

---

# **EDFlow**

**Mimo Tilbich**

**Jul 21, 2020**



# CONTENTS

<b>1</b>	<b>Table of Contents</b>	<b>3</b>
<b>2</b>	<b>Installation</b>	<b>5</b>
<b>3</b>	<b>Getting Started</b>	<b>7</b>
3.1	TensorFlow Eager . . . . .	7
3.2	PyTorch . . . . .	11
<b>4</b>	<b>Documentation</b>	<b>15</b>
4.1	Intro . . . . .	15
4.2	What Happens When I Run EDflow . . . . .	15
4.3	Tutorial . . . . .	16
4.4	Data Sets and Batching . . . . .	21
4.5	Hooks . . . . .	24
4.6	Models . . . . .	24
4.7	Iterators . . . . .	25
4.8	Integrations . . . . .	26
4.9	Contributions . . . . .	27
4.10	FAQ . . . . .	29
4.11	edflow package . . . . .	30
<b>5</b>	<b>Indices and tables</b>	<b>121</b>
	<b>Python Module Index</b>	<b>123</b>
	<b>Index</b>	<b>125</b>



A framework independent engine for training and evaluating in batches.



## TABLE OF CONTENTS

- 1. *Installation*
- 2. *Getting Started*
  - 1. *TensorFlow Eager*
  - 2. *PyTorch*





## INSTALLATION

```
git clone https://github.com/pesser/edflow.git
cd edflow
pip install .
```



## GETTING STARTED

```
cd examples
```

### 3.1 TensorFlow Eager

You provide an implementation of a model and an iterator and use edflow to train and evaluate your model. An example can be found in `template_tfe/edflow.py`:

```
import functools
import tensorflow as tf

tf.enable_eager_execution()
import tensorflow.keras as tfk
import numpy as np
from edflow import TemplateIterator, get_logger

class Model(tfk.Model):
    def __init__(self, config):
        super().__init__()
        self.conv1 = tfk.layers.Conv2D(filters=6, kernel_size=5)
        self.pool = tfk.layers.MaxPool2D(pool_size=2, strides=2)
        self.conv2 = tfk.layers.Conv2D(filters=16, kernel_size=5)
        self.fc1 = tfk.layers.Dense(units=120)
        self.fc2 = tfk.layers.Dense(units=84)
        self.fc3 = tfk.layers.Dense(units=config["n_classes"])

        input_shape = (config["batch_size"], 28, 28, 1)
        self.build(input_shape)

    def call(self, x):
        x = self.pool(tf.nn.relu(self.conv1(x)))
        x = self.pool(tf.nn.relu(self.conv2(x)))
        x = tf.reshape(x, [tf.shape(x)[0], -1])
        x = tf.nn.relu(self.fc1(x))
        x = tf.nn.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class Iterator(TemplateIterator):
    def __init__(self, *args, **kwargs):
```

(continues on next page)

```

super().__init__(*args, **kwargs)
# loss and optimizer
self.criterion = functools.partial(
    tfk.losses.sparse_categorical_crossentropy, from_logits=True
)
self.optimizer = tf.train.MomentumOptimizer(learning_rate=0.001, momentum=0.9)
# to save and restore
self.tfcheckpoint = tf.train.Checkpoint(
    model=self.model, optimizer=self.optimizer
)

def save(self, checkpoint_path):
    self.tfcheckpoint.write(checkpoint_path)

def restore(self, checkpoint_path):
    self.tfcheckpoint.restore(checkpoint_path)

def step_op(self, model, **kwargs):
    # get inputs
    inputs, labels = kwargs["image"], kwargs["class"]

    # compute loss
    with tf.GradientTape() as tape:
        outputs = model(inputs)
        loss = self.criterion(y_true=labels, y_pred=outputs)
        mean_loss = tf.reduce_mean(loss)

    def train_op():
        grads = tape.gradient(mean_loss, model.trainable_variables)
        self.optimizer.apply_gradients(zip(grads, model.trainable_variables))

    def log_op():
        # the default logger understands the keys "images" (written as png
        # in the log directory) and "scalars" (written to stdout and the
        # log file).
        acc = np.mean(np.argmax(outputs, axis=1) == labels)
        min_loss = np.min(loss)
        max_loss = np.max(loss)
        return {
            "images": {"inputs": inputs},
            "scalars": {
                "min_loss": min_loss,
                "max_loss": max_loss,
                "mean_loss": mean_loss,
                "acc": acc,
            },
        }

    def eval_op():
        # values under "labels" are written into a single file,
        # remaining values are written into one file per example.
        # Here, "outputs" would be small enough to write into labels, but
        # for illustration we do not write it as labels.
        return {"outputs": np.array(outputs), "labels": {"loss": np.array(loss)}}

    return {"train_op": train_op, "log_op": log_op, "eval_op": eval_op}

```

Specify your parameters in a yaml config file, e.g. `template_tfe/config.yaml`:

```
dataset: edflow.datasets.fashionmnist.FashionMNIST
model: template_tfe.edflow.Model
iterator: template_tfe.edflow.Iterator
batch_size: 4
num_epochs: 2

n_classes: 10
```

### 3.1.1 Train

To start training, specify configuration files with `-b/--base <config>` command-line option, use the `-t/--train` flag to enable training mode and, optionally, the `-n/--name <name>` option to more easily find your experiments later on:

```
$ edflow -b template_tfe/config.yaml -t -n hello_tfe
[INFO] [train]: Starting Training.
[INFO] [train]: Instantiating dataset.
[INFO] [FashionMNIST]: Using split: train
[INFO] [train]: Number of training samples: 60000
[INFO] [train]: Warm up batches.
[INFO] [train]: Reset batches.
[INFO] [train]: Instantiating model.
[INFO] [train]: Instantiating iterator.
[INFO] [train]: Initializing model.
[INFO] [train]: Starting Training with config:
batch_size: 4
dataset: edflow.datasets.fashionmnist.FashionMNIST
hook_freq: 1
iterator: template_tfe.edflow.Iterator
model: template_tfe.edflow.Model
n_classes: 10
num_epochs: 2
num_steps: 30000

[INFO] [train]: Saved config at logs/2019-08-05T18:55:20_hello_tfe/configs/train_2019-
↪08-05T18:55:26.yaml
[INFO] [train]: Iterating.
[INFO] [LoggingHook]: global_step: 0
[INFO] [LoggingHook]: acc: 0.25
[INFO] [LoggingHook]: max_loss: 2.3287339210510254
[INFO] [LoggingHook]: mean_loss: 2.256807565689087
[INFO] [LoggingHook]: min_loss: 2.2113394737243652
[INFO] [LoggingHook]: project root: logs/2019-08-05T18:55:20_hello_tfe/train
...
```

edflow shows the progress of your training and scalar logging values. The log file, log outputs and checkpoints can be found in the `train` folder of the project root at `logs/2019-08-05T18:55:20_hello_tfe/`. By default, checkpoints are written after each epoch, or when an exception is encountered, including a `KeyboardInterrupt`. The checkpoint frequency can be adjusted with a `ckpt_freq: <frequency>` entry in the config file. All config file entries can also be specified on the command line as, e.g., `--ckpt_freq <frequency>`.

### 3.1.2 Interrupt and Resume

Use CTRL-C to interrupt the training:

```
[INFO] [LambdaCheckpointHook]: Saved model to logs/2019-08-05T18:55:20_hello_tfe/
↳train/checkpoints/model-1207.ckpt
```

To resume training, run

```
edflow -b template_tfe/config.yaml -t -p logs/2019-08-05T18:55:20_hello_tfe/
```

It will load the last checkpoint in the project folder and continue training and logging into the same folder. This lets you easily adjust parameters without having to start training from scratch, e.g.

```
edflow -b template_tfe/config.yaml -t -p logs/2019-08-05T18:55:20_hello_tfe/ --batch_
↳size 32
```

will continue with an increased batch size. Instead of loading the latest checkpoint, you can load a specific checkpoint by adding `-c <path to checkpoint>`:

```
edflow -b template_tfe/config.yaml -t -p logs/2019-08-05T18:55:20_hello_tfe/ -c logs/
↳2019-08-05T18:55:20_hello_tfe/train/checkpoints/model-1207.ckpt
```

### 3.1.3 Evaluate

Evaluation mode will write all outputs of `eval_op` to disk and prepare them for consumption by your evaluation functions. Just remove the training flag `-t`:

```
edflow -b template_tfe/config.yaml -p logs/2019-08-05T18:55:20_hello_tfe/ -c logs/
↳2019-08-05T18:55:20_hello_tfe/train/checkpoints/model-1207.ckpt
```

If `-c` is not specified, it will evaluate the latest checkpoint. The evaluation mode will finish with

```
[INFO] [EvalHook]: All data has been produced. You can now also run all callbacks_
↳using the following command:
edeval -c logs/2019-08-05T18:55:20_hello_tfe/eval/2019-08-05T19:22:23/1207/model_
↳output.csv -cb <name>:<your callback>
```

Your callbacks will get the path to the evaluation folder, the input dataset as seen by your model, an output dataset which contains the corresponding outputs of your model and the config used for evaluation. `template_tfe/edflow.py` contains an example callback computing the average loss and accuracy:

```
def acc_callback(root, data_in, data_out, config):
    from tqdm import trange

    logger = get_logger("acc_callback")
    correct = 0
    seen = 0
    # labels are loaded directly into memory
    loss1 = np.mean(data_out.labels['loss'])
    loss2 = 0.0
    for i in trange(len(data_in)):
        # data_in is the dataset that was used for evaluation
        labels = data_in[i]["class"]
        # data_out contains all the keys that were specified in the eval_op
```

