delira Documentation

Release v0.4.1+262.g3b7794b.dirty

Justus Schock, Michael Baumgartner, Oliver Rippel, Christoph Ha

Aug 07, 2019

GETTING STARTED

CHAPTER

ONE

GETTING STARTED

1.1 Backends

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

The currently supported backends are:

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

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

- torchscript : Suffix `torchscript`

  Note: `delira` with `torchscript` backend dies currently not support Multi-GPU training.

- tensorflow eager execution: Suffix `tensorflow`

  Note: `delira` with `tensorflow eager` backend dies currently not support Multi-GPU training.

- tensorflow graph mode: Suffix `tensorflow`

  Note: `delira` with `tensorflow graph` backend dies currently not support Multi-GPU training.

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

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

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

1.2 Installation

Backend	Binary Installation	Source Installation	Notes
None	`pip install delira`	`pip install git+https://github.com/delira-dev/delira.git`	Training not possible if backend is not installed separately
torch	`pip install delira[torch]`	`git clone https://github.com/delira-dev/delira.git && cd delira && pip install .[torch]`	`delira` with `torch` backend supports mixed-precision training via NVIDIA/apex (must be installed separately).
torchscript	`pip install delira[torchscript]`	`git clone https://github.com/delira-dev/delira.git && cd delira && pip install .[torchscript]`	The `torchscript` backend currently supports only single-GPU-training
tensorflow eager	`pip install delira[tensorflow]`	`git clone https://github.com/delira-dev/delira.git && cd delira && pip install .[tensorflow]`	the `tensorflow` backend is still very experimental and lacks some features
tensorflow graph	`pip install delira[tensorflow]`	`git clone https://github.com/delira-dev/delira.git && cd delira && pip install .[tensorflow]`	the `tensorflow` backend is still very experimental and lacks some features
scikit-learn	`pip install delira`	`pip install git+https://github.com/delira-dev/delira.git`	/
'chainer'	`pip install delira[chainer]`	`git clone https://github.com/delira-dev/delira.git && cd delira && pip install .[chainer]`	/
Full	`pip install delira[full]`	`git clone https://github.com/delira-dev/delira.git && cd delira && pip install .[full]`	All backends will be installed

CHAPTER

TWO

DELIRA INTRODUCTION

Last updated: 09.05.2019

Authors: Justus Schock, Christoph Haarburger

2.1 Loading Data

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

2.1.1 The Dataset

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

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

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

This means, you have to pass the path to the directory containing your data (`data_path`), a function to load a single sample of your data (`load_fn`). To get a single sample of your dataset after creating it, you can index it like this: `dataset[0]`. Additionally you can iterate over your dataset just like over any other `python` iterator via

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

or enumerate it via

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

.

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

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

```
from delira.data_loading import BaseLazyDataset, default_load_fn_2d
dataset_train = BaseLazyDataset("/images/datasets/external/mnist/train",
                                default_load_fn_2d, img_shape=(224, 224))
```

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

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

Now, let's just initialize our trainset:

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

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

```
dataset_train[0]
```

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

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

2.1.2 The Dataloader

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

```
from delira.data_loading import BaseDataLoader

batch_size = 32

loader_train = BaseDataLoader(dataset_train, batch_size)
```

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

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

2.1.3 The Datamanager

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

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

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

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

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

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

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

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

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

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

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

Iterating over a DataManager is possible in simple loops:

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

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

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

2.1.4 Sampler

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

- `AbstractSampler`

- SequentialSampler
- PrevalenceSequentialSampler
- RandomSampler
- PrevalenceRandomSampler
- WeightedRandomSampler
- LambdaSampler

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

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

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

2.2 Models

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

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

- __init__
- closure
- prepare_batch
- __call__

2.2.1 __init__

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

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

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

    Parameters
    ----------
```

(continues on next page)

(continued from previous page)

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

2.2.2 closure

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

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

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

2.2.3 prepare_batch

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

2.3 Abstract Networks for specific Backends

2.3.1 PyTorch

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

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

forward

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

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

(continues on next page)

(continued from previous page)

```
    out_3 = self.conv3(out2)

    return {"pred": out_3}
```

prepare_batch

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

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

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

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

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

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

    return return_dict
```

and can be customized by subclassing the `AbstractPyTorchNetwork`.

closure example

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

    """

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

    loss_vals = {}
    metric_vals = {}
    total_loss = 0

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

    else:
        context_man = torch.no_grad

    with context_man():

        inputs = data_dict.pop("data")
        # obtain outputs from network
        preds = model(inputs)["pred"]

        if data_dict:

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

(continues on next page)

(continued from previous page)

```
                    total_loss += _loss_val

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

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

        else:

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

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

            loss_vals = eval_loss_vals
            metric_vals = eval_metrics_vals

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

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

        return metric_vals, loss_vals, preds

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

2.3.2 Other examples

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

2.4 Training

2.4.1 Parameters

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

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

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

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

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

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

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

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

2.4.2 Trainer

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

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

Training your network might look like this:

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

# path where checkpoints should be saved
```

(continues on next page)

(continued from previous page)

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

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

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

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

2.4.3 Experiment

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

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

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

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

experiment.run(manager_train, manager_val)
```

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

2.5 Logging

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

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

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

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

2.5.1 `MultiStreamHandler`

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

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

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

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

```
NUM_ITERS = 4

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

2.6 More Examples

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

CHAPTER

THREE

CLASSIFICATION WITH DELIRA - A VERY SHORT INTRODUCTION

Author: Justus Schock

Date: 04.12.2018

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

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

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

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

3.1 Logging and Visualization

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

```
visdom -port=9999
```

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

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

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

logger_cls = PytorchVisdomLogger

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


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

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

3.2 Data Preparation

3.2.1 Loading

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

```
from delira.data_loading import TorchvisionClassificationDataset

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

3.2.2 Augmentation

For Data-Augmentation we will apply a few transformations:

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

transforms = Compose([
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

3.3 Training

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

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


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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

3.4 See Also

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

CHAPTER

FOUR

GENERATIVE ADVERSARIAL NETS WITH DELIRA - A VERY SHORT INTRODUCTION

Author: Justus Schock

Date: 04.12.2018

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

4.1 HyperParameters

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

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

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

4.2 Logging and Visualization

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

```
visdom -port=9999
```

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

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

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

logger_cls = PytorchVisdomLogger

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


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

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

4.3 Data Preparation

4.3.1 Loading

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

```
from delira.data_loading import TorchvisionClassificationDataset

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

4.3.2 Augmentation

For Data-Augmentation we will apply a few transformations:

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

4.4 Training

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

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


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

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

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

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

4.5 See Also

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

CHAPTER

FIVE

SEGMENTATION IN 2D USING U-NETS WITH DELIRA - A VERY SHORT INTRODUCTION

Author: Justus Schock, Alexander Moriz

Date: 17.12.2018

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

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

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

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

5.1 Logging and Visualization

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

```
visdom -port=9999
```

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

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

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

logger_cls = PytorchVisdomLogger

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


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

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

5.2 Data Praparation

5.2.1 Loading

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

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

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

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

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

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

```
import SimpleITK as sitk
import numpy as np

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

```
import matplotlib.pyplot as plt

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

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

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

```
from delira.data_loading import AbstractDataset

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

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

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

Now, we can finally instantiate our datasets:

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

5.2.2 Augmentation

For Data-Augmentation we will apply a few transformations:

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

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

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

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

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

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

5.3 Training

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

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


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

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

(continues on next page)

(continued from previous page)

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

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

5.4 See Also

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

CHAPTER

SIX

SEGMENTATION IN 3D USING U-NETS WITH DELIRA - A VERY SHORT INTRODUCTION

Author: Justus Schock, Alexander Moriz

Date: 17.12.2018

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

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

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

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

6.1 Logging and Visualization

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

```
visdom -port=9999
```

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

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

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

logger_cls = PytorchVisdomLogger

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


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

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

6.2 Data Praparation

6.2.1 Loading

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

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

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

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

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

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

```
import SimpleITK as sitk
import numpy as np

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

```
from delira.data_loading import AbstractDataset

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

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

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

Now, we can finally instantiate our datasets:

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

6.2.2 Augmentation

For Data-Augmentation we will apply a few transformations:

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

6.3 Training

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

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


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

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

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

6.4 See Also

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

CHAPTER

SEVEN

HOW TO: INTEGRATE YOUR OWN COMPUTATION BACKEND

Author: Justus Schock

Date: 15.05.2019

This howto will take you on a trip through the `delira` internals, while we will see, how to add a custom computation backend on the examplaric case of the `torch.jit` or `TorchScript` backend

7.1 Model Definitions

In order to implement a network, we will first have to define the network itself. In `delira` there is a single backend-specific implementation of an abstract network class for each of the backends. These interface classes are all based on the `AbstractNetwork`-class, defining the major API.

So let's start having a look at this class to see, what we will have to implement for our own backend.

Of course we will have to implement an `__init__` defining our class. The `__init__` of `AbstractNetwork` (which should be called during our the `__init__` of our baseclass) accepts a number of kwargs and simply registers them to be `init_kwargs`, so there is nothing we have to take care of.

The next function to inspect is the `__call__` function, which makes the class callable and the docstrings indicate, that it should take care of our model's forward-pass.

After the `__call__` we now have the `closure` function, which defines a single training step (including, but not limited to, forward-pass, calculation of losses and train-metrics, backward-pass and optimization).

The last method to implement is the `prepare_batch` function which converts the input to a suitable format and the correct data-type and device.

7.1.1 TorchScript Limitations

Since we want to implement an abstract network class for this specific backend, we should have a look on how to generally implement models in this backend.

According the the PyTorch docs this works as follows:

> You can write TorchScript code directly using Python syntax. You do this using the `torch.jit.script` decorator (for functions) or `torch.jit.script_method` decorator (for methods) on subclasses of `ScriptModule`. With this decorator the body of the annotated function is directly translated into TorchScript. TorchScript itself is a subset of the Python language, so not all features in Python work, but we provide enough functionality to compute on tensors and do control-dependent operations.

Since our use-case is to implement the interface class for networks, we want to use the way of subclassing `torch.jit.ScriptModule`, implement it's `forward` and use the `torch.jit.script_method` decorator on it.

The example given in the very same docs for this case is:

