNetworkTrainer method), 114
save_state() (PyTorchNetworkTrainer method), 117
SequentialSampler (class in delira.data_loading.sampler), 40
set_from_string() (LookupConfig method), 150
set_from_string() (Parameters method), 110
set_name() (MultiStreamHandler method), 44
set_name() (TrixiHandler method), 45
set_thread_id() (BaseDataLoader method), 36
set_with_decode() (LookupConfig method), 151
set_with_decode() (Parameters method), 110
setdefault() (LookupConfig method), 151
setdefault() (Parameters method), 110
setFormatter() (MultiStreamHandler method), 44
setFormatter() (TrixiHandler method), 45
setLevel() (MultiStreamHandler method), 44
setLevel() (TrixiHandler method), 45
setup() (AbstractExperiment method), 119
setup() (PyTorchExperiment method), 120
share_memory() (AbstractPyTorchNetwork method), 56
share_memory() (BCEFocalLossLogitPyTorch method), 141
share_memory() (BCEFocalLossPyTorch method), 133
share_memory() (ClassificationNetworkBasePyTorch method), 65
share_memory() (GenerativeAdversarialNetworkBasePyTorch method), 84
share_memory() (UNet2dPyTorch method), 94

share_memory() (UNet3dPyTorch method), 104
share_memory() (VGG3DClassificationNetworkPyTorch method), 75
sitk_copy_metadata() (in module delira.utils.imageops), 145
sitk_new_blank_image() (in module delira.utils.imageops), 146
sitk_resample_to_image() (in module delira.utils.imageops), 146
sitk_resample_to_shape() (in module delira.utils.imageops), 146
sitk_resample_to_spacing() (in module delira.utils.imageops), 147
state_dict() (AbstractPyTorchNetwork method), 56
state_dict() (BCEFocalLossLogitPyTorch method), 141
state_dict() (BCEFocalLossPyTorch method), 133
state_dict() (ClassificationNetworkBasePyTorch method), 65
state_dict() (GenerativeAdversarialNetworkBasePyTorch method), 84
state_dict() (UNet2dPyTorch method), 94
state_dict() (UNet3dPyTorch method), 104
state_dict() (VGG3DClassificationNetworkPyTorch method), 75
StepLRCallback (class in delira.training.callbacks.pytorch_schedulers), 125
StepLRCallbackPyTorch (in module delira.training.callbacks), 126
StoppingPrevalenceRandomSampler (class in delira.data_loading.sampler), 40
StoppingPrevalenceSequentialSampler (class in delira.data_loading.sampler), 41
subdirs() (in module delira.utils.path), 147

T

to() (AbstractPyTorchNetwork method), 56
to() (BCEFocalLossLogitPyTorch method), 141
to() (BCEFocalLossPyTorch method), 133
to() (ClassificationNetworkBasePyTorch method), 66
to() (GenerativeAdversarialNetworkBasePyTorch method), 84
to() (UNet2dPyTorch method), 94
to() (UNet3dPyTorch method), 104
to() (VGG3DClassificationNetworkPyTorch method), 75
to_cmd_args_str() (LookupConfig method), 151
to_cmd_args_str() (Parameters method), 110
torch_module_func() (in module delira.utils.decorators), 145
torch_tensor_func() (in module delira.utils.decorators), 145
train() (AbstractNetworkTrainer method), 114
train() (AbstractPyTorchNetwork method), 57
train() (BCEFocalLossLogitPyTorch method), 142
train() (BCEFocalLossPyTorch method), 134
train() (ClassificationNetworkBasePyTorch method), 67
train() (GenerativeAdversarialNetworkBasePyTorch method), 85
train() (PyTorchNetworkTrainer method), 118
train() (UNet2dPyTorch method), 95
train() (UNet3dPyTorch method), 105
train() (VGG3DClassificationNetworkPyTorch method), 76
train_test_split() (AbstractDataset method), 34
train_test_split() (BaseCacheDataset method), 35
train_test_split() (BaseDataManager method), 37
train_test_split() (BaseLazyDataset method), 34
training_on_top (Parameters attribute), 111
TrixiHandler (class in delira.logging), 44
type() (AbstractPyTorchNetwork method), 58
type() (BCEFocalLossLogitPyTorch method), 142
type() (BCEFocalLossPyTorch method), 134
type() (ClassificationNetworkBasePyTorch method), 67
type() (GenerativeAdversarialNetworkBasePyTorch method), 86
type() (UNet2dPyTorch method), 95
type() (UNet3dPyTorch method), 105
type() (VGG3DClassificationNetworkPyTorch method), 76

U

UNet2dPyTorch (class in delira.models.segmentation), 86
UNet2dPyTorch.to() (in module delira.models.segmentation), 94
UNet3dPyTorch (class in delira.models.segmentation), 96
UNet3dPyTorch.to() (in module delira.models.segmentation), 104
update() (LookupConfig method), 151
update() (Parameters method), 111
update_missing() (LookupConfig method), 152
update_missing() (Parameters method), 111
update_state() (AbstractNetworkTrainer method), 114
update_state() (PyTorchNetworkTrainer method), 118

V

values() (LookupConfig method), 152
values() (Parameters method), 112
variability_on_top (Parameters attribute), 112
VGG3DClassificationNetworkPyTorch (class in delira.models.classification), 67
VGG3DClassificationNetworkPyTorch.to() (in module delira.models.classification), 75
VisdomImageHandler (class in delira.logging.deprecated), 46
VisdomImageSaveHandler (class in delira.logging.deprecated), 47
VisdomImageSaveStreamHandler (class in delira.logging.deprecated), 47

VisdomStreamHandler (class in delira.logging.deprecated), 48

W

weight_init() (UNet2dPyTorch static method), 95
weight_init() (UNet3dPyTorch static method), 105
WeightedRandomSampler (class in delira.data_loading.sampler), 41

Z

zero_grad() (AbstractPyTorchNetwork method), 58
zero_grad() (BCEFocalLossLogitPyTorch method), 142
zero_grad() (BCEFocalLossPyTorch method), 134
zero_grad() (ClassificationNetworkBasePyTorch method), 67
zero_grad() (GenerativeAdversarialNetworkBasePyTorch method), 86
zero_grad() (UNet2dPyTorch method), 96
zero_grad() (UNet3dPyTorch method), 105
zero_grad() (VGG3DClassificationNetworkPyTorch method), 76