(continues on next page)

(continued from previous page)

```

outputs = data_out[i]["outputs"]
# labels are also available on each example
loss = data_out[i]["labels_"]["loss"]

prediction = np.argmax(outputs, axis=0)
correct += labels == prediction
loss2 += loss
logger.info("Loss1: {}".format(loss1))
logger.info("Loss2: {}".format(loss2 / len(data_in)))
logger.info("Accuracy: {}".format(correct / len(data_in)))

```

which can be executed with:

```

$ edeval -c logs/2019-08-05T18:55:20_hello_tfe/eval/2019-08-05T19:22:23/1207/model_
↪output.csv -cb tfe_cb:template_tfe.edflow.acc_callback
...
INFO:acc_callback:Loss1: 0.4174468219280243
INFO:acc_callback:Loss2: 0.4174468546746697
INFO:acc_callback:Accuracy: 0.8484

```

## 3.2 PyTorch

The same example as implemented by *TensorFlow Eager*, can be found for PyTorch in `template_pytorch/edflow.py` and requires only slightly different syntax:

```

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

import numpy as np
from edflow import TemplateIterator, get_logger

class Model(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 4 * 4, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, config["n_classes"])

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

```

(continues on next page)

(continued from previous page)

```

class Iterator(TemplateIterator):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # loss and optimizer
        self.criterion = nn.CrossEntropyLoss(reduction="none")
        self.optimizer = optim.SGD(self.model.parameters(), lr=0.001, momentum=0.9)

    def save(self, checkpoint_path):
        state = {
            "model": self.model.state_dict(),
            "optimizer": self.optimizer.state_dict(),
        }
        torch.save(state, checkpoint_path)

    def restore(self, checkpoint_path):
        state = torch.load(checkpoint_path)
        self.model.load_state_dict(state["model"])
        self.optimizer.load_state_dict(state["optimizer"])

    def step_op(self, model, **kwargs):
        # get inputs
        inputs, labels = kwargs["image"], kwargs["class"]
        inputs = torch.tensor(inputs)
        inputs = inputs.permute(0, 3, 1, 2)
        labels = torch.tensor(labels, dtype=torch.long)

        # compute loss
        outputs = model(inputs)
        loss = self.criterion(outputs, labels)
        mean_loss = torch.mean(loss)

    def train_op():
        self.optimizer.zero_grad()
        mean_loss.backward()
        self.optimizer.step()

    def log_op():
        acc = np.mean(
            np.argmax(outputs.detach().numpy(), axis=1) == labels.detach().numpy()
        )
        min_loss = np.min(loss.detach().numpy())
        max_loss = np.max(loss.detach().numpy())
        return {
            "images": {
                "inputs": inputs.detach().permute(0, 2, 3, 1).numpy()
            },
            "scalars": {
                "min_loss": min_loss,
                "max_loss": max_loss,
                "mean_loss": mean_loss,
                "acc": acc,
            },
        }

    def eval_op():
        return {
            "outputs": np.array(outputs.detach().numpy()),

```

(continues on next page)



(continued from previous page)

```
        "labels": {"loss": np.array(loss.detach().numpy())},  
    }  
  
    return {"train_op": train_op, "log_op": log_op, "eval_op": eval_op}
```

You can experiment with it in the exact same way as *above*. For example, to *start training* run:

```
edflow -b template_pytorch/config.yaml -t -n hello_pytorch
```

See also *interrupt and resume* and *evaluation*.



## DOCUMENTATION

### 4.1 Intro

**EDFlow** is a training engine that is meant to save you time and code. While taking snapshots of your code **EDFlow** helps you to manage data logging, create batches from data and run evaluation. All in all **EDFlow** runs all the repetitive tasks you usually cannot quite copy-and-paste.

It is relatively easy to translate your current non-EDFlow learning script into an EDFlow compatible one. Although we do not have an auto-translation tool (yet), feel free to take look at our [Tutorial](#). This also serves as a nice practical introduction to **EDFlow**.

Overall **EDFlow** allows you to recycle as much code as possible throughout your projects in the easiest possible way. We hope you enjoy your future with **EDFlow** :\*.

Yours truly, Mimo Tillbich

### 4.2 What Happens When I Run EDflow

#### 4.2.1 config

At the heart of every training or evaluation is the **config** file. It is a `dict` that contains the keywords and values you specify in `train.yaml`. Some keys are mandatory, like: - `dataset` package link to the data set class - `model` package link to the model class - `iterator` package link to the iterator class - `batch_size` how large a batch should be - `num_steps` or `num_epochs` how long should the training be

EDFlow is able to handle multiple config files but typically it is recommended to have a base config file, which is included with the `-b` option and separate training and evaluation configs can be included on top of that, if needed.

- `Test_mode` is set to `true` (e.g. for dropout)

#### 4.2.2 Workflow

When you have successfully built your model your model with:

```
edflow -t your_model/train.yaml
```

This triggers EDFlow's signature workflow:

1. The `ProjectManager` is initialized
  - It creates the folder structure, takes a snapshot of the code and keeps track directory addresses through attributes
  - It is still to decide on the best way to take the snapshot, feel free to participate and [contribute](#)

2. All processes are initialized
  - if `-t` option is given, a training process is started
  - for each `-e` option an evaluation process is called
3. The training process
  - `Logger` is initialized
  - `Dataset` is initialized
  - The batches are built
  - `model` is initialized
    - `#TODO` initialize a dummy if no model is given
  - `Trainer/Iterator` is initialized
  - if `--checkpoint` is given, load checkpoint
  - If `--retrain` is given, reset global step (begin training with pre-trained model)
  - `Iterator.iterate` is called
    - This is the data loop, only argument is the batched data
    - `tqdm tqdm.github.io` is called: for epoch in epochs, for batch in batches
    - initialize fetches
      - \* nested dict
      - \* leaves must be functions i.e. `{global_step:get_global_step() }`
    - feeds are initialized as a copy of batch (this allows to manipulate the feed)
    - all hook s' `before_step(global_step, fetches, feeds, batch)` is called
      - \* hook s can add data, manipulate feeds(i.e. make numpy arrays tf objects), log batch data...
    - `self.run(fetches, feeds)` is called
      - \* every function in fetches is called with feeds as argument
    - `global_step` is incremented
    - all hook s' `after_step(global_step, fetches, feeds, batch)` is called

---

**Note:** here goes a nice gif of edflow in action

---

## 4.3 Tutorial

### 4.3.1 PyTorch

We think that a good way to learn edflow is by example(s). Thus, we translate a simple classification code (the introductory PyTorch [example](#) running on the CIFAR10 dataset) written in PyTorch to the appropriate edflow code. In particular, a detailed step-by-step explanation of the following parts is provided:

- How to set up the (*required*) `dataset` class for edflow

- How to include the classification network (which can then be replaced by any other network in new projects) in the (*required*) `Model` class.
- Setting up an `Iterator` (often called *Trainer*) to execute training via the `step_ops` method.

As a plus, a brief introduction to data logging via pre-build and custom `Hooks` is given.

## The config file

As mentioned before, each `edflow` training is fully set up by its **config** file (e.g. `train.yaml`). This file specifies all (tunable) hyper-parameters and paths to the `Dataset`, `Model` and `Iterator` used in the project.

Here, the `config.yaml` file is rather short:

```
dataset: tutorial_pytorch.edflow.Dataset
model: tutorial_pytorch.edflow.Model
iterator: tutorial_pytorch.edflow.Iterator
batch_size: 4
num_epochs: 2

n_classes: 10
```

Note that the first five keys are **required** by `edflow`. The key `n_classes` is set to illustrate the usage of custom keys (e.g. if training only on a subset of all CIFAR10 classes, ...)

## Setting up the data

### Necessary Imports

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from edflow.data.dataset import DatasetMixin
from edflow.iterators.model_iterator import PyHookedModelIterator
from edflow.hooks.pytorch_hooks import PyCheckpointHook
from edflow.hooks.hook import Hook
from edflow.hooks.checkpoint_hooks.torch_checkpoint_hook import _
↳ RestorePytorchModelHook
from edflow.project_manager import ProjectManager
```

Every `edflow` program requires a dataset class:

```
class Dataset(DatasetMixin):
    """We just initialize the same dataset as in the tutorial and only have to
    implement __len__ and get_example."""

    def __init__(self, config):
        self.train = not config.get("test_mode", False)

        transform = transforms.Compose(
            [
                transforms.ToTensor(),
```

(continues on next page)

(continued from previous page)

```

        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    ]
)
dataset = torchvision.datasets.CIFAR10(
    root="./data", train=self.train, download=True, transform=transform
)
self.dataset = dataset

```

Our dataset is thus conceptually similar to the PyTorch dataset. The `__getitem__()` method required for pytorch datasets is overwritten by `get_example()`. We set an additional `self.train` flag to unify train- and testdata in this class and make switching between them convenient. It is noteworthy that a `Dataloader` is *not* required in `edflow`; dataloading methods are inherited from the base class.

Note that every custom dataset has to implement the methods `__len__()` and `get_example(index)`. Here, `get_example(self, index)` just indexes the `torchvision.dataset` and returns the according numpy arrays (transformed from `torch.tensor`).