```
import torch
class MyScriptModule(torch.jit.ScriptModule):
    def __init__(self, N, M):
        super().__init__()
        self.weight = torch.nn.Parameter(torch.rand(N, M))

    @torch.jit.script_method
    def forward(self, input):
        return self.weight.mv(input)

my_script_module = MyScriptModule(5, 3)
input_tensor = torch.rand(3)
my_script_module(input_tensor)
```

```
tensor([0.4997, 0.2955, 0.1588, 0.1873, 0.4753], grad_fn=<MvBackward>)
```

7.1.2 Merging TorchScript into our Abstract Class

This little example gives us a few things, we have to do for a successful definition of our base class:

1.) Our class has to subclass both, the `AbstractNetwork` and the `torch.jit.ScriptModule` classes.

2.) We need to implement a `forward` method, which takes care of the forward-pass (as it's name indicates).

3.) We don't have to take care of the backward-pass (thanks to `PyTorch`'s and `TorchScript`'s AutoGrad (which is a framework for automatic differentiation).

4.) Since `torch.jit.ScriptModule` is callable (seen in the example), it already implements a `__call__` method and we may simply use this one.

5.) The `closure` is completely network-dependent and thus has to remain an abstract method here.

6.) The `prepare_batch` function also depends on the combination of network, inputs and loss functions to use, but we can at least give a prototype of such an function, which handles the devices correctly and converts everything to `float`

7.1.3 Actual Implementation

Now, let's start with the actual implementation and do one function by another and keep the things in mind, we just discovered.

Class Signature and `__init__`-Method

To subclass both networks, we cannot use the simple `super().__init__` approach, because we have to init both parent classes, so we do

```
class AbstractTorchScriptNetwork(AbstractNetwork, torch.jit.ScriptModule):

    @abc.abstractmethod
    def __init__(self, optimize=True, **kwargs):
        """

        Parameters
```

(continues on next page)

(continued from previous page)

```
        ----------
        optimize : bool
            whether to optimize the network graph or not; default: True
        **kwargs :
            additional keyword arguments (passed to :class:`AbstractNetwork`)
        """
        torch.jit.ScriptModule.__init__(self, optimize=optimize)
        AbstractNetwork.__init__(self, **kwargs)
```

instead. This ensures all parent classes to be initialized correctly.

`__call__`-Method

As mentioned above, the `__call__` method is very easy to implement, because we can simply use the implementation of our `TorchScript` base class like this:

```
def __call__(self, *args, **kwargs):
    """
    Calls Forward method

    Parameters
    ----------
    *args :
        positional arguments (passed to `forward`)
    **kwargs :
        keyword arguments (passed to `forward`)

    Returns
    -------
    Any
        result: module results of arbitrary type and number

    """
    return torch.jit.ScriptModule.__call__(self, *args, **kwargs)
```

This also ensures, that we can pass an arbitrary number or positional and keyword arguments of arbitrary types to it (which are all passed to the `forward`-function). The advantage over directly calling the `forward` method here, is that the `ScriptModule.__call__` already does the handling of forward-pre-hooks, forward-hooks and backward-hooks.

`closure`-Method

Since this method is highly model-dependant, we just don't implement it, which forces the user to implement it (since it is marked as an `abstractmethod` in `AbstractExperiment`).

`prepare_batch`-Method

The above mentioned prototype of pushing everything to the correct device and convert it to float looks like this:

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

(continues on next page)

(continued from previous page)

```
    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).to(torch.float)}

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

    return return_dict
```

Since we don't want to use any of the model's attributes here (and for conformity with the `AbstractNetwork` class), this method is defined as `staticmethod`, meaning it is class-bound, not instance-bound. The `closure` method has to be a `staticmethod` too.

`forward`-Method

The only thing left now, is the `forward` method, which is internally called by `ScriptModule.__call__`. The bad news is: We currently can't implement it. Subclassing a `ScriptModule` to overwrite a function decorated with `torch.jit.script_method` is not (yet) supported, but will be soon, once this PR is merged and released.

For now: you simply have to implement this method in your own network despite the missing of an abstract interface-method.

Putting it all together

If we combine all the function implementations to one class, it looks like this:

```
class AbstractTorchScriptNetwork(AbstractNetwork, torch.jit.ScriptModule):

    """
    Abstract Interface Class for TorchScript Networks. For more information
    have a look at https://pytorch.org/docs/stable/jit.html#torchscript

    Warnings
    --------
    In addition to the here defined API, a forward function must be
    implemented and decorated with ``@torch.jit.script_method``
```

(continues on next page)

(continued from previous page)

```
    """
    @abc.abstractmethod
    def __init__(self, optimize=True, **kwargs):
        """

        Parameters
        ----------
        optimize : bool
            whether to optimize the network graph or not; default: True
        **kwargs :
            additional keyword arguments (passed to :class:`AbstractNetwork`)
        """
        torch.jit.ScriptModule.__init__(self, optimize=optimize)
        AbstractNetwork.__init__(self, **kwargs)

    def __call__(self, *args, **kwargs):
        """
        Calls Forward method

        Parameters
        ----------
        *args :
            positional arguments (passed to `forward`)
        **kwargs :
            keyword arguments (passed to `forward`)

        Returns
        -------
        Any
            result: module results of arbitrary type and number

        """
        return torch.jit.ScriptModule.__call__(self, *args, **kwargs)

    @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

        """
```

(continues on next page)

(continued from previous page)

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

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

        return return_dict
```

7.2 Saving and loading

Now that we have the ability to implement `delira`-suitable TorchScript models, we want to store them on disk and load them again, so that we don't have to retrain them every time we want to use them. These I/O functions are usually located in `delira.io`.

7.2.1 Saving

Our saving function utilizes multiple functions: `torch.jit.save` to simply save the model (including it's graph) and the `save_checkpoint_torch` function implemented for the `PyTorch` backend to store the trainer state, since `TorchScript` allows us to use plain `PyTorch` optimizers.

The implementation of the function looks like this:

```
def save_checkpoint_torchscript(file: str, model=None, optimizers={},
                                epoch=None, **kwargs):
    """
    Save current checkpoint to two different files:
        1.) ``file + "_model.ptj"``: Will include the state of the model
            (including the graph; this is the opposite to
            :func:`save_checkpoint`)
        2.) ``file + "_trainer_state.pt"``: Will include the states of all
            optimizers and the current epoch (if given)

    Parameters
    ----------
    file : str
        filepath the model should be saved to
    model : AbstractPyTorchJITNetwork 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)

    """

    # remove file extension if given
    if any([file.endswith(ext) for ext in [".pth", ".pt", ".ptj"]]):
        file = file.rsplit(".", 1)[0]

    if isinstance(model, AbstractPyTorchJITNetwork):
        torch.jit.save(model, file + "_model.ptj")
```

(continues on next page)

(continued from previous page)

```
    if optimizers or epoch is not None:
        save_checkpoint_torch(file + "_trainer_state.pt", None,
                              optimizers=optimizers, epoch=epoch, **kwargs)
```

7.2.2 Loading

To load a model, which has been saved to disk by this function we have to revert each part of it. We do this by using `torch.jit.load` for the model (and the graph) and `load_checkpoint_torch` by the `PyTorch` backend. The actual implementation is given here:

```
def load_checkpoint_torchscript(file: str, **kwargs):
    """
    Loads a saved checkpoint consisting of 2 files
    (see :func:`save_checkpoint_jit` for details)

    Parameters
    ----------
    file : str
        filepath to a file containing a saved model
    **kwargs:
        Additional keyword arguments (passed to torch.load)
        Especially "map_location" is important to change the device the
        state_dict should be loaded to

    Returns
    -------
    OrderedDict
        checkpoint state_dict

    """
    # remove file extensions
    if any([file.endswith(ext) for ext in [".pth", ".pt", ".ptj"]]):
        file = file.rsplit(".", 1)[0]

    # load model
    if os.path.isfile(file + ".ptj"):
        model_file = file
    elif os.path.isfile(file + "_model.ptj"):
        model_file = file + "_model.ptj"
    else:
        raise ValueError("No Model File found for %s" % file)

    # load trainer state (if possible)
    trainer_file = model_file.replace("_model.ptj", "_trainer_state.pt")
    if os.path.isfile(trainer_file):
        trainer_state = load_checkpoint_torch(trainer_file, **kwargs)

    else:
        trainer_state = {"optimizer": {},
                         "epoch": None}

    trainer_state.update({"model": torch.jit.load(model_file)})

    return trainer_state
```

7.3 A Trainer to train

Now, that we can define and save/load our models, we want to train them. Luckily `delira` has already implemented a very modular backend-agnostic trainer (the `BaseNetworkTrainer`) and build upon this a `PyTorchNetworkTrainer`. Since the training process in PyTorch and TorchScript is nearly the same, we can just extend the `PyTorchNetworkTrainer`. Usually one would have to extend the `BaseNetworkTrainer` to provide some backend specific functions (like necessary initializations, optimizer setup, seeding etc.). To see how this is done, you could either have a look at the `PyTorchNetworkTrainer` or the `TfNetworkTrainer` for tensorflow, which are both following this principle. Usually the only stuff to completely change is the loading/saving behavior and the `_setup` function, which defines the backend-specific initialization. Some other functions may have to be extended (by implementing the extension and calling the parent-classes function).

7.3.1 Things to change:

By Subclassing the `PyTorchNetworkTrainer` we have to change the following things:

- The trainer's default arguments
- The behavior for trying to resume a previous training
- The saving, loading and updating behavior

We will access this one by one:

The Default Arguments

We want to use `AbstractTorchScriptNetworks` instead of `AbstractPyTorchNetworks` here and we have to change the behavior if passing multiple GPUs, because currently Multi-GPU training is not supported by `TorchScript`.