```

def __len__(self):
    return len(self.dataset)

def get_example(self, i):
    """edflow assumes a dictionary containing values that can be stacked
    by np.stack(), e.g. numpy arrays or integers."""
    x, y = self.dataset[i]
    return {"x": x.numpy(), "y": y}

```

## Building the model

Having specified a dataset we need to define a model to actually run a training. `edflow` expects a `Model` object which initializes the underlying `nn.Module` model. Here, `Net` is the same model that is used in the official PyTorch tutorial; we just recycle it here.

```

class Model(object):
    def __init__(self, config):
        """For illustration we read `n_classes` from the config."""
        self.net = Net(n_classes=config["n_classes"])

    def __call__(self, x):
        return self.net(torch.tensor(x))

    def parameters(self):
        return self.net.parameters()

```

Nothing unusual here (model definition)...

```

class Net(nn.Module):
    def __init__(self, n_classes):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, n_classes)

```

(continues on next page)

(continued from previous page)

```
def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = x.view(-1, 16 * 5 * 5)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x
```

## How to actually train (Iterator)

Right now we have a rather static model and a dataset but can not do much with it - that's where the `Iterator` comes into play. For PyTorch, this class inherits from `PyHookedModelIterator` as follows:

```
from edflow.iterators.model_iterator import PyHookedModelIterator

class Iterator(PyHookedModelIterator):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.criterion = nn.CrossEntropyLoss()
        self.optimizer = optim.SGD(self.model.parameters(), lr=0.001, momentum=0.9)
```

An `Iterator` can for example hold the optimizers used for training, as well as the loss functions. In our example we use a standard stochastic gradient descent optimizer and cross-entropy loss. Most important, however, is the (*required*) `step_ops()` method: This method provides a pointer towards the function used to do operations on the data, i.e. as returned by the `get_example()` method. In the example at hand this is the `train_op()` method. Note that all ops which should be run as `step_ops()` require the model and the keyword arguments as returned by the `get_example()` method (strictly in this order). We add an if-else statement to directly distinguish between training and testing mode. This is not necessary; we could also define an `Evaluator` (based on `PyHookedModelIterator`) and point to it in a `test.yaml` file.

```
def step_ops(self):
    if self.config.get("test_mode", False):
        return self.test_op
    else:
        return self.train_op

    def train_op(self, model, x, y, **kwargs):
        """All ops to be run as step ops receive model as the first argument
        and keyword arguments as returned by get_example of the dataset."""

        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = x, y
```

Thus, having defined an `Iterator` makes the usual

```
for epoch in epochs:
    for data in dataloader:
        # do something fancy
```

loops obsolete (compare to the 'classic' pytorch example).

The following block contains the full `Iterator`:

```

class Iterator(PyHookedModelIterator):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.criterion = nn.CrossEntropyLoss()
        self.optimizer = optim.SGD(self.model.parameters(), lr=0.001, momentum=0.9)
        self.running_loss = 0.0

        self.restorer = RestorePytorchModelHook(
            checkpoint_path=ProjectManager.checkpoints, model=self.model.net
        )
        if not self.config.get("test_mode", False):
            # we add a hook to write checkpoints of the model each epoch or when
            # training is interrupted by ctrl-c
            self.ckpt_hook = PyCheckpointHook(
                root_path=ProjectManager.checkpoints, model=self.model.net
            ) # PyCheckpointHook expects a torch.nn.Module
            self.hooks.append(self.ckpt_hook)
        else:
            # evaluate accuracy
            self.hooks.append(AccuracyHook(self))

    def initialize(self, checkpoint_path=None):
        # restore model from checkpoint
        if checkpoint_path is not None:
            self.restorer(checkpoint_path)

    def step_ops(self):
        if self.config.get("test_mode", False):
            return self.test_op
        else:
            return self.train_op

    def train_op(self, model, x, y, **kwargs):
        """All ops to be run as step ops receive model as the first argument
        and keyword arguments as returned by get_example of the dataset."""

        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = x, y

        # zero the parameter gradients
        self.optimizer.zero_grad()

        # forward + backward + optimize
        outputs = self.model(inputs)
        loss = self.criterion(outputs, torch.tensor(labels))
        loss.backward()
        self.optimizer.step()

        # print statistics
        self.running_loss += loss.item()
        i = self.get_global_step()
        if i % 200 == 199: # print every 200 mini-batches
            # use the logger instead of print to obtain both console output and
            # logging to the logfile in project directory
            self.logger.info("[%5d] loss: %.3f" % (i + 1, self.running_loss / 200))
            self.running_loss = 0.0

```

(continues on next page)



(continued from previous page)

```
def test_op(self, model, x, y, **kwargs):
    """Here we just run the model and let the hook handle the output."""
    images, labels = x, y
    outputs = self.model(images)
    return outputs, labels
```

To run the code, just enter

```
$ edflow -t tutorial_pytorch/config.yaml
```

into your terminal.

## Hooks

Coming soon. Stay tuned :)

### 4.3.2 Tensorflow

#TODO

## 4.4 Data Sets and Batching

### 4.4.1 Basics

**edflow** is pretty much built around your data. At the core of every training or evaluation is the data, that is utilized. Through **edflow** it is easier than ever to reuse data sets, give them additional features or prepare them for evaluation.

To begin with, you have to inherit from a data set call from `edflow.data.dataset` e.g. `DatasetMixin`. Each class comes with practical features that save code and are (or should) be tested thoroughly.

Every `Dataset` class **must** include the methods `get_example(self, idx)`, where `idx` is an `int`, and `__len__(self)`. `__len__(self)` returns the length of your data set i.e. the number of images. Later on, one epoch is defined as iterating through all indices from 0 to `__len__(self) - 1`.

`get_example(self, index)` gets the current index as argument. Normally, these indices are drawn at random but every index is used once in an epoch, which makes for nice, evenly distributed data. The method must return a dict with strings as keys and the data as element. A nice example would be MNIST. Typically, `get_example` would return a dict like:

```
{label: int, image: np.array}
```

Naturally, you do not have to use these keys and the dict can contain as many keys and data of any type as you want.

### 4.4.2 Batches

If you want to use batches of data you do not have to change anything but the config. Batches are automatically created based on the key `batch_size` which you specify in the config.

A cool feature when working with examples of nested dictionaries is, that they behave the same as their batch versions! I.e. you can access the same keys in the same order in a single example and in a batch of examples and still end up at the value or batch of values you would expect.

```
example = {'a': 1, 'b': {'c': 1}, 'd': [1, 2]}

# after applying our batching algorithm on a list of three of the above examples:
batch_of_3_examples = {'a': [1, 1, 1], 'b': {'c': [1, 1, 1]}, 'd': [[1, 1, 1], [2, 2, ↵
↵2]]}

example['a'] == 1 # True
example['d'][0] == 1 # True

batch_of_3_examples['a'] == [1, 1, 1] # True
batch_of_3_examples['d'][0] == [1, 1, 1] # True
```

This comes in especially handy when you use the utility functions found at `edflow.util` for handling nested structures, as you now can use the same keys anytime:

```
from edflow.util import retrieve

retrieve(example, 'a') == 1 # True
retrieve(example, 'd/0') == 1 # True

retrieve(batch_of_3_examples, 'a') == [1, 1, 1] # True
retrieve(batch_of_3_examples, 'd/0') == [1, 1, 1] # True
```

### 4.4.3 Advanced Data Sets

There is a wealth of Dataset manipulation classes, which almost all manipulate the base dataset by manipulating the indices passed to the dataset.

- SubDataset
- SequenceDataset
- ConcatenatedDataset
- ExampleConcatenatedDataset

More exist, but the above are the ones used most as a recent survey has shown<sup>2</sup>.

<sup>2</sup> Johannes Haux: I use SubDataset, SequenceDataset, ConcatenatedDataset, ExampleConcatenatedDataset. The rest I do not use.

#### 4.4.4 Dataset Workflow

**Warning:** Datasets, which are specified in the edflow config, must accept one positional argument `config`!

A basic workflow with data in **edflow** looks like this:

1. Load the raw data into some `DatasetMixin` derived custom class.
2. Use this dataset in a different class, which accepts a `config`-dictionary, containing all relevant parameters, e.g. for making splits (e.g. train, valid).

This workflow allows to separate the raw loading of the data and reusing it in various settings. Of course you can merge both steps or add many more.

**Note:** You can also define a function, which accepts a `config`, to build you `Dataset __class__`. During construction of the dataset, edflow only expects the module defined in the `config` behind `dataset` to accept the `config` as parameter. This behaviour is discouraged though, as one cannot inherit from those functions, limiting reusability.

It is also worth noting, that limiting the nestedness of your Dataset pipeline greatly increases reusability as it helps understanding what is happening to the raw data.

To further increase the usefulness of your datasets always add documentation and especially add an example, of what an example from you dataset might look like. This can be beautifully done using the function `edflow.util.pp2mkdtable()`, which formats the content of the example as markdown grid-table:

```
from edflow.util import pp2mkdtable

D = MyDataset()
example = D[10]

nicely_formatted_string = pp2mkdtable(example)

# Just copy it from the terminal
print(nicely_formatted_string)

# Or write it to a file
with open('output.md', 'w+') as example_file:
    example_file.write(nicely_formatted_string)
```

### 4.4.5 SubDataset

Given a dataset and an arbitrary list of indices, which must be in the range `[0, len(dataset_)]`, it will change the way the indices are interpreted.

## 4.5 Hooks

Hooks are a distinct EDFlow feature. You can think of them as plugins for your trainer.

Each Hook is inherited from `edflow.hooks.hook.Hook` or one of its inherited classes. It contains methods for different parts of a training loop:

- `before_epoch(epoch)`
- `before_step(step, fetches, feeds, batch)`
- `after_step(step, last_results)`
- `after_epoch(epoch)`
- `at_exception(exception)`

Coming soon:

- `before_training`
- `after_training`

EDFlow already comes with a number of hooks that allow for conversion of arrays to tensors, save checkpoints, call other hooks on intervals, log your data... All of this functionality can be expanded and transferred easily between projects which is one of the main assets of EDFlow.

In order to add a hook to your iterator simply expand the list of current hooks (some come ‘pre-installed’ with an iterator) like that:

```
self.hooks += [hook, another_hook, so_many_hooks]
```

after you initialized each hook with its respective parameters.

Once you seized the concept of hooks, they really are one of EDFlows greatest tools and come with all the advantages of modularity.

## 4.6 Models

Models can, but may not be your way to set up your machine learning model. For simple feed-forward networks it is a good idea to implement them as a model class (inherited from `object`) with simple `input` and `output` methods. Usually the whole actual model in between is defined in the `__init__` method.

The iterator then takes the model as one of its arguments and adds the optimizer logic to the respective model. This allows for easy exchange between models that only requires changing one line of code in the `config.yaml`.

More advanced models that may require to reuse parts of the model should only define the architecture but leave the inputs and outputs to the iterator.

## 4.7 Iterators

Iterators are the ‘main hub’ in EDFlow. They combine all other elements and manage the actual workflow.

You may have noticed that iterators are sometimes also called ‘Trainers’. That’s because the Iterator actually trains the model during the training phase. Afterwards the evaluator also qualifies as a more or less altered iterator.

The iterators `init` **must** include:

- initialisation of the model
- initialisation of the hooks and extension to the list of hooks
- `super().__init__(*args, **kwargs)` to invoke the `__init__` of its parent
- a `step_ops()` method that return a list of operations to execute on the feeds.

For machine learning purposes the `step_ops` method should always return a `train_ops` operation which calculates losses for the optimizer and returns the loss score.

Training logic should be implemented in the `run(fetches, feed_dict)` method. For instance, alternating training steps for GANs can be achieved by adding/removing the respective training operations from `fetches`. Many more possibilities, like exchanging the optimizer etc. are imaginable.

EDFlow provides a number of iterators out of the box, that feature most tools usually needed.

- `PyHookedModelIterator`
- `TFHookedModelIterator`
- `TFBaseTrainer`
- `TFBaseEvaluator`
- `TFFrequencyTrainer`
- `TFListTrainer`
- `TorchHookedModelIterator`

### 4.7.1 Epochs and Global Step

In one epoch you iterate through your whole dataset. If you specify a number of training steps then EDFlow will run as many epochs as possible with the given dataset but not finish an epoch if the desired training step is reached. `num_steps` trumps `num_epochs`

### 4.7.2 Accessing a Set of Fixed Examples

Sometimes it is useful to log the behavior of the model on a fixed batch of examples. To do this, one can access a set of fixed examples through `iterators.template_iterator.TemplateIterator.get_fixed_examples`. The indices of the fixed examples are chosen randomly on the length of the dataset by default. One can specify a list of fixed indices through the config in the following way.

```
fixed_example_indices:
  train: [0, 1, 2, 3]
  validation: [0, 1, 2, 3]
```

From within the iterator, the fixed examples can be access as follows

```
def step_op(self, model, **kwargs):
    def log_op():
        fixed_examples = self.get_fixed_examples("validation")
        # Do fancy stuff
        logs["fixed_metric"] = foo_bar(fixed_examples)
```

## 4.8 Integrations

### 4.8.1 git

Git integration can be enabled with the config parameter `--integrations/git True`. This assumes that you are starting edflow in a directory which is part of a git repository. For every run of edflow, git integration amounts to creating a tagged commit that contains a snapshot of all `.py` and `.yaml` files found under `code_root`, and all git tracked files at the time of the run. The name of the tag can be found as `git_tag: <tag>` in the `log.txt` of the run directory. You can get an overview of all tags with `git tag`. This allows you to easily compare your working directory to the code used in a previous experiment with `git diff <tag>` (git ignores untracked files in the working directory for its diff, so you might want to add them first), or two experiments you ran with `git diff <tag1> <tag2>`. Furthermore, it allows you to reproduce or continue training of an old experiment with `git checkout <tag>`.

### 4.8.2 wandb

Weights and biases integration can be enabled with the config parameter `--integrations/wandb/active True`. By default, this will log the config, scalar logging values and images to weights and biases. To disable image logging use `--integrations/wandb/handlers ["scalars"]`.

### 4.8.3 tensorboard

Tensorboard integration can be enabled with the config parameter `--integrations/tensorboard/active True`. By default, this will log the config, scalar logging values and images to tensorboard. To disable image logging use `--integrations/tensorboard/handlers ["scalars"]`.

### 4.8.4 minimal\_logging

By default, each image output is logged to a folder in `./logs/xxx`. The image format can be changed by changing the config as follows

```
integrations:
  minimal_logging:
    image_format: png # png or jpg. jpg saves a lot of space
```

## 4.9 Contributions

If you have any new applications that require custom hooks or iterators feel free to contribute at any time.

EDFlow is continuously expanded and gains new capabilities with every use. Examples of models are always welcome and we are happy if want to contribute in any way.

We are working on github and celebrate every pull request.

### 4.9.1 black

Before requesting a pull please run [black](#) for better code style or simply add [black](#) to your pre-commit hook:

0. Install [black](#) with

```
$ pip install black
```

1. Paste the following into at the top `<project-root>/git/hooks/pre-commit.sample`:

```
# run black on all staged files
staged=$(git diff --name-only --cached)
black $staged
# add them again after formatting
git add $staged
```

2. Rename `pre-commit.sample` to `pre-commit`
3. Make it executable using:

```
$ chmod +x pre-commit
```

4. Done!

Or run black by hand and use this command before every commit:

```
black ./
```

### 4.9.2 Continuous Integration

We use [travisCI](#) for continuous integration. You do not need to worry about it as long as your code passes all tests (this includes a formatting test with black).

---

**Note:** this should include an example to run the tests locally as well

---

### 4.9.3 Documentation

This is a short summary how the documentation works and how it can be built

The documentation uses [sphinx](#) and is available under [readthedocs.org](#). It also uses [all-contributors](#) for honoring contributors.

#### sphinx

To build the documentation locally, install *sphinx*

```
pip install sphinx sphinx_rtd_theme sphinxcontrib-apidoc
```

and run

```
$ cd docs
$ make html
```

The html files are available under the then existing directory `docs/_build/html/`

The preferred docstring format is [numpy](#).

We use *sphinx-apidoc* to track all files automatically::

```
$ cd docs
$ sphinx-apidoc -o ./source/source_files ../edflow
```

`docs/conf.py` contains a list of mocked dependencies. Make sure to add newly introduced dependencies to that list.

#### all-contributors

We use *all-contributors* locally and manage the contributors by hand.

To do so, install *all-contributors* as described here (we advise you to install it inside the repo but unstage the added files). Then run the following command to add a contributor or contribution::

```
all-contributors add <username> <contribution>
```

If this does not work for you (sometimes with npm the case) use::