To do this: we implement the functions `__init__`, apply our changes and forward these changes to the call of the base-classes `__init__` like this (omitted docstrings for the sake of shortness):

```
class TorchScriptNetworkTrainer(PyTorchNetworkTrainer):
        def __init__(self,
                     network: AbstractTorchScriptNetwork,
                     save_path: str,
                     key_mapping,
                     losses=None,
                     optimizer_cls=None,
                     optimizer_params={},
                     train_metrics={},
                     val_metrics={},
                     lr_scheduler_cls=None,
                     lr_scheduler_params={},
                     gpu_ids=[],
                     save_freq=1,
                     optim_fn=create_optims_default,
                     logging_type="tensorboardx",
                     logging_kwargs={},
                     fold=0,
                     callbacks=[],
                     start_epoch=1,
                     metric_keys=None,
                     convert_batch_to_npy_fn=convert_torch_tensor_to_npy,
```

(continues on next page)

(continued from previous page)

```
                    criterions=None,
                    val_freq=1,
                    **kwargs):

            if len(gpu_ids) > 1:
                # only use first GPU due to
                # https://github.com/pytorch/pytorch/issues/15421
                gpu_ids = [gpu_ids[0]]
                logging.warning("Multiple GPUs specified. Torch JIT currently "
                                "supports only single-GPU training. "
                                "Switching to use only the first GPU for now...")

            super().__init__(network=network, save_path=save_path,
                             key_mapping=key_mapping, losses=losses,
                             optimizer_cls=optimizer_cls,
                             optimizer_params=optimizer_params,
                             train_metrics=train_metrics,
                             val_metrics=val_metrics,
                             lr_scheduler_cls=lr_scheduler_cls,
                             lr_scheduler_params=lr_scheduler_params,
                             gpu_ids=gpu_ids, save_freq=save_freq,
                             optim_fn=optim_fn, logging_type=logging_type,
                             logging_kwargs=logging_kwargs, fold=fold,
                             callbacks=callbacks,
                             start_epoch=start_epoch, metric_keys=metric_keys,
                             convert_batch_to_npy_fn=convert_batch_to_npy_fn,
                             mixed_precision=False, mixed_precision_kwargs={},
                             criterions=criterions, val_freq=val_freq, **kwargs
                             )
```

Resuming Training

For resuming the training, we have to completely change the `try_resume_training` function and cannot reuse the parent's implementation of it. Thus, we don't call `super().try_resume_training` here, but completely reimplement it from scratch:

```
def try_resume_training(self):
    """
    Load the latest state of a previous training if possible

    """
    # Load latest epoch file if available
    if os.path.isdir(self.save_path):
        # check all files in directory starting with "checkpoint" and
        # not ending with "_best.pth"
        files = [x for x in os.listdir(self.save_path)
                 if os.path.isfile(os.path.join(self.save_path, x))
                 and x.startswith("checkpoint")
                 and not x.endswith("_best.ptj")
                 ]

        # if list is not empty: load previous state
        if files:

            latest_epoch = max([
```

(continues on next page)

(continued from previous page)

```
                int(x.rsplit("_", 1)[-1].rsplit(".", 1)[0])
                for x in files])

            latest_state_path = os.path.join(self.save_path,
                                             "checkpoint_epoch_%d.ptj"
                                             % latest_epoch)

            # if pth file does not exist, load pt file instead
            if not os.path.isfile(latest_state_path):
                latest_state_path = latest_state_path[:-1]

            logger.info("Attempting to load state from previous \
                        training from %s" % latest_state_path)
            try:
                self.update_state(latest_state_path)
            except KeyError:
                logger.warning("Previous State could not be loaded, \
                                although it exists.Training will be \
                                restarted")
```

Saving and Loading

Now we need to change the saving and loading behavior. As always we try to reuse as much code as possible to avoid code duplication.

Saving

To save the current training state, we simply call the `save_checkpoint_torchscript` function:

```
def save_state(self, file_name, epoch, **kwargs):
    """
    saves the current state via
    :func:`delira.io.torch.save_checkpoint_jit`

    Parameters
    ----------
    file_name : str
        filename to save the state to
    epoch : int
        current epoch (will be saved for mapping back)
    **kwargs :
        keyword arguments

    """
    if file_name.endswith(".pt") or file_name.endswith(".pth"):
        file_name = file_name.rsplit(".", 1)[0]

    save_checkpoint_torchscript(file_name, self.module, self.optimizers,
                                **kwargs)
```

Loading

To load the training state, we simply return the state loaded by `load_checkpoint_torchscript`. Since we don't use any arguments of the trainer itself here, the function is a `staticmethod`:

```
@staticmethod
def load_state(file_name, **kwargs):
    """
    Loads the new state from file via
    :func:`delira.io.torch.load_checkpoint:jit`

    Parameters
    ----------
    file_name : str
        the file to load the state from
    **kwargs : keyword arguments

    Returns
    -------
    dict
        new state

    """
    return load_checkpoint_torchscript(file_name, **kwargs)
```

Updating

After we loaded the new state, we need to update the trainer's internal state by this new state.

We do this by directly assigning the model here (since the graph was stored/loaded too) instead of only updating the state_dict and calling the parent-classes method afterwards:

```
def _update_state(self, new_state):
    """
    Update the state from a given new state

    Parameters
    ----------
    new_state : dict
        new state to update internal state from

    Returns
    -------
    :class:`PyTorchNetworkJITTrainer`
        the trainer with a modified state

    """
    if "model" in new_state:
        self.module = new_state.pop("model").to(self.input_device)

    return super()._update_state(new_state)
```

7.3.2 A Whole Trainer

After combining all the changes above, we finally get our new trainer as:

```
class TorchScriptNetworkTrainer(PyTorchNetworkTrainer):
    def __init__(self,
                 network: AbstractTorchScriptNetwork,
                 save_path: str,
                 key_mapping,
                 losses=None,
                 optimizer_cls=None,
                 optimizer_params={},
                 train_metrics={},
                 val_metrics={},
                 lr_scheduler_cls=None,
                 lr_scheduler_params={},
                 gpu_ids=[],
                 save_freq=1,
                 optim_fn=create_optims_default,
                 logging_type="tensorboardx",
                 logging_kwargs={},
                 fold=0,
                 callbacks=[],
                 start_epoch=1,
                 metric_keys=None,
                 convert_batch_to_npy_fn=convert_torch_tensor_to_npy,
                 criterions=None,
                 val_freq=1,
                 **kwargs):
        """

        Parameters
        ----------
        network : :class:`AbstractPyTorchJITNetwork`
            the network to train
        save_path : str
            path to save networks to
        key_mapping : dict
            a dictionary containing the mapping from the ``data_dict`` to
            the actual model's inputs.
            E.g. if a model accepts one input named 'x' and the data_dict
            contains one entry named 'data' this argument would have to
            be ``{'x': 'data'}``
        losses : dict
            dictionary containing the training losses
        optimizer_cls : subclass of tf.train.Optimizer
            optimizer class implementing the optimization algorithm of
            choice
        optimizer_params : dict
            keyword arguments passed to optimizer during construction
        train_metrics : dict, optional
            metrics, which will be evaluated during train phase
            (should work on framework's tensor types)
        val_metrics : dict, optional
            metrics, which will be evaluated during test phase
            (should work on numpy arrays)
        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
```

(continues on next page)

(continued from previous page)

```
        list containing ids of GPUs to use; if empty: use cpu instead
        Currently ``torch.jit`` only supports single GPU-Training,
        thus only the first GPU will be used if multiple GPUs are passed
    save_freq : int
        integer specifying how often to save the current model's state.
        State is saved every state_freq epochs
    optim_fn : function
        creates a dictionary containing all necessary optimizers
    logging_type : str or callable
        the type of logging. If string: it must be one of
        ["visdom", "tensorboardx"]
        If callable: it must be a logging handler class
    logging_kwargs : dict
        dictionary containing all logging keyword arguments
    fold : int
        current cross validation fold (0 per default)
    callbacks : list
        initial callbacks to register
    start_epoch : int
        epoch to start training at
    metric_keys : dict
        dict specifying which batch_dict entry to use for which metric as
        target; default: None, which will result in key "label" for all
        metrics
    convert_batch_to_npy_fn : type, optional
        function converting a batch-tensor to numpy, per default this is
        a function, which detaches the tensor, moves it to cpu and the
        calls ``.numpy()`` on it
    mixed_precision : bool
        whether to use mixed precision or not (False per default)
    mixed_precision_kwargs : dict
        additional keyword arguments for mixed precision
    val_freq : int
        validation frequency specifying how often to validate the trained
        model (a value of 1 denotes validating every epoch,
        a value of 2 denotes validating every second epoch etc.);
        defaults to 1
    **kwargs :
        additional keyword arguments

    """

    if len(gpu_ids) > 1:
        # only use first GPU due to
        # https://github.com/pytorch/pytorch/issues/15421
        gpu_ids = [gpu_ids[0]]
        logging.warning("Multiple GPUs specified. Torch JIT currently "
                        "supports only single-GPU training. "
                        "Switching to use only the first GPU for now...")

    super().__init__(network=network, save_path=save_path,
                     key_mapping=key_mapping, losses=losses,
                     optimizer_cls=optimizer_cls,
                     optimizer_params=optimizer_params,
                     train_metrics=train_metrics,
                     val_metrics=val_metrics,
                     lr_scheduler_cls=lr_scheduler_cls,
```

(continues on next page)

(continued from previous page)

```
                        lr_scheduler_params=lr_scheduler_params,
                        gpu_ids=gpu_ids, save_freq=save_freq,
                        optim_fn=optim_fn, logging_type=logging_type,
                        logging_kwargs=logging_kwargs, fold=fold,
                        callbacks=callbacks,
                        start_epoch=start_epoch, metric_keys=metric_keys,
                        convert_batch_to_npy_fn=convert_batch_to_npy_fn,
                        mixed_precision=False, mixed_precision_kwargs={},
                        criterions=criterions, val_freq=val_freq, **kwargs
                        )

    def try_resume_training(self):
        """
        Load the latest state of a previous training if possible

        """
        # Load latest epoch file if available
        if os.path.isdir(self.save_path):
            # check all files in directory starting with "checkpoint" and
            # not ending with "_best.pth"
            files = [x for x in os.listdir(self.save_path)
                     if os.path.isfile(os.path.join(self.save_path, x))
                     and x.startswith("checkpoint")
                     and not x.endswith("_best.ptj")
                     ]

            # if list is not empty: load previous state
            if files:

                latest_epoch = max([
                    int(x.rsplit("_", 1)[-1].rsplit(".", 1)[0])
                    for x in files])

                latest_state_path = os.path.join(self.save_path,
                                                 "checkpoint_epoch_%d.ptj"
                                                 % latest_epoch)

                # if pth file does not exist, load pt file instead
                if not os.path.isfile(latest_state_path):
                    latest_state_path = latest_state_path[:-1]

                logger.info("Attempting to load state from previous \
                            training from %s" % latest_state_path)
                try:
                    self.update_state(latest_state_path)
                except KeyError:
                    logger.warning("Previous State could not be loaded, \
                                    although it exists.Training will be \
                                    restarted")

    def save_state(self, file_name, epoch, **kwargs):
        """
        saves the current state via
        :func:`delira.io.torch.save_checkpoint_jit`

        Parameters
        ----------
```

(continues on next page)

(continued from previous page)

```
        file_name : str
            filename to save the state to
        epoch : int
            current epoch (will be saved for mapping back)
        **kwargs :
            keyword arguments

        """
        if file_name.endswith(".pt") or file_name.endswith(".pth"):
            file_name = file_name.rsplit(".", 1)[0]

        save_checkpoint_torchscript(file_name, self.module, self.optimizers,
                                    **kwargs)

    @staticmethod
    def load_state(file_name, **kwargs):
        """
        Loads the new state from file via
        :func:`delira.io.torch.load_checkpoint:jit`

        Parameters
        ----------
        file_name : str
            the file to load the state from
        **kwargs : keyword arguments

        Returns
        -------
        dict
            new state

        """
        return load_checkpoint_torchscript(file_name, **kwargs)

    def _update_state(self, new_state):
        """
        Update the state from a given new state

        Parameters
        ----------
        new_state : dict
            new state to update internal state from

        Returns
        -------
        :class:`PyTorchNetworkJITTrainer`
            the trainer with a modified state

        """
        if "model" in new_state:
            self.module = new_state.pop("model").to(self.input_device)

        return super()._update_state(new_state)
```

7.4 Wrapping it all in an Experiment

To have access to methods like a K-Fold (and the not yet finished) hyperparameter tuning, we need to wrap the trainer in an Experiment. We will use the same approach as we did for implementing the trainer: Extending an already provided class.

This time we extend the `PyTorchExperiment` which itself extends the `BaseExperiment` by some backend-specific defaults, types and seeds.

Our whole class definition just changes the default arguments of the `PyTorchExperiment` and thus, we only have to implenent it's `__init__`:

```
class TorchScriptExperiment(PyTorchExperiment):
    def __init__(self,
                 params: typing.Union[str, Parameters],
                 model_cls: AbstractTorchScriptNetwork, # not AbstractPyTorchNetwork␣
↪anymore
                 n_epochs=None,
                 name=None,
                 save_path=None,
                 key_mapping=None,
                 val_score_key=None,
                 optim_builder=create_optims_default_pytorch,
                 checkpoint_freq=1,
                 trainer_cls=TorchScriptNetworkTrainer, # not PyTorchNetworkTrainer␣
↪anymore
                 **kwargs):
        """

        Parameters
        ----------
        params : :class:`Parameters` or str
            the training parameters, if string is passed,
            it is treated as a path to a pickle file, where the
            parameters are loaded from
        model_cls : Subclass of :class:`AbstractTorchScriptNetwork`
            the class implementing the model to train
        n_epochs : int or None
            the number of epochs to train, if None: can be specified later
            during actual training
        name : str or None
            the Experiment's name
        save_path : str or None
            the path to save the results and checkpoints to.
            if None: Current working directory will be used
        key_mapping : dict
            mapping between data_dict and model inputs (necessary for
            prediction with :class:`Predictor`-API), if no keymapping is
            given, a default key_mapping of {"x": "data"} will be used here
        val_score_key : str or None
            key defining which metric to use for validation (determining
            best model and scheduling lr); if None: No validation-based
            operations will be done (model might still get validated,
            but validation metrics can only be logged and not used further)
        optim_builder : function
            Function returning a dict of backend-specific optimizers.
            defaults to :func:`create_optims_default_pytorch`
```

(continues on next page)

(continued from previous page)

```
        checkpoint_freq : int
            frequency of saving checkpoints (1 denotes saving every epoch,
            2 denotes saving every second epoch etc.); default: 1
        trainer_cls : subclass of :class:`TorchScriptNetworkTrainer`
            the trainer class to use for training the model, defaults to
            :class:`TorchScriptNetworkTrainer`
        **kwargs :
            additional keyword arguments

        """
        super().__init__(params=params, model_cls=model_cls,
                         n_epochs=n_epochs, name=name, save_path=save_path,
                         key_mapping=key_mapping,
                         val_score_key=val_score_key,
                         optim_builder=optim_builder,
                         checkpoint_freq=checkpoint_freq,
                         trainer_cls=trainer_cls,
                         **kwargs)
```

7.5 Testing it

Now that we finished the implementation of the backend (which is the outermost wrapper; Congratulations!), we can just test it. We'll use a very simple network and test it with dummy data. We also only test the `run` and `test` functionality of our experiment, since everything else is just used for setting up the internal state or a composition of these two methods and already tested: Now, let's just define our dataset, instantiate it three times (for training, validation and testing) and wrap each of them into a `BaseDataManager`:

```
from delira.data_loading import AbstractDataset
from delira.data_loading import BaseDataManager


class DummyDataset(AbstractDataset):
    def __init__(self, length):
        super().__init__(None, None)
        self.length = length

    def __getitem__(self, index):
        return {"data": np.random.rand(32),
                "label": np.random.randint(0, 1, 1)}

    def __len__(self):
        return self.length

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

dset_train = DummyDataset(500)
dset_val = DummyDataset(50)
dset_test = DummyDataset(10)

# training, validation and testing with
#a batchsize of 16, 1 loading thread and no transformations.
dmgr_train = BaseDataManager(dset_train, 16, 1, None)
dmgr_val = BaseDataManager(dset_val, 16, 1, None)
```

(continues on next page)

(continued from previous page)

```
dmgr_test = BaseDataManager(dset_test, 16, 1, None)
```

Now, that we have created three datasets, we need to define our small dummy network. We do this by subclassing `delira.models.AbstractTorchScriptNetwork` (which is the exactly implementation given above, be we need to use the internal one, because there are some typechecks against this one).

```
from delira.models import AbstractTorchScriptNetwork
import torch


class DummyNetworkTorchScript(AbstractTorchScriptNetwork):
    __constants__ = ["module"]

    def __init__(self):
        super().__init__()
        self.module = self._build_model(32, 1)

    @torch.jit.script_method
    def forward(self, x):
        return {"pred": self.module(x)}

    @staticmethod
    def prepare_batch(batch_dict, input_device, output_device):
        return {"data": torch.from_numpy(batch_dict["data"]
                                         ).to(input_device,
                                              torch.float),
                "label": torch.from_numpy(batch_dict["label"]
                                          ).to(output_device,
                                               torch.float)}

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


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

(continues on next page)

(continued from previous page)

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

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

        """

        assert (optimizers and losses) 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 losses.items():
                    _loss_val = crit_fn(preds["pred"], *data_dict.values())
                    loss_vals[key] = _loss_val.item()
                    total_loss += _loss_val

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

        if optimizers:
            optimizers['default'].zero_grad()
            # perform loss scaling via apex if half precision is enabled
            with optimizers["default"].scale_loss(total_loss) as scaled_loss:
                scaled_loss.backward()
            optimizers['default'].step()

        else:
```

(continues on next page)

(continued from previous page)

```
            # 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

        return metric_vals, loss_vals, {k: v.detach()
                                        for k, v in preds.items()}

    @staticmethod
    def _build_model(in_channels, n_outputs):
        return torch.nn.Sequential(
            torch.nn.Linear(in_channels, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, n_outputs)
        )
```

Now, that we defined our model, let's just test, if we really can forward some tensors through it. We will just use some random `torch.Tensors` (created by `torch.rand`). Since our model accepts 1d inputs of length 32, we need to pass 2d tensors to it (the additional dimension is the batch-dimension).

```
input_tensor_single = torch.rand(1, 32) # use a single-sample batch (batchsize=1) here
input_tensor_batched = torch.rand(4, 32) # use a batch with batchsize 4 here

# create model instance
model = DummyNetworkTorchScript()

outputs = {"single": model(input_tensor_single)["pred"], "batched": model(input_
↪tensor_batched)["pred"]}
outputs
```

```
{'single': tensor([[-0.1934]], grad_fn=<DifferentiableGraphBackward>),
 'batched': tensor([[-0.0525],
         [-0.0884],
         [-0.1492],
         [-0.0431]], grad_fn=<DifferentiableGraphBackward>)}
```

```
from sklearn.metrics import mean_absolute_error
from delira.training.callbacks import ReduceLROnPlateauCallbackPyTorch
from delira.training import Parameters
params = Parameters(fixed_params={
                    "model": {},
                    "training": {
                        "losses": {"CE": torch.nn.BCEWithLogitsLoss()},
                        "optimizer_cls": torch.optim.Adam,
                        "optimizer_params": {"lr": 1e-3},
                        "num_epochs": 2,
                        "val_metrics": {"mae": mean_absolute_error},
                        "lr_sched_cls": ReduceLROnPlateauCallbackPyTorch,
                        "lr_sched_params": {"mode": "min"}
                    }
```

(continues on next page)

(continued from previous page)

```
                }
            )

from delira.training import TorchScriptExperiment

exp = TorchScriptExperiment(params, DummyNetworkTorchScript,
                            key_mapping={"x": "data"},
                            val_score_key="mae",
                            val_score_mode="min")

trained_model = exp.run(dmgr_train, dmgr_val)
exp.test(trained_model, dmgr_test, params.nested_get("val_metrics"))
```

Congratulations. You have implemented your first fully-workable `delira`-Backend. Wasn't that hard, was it?