```
./node_modules/.bin/all-contributors add <username> <contribution>
```

### 4.9.4 Known Issues

We noticed that mocking *numpy* in `config.py` will not work due to some requirements when importing *numpy* in EDFlow. Thus we need to require *numpy* when building the documentation.

Locally, this means that you need to have *numpy* installed in your environment.

Concerning [readthedocs.org](#), this means that we require a `readthedocs.yml` in the source directory which points to `extra_requirements` in `setup.py`, where *numpy* is a dependency. Other dependencies are *sphinx* and *sphinx\_rtd\_theme*.



## 4.10 FAQ

**How do I set a random seed?** Iterators or models or datasets can use a random seed from the config. How and where to set such seeds is application specific. It is recommended to create local pseudo-random-number-generators whenever possible, e.g. using `RandomState` for *numpy*.

Note that loading examples from a dataset happens in multiple processes, and the same random seed is copied to all child processes. If your `edflow.data.dataset.DatasetMixin.get_example()` method relies on random numbers, you should use `edflow.util.PRNGMixin` to make sure examples in your batches are independent. This will add a `prng` attribute (a *RandomState* instance) to your class, which will be seeded differently in each process.

**How do I run tests locally?** We use *pytest* for our tests and you can run `pytest --ignore="examples"` to run the general tests. To run framework dependent tests and see the precise testing protocol executed by *travis*, see `.travis.yml`.

**Why can't my implementations be imported?** In general, it is your responsibility to make sure *python* can import your implementations (e.g. install your implementations or add their location to your *PYTHONPATH*). To support the common practice of executing *edflow* one directory above your implementations, we add the current working directory to *python*'s import path.

For example, if `/a/b/myimplementations/c/d.py` contains your *MyModel* class, you can specify `myimplementations.c.d.MyModel` for your *model* config parameter if you run *edflow* in `a/b/`.

**Why is my code not copied to the log folder?** You can always specify the path to your code to copy with the `code_root` config option. Similar to how implementations are found (see previous question), we support the common practice of executing *edflow* one directory above your implementations.

For example, if `/a/b/myimplementations/c/d.py` contains your *MyModel* class and you specify `myimplementations.c.d.MyModel` for your *model* config parameter, *edflow* will use `$(pwd)/myimplementations` as the code root which assumes you are executing *edflow* in `/a/b/`.

**How can I kill edflow zombie processes?** You can use *edlist* to show all *edflow* processes. All sub-processes share the same process group id (*pgid*), so you can easily send all of them a signal with `kill -<pgid>`.

**How do I set breakpoints? `import pdb; pdb.set_trace()` is not working.** Use `import edflow.fpdb as pdb; pdb.set_trace()` instead. *edflow* runs trainings and evaluations in their own processes. Hence, `sys.stdin` must be set properly to be able to interact with the debugger.

**Error when using TFE: `NotImplementedError: object proxy must define reduce_ex()`.** This was addressed in this issue : <https://github.com/pessier/edflow/issues/240> When adding the config to a model that inherits from *tf.keras.Model*, the config cannot be dumped. It looks like keras changes lists within the config to a *ListWrapper* object, which are not reducable by `yaml.dump`

**Workaround** is to simply not do `self.config = config` and save everything you need in a field in the model.

## 4.11 edflow package

Submodules:

### 4.11.1 edflow.args module

#### Summary

Functions:

---

<code>get_parser</code>
-------------------------

---

#### Reference

`edflow.args.get_parser(**parser_kwargs)`

### 4.11.2 edflow.custom\_logging module

Module to handle logging in edflow.

Can be imported by application code to get loggers and find out where outputs should be stored.

#### Summary

Classes:

---

<code>ColorLineFormatter</code>	
<code>LogSingleton</code>	alias of <code>edflow.custom_logging.log</code>
<code>TqdmHandler</code>	A logging handler compatible with tqdm progress bars.
<code>log</code>	Singleton managing all loggers for a run.
<code>run</code>	Singleton managing all directories for a run.

---

Functions:

---

<code>get_logger</code>	Get logger.
-------------------------	-------------

---

#### Reference

**class** `edflow.custom_logging.run`

Bases: `object`

Singleton managing all directories for a run.

Calling the `init` method below will set up a logging directory structure that should be used for this run. Application code can import this class and use its attributes to figure out where to store their outputs.

---

**Note:** This class is intended to provide run information without the need to pass it through. Thus it behaves

like a singleton by storing all information on the class object itself and not an instance of the class.

---

**now**

Representing time of initialization.

**Type** str

**postfix**

User specified postfix of run directory or eval directory.

**Type** str

**name**

The name of the current run. Stays consistent on resuming.

**Type** str

**git\_tag**

If activated and a git repo was found, this attribute contains the tag name pointing to a commit recording the state of the repository when this run was started.

**Type** str

**resumed**

True if this run was resumed.

**Type** bool

**code\_root**

Path where code is copied from.

**Type** str

**code**

Path where code is copied to.

**Type** str

**root**

Path under which all outputs of the run should be stored.

**Type** str

**train**

Path to store train outputs in.

**Type** str

**eval**

Path to eval subfolders.

**Type** str

**latest\_eval**

Path to store eval outputs in.

**Type** str

**configs**

Path to store configs in.

**Type** str

**checkpoints**

Path to store checkpoints in.

**Type** str

**exists** = False

True if log structure was initialized.

**classmethod** **init** (*log\_dir=None, run\_dir=None, code\_root='.', postfix=None, log\_level='info', git=False*)

Initialize logging for this run.

After execution of this method, the log directory structure was created, code was copied and committed if desired, and some basic system information has been logged. Subsequent use of loggers from `log.get_logger` will result in log files written to the run directory.

#### Parameters

- **log\_dir** (*str*) – Create new run directory under this directory.
- **run\_dir** (*str*) – Resume in existing run directory.
- **code\_root** (*str*) – Path to where the code lives. py and yaml files will be copied into run directory.
- **postfix** (*str*) – Identifier appended to run directory if non-existent else to latest eval directory.
- **log\_level** (*str*) – Default log level for loggers.
- **git** (*bool*) – If True, put code into tagged commit.

**class** `edflow.custom_logging.TqdmHandler` (*stream=None*)

Bases: `logging.StreamHandler`

A logging handler compatible with tqdm progress bars.

**emit** (*record*)

Emit a record.

If a formatter is specified, it is used to format the record. The record is then written to the stream with a trailing newline. If exception information is present, it is formatted using `traceback.print_exception` and appended to the stream. If the stream has an ‘encoding’ attribute, it is used to determine how to do the output to the stream.

**class** `edflow.custom_logging.log`

Bases: `object`

Singleton managing all loggers for a run.

---

**Note:** This class is intended to provide logging facilities without the need to pass it through. Thus it behaves like a singleton by storing all information on the class object itself and not an instance of the class.

---

**target** = 'root'

Current default target to write log file to.

**level** = 20

Current default log level for new loggers.

**loggers** = []

List of all loggers.

**classmethod** **set\_log\_target** (*which*)

Set default target where log file is written to.

**classmethod** `get_logger` (*name*, *which=None*, *level=None*)

Get logger.

If run was initialized, returns a logger which is compatible with tqdm progress bars and logs into a file in the run directory. Otherwise, returns a basic logger.

#### Parameters

- **name** (*str* or *object*) – Name of the logger. If not a string, the name of the given object class is used.
- **which** (*str*) – Subdirectory in the project folder where log file is written to.
- **level** (*str*) – Log level of the logger.

**classmethod** `set_log_level` (*level*)

Set log level of all existing and default log level of all future loggers.

**class** `edflow.custom_logging.ColorLineFormatter` (*use\_color*, *\*args*, *\*\*kwargs*)

Bases: `logging.Formatter`

**\_\_init\_\_** (*use\_color*, *\*args*, *\*\*kwargs*)

Initialize the formatter with specified format strings.

Initialize the formatter either with the specified format string, or a default as described above. Allow for specialized date formatting with the optional `datefmt` argument. If `datefmt` is omitted, you get an ISO8601-like (or RFC 3339-like) format.

Use a style parameter of `'%'`, `'{'` or `'$'` to specify that you want to use one of `%`-formatting, `str.format()` (`{}`) formatting or `string.Template` formatting in your format string.

Changed in version 3.2: Added the `style` parameter.

**format** (*record*)

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using `LogRecord.getMessage()`. If the formatting string uses the time (as determined by a call to `usesTime()`, `formatTime()` is called to format the event time. If there is exception information, it is formatted using `formatException()` and appended to the message.

`edflow.custom_logging.LogSingleton`

alias of `edflow.custom_logging.log`

`edflow.custom_logging.get_logger` (*name*, *which=None*, *level=None*)

Get logger.

If run was initialized, returns a logger which is compatible with tqdm progress bars and logs into a file in the run directory. Otherwise, returns a basic logger.

#### Parameters

- **name** (*str* or *object*) – Name of the logger. If not a string, the name of the given object class is used.
- **which** (*str*) – Subdirectory in the project folder where log file is written to.
- **level** (*str*) – Log level of the logger.

### 4.11.3 edflow.debug module

#### Summary

Classes:

---

*ConfigDebugDataset*

---

*DebugDataset*

---

*DebugIterator*

---

*DebugModel*

---

Functions:

---

*debug\_step\_op*

---

#### Reference

**class** edflow.debug.**DebugModel** (\*a, \*\*k)

Bases: object

**\_\_init\_\_** (\*a, \*\*k)

Initialize self. See help(type(self)) for accurate signature.

edflow.debug.**debug\_step\_op** (model, \*args, \*\*kwargs)

**class** edflow.debug.**DebugIterator** (\*args, \*\*kwargs)

Bases: *edflow.iterators.model\_iterator.PyHookedModelIterator*

**\_\_init\_\_** (\*args, \*\*kwargs)

Constructor.

##### Parameters

- **model** (*object*) – Model class.
- **num\_epochs** (*int*) – Number of times to iterate over the data.
- **hooks** (*list*) – List containing Hook instances.
- **hook\_freq** (*int*) – Frequency at which hooks are evaluated.
- **bar\_position** (*int*) – Used by tqdm to place bars at the right position when using multiple Iterators in parallel.

**step\_ops** ()

Defines ops that are called at each step.

##### Returns

**Return type** The operation run at each step.

**class** edflow.debug.**DebugDataset** (size=100, offset=0, other\_labels=False, other\_ex\_keys=False, \*args, \*\*kwargs)

Bases: *edflow.data.dataset\_mixin.DatasetMixin*

**\_\_init\_\_** (size=100, offset=0, other\_labels=False, other\_ex\_keys=False, \*args, \*\*kwargs)

Initialize self. See help(type(self)) for accurate signature.

**get\_example** (i)

---

**Note:** Please the documentation of `DatasetMixin` to not be confused.

---

Add default behaviour for datasets defining an attribute `data`, which in turn is a dataset. This happens often when stacking several datasets on top of each other.

The default behaviour now is to return `self.data.get_example(idx)` if possible, and otherwise revert to the original behaviour.

#### property labels

Add default behaviour for datasets defining an attribute `data`, which in turn is a dataset. This happens often when stacking several datasets on top of each other.

The default behaviour is to return `self.data.labels` if possible, and otherwise revert to the original behaviour.