Before you start implementing backends for all the other frameworks out there, let me just give you some advices:

- You should test everything you implement or extend
- Make sure, to keep your backend-specification in mind
- Always follow the API of already existing backends. If this is not possible: test this extensively
- If you extend another backend (like we did here; we extended the `PyTorch`-backend for `TorchScript`), make sure, that the "base-backend" is always installed (best if they can only be installed together)
- If you have questions regarding the implementation, don't hestiate to contact us.

CHAPTER

EIGHT

API DOCUMENTATION

8.1 Delira

8.1.1 Data Loading

This module provides Utilities to load the Data

Arbitrary Data

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

Datasets

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

Datasets can be indexed by integers and return single samples.

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

AbstractDataset

class AbstractDataset(*data_path: str*, *load_fn: Callable*)

Bases: `object`

Base Class for Dataset

abstract _make_dataset(*path: str*)

Create dataset

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

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

Return type list

get_sample_from_index(*index*)

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

See also:

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

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

Returns sample corresponding to given index

Return type Any

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

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

Returns the subset

Return type BlankDataset

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

Parameters

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

Returns

- BlankDataset – train dataset
- BlankDataset – test dataset

See also:

sklearn.model_selection.train_test_split

BaseLazyDataset

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

Dataset to load data in a lazy way

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

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

Returns list of sample paths

Return type list

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

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

See also:

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

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

Returns sample corresponding to given index

Return type Any

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

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

Returns the subset

Return type BlankDataset

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

Parameters

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

Returns

- BlankDataset – train dataset
- BlankDataset – test dataset

See also:

sklearn.model_selection.train_test_split

BaseCacheDataset

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

Dataset to preload and cache data

Notes

data needs to fit completely into RAM!

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

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

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

Return type list

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

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

See also:

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

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

Returns sample corresponding to given index

Return type Any

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

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

Returns the subset

Return type BlankDataset

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

Parameters

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

Returns

- BlankDataset – train dataset
- BlankDataset – test dataset

See also:

sklearn.model_selection.train_test_split

BaseExtendCacheDataset

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

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

Notes

data needs to fit completely into RAM!

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

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

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

Return type list

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

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

See also:

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

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

Returns sample corresponding to given index

Return type Any

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

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

Returns the subset

Return type BlankDataset

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

Parameters

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

Returns

- BlankDataset – train dataset
- BlankDataset – test dataset

See also:

sklearn.model_selection.train_test_split

ConcatDataset

class ConcatDataset(**datasets*)
Bases: delira.data_loading.dataset.AbstractDataset

abstract _make_dataset(*path: str*)
Create dataset

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

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

Return type list

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

See also:

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

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

Returns sample corresponding to given index

Return type Any

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

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

Returns the subset

Return type BlankDataset

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

Parameters

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

Returns

- BlankDataset – train dataset
- BlankDataset – test dataset

See also:

sklearn.model_selection.train_test_split

BlankDataset

Nii3DLazyDataset

Nii3DCacheDataset

TorchvisionClassificationDataset:

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

Wrapper for torchvision classification datasets to provide consistent API

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

Parameters

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

Returns actual Dataset

Return type torchvision.Dataset

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

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

See also:

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

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

Returns sample corresponding to given index

Return type Any

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

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

Returns the subset

Return type BlankDataset

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

Parameters

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

Returns

- BlankDataset – train dataset
- BlankDataset – test dataset

See also:

sklearn.model_selection.train_test_split

Dataloader

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

ToDo: add flow chart diagramm

BaseDataLoader

class BaseDataLoader(*dataset: delira.data_loading.dataset.AbstractDataset*, *sampler_queues: list*, *batch_size=1*, *num_batches=None*, *seed=1*)
Bases: batchgenerators.dataloading.data_loader.SlimDataLoaderBase

Class to create a data batch out of data samples

_get_sample(*index*)
Helper functions which returns an element of the dataset

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

Returns Returned Data

Return type dict

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

Returns data and labels

Return type dict

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

Datamanager

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

BaseDataManager

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

Class to Handle Data Creates Dataset , Dataloader and BatchGenerator

property batch_size
Property to access the batchsize

Returns the batchsize

Return type int

property data_loader_cls
Property to access the current data loader class

Returns Subclass of SlimDataLoaderBase

Return type type

property dataset
Property to access the current dataset

Returns the current dataset

Return type AbstractDataset

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

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

Returns Batchgenerator

Return type *Augmenter*

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

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

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

Returns manager containing the subset

Return type *BaseDataManager*

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

Returns Number of Batches

Return type int

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

property n_process_augmentation
Property to access the number of augmentation processes

Returns number of augmentation processes

Return type int

property n_samples
Number of Samples

Returns Number of Samples

Return type int

property sampler
Property to access the current sampler

Returns the current sampler

Return type AbstractSampler

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

Parameters

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

property transforms
Property to access the current data transforms

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

Return type None, AbstractTransform

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

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

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

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

Raises **KeyError** – Invalid keys are specified

Augmenter

class Augmenter(*data_loader: delira.data_loading.data_loader.BaseDataLoader*, *transforms*, *n_process_augmentation*, *sampler*, *sampler_queues: list*, *num_cached_per_queue=2*, *seeds=None*, ***kwargs*)

Bases: object

Class wrapping MultiThreadedAugmentor and SingleThreadedAugmenter to provide a uniform API and to disable multiprocessing/multithreading inside the dataloading pipeline

static _Augmenter__identity_fn(**args*, ***kwargs*)

Helper function accepting arbitrary args and kwargs and returning without doing anything

Parameters

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

_finish()

Property to provide uniform API of _finish

Returns either the augmenter's _finish method (if available) or __identity_fn (if not available)

Return type Callable

_fn_checker(*function_name*)

Checks if the internal augmenter has a given attribute and returns it. Otherwise it returns __identity_fn

Parameters **function_name** (*str*) – the function name to check for

Returns either the function corresponding to the given function name or __identity_fn

Return type Callable

_next_queue()

property _start

Property to provide uniform API of _start

Returns either the augmenter's _start method (if available) or __identity_fn (if not available)

Return type Callable

next()

Function to sample and load

Returns the next batch

Return type dict

property num_batches
Property returning the number of batches

Returns number of batches

Return type int

property num_processes
Property returning the number of processes to use for loading and augmentation

Returns number of processes to use for loading and augmentation

Return type int

restart()
Property to provide uniform API of `restart`

Returns either the augmenter's `restart` method (if available) or `__identity_fn` (if not available)

Return type Callable

Utils

norm_range

norm_range(*mode*)
Closure function for range normalization

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

Returns normalization function

Return type callable

norm_zero_mean_unit_std

norm_zero_mean_unit_std(*data*)
Return normalized data with mean 0, standard deviation 1

Parameters **data** (*np.nadarray*) –

Returns normalized data

Return type np.ndarray

is_valid_image_file

is_valid_image_file(*fname*, *img_extensions*, *gt_extensions*)
Helper Function to check wheter file is image file and has at least one label file

Parameters

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

- **img_extensions** (*list*) – list of valid image file extensions
- **gt_extensions** (*list*) – list of valid gt file extensions

Returns is valid data sample

Return type bool

default_load_fn_2d

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

Parameters

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

Returns

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

LoadSample

class LoadSample(*sample_ext: dict*, *sample_fn: collections.abc.Callable*, *dtype=None*, *normalize=()*, *norm_fn=<function norm_range.<locals>.norm_fn>*, ***kwargs*)
Bases: object

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

Nii-Data

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

load_nii

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

Returns numpy array containing the loaded data

Return type np.ndarray

BaseLabelGenerator

class BaseLabelGenerator(*fpath*)
Bases: object

Base Class to load labels from json files

_load()
Private Helper function to load the file

Returns loaded values from file

Return type Any

abstract get_labels()
Abstractmethod to get labels from class

Raises **NotImplementedError** – if not overwritten in subclass

load_sample_nii

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

Parameters

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

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

Return type dict

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

Sampler

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

AbstractSampler

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

Class to define an abstract Sampling API

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

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

Returns number of indices to sample (truncated if necessary)

Return type int

Raises **StopIteration** – if enough batches sampled

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

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

Returns List with sampled indices

Return type list

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

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

Returns The initialzed sampler

Return type *AbstractSampler*

LambdaSampler

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

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

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

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

Returns number of indices to sample (truncated if necessary)

Return type int

Raises **StopIteration** – if enough batches sampled

_get_indices(*n_indices*)
Actual Sampling

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

Returns list of sampled indices

Return type list

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

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

Returns The initialzed sampler

Return type *AbstractSampler*

RandomSampler

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

Implements Random Sampling With Replacement from whole Dataset

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

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

Returns number of indices to sample (truncated if necessary)

Return type int

Raises **StopIteration** – if enough batches sampled

_get_indices(*n_indices*)
Actual Sampling

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

Returns list of sampled indices

Return type list

Raises **StopIteration** – If maximal number of samples is reached

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

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

Returns The initialzed sampler

Return type *AbstractSampler*

PrevalenceRandomSampler

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

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

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

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

Returns number of indices to sample (truncated if necessary)

Return type int

Raises **StopIteration** – if enough batches sampled

_get_indices(*n_indices*)
Actual Sampling

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

Returns list of sampled indices

Return type list

Raises **StopIteration** – If maximal number of samples is reached

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

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

Returns The initialized sampler

Return type *AbstractSampler*

StoppingPrevalenceRandomSampler

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

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

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

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

Returns number of samples per class to return

Return type dict

_get_indices(*n_indices*)
Actual Sampling

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

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

Returns **list**

Return type list of sampled indices

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

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

Returns The initialized sampler

Return type *AbstractSampler*

SequentialSampler

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

Implements Sequential Sampling from whole Dataset

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

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

Returns number of indices to sample (truncated if necessary)

Return type int

Raises **StopIteration** – if enough batches sampled

_get_indices(*n_indices*)
Actual Sampling

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

:raises StopIteration : If end of dataset reached:

Returns list of sampled indices

Return type list

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

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

Returns The initialzed sampler

Return type *AbstractSampler*

PrevalenceSequentialSampler

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

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

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

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

Returns number of indices to sample (truncated if necessary)

Return type int

Raises **StopIteration** – if enough batches sampled

_get_indices(*n_indices*)
Actual Sampling

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

:raises StopIteration : If end of class indices is reached:

Returns list of sampled indices

Return type list

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

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

Returns The initialzed sampler

Return type *AbstractSampler*

StoppingPrevalenceSequentialSampler

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

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

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

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

Returns number of samples per class to return

Return type dict

_get_indices(*n_indices*)
Actual Sampling

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

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

Returns list of sampled indices

Return type list

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

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

Returns The initialzed sampler

Return type *AbstractSampler*

WeightedRandomSampler

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

Implements Weighted Random Sampling

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

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

Returns number of indices to sample (truncated if necessary)

Return type int

Raises **StopIteration** – if enough batches sampled

_get_indices(*n_indices*)
Actual Sampling

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

Returns list of sampled indices

Return type list

Raises

- **StopIteration** – If maximal number of samples is reached
- **ValueError** – if weights or cum_weights don't match the population

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

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

Returns The initialzed sampler

Return type *AbstractSampler*

8.1.2 IO

if "CHAINER" in get_backends(): from delira.io.chainer import save_checkpoint as chainer_save_checkpoint from delira.io.chainer import load_checkpoint as chainer_load_checkpoint

if "SKLEARN" in get_backends(): from delira.io.sklearn import load_checkpoint as sklearn_load_checkpoint from delira.io.sklearn import save_checkpoint as sklearn_save_checkpoint

torch_load_checkpoint

torch_load_checkpoint(*file*, ***kwargs*)
Loads a saved model

Parameters

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

Returns checkpoint state_dict

Return type OrderedDict

torch_save_checkpoint

torch_save_checkpoint(*file: str*, *model=None*, *optimizers=None*, *epoch=None*, ***kwargs*)
Save checkpoint

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)

torchscript_load_checkpoint

torchscript_load_checkpoint(*file: str*, ***kwargs*)
Loads a saved checkpoint consisting of 2 files (see `save_checkpoint_jit()` for details)

Parameters

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

Returns checkpoint state_dict

Return type OrderedDict

torchscript_save_checkpoint

torchscript_save_checkpoint(*file: str*, *model=None*, *optimizers=None*, *epoch=None*, ***kwargs*)

Save current checkpoint to two different files:

1.) **`file + "_model.ptj"`: Will include the state of the model** (including the graph; this is the opposite to `save_checkpoint()`)

2.) **`file + "_trainer_state.pt"`: Will include the states of all** optimizers and the current epoch (if given)

Parameters

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

tf_load_checkpoint

tf_load_checkpoint(*file: str*, *model=None*)

Loads a saved model

Parameters

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

tf_save_checkpoint

tf_save_checkpoint(*file: str*, *model=None*)

Save model's parameters contained in it's graph

Parameters

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

tf_eager_load_checkpoint

tf_eager_load_checkpoint(*file*, *model: delira.models.backends.tf_eager.abstract_network.AbstractTfEagerNetwork = None*, *optimizer: Dict[str, tensorflow.train.Optimizer] = None*)

tf_eager_save_checkpoint

tf_eager_save_checkpoint(*file*, *model: delira.models.backends.tf_eager.abstract_network.AbstractTfEagerNetwork = None*, *optimizer: Dict[str, tensorflow.train.Optimizer] = None*, *epoch=None*)

chainer_load_checkpoint

chainer_load_checkpoint(*file*, *old_state: dict = None*, *model: chainer.link.Link = None*, *optimizers: dict = None*)

Loads a state from a given file

Parameters

- **file** (*str*) – string containing the path to the file containing the saved state
- **old_state** (*dict*) – dictionary containing the modules to load the states to
- **model** (`chainer.link.Link`) – the model the state should be loaded to; overwrites the `model` key in `old_state` if not None
- **optimizers** (*dict*) – dictionary containing all optimizers. overwrites the `optimizers` key in `old_state` if not None

Returns the loaded state

Return type dict

chainer_save_checkpoint

chainer_save_checkpoint(*file*, *model=None*, *optimizers=None*, *epoch=None*)

Saves the given checkpoint

Parameters

- **file** (*str*) – string containing the path, the state should be saved to
- **model** (`AbstractChainerNetwork`) –
- **optimizers** (*dict*) – dictionary containing all optimizers
- **epoch** (*int*) – the current epoch

sklearn_load_checkpoint

sklearn_load_checkpoint(*file*, ***kwargs*)

Loads a saved model

Parameters

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

Returns checkpoint state_dict

Return type OrderedDict

sklearn_save_checkpoint

sklearn_save_checkpoint(*file: str*, *model=None*, *epoch=None*, ***kwargs*)

Save model's parameters

Parameters

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

- **model** (*AbstractNetwork or None*) – the model which should be saved if None: empty dict will be saved as state dict
- **epoch** (*int*) – current epoch (will also be pickled)

8.1.3 Logging

The logging module provides the utilities for logging arbitrary values to different backends and a logger registry.

Logger <base_logger> Logging Backends <backends> Logging Context <logging_context> Registry <registry>

8.1.4 Models

`delira` comes with it's own model-structure tree - with `AbstractNetwork` at it's root - and integrates several backends deeply into it's structure.

AbstractNetwork

class AbstractNetwork(*type*)

Bases: `object`

Abstract class all networks should be derived from

_init_kwargs = {}

abstract static closure(*model*, *data_dict: dict*, *optimizers: dict*, *losses=None*, *metrics=None*, *fold=0*, ***kwargs*)

Function which handles prediction from batch, logging, loss calculation and optimizer step

Parameters

- **model** (*AbstractNetwork*) – model to forward data through
- **data_dict** (*dict*) – dictionary containing the data
- **optimizers** (*dict*) – dictionary containing all optimizers to perform parameter update
- **losses** (*dict*) – Functions or classes to calculate losses
- **metrics** (*dict*) – Functions or classes to calculate other metrics
- **fold** (*int*) – Current Fold in Crossvalidation (default: 0)
- **kwargs** (*dict*) – additional keyword arguments

Returns

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

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

property init_kwargs

Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

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

Parameters

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

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

Return type dict

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

Backends

Chainer

AbstractChainerNetwork

class AbstractChainerNetwork(***kwargs*)
Bases: chainer.Chain, delira.models.backends.chainer.abstract_network.ChainerMixin

Abstract Class for Chainer Networks

_init_kwargs = {}

static closure(*model*, *data_dict: dict*, *optimizers: dict*, *losses={}*, *metrics={}*, *fold=0*, ***kwargs*)
default closure method to do a single training step; Could be overwritten for more advanced models

Parameters

- **model** (*AbstractChainerNetwork*) – trainable model
- **data_dict** (*dict*) – dictionary containing the data
- **optimizers** (*dict*) – dictionary of optimizers to optimize model's parameters; ignored here, just passed for compatibility reasons
- **losses** (*dict*) – dict holding the losses to calculate errors; ignored here, just passed for compatibility reasons
- **metrics** (*dict*) – dict holding the metrics to calculate
- **fold** (*int*) – Current Fold in Crossvalidation (default: 0)
- ****kwargs** – additional keyword arguments

Returns

- *dict* – Metric values (with same keys as input dict metrics)
- *dict* – Loss values (with same keys as input dict losses; will always be empty here)
- *dict* – dictionary containing all predictions

abstract forward(**args*, ***kwargs*) → dict
Feeds Arguments through the network

Parameters

- ***args** – positional arguments of arbitrary number and type
- ****kwargs** – keyword arguments of arbitrary number and type

Returns dictionary containing all computation results

Return type dict

property init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

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

Parameters

- **batch** (*dict*) – dictionary containing all the data
- **input_device** (*chainer.backend.Device or string*) – 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

DataParallelChainerNetwork

class DataParallelChainerNetwork(*module: delira.models.backends.chainer.abstract_network.AbstractChainerNetwork*, *devices: list*, *output_device=None*, *batch_dim=0*)
Bases: delira.models.backends.chainer.abstract_network.AbstractChainerNetwork

A Wrapper around a *AbstractChainerNetwork* instance to implement parallel training by splitting the batches

static _gather(*predictions*, *dim*, *target_device*)
Re-Builds batches on the target device

Parameters

- **predictions** (*list*) – list containing the predictions from all replicated models
- **dim** (*int*) – dimension to use for concatenating single predictions
- **target_device** (*str or chainer.backend.Device*) – the device, the re-built batch should lie on

Returns the rebuild batch (lying on target_device)

Return type Any

_init_kwargs = {}

static _scatter(*inputs*, *kwargs*, *target_devices: list*, *dim=0*)
Scatters all inputs (args and kwargs) to target devices and splits along given dimension

Parameters

- **inputs** (*list or tuple*) – positional arguments
- **kwargs** (*dict*) – keyword arguments
- **target_devices** (*list*) – list of target device (either string or chainer.backend.Device)
- **dim** (*int*) – the dimension, which should be used for splitting the batch

Returns

- *tuple* – scattered positional arguments
- *tuple* – scattered keyword arguments

cleargrads()

property closure

default closure method to do a single training step; Could be overwritten for more advanced models

Parameters

- **model** (*AbstractChainerNetwork*) – trainable model
- **data_dict** (*dict*) – dictionary containing the data
- **optimizers** (*dict*) – dictionary of optimizers to optimize model's parameters; ignored here, just passed for compatibility reasons
- **losses** (*dict*) – dict holding the losses to calculate errors; ignored here, just passed for compatibility reasons
- **metrics** (*dict*) – dict holding the metrics to calculate
- **fold** (*int*) – Current Fold in Crossvalidation (default: 0)
- ****kwargs** – additional keyword arguments

Returns

- *dict* – Metric values (with same keys as input dict metrics)
- *dict* – Loss values (with same keys as input dict losses; will always be empty here)
- *dict* – dictionary containing all predictions

forward(**args*, ***kwargs*)