```
class edflow.debug.ConfigDebugDataset (config)
    Bases: edflow.debug.DebugDataset

    __init__ (config)
        Initialize self. See help(type(self)) for accurate signature.
```

## 4.11.4 edflow.explore module

### Summary

Functions:

<i>custom_visualizations</i>	Displays custom visualizations in streamlit
<i>display</i>	Display item in streamlit
<i>display_default</i>	Find out how item could be displayed
<i>explore</i>	Explore dataset specified in config
<i>selector</i>	Show select box to choose display mode of obj in streamlit
<i>show_example</i>	Show example of dataset

### Reference

`edflow.explore.display_default (obj)`

Find out how item could be displayed

**Parameters** `obj` (*Any*) – Item of example

**Returns** One of “Image”, “Text”, “Flow”, “Segmentation”, “None”

**Return type** `str`

`edflow.explore.display (key, obj)`

Display item in streamlit

**Parameters**

- **key** (*str*) – Subheader to be displayed
- **obj** (*Any*) – Item of example to be displayed

`edflow.explore.selector(key, obj)`  
 Show select box to choose display mode of obj in streamlit

**Parameters**

- **key** (*str*) – Key of item to be displayed
- **obj** (*Any*) – Item to be displayed

**Returns** Selected display method for item

**Return type** `str`

`edflow.explore.custom_visualizations(ex, config)`  
 Displays custom visualizations in streamlit

The visualizations can be inserted to the config via their import path. Everyone can implement a custom visualization for an example.

The visualization functions must accept the example and config as positional arguments.

## Examples

Add visualizations to the text box with their import path. For example:

```
edflow.util.edexplore.display_flow_on_image
```

A valid visualization function could look like for example:

```
import streamlit as st
from edflow.util.edexplore import isimage, st_get_list_or_dict_item

def my_visualization(ex, config):
    st.write("This is my visualization")

    image1, image1_key = st_get_list_or_dict_item(ex, "image1", filter_fn=isimage)

    st.image((image1 + 1) / 2)

    image2 = ex["image2"]
    image3 = ex["image3"]

    st.image((image2 + 1) / 2)
    st.image((image3 + 1) / 2)
```

Visualizations can be displayed by default, if they are specified in the config. An example for the configuration yaml file would be:

```
edexplore:
  visualizations:
    custom:
      vis1:
        path: my_package.visualizations.my_visualization
```

**Parameters**

- **ex** (*dict*) – Example to be visualized
- **config** (*dict*) – Edexplore config



`edflow.explore.show_example(dset, idx, config)`

Show example of dataset

#### Parameters

- **dset** (*DatasetMixin*) – Dataset to be shown
- **idx** (*int*) – Index to be shown
- **config** (*dict*) – Config used to show example

`edflow.explore.explore(config, disable_cache=False)`

Explore dataset specified in config

#### Parameters

- **config** (*dict*) – Edflow config dict used to explore dataset
- **disable\_cache** (*bool, optional*) – Disable cache while loading dataset, by default False

### 4.11.5 edflow.fpdb module

#### Summary

Classes:

<i>ForkedPdb</i>	Pdb subclass which works in subprocesses.
------------------	---

#### Reference

**class** `edflow.fpdb.ForkedPdb` (*completekey='tab', stdin=None, stdout=None, skip=None, nosigint=False, readrc=True*)

Bases: `pdb.Pdb`

Pdb subclass which works in subprocesses. We need to set `stdin` to be able to interact with the debugger. `os.fdopen` instead of `open("/dev/stdin")` keeps `readline` working. <https://stackoverflow.com/a/31821795>

**interaction** (*\*args, \*\*kwargs*)

### 4.11.6 edflow.main module

#### Summary

Functions:

<i>test</i>	Run tests.
<i>train</i>	Run training.

## Reference

`edflow.main.train` (*config, root, checkpoint=None, retrain=False, debug=False*)

Run training. Loads model, iterator and dataset according to config.

`edflow.main.test` (*config, root, checkpoint=None, nogpu=False, bar\_position=0, debug=False*)

Run tests. Loads model, iterator and dataset from config.

### 4.11.7 edflow.project\_manager module

Subpackages:

### 4.11.8 edflow.applications package

Submodules:

#### edflow.applications.tf\_perceptual\_loss module

##### Summary

Classes:

---

*VGG19Features*

---

Functions:

---

<i>preprocess_input</i>	Preprocesses a tensor encoding a batch of images.
-------------------------	---

---

## Reference

`edflow.applications.tf_perceptual_loss.preprocess_input` (*x*)

Preprocesses a tensor encoding a batch of images. :param *x*: input tensor, 4D in [-1,1] :type *x*: tf.Tensor

**Returns** Preprocessed tensor

**Return type** tf.Tensor

**class** `edflow.applications.tf_perceptual_loss.VGG19Features` (*session, feature\_layers=None, feature\_weights=None, gram\_weights=None, default\_gram=0.1, original\_scale=False*)

Bases: object

**\_\_init\_\_** (*session, feature\_layers=None, feature\_weights=None, gram\_weights=None, default\_gram=0.1, original\_scale=False*)

Initialize self. See help(type(self)) for accurate signature.

**extract\_features** (*x*)

x should be rgb in [-1,1].

**make\_feature\_ops** (*x*)  
 x should be rgb tensor in [-1,1].

**grams** (*f<sub>s</sub>*)

**make\_loss\_op** (*x*, *y*)  
 x, y should be rgb tensors in [-1,1]. Uses l1 and spatial average.

**make\_nll\_op** (*x*, *y*, *log\_variances*, *gram\_log\_variances=None*, *calibrate=True*)  
 x, y should be rgb tensors in [-1,1]. This version treats every layer independently.

**make\_l1\_nll\_op** (*x*, *y*, *log\_variance*)  
 x, y should be rgb tensors in [-1,1]. Uses make\_loss\_op to compute version compatible with previous experiments.

**make\_style\_op** (*x*, *y*)

### 4.11.9 edflow.config package

Submodules:

#### edflow.config.commandline\_kwargs module

#### Summary

Functions:

<i>parse_unknown_args</i>	
<i>update_config</i>	additional_kwargs are added in order of the keys' length, e.g.

#### Reference

edflow.config.commandline\_kwargs.**update\_config** (*config*, *additional\_kwargs*)  
 additional\_kwargs are added in order of the keys' length, e.g. 'a' is overridden by 'a/b'.

edflow.config.commandline\_kwargs.**parse\_unknown\_args** (*unknown*,  
*is\_wandb\_sweep=False*)

### 4.11.10 edflow.data package

Submodules:

#### edflow.data.dataset module

##### Datasets TLDR

Datasets contain examples, which can be accessed by an index:

```
example = Dataset[index]
```

Each example is annotated by labels. These can be accessed via the `labels` attribute of the dataset:

```
label = Dataset.labels[key][index]
```

To make a working dataset you need to implement a `get_example()` method, which must return a dict, a `__len__()` method and define the `labels` attribute, which must be a dict, that can be empty.

**Warning:** Dataset, which are specified in the edflow config must accept one positional argument `config`!

If you have to worry about dataloading take a look at the `LateLoadingDataset`. You can define datasets to return examples containing callables for heavy dataloading, which are only executed by the `LateLoadingDataset`. Having this class as the last in your dataset pipeline can potentially speed up your data loading.

##### Summary

##### Reference

#### edflow.data.dataset\_mixin module

##### Summary

Classes:

<i>ConcatenatedDataset</i>	A dataset which concatenates given datasets.
<i>DatasetMixin</i>	Our fork of the <code>chainer-Dataset</code> class. Every Dataset used with edflow should at some point inherit from this baseclass..
<i>SubDataset</i>	A subset of a given dataset.

## Reference

**class** `edflow.data.dataset_mixin.DatasetMixin`

Bases: `object`

Our fork of the `chainer-Dataset` class. Every Dataset used with edflow should at some point inherit from this baseclass.

## Notes

### Necessary and best practices

When implementing your own dataset you need to specify the following methods:

- `__len__` defines how many examples are in the dataset
- `get_example` returns one of those examples given an index. The example must be a dictionary

### Labels

Additionally the dataset class should specify an attribute `labels`, which works like a dictionary with lists or arrays behind each keyword, that have the same length as the dataset. The dictionary can also be empty if you do not want to define labels.

The philosophy behind having both a `get_example()` method and the `labels` attribute is to split the dataset into compute heavy and easy parts. Labels should be quick to load at construction time, e.g. by loading a `.npy` file or a `.csv`. They can then be used to quickly manipulate the dataset. When getting the actual example we can do the heavy lifting like loading and/or manipulating images.

**Warning:** Labels must be `dict`s of `numpy` arrays and not `list`s! Otherwise many operations do not work and result in incomprehensible errors.

### Batching

As one usually works with batched datasets, the compute heavy steps can be hidden through parallelization. This is all done by the `make_batches()`, which is invoked by edflow automatically.

### Default Behaviour

As one sometimes stacks and chains multiple levels of datasets it can become cumbersome to define `__len__`, `get_example` and `labels`, if all one wants to do is evaluate their respective implementations of some other dataset, as can be seen in the code example below:

```
SomeDerivedDataset(DatasetMixin):
    def __init__(self):
        self.other_data = SomeOtherDataset()
        self.labels = self.other_data.labels

    def __len__(self):
        return len(self.other_data)

    def get_example(self, idx):
        return self.other_data[idx]
```

This can be omitted when defining a data attribute when constructing the dataset. `DatasetMixin` implements these methods with the default behaviour to wrap around the corresponding methods of the underlying data attribute. Thus the above example becomes

```
SomeDerivedDataset (DatasetMixin):
    def __init__(self):
        self.data = SomeOtherDataset()
```

If `self.data` has a `labels` attribute, labels of the derived dataset will be taken from `self.data`.

``+`` and ``\*``

Sometimes you want to concatenate two datasets or multiply the length of one dataset by concatenating it several times to itself. This can easily be done by adding Datasets or multiplying one by an integer factor.

```
A = C + B # Adding two Datasets
D = 3 * A # Multiplying two datasets
```

The above is equivalent to

```
A = ConcatenatedDataset(C, B) # Adding two Datasets
D = ConcatenatedDataset(A, A, A) # Multiplying two datasets
```

### Labels in the example ``dict``

Oftentimes it is good to store and load some values as labels as it can increase performance and decrease storage size, e.g. when storing scalar values. If you need these values to be returned by the `get_example()` method, simply activate this behaviour by setting the attribute `append_labels` to `True`.

```
SomeDerivedDataset (DatasetMixin):
    def __init__(self):
        self.labels = {'a': [1, 2, 3]}
        self.append_labels = True

    def get_example(self, idx):
        return {'a' : idx**2, 'b': idx}

    def __len__(self):
        return 3

S = SomeDerivedDataset()
a = S[2]
print(a) # {'a': 3, 'b': 2}

S.append_labels = False
a = S[2]
print(a) # {'a': 4, 'b': 2}
```

Labels are appended to your example, after all code is executed from your `get_example` method. Thus, if there are keys in your labels, which can also be found in the examples, the label entries will override the values in your example, as can be seen in the example above.

`get_example(*args, **kwargs)`

---

**Note:** Please the documentation of `DatasetMixin` to not be confused.

---

Add default behaviour for datasets defining an attribute `data`, which in turn is a dataset. This happens often when stacking several datasets on top of each other.

The default behaviour now is to return `self.data.get_example(idx)` if possible, and otherwise revert to the original behaviour.

**property labels**

Add default behaviour for datasets defining an attribute `data`, which in turn is a dataset. This happens often when stacking several datasets on top of each other.

The default behaviour is to return `self.data.labels` if possible, and otherwise revert to the original behaviour.

**property append\_labels****property expand**

**class** `edflow.data.dataset_mixin.ConcatenatedDataset` (*\*datasets, balanced=False*)

Bases: `edflow.data.dataset_mixin.DatasetMixin`

A dataset which concatenates given datasets.

**\_\_init\_\_** (*\*datasets, balanced=False*)

**Parameters**

- **\*datasets** (`DatasetMixin`) – All datasets we want to concatenate
- **balanced** (*bool*) – If `True` all datasets are padded to the length of the longest dataset. Padding is done in a cycled fashion.

**get\_example** (*i*)

Get example and add dataset index to it.

**property labels**

Add default behaviour for datasets defining an attribute `data`, which in turn is a dataset. This happens often when stacking several datasets on top of each other.

The default behaviour is to return `self.data.labels` if possible, and otherwise revert to the original behaviour.

**class** `edflow.data.dataset_mixin.SubDataset` (*data, subindices*)

Bases: `edflow.data.dataset_mixin.DatasetMixin`

A subset of a given dataset.

**\_\_init\_\_** (*data, subindices*)

Initialize self. See `help(type(self))` for accurate signature.

**get\_example** (*i*)

Get example and process. Wrapped to make sure stacktrace is printed in case something goes wrong and we are in a `MultiprocessIterator`.

**property labels**

Add default behaviour for datasets defining an attribute `data`, which in turn is a dataset. This happens often when stacking several datasets on top of each other.

The default behaviour is to return `self.data.labels` if possible, and otherwise revert to the original behaviour.

Subpackages:

## edflow.data.agnostics package

Submodules:

### edflow.data.agnostics.concatenated module

#### Summary

Classes:

<i>DisjunctExampleConcatenatedDataset</i>	Concatenates a list of disjunct datasets.
<i>ExampleConcatenatedDataset</i>	Concatenates a list of datasets along the example axis.

#### Reference

**class** edflow.data.agnostics.concatenated.**ExampleConcatenatedDataset** (\*datasets)

Bases: *edflow.data.dataset\_mixin.DatasetMixin*

Concatenates a list of datasets along the example axis.

---

**Note:** All datasets must be of same length and must return examples with the same keys and behind those keys with the same type and shape.

---

If dataset A returns examples of form {'a': x, 'b': x} and dataset B of form {'a': y, 'b': y} the ExampleConcatenatedDataset (A, B) return examples of form {'a': [x, y], 'b': [x, y]}.

**\_\_init\_\_** (\*datasets)

**Parameters** \*datasets (*DatasetMixin*) – All the datasets to concatenate.

**set\_example\_pars** (start=None, stop=None, step=None)

Allows to manipulate the length and step of the returned example lists.

**property labels**

Now each index corresponds to a sequence of labels.

**get\_example** (i)

---

**Note:** Please the documentation of DatasetMixin to not be confused.

---

Add default behaviour for datasets defining an attribute data, which in turn is a dataset. This happens often when stacking several datasets on top of each other.

The default behaviour now is to return self.data.get\_example (idx) if possible, and otherwise revert to the original behaviour.

**class** edflow.data.agnostics.concatenated.**DisjunctExampleConcatenatedDataset** (\*datasets, dis-junct=True, same\_length=True)

Bases: *edflow.data.dataset\_mixin.DatasetMixin*



Concatenates a list of disjunct datasets.

---

**Note:** All datasets must be of same length and labels and returned keys must be disjunct. If labels or keys are not disjunct, set the optional parameter *disjunct* to False, to use the value of the last dataset containing the key. Datasets can have different length if *same\_length* is set to False.

---

If dataset A returns examples of form {'a': w, 'b': x} and dataset B of form {'c': y, 'd': z} the `DisjunctExampleConcatenatedDataset(A, B)` return examples of form {'a': w, 'b': x, 'c': y, 'd': z}.

`__init__` (\*datasets, disjunct=True, same\_length=True)

#### Parameters

- **\*datasets** (`DatasetMixin`) – All the datasets to concatenate.
- **disjunct** (`bool`) – labels and returned keys do not have to be disjunct. Last dataset overwrites values
- **same\_length** (`bool`) – Datasets do not have to be of same length. Concatenated dataset has length of smallest dataset.

`get_example` (i)

---

**Note:** Please the documentation of `DatasetMixin` to not be confused.

---

Add default behaviour for datasets defining an attribute `data`, which in turn is a dataset. This happens often when stacking several datasets on top of each other.

The default behaviour now is to return `self.data.get_example(idx)` if possible, and otherwise revert to the original behaviour.

## edflow.data.agnostics.csv\_dset module

### Summary

Classes:

<code>CsvDataset</code>	Using a csv file as index, this Dataset returns only the entries in the csv file, but can be easily extended to load other data using the <code>ProcessedDatasets</code> .
-------------------------	--

### Reference

**class** `edflow.data.agnostics.csv_dset.CsvDataset` (csv\_root, \*\*pandas\_kwargs)

Bases: `edflow.data.dataset_mixin.DatasetMixin`

Using a csv file as index, this Dataset returns only the entries in the csv file, but can be easily extended to load other data using the `ProcessedDatasets`.

`__init__` (csv\_root, \*\*pandas\_kwargs)

#### Parameters

- **csv\_root** (*str*) – Path/to/the/csv containing all datapoints. The first line in the file should contain the names for the attributes in the corresponding columns.
- **pandas\_kwargs** (*kwargs*) – Passed to `pandas.read_csv()` when loading the csv file.

**get\_example** (*idx*)

Returns all entries in row *idx* of the labels.

## edflow.data.agnostics.late\_loading module

### Summary

Classes:

---

*LateLoadingDataset*

The *LateLoadingDataset* allows to work with examples containing *Callables*, which are evaluated by this Dataset.

---

Functions:

---

*expand*

---

### Reference

**class** edflow.data.agnostics.late\_loading.**LateLoadingDataset** (*base\_dset*)

Bases: *edflow.data.dataset\_mixin.DatasetMixin*

The *LateLoadingDataset* allows to work with examples containing *Callables*, which are evaluated by this Dataset. This way you can define data loading routines for images or other time consuming things in a base dataset, then add lots of data rearranging logic on top of this base dataset and in the end only load the subset of examples, you really want to use by calling the routines.

```
class BaseDset:
    def get_example(self, idx):
        def _loading_routine():
            load_image(idx)

        return {'image': _loading_routine}

class AnchorDset:
    def __init__(self):
        B = BaseDset()

        self.S = SequenceDataset(B, 5)

    def get_example(self, idx):
        ex = self.S[idx]

        out = {}
        out['anchor1'] = ex['image'][0]
        out['anchor2'] = ex['image'][-1]
```

(continues on next page)

(continued from previous page)

```

    return out

final_dset = LateLoadingDataset(AnchorDset())

```

```

__init__(base_dset)
    Initialize self. See help(type(self)) for accurate signature.

get_example(idx)

```

---

**Note:** Please the documentation of `DatasetMixin` to not be confused.

---

Add default behaviour for datasets defining an attribute `data`, which in turn is a dataset. This happens often when stacking several datasets on top of each other.

The default behaviour now is to return `self.data.get_example(idx)` if possible, and otherwise revert to the original behaviour.