Scatters the inputs (both positional and keyword arguments) across all devices, feeds them through model replicas and re-builds batches on output device

Parameters

- ***args** – positional arguments of arbitrary number and type
- ****kwargs** – keyword arguments of arbitrary number and type

Returns combined output from all scattered models

Return type Any

property init_kwargs

Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

params(*include_uninit=True*)
Only the parameters of the module on the first device will actually be updated, all the other parameters will be replicated by the optimizer after an update

Parameters **include_uninit** (*bool*) –

Returns

Return type a generator holding the root-modules parameters

property prepare_batch
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** (*chainer.backend.Device or string*) – 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

zerograds()

DataParallelChainerOptimizer

class DataParallelChainerOptimizer(*optimizer*)
Bases: chainer.Optimizer

An Optimizer-Wrapper to enable DataParallel. Basically this forwards all functions to the interal optimizer, but registers the additional hooks needed for DataParallel (namely `ParallelOptimizerUpdateModelParameters` as a post-update hook and *ParallelOptimizerCumulateGradientsHook* as a pre-update hook)

property _loss_scale

property _loss_scale_max

property _loss_scaling_is_dynamic

property _pre_update_hooks

property add_hook

property call_hooks

property check_nan_in_grads

property epoch

classmethod from_optimizer_class(*optim_cls*, **args*, ***kwargs*)

Parameters

- **optim_cls** (subclass of chainer.Optimizer) – the optimizer to use internally
- ***args** – arbitrary positional arguments (will be used for initialization of internally used optimizer)

- ****kwargs** – arbitrary keyword arguments (will be used for initialization of internally used optimizer)

property is_safe_to_update

property loss_scaling

property new_epoch

property remove_hook

property serialize

property set_loss_scale

setup (*link*)

Calls the setup method of the internal optimizer and registers the necessary grads for data-parallel behavior

Parameters **link** (DataParallel) – the target, whoose parameters should be updated

property target

property update

property update_loss_scale

property use_auto_new_epoch

ParallelOptimizerUpdateModelParameters

ParallelOptimizerCumulateGradientsHook

class ParallelOptimizerCumulateGradientsHook

Bases: object

A hook which sums up all replication's gradients in a DataParallel-Scenario

call_for_each_param = False

name = 'DataParallelCumulateGradients'

timing = 'pre'

SciKit-Learn

SklearnEstimator

class SklearnEstimator (*module: sklearn.base.BaseEstimator*)

Bases: delira.models.abstract_network.AbstractNetwork

Wrapper Class to wrap all sklearn estimators and provide delira compatibility

_init_kwargs = {}

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

default closure method to do a single training step; Could be overwritten for more advanced models

Parameters

- **model** (SkLearnEstimator) – trainable model
- **data_dict** (*dict*) – dictionary containing the data

- **optimizers** (*dict*) – dictionary of optimizers to optimize model's parameters; ignored here, just passed for compatibility reasons
- **losses** (*dict*) – dict holding the losses to calculate errors; ignored here, just passed for compatibility reasons
- **metrics** (*dict*) – dict holding the metrics to calculate
- **fold** (*int*) – Current Fold in Crossvalidation (default: 0)
- ****kwargs** – additional keyword arguments

Returns

- *dict* – Metric values (with same keys as input dict metrics)
- *dict* – Loss values (with same keys as input dict losses; will always be empty here)
- *dict* – dictionary containing all predictions

property init_kwargs

Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

property iterative_training

Property indicating, whether a the current module can be trained iteratively (batchwise)

Returns True: if current module can be trained iteratively False: else

Return type bool

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** (*Any*) – device for module inputs (will be ignored here; just given for compatibility)
- **output_device** (*Any*) – device for module outputs (will be ignored here; just given for compatibility)

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

Return type dict

TensorFlow Eager Execution

AbstractTfEagerNetwork

class AbstractTfEagerNetwork(*data_format='channels_first'*, *trainable=True*, *name=None*, *dtype=None*, ***kwargs*)

Bases: `delira.models.abstract_network.AbstractNetwork`, `tensorflow.keras.layers.Layer`

Abstract Network for TF eager execution backend. All models to use with this backend should be derived from this class

_init_kwargs = {}

abstract call(**args*, ***kwargs*)
Defines the model's forward pass

Parameters

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

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

static closure(*model, data_dict: dict, optimizers: Dict[str, tensorflow.train.Optimizer], losses={}, metrics={}, fold=0, **kwargs*)
Function which handles prediction from batch, logging, loss calculation and optimizer step

Parameters

- **model** (AbstractNetwork) – model to forward data through
- **data_dict** (*dict*) – dictionary containing the data
- **optimizers** (*dict*) – dictionary containing all optimizers to perform parameter update
- **losses** (*dict*) – Functions or classes to calculate losses
- **metrics** (*dict*) – Functions or classes to calculate other metrics
- **fold** (*int*) – Current Fold in Crossvalidation (default: 0)
- **kwargs** (*dict*) – additional keyword arguments

Returns

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

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

property init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

static prepare_batch(*batch: dict*, *input_device*, *output_device*)
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** (*str*) – device for module inputs
- **output_device** (*str*) – device for module outputs

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

Return type dict

DataParallelTfEagerNetwork

class DataParallelTfEagerNetwork(*module*, *devices*)

Bases: delira.models.backends.tf_eager.abstract_network.AbstractTfEagerNetwork

DataParallel Module for the TF eager execution backend

> **Warning:** This Module is highly experimental and not guaranteed to work properly!

_init_kwargs = {}

call(**args*, ***kwargs*)

Defines the forward pass of the module

Parameters

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

property closure

Function which handles prediction from batch, logging, loss calculation and optimizer step

Parameters

- **model** (AbstractNetwork) – model to forward data through
- **data_dict** (*dict*) – dictionary containing the data
- **optimizers** (*dict*) – dictionary containing all optimizers to perform parameter update
- **losses** (*dict*) – Functions or classes to calculate losses
- **metrics** (*dict*) – Functions or classes to calculate other metrics
- **fold** (*int*) – Current Fold in Crossvalidation (default: 0)
- **kwargs** (*dict*) – additional keyword arguments

Returns

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

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

property init_kwargs

Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

property prepare_batch

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** (*str*) – device for module inputs
- **output_device** (*str*) – device for module outputs

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

Return type dict

TensorFlow Graph Execution

AbstractTfGraphNetwork

class AbstractTfGraphNetwork(*sess=tensorflow.Session*, ***kwargs*)

Bases: delira.models.abstract_network.AbstractNetwork

Abstract Class for Tf Networks

See also:

AbstractNetwork

_abc_impl = <_abc_data object>

_add_losses(*losses: dict*)

Adds losses to model that are to be used by optimizers or during evaluation. Can be overwritten for more advanced loss behavior

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

_add_optims(*optims: dict*)

Adds optims to model that are to be used by optimizers or during training. Can be overwritten for more advanced optimizers

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

_init_kwargs = {}

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

default closure method to do a single training step; Could be overwritten for more advanced models

Parameters

- **model** (SkLearnEstimator) – trainable model
- **data_dict** (*dict*) – dictionary containing the data
- **optimizers** (*dict*) – dictionary of optimizers to optimize model's parameters; ignored here, just passed for compatibility reasons
- **losses** (*dict*) – dict holding the losses to calculate errors; ignored here, just passed for compatibility reasons
- **metrics** (*dict*) – dict holding the metrics to calculate
- **fold** (*int*) – Current Fold in Crossvalidation (default: 0)
- ****kwargs** – additional keyword arguments

Returns

- *dict* – Metric values (with same keys as input dict metrics)
- *dict* – Loss values (with same keys as input dict losses; will always be empty here)

- *dict* – dictionary containing all predictions

property init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

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

Parameters

- **batch** (*dict*) – dictionary containing all the data
- **input_device** (*Any*) – device for module inputs (will be ignored here; just given for compatibility)
- **output_device** (*Any*) – device for module outputs (will be ignored here; just given for compatibility)

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

Return type dict

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

Parameters

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

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

Return type dict

PyTorch

AbstractPyTorchNetwork

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

Abstract Class for PyTorch Networks

See also:

`None`, `AbstractNetwork`

_init_kwargs = {}

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

Parameters

- **model** (*AbstractPyTorchNetwork*) – trainable model
- **data_dict** (*dict*) – dictionary containing the data

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

Returns

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

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

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

Returns result: module results of arbitrary type and number

Return type Any

property init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

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

Parameters

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

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

Return type dict

DataParallelPyTorchNetwork

class DataParallelPyTorchNetwork(*module: delira.models.backends.torch.abstract_network.AbstractPyTorchNetwork*, *device_ids=None*, *output_device=None*, *dim=0*)
Bases: delira.models.backends.torch.abstract_network.AbstractPyTorchNetwork, torch.nn.DataParallel

A Wrapper around a *AbstractPyTorchNetwork* instance to implement parallel training by splitting the batches

_init_kwargs = {}

property closure
closure method to do a single backpropagation step

Parameters

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

Returns

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

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

forward(**args*, ***kwargs*)
Scatters the inputs (both positional and keyword arguments) across all devices, feeds them through model replicas and re-builds batches on output device

Parameters

- ***args** – positional arguments of arbitrary number and type
- ****kwargs** – keyword arguments of arbitrary number and type

Returns combined output from all scattered models

Return type Any

property init_kwargs
Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

property prepare_batch
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

scale_loss

scale_loss(*loss*, *optimizers*, *loss_id=0*, *model=None*, *delay_unscale=False*, ***kwargs*)

Context Manager which automatically switches between loss scaling via apex.amp (if apex is available) and no loss scaling

Parameters

- **loss** (`torch.Tensor`) – a pytorch tensor containing the loss value
- **optimizers** (*list*) – a list of `torch.optim.Optimizer` containing all optimizers, which are holding paraneters affected by the backpropagation of the current loss value
- **loss_id** (*int*) – When used in conjunction with the `num_losses` argument to `amp.initialize`, enables Amp to use a different loss scale per loss. `loss_id` must be an integer between 0 and `num_losses` that tells Amp which loss is being used for the current backward pass. If `loss_id` is left unspecified, Amp will use the default global loss scaler for this backward pass.
- **model** (*AbstractPyTorchNetwork* or None) – Currently unused, reserved to enable future optimizations.
- **delay_unscale** (*bool*) – `delay_unscale` is never necessary, and the default value of `False` is strongly recommended. If `True`, Amp will not unscale the gradients or perform model->master gradient copies on context manager exit. `delay_unscale=True` is a minor ninja performance optimization and can result in weird gotchas (especially with multiple models/optimizers/losses), so only use it if you know what you're doing.
- ****kwargs** – additional keyword arguments; currently unused, but provided for the case amp decides to extend the functionality here

Yields `torch.Tensor` – the new loss value (scaled if apex.amp is available and was configured to do so, unscaled in all other cases)

TorchScript

AbstractTorchScriptNetwork

class AbstractTorchScriptNetwork(*optimize=True*, ***kwargs*)

Bases: `delira.models.abstract_network.AbstractNetwork`, `torch.jit.ScriptModule`

Abstract Interface Class for TorchScript Networks. For more information have a look at https://pytorch.org/docs/stable/jit.html#torchscript

> **Warning:** In addition to the here defined API, a forward function must be implemented and decorated with `@torch.jit.script_method`

_init_kwargs = {}

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

closure method to do a single backpropagation step

Parameters

- **model** (*AbstractTorchScriptNetwork*) – trainable model
- **data_dict** (*dict*) – dictionary containing the data

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

Returns

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

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

property init_kwargs

Returns all arguments registered as init kwargs

Returns init kwargs

Return type dict

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

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

Parameters

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

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

Return type dict

8.1.5 Training

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

Parameters

Parameters

NetworkTrainer

The network trainer implements the actual training routine and can be subclassed for special routines. More specific trainers can be found in the backend-specific sections

BaseNetworkTrainer

Predictor

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

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

Predictor

Experiments

Experiments are the outermost class to control your training, it wraps your NetworkTrainer and provides utilities for cross-validation. More Experiments can be found in the sections for the specific backends

BaseExperiment

Backends

The following section contains all backends which are implemented, developed and maintained for usage with `delira`.

A single backend usually contains at least a trainer, an experiment and some models (which are capsuled in the **'models<../../models/models>'_** section.

Chainer

ChainerNetworkTrainer

ChainerExperiment

convert_chainer_to_numpy

create_chainer_optims_default

SciKit-Learn

SklearnEstimatorTrainer

SklearnExperiment

create_sklearn_optims_default

TensorFlow Eager Execution

TfEagerNetworkTrainer

TfEagerExperiment

create_tfeager_optims_default

convert_tfeager_to_numpy

TensorFlow Graph Execution

TfGraphNetworkTrainer

TfGraphExperiment

initialize_uninitialized

PyTorch

PyTorchNetworkTrainer

PyTorchExperiment

create_pytorch_optims_default

convert_torch_to_numpy

TorchScript

TorchScriptNetworkTrainer

TorchScriptExperiment

Callbacks

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

AbstractCallback

class AbstractCallback(**args*, ***kwargs*)

Bases: `object`

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

See also:

`AbstractNetworkTrainer`

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

Function which will be executed at begin of each epoch

Parameters

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

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

Return type dict

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

Parameters

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

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

Return type dict

EarlyStopping

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

Implements Early Stopping as callback

See also:

AbstractCallback

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

Parameters **metric** – current metric value

Returns whether this metric is the new best metric or not

Return type bool

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

Parameters

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

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

Return type dict

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

Parameters

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

Returns trainer with modified attributes

Return type AbstractNetworkTrainer

DefaultPyTorchSchedulerCallback

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

Bases: delira.training.callbacks.abstract_callback.AbstractCallback

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

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

Function which will be executed at begin of each epoch

Parameters

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

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

Return type dict

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

Executes a single scheduling step

Parameters

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

Returns modified trainer

Return type PyTorchNetworkTrainer

CosineAnnealingLRCallback

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

Bases: delira.training.callbacks.pytorch_schedulers.DefaultPyTorchSchedulerCallback

Wraps PyTorch's *CosineAnnealingLR* Scheduler as callback

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

Function which will be executed at begin of each epoch

Parameters

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

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

Return type dict

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

Parameters

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

Returns modified trainer

Return type PyTorchNetworkTrainer

ExponentialLRCallback

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

Wraps PyTorch's *ExponentialLR* Scheduler as callback

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

Parameters

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

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

Return type dict

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

Parameters

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

Returns modified trainer

Return type PyTorchNetworkTrainer

LambdaLRCallback

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

Wraps PyTorch's *LambdaLR* Scheduler as callback

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

Parameters

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

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

Return type dict

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

Parameters

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

Returns modified trainer

Return type PyTorchNetworkTrainer

MultiStepLRCallback

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

Wraps PyTorch's *MultiStepLR* Scheduler as callback

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

Parameters

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

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

Return type dict

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

Parameters

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

Returns modified trainer

Return type PyTorchNetworkTrainer

ReduceLROnPlateauCallback

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

Wraps PyTorch's *ReduceLROnPlateau* Scheduler as Callback

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

Parameters

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

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

Return type dict

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

Parameters

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

Returns modified trainer

Return type PyTorchNetworkTrainer

StepLRCallback

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

Wraps PyTorch's *StepLR* Scheduler as callback

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

Parameters

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

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

Return type dict

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

Parameters

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

Returns modified trainer

Return type PyTorchNetworkTrainer

CosineAnnealingLRCallbackPyTorch

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

ExponentialLRCallbackPyTorch

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

LambdaLRCallbackPyTorch

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

MultiStepLRCallbackPyTorch

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

ReduceLROnPlateauCallbackPyTorch

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

StepLRCallbackPyTorch

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

Custom Loss Functions

BCEFocalLossPyTorch

BCEFocalLossLogitPyTorch

Metrics

SklearnClassificationMetric

SklearnAccuracyScore

SklearnBalancedAccuracyScore

SklearnF1Score

SklearnFBetaScore

SklearnHammingLoss

SklearnJaccardSimilarityScore

SklearnLogLoss

SklearnMatthewsCorrCoeff

SklearnPrecisionScore

SklearnRecallScore

SklearnZeroOneLoss

AurocMetric

def recursively_convert_elements(element, check_type, conversion_fn):

def convert_to_numpy_identity(*args, **kwargs):

Utilities

recursively_convert_elements

recursively_convert_elements(*element*, *check_type*, *conversion_fn*)
Function to recursively convert all elements

Parameters

- **element** (*Any*) – the element to convert
- **check_type** (*Any*) – if `element` is of type `check_type`, the conversion function will be applied to it
- **conversion_fn** (*Any*) – the function to apply to `element` if it is of type `check_type`

Returns the converted element

Return type Any

convert_to_numpy_identity

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

Parameters

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

8.1.6 Utils

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

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

Context Manager to disable the debug mode for the wrapped context

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

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

Context Manager to enable the debug mode for the wrapped context

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

class DebugMode(*mode*)
Bases: `object`

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

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

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

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

add_param_group(*param_group*)

load_state_dict(*state_dict*)

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

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

state_dict()

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

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

zero_grad()

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

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

Returns

Return type Wrapped Function

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

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

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

Returns

Return type Wrapped Function

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

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

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

Returns

Return type Wrapped Function

Raises **Deprecation Warning** –

numpy_array_func(*func*)

torch_module_func(*func*)

torch_tensor_func(*func*)

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

Parameters

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

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

Return type tuple

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

Parameters

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

Returns origin offset

Return type np.ndarray

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

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

Returns slice index

Return type int

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

Parameters

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

Returns Target image with copied metadata

Return type SimpleITK.Image

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

Parameters

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

Returns Blank image with given properties

Return type SimpleITK.Image

sitk_resample_to_image(*image*, *reference_image*, *default_value=0.0*, *interpolator=SimpleITK.sitkLinear*, *transform=None*, *output_pixel_type=None*)
Resamples Image to reference image

Parameters

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

Returns resampled image

Return type SimpleITK.Image

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

Parameters

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

Returns Resampled Image

Return type SimpleITK.Image

sitk_resample_to_spacing(*image*, *new_spacing=(1.0, 1.0, 1.0)*, *interpolator=SimpleITK.sitkLinear*, *default_value=0.0*)
Resamples SITK Image to a given spacing

Parameters

- **image** (*SimpleITK.Image*) – image which should be resampled
- **new_spacing** (*list or np.ndarray or tuple*) – target spacing

- **interpolator** (*Any*) – implements the actual interpolation
- **default_value** (*float*) – default value

Returns resampled Image with target spacing

Return type SimpleITK.Image

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

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

Returns list of strings of all subdirectories

Return type list

now()
Return current time as YYYY-MM-DD_HH-MM-SS

Returns current time

Return type string

class LookupConfig(**args*, ***kwargs*)
Bases: `trixi.util.Config`

Helper class to have nested lookups in all subdicts of Config

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

Parameters

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

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

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

Return type Any

8.1.7 Backend Resolution

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

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

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

get_backends

get_backends()
Return List of currently available backends

Returns list of strings containing the currently installed backends

Return type list

def get_current_debug_mode(): “”” Getter function for the current debug mode Returns ——- bool

current debug mode

“”” return __DEBUG_MODE

def switch_debug_mode(): “”” Alternates the current debug mode “”” set_debug_mode(not get_current_debug_mode())

def set_debug_mode(mode: bool): “”” Sets a new debug mode Parameters ——-- mode : bool

the new debug mode

“”” global __DEBUG_MODE __DEBUG_MODE = mode

8.1.8 Debug Mode

Delira now contains a fully-fledged *Debug* mode, which disables all kinds of multiprocessing.

get_current_debug_mode

get_current_debug_mode()
Getter function for the current debug mode :returns: current debug mode :rtype: bool

switch_debug_mode

switch_debug_mode()
Alternates the current debug mode

set_debug_mode

set_debug_mode(*mode: bool*)
Sets a new debug mode :param mode: the new debug mode :type mode: bool

8.1.9 Class Hierarchy Diagrams

Contents

- Coarse
- Fine

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

P

R

S

T

U

W

Z