```
edflow.data.agnostics.late_loading.expand(value)
```

## edflow.data.agnostics.subdataset module

## edflow.data.believers package

Submodules:

## edflow.data.believers.meta module

## Summary

Classes:

<i>MetaDataset</i>	The <i>MetaDataset</i> allows for easy data reading using a simple interface.
--------------------	---

Functions:

<i>clean_keys</i>	Removes all loader information from the keys.
<i>load_labels</i>	<b>param root</b> Where to look for the labels.
<i>loader_from_key</i>	Returns the name, loader pair given a key.
<i>setup_loaders</i>	Creates a map of key -> function pairs, which can be used to postprocess label values at each <code>__getitem__</code> call.

## Reference

**class** edflow.data.believers.meta.**MetaDataset** (*root*)

Bases: *edflow.data.dataset\_mixin.DatasetMixin*

The *MetaDataset* allows for easy data reading using a simple interface.

All you need to do is hand the constructor a path and it will look for all data in a special format and load it as numpy arrays. If further specified in a meta data file or the name of the label array, when calling the `getitem` method of the dataset, a special loader function will be called.

Let's take a look at an example data folder of the following structure:

```
root/
├ meta.yaml
├ images/
│   ├── image_1.png
│   ├── image_2.png
│   └── image_3.png
├ ...
├   └── image_10000.png
├ image:image-*--10000-*--str.npy
├ attr1-*--10000-*--int.npy
├ attr2-*--10000x2-*--int.npy
└ kps-*--10000x17x3-*--int.npy
```

The `meta.yaml` file looks like this:

```
description: |
    This is a dataset which loads images.
    All paths to the images are in the label `image`.

loader_kwargs:
    image:
        support: "-1->1"
```

The resulting dataset has the following labels:

- `image_`: the paths to the images. Note the extra `_` at the end.
- `attr1`
- `attr2`
- `kps`

When using the `__getitem__` method of the dataset, the image loader will be applied to the image label at the given index and the image will be loaded from the given path.

As we have specified loader keyword arguments, we will get the images with a support of `[-1, 1]`.

**\_\_init\_\_** (*root*)

**Parameters** *root* (*str*) – Where to look for all the data.

**get\_example** (*idx*)

Loads all loadable data from the labels.

**Parameters** *idx* (*int*) – The index of the example to load

**show** ()

`edflow.data.believers.meta.setup_loaders(labels, meta_dict)`

Creates a map of key -> function pairs, which can be used to postprocess label values at each `__getitem__` call.

Loaders defined in `meta_dict` supersede those define in the label keys.

#### Parameters

- **labels** (*dict*(*str*, *numpy.memmap*)) – Labels contain all load-easy dataset relevant data. If the key follows the pattern `name:loader`, this function will try to finde the corresponding loader in `DEFAULT_LOADERS`.
- **meta\_dict** (*dict*) – A dictionary containing all dataset relevent information, which is the same for all examples. This function will try to find the entry `loaders` in the dictionary, which must contain another `dict` with `name:loader` pairs. Here `loader` must be either an entry in `DEFAULT_LOADERS` or a loadable import path. You can additionally define an entry `loader_kwargs`, which must contain `name:dict` pairs. The dictionary is passed as keyword arguments to the loader corresponding to `name`.

#### Returns

- **loaders** (*dict*) – Name, function pairs, to apply loading logic based on the labels with the specified names.
- **loader\_kwargs** (*dict*) – Name, dict pairs. The dicts are passed to the loader functions as keyword arguments.

`edflow.data.believers.meta.load_labels(root)`

**Parameters** `root` (*str*) – Where to look for the labels.

**Returns** `labels` – All labels as `np.memmap` s.

**Return type** `dict`

`edflow.data.believers.meta.clean_keys(labels, loaders)`

Removes all loader information from the keys.

**Parameters** `labels` (*dict*(*str*, *numpy.memmap*)) – Labels contain all load-easy dataset relevant data.

**Returns** `labels` – The original labels, with keys without the `:loader` part.

**Return type** `dict(str, numpy.memmap)`

`edflow.data.believers.meta.loader_from_key(key)`

Returns the name, loader pair given a key.

## edflow.data.believers.meta\_loaders module

### Summary

Functions:

---

<code>category</code>	Turns an abstract category label into a readable label.
-----------------------	---

---

`image_loader`

**param path** Where to finde the image.

---

continues on next page

Table 19 – continued from previous page

*numpy\_loader***param path** Where to finde the array.

## Reference

`edflow.data.believers.meta_loaders.image_loader(path, root="", support='0->255', resize_to=None)`

### Parameters

- **path** (*str*) – Where to finde the image.
- **root** (*str*) – Root path, at which the suuplied path starts. E.g. if all paths supplied to this function are relative to `/export/scratch/you_are_great/dataset`, this path would be root.
- **support** (*str*) –  
**Defines the support and data type of the loaded image. Must be one of**
  - `0->255`: The PIL default. Datatype is `np.uint8` and all values are integers between 0 and 255.
  - `0->1`: Datatype is `np.float32` and all values are floats between 0 and 1.
  - `-1->1`: Datatype is `np.float32` and all values are floats between -1 and 1.
- **resize\_to** (*list*) – If not `None`, the loaded image will be resized to these dimensions. Must be a list of two integers or a single integer, which is interpreted as list of two integers with same value.

**Returns im** – An image loaded using PIL. Image and adjusted to the range as specified.

**Return type** `np.array`

`edflow.data.believers.meta_loaders.numpy_loader(path, root="")`

**Parameters path** (*str*) – Where to finde the array.

**Returns arr** – An array loaded using `np.load`

**Return type** `np.array`

`edflow.data.believers.meta_loaders.category(index, categories)`

Turns an abstract category label into a readable label.

Example:

Your dataset has the label `pid` which has integer entries like `[0, 0, 0, ..., 2, 2]` between 0 and 3.

Inside the dataset's `meta.yaml` you define

```
# meta.yaml
# ...
loaders:
  pid: category
loader_kwargs:
  pid:
    categories: ['besser', 'pesser', 'Mimo Tilbich']
```

Now examples will be annotated with `{pid: 'besser'}` if the pid is 0, `{pid: 'pesser'}` if pid is 1 or `{pid: 'Mimo Tilbich'}` if the pid is 2.

Note that categories can be anything that implements a `__getitem__` method. You simply need to take care, that it understands the `index` value it is passed by this loader function.

#### Parameters

- **index** (*int*, *Hashable*) – Some value that will be passed to categories's `__getitem__()` method. I.e. categories can be a list or dict or whatever you want!
- **categories** (list, dict, object with `__getitem__` method) – Defines the categories you have in you dataset. Will be accessed like `categories[index]`

**Returns** `category` – `categories[index]`

**Return type** `object`

### edflow.data.believers.meta\_util module

#### Summary

Functions:

---

<code>store_label_mmap</code>	Stores the numpy array data as numpy <i>MemoryMap</i> with the naming convention, that is loadable by <i>MetaDataset</i> .
-------------------------------	--

---

#### Reference

`edflow.data.believers.meta_util.store_label_mmap(data, root, name)`

Stores the numpy array data as numpy *MemoryMap* with the naming convention, that is loadable by *MetaDataset*.

#### Parameters

- **data** (*numpy.ndarray*) – The data to store.
- **root** (*str*;) – Where to store the memory map.
- **name** (*str*) – The name of the array. If loaded by *MetaDataset* this will be the key in the labels dictionary at which one can find the data.

### edflow.data.believers.meta\_view module

#### Summary

Classes:

---

<code>MetaViewDataset</code>	The <i>MetaViewDataset</i> implements a way to render out a view of a base dataset without the need to rewrite/copy the load heavy data in the base dataset.
------------------------------	--

---

## Reference

**class** `edflow.data.believers.meta_view.MetaViewDataset` (*root*)

Bases: `edflow.data.believers.meta.MetaDataset`

The `MetaViewDataset` implements a way to render out a view of a base dataset without the need to rewrite/copy the load heavy data in the base dataset.

**To use the `MetaViewDataset` you need to define two things:**

1. **A base dataset as import string in the `meta.yaml` file.** Use the key `base_dset` for this. This should preferably be a function or class, which is passed the kwargs `base_kwargs` as defined in the `meta.yaml`.
2. **A view in the form of a numpy memmap or a nested object of `dict`s` and `list`s`** with `memmaps` at the leaves, each storing the indices used for the view in this dataset. The arrays can be of any dimensionality, but no value must be outside the range `[0, len(base_dataset)]` and they must all be of the same length.

The dimensionality of the view is reflected in the nestedness of the resulting examples.

### Example

You have a base dataset, which contains video frames. It has length `N`.

Say you want to have a combination of two views on your dataset: One contains all `M` possible subsequences of length 5 of videos contained in the dataset and one contains an appearance image per each example with the same person as in the sequence.

All you need is to define two numpy arrays, one with the indices belonging to the sequenced frames and one with indices of examples of the appearance images. They should look something like this:

```
# Sequence indices
seq_idxxs = [[0, 1, 2, 3, 4],
             [1, 2, 3, 4, 5],
             [2, 3, 4, 5, 6],
             [3, 4, 5, 6, 7],
             ...
             [N-4, N-3, N-2, N-1, N],
print(seq_idxxs.shape) # [M, 5]

# Sequence indices
app_idxxs = [12,
             12,
             15,
             10,
             ..
             109],
print(app_idxxs.shape) # [M]
```

Knowing your views, create a folder, where you want to store your view dataset, i.e. at some path `ROOT`. Create a folder `ROOT/labels` and store the views according to the label naming scheme as defined in the `MetaDataset`. You can use the function `edflow.data.believers.meta_util.store_label_mmap()` for this. You can also store the views in any subfolder of labels, which might come in handy if you have a lot of labels and want to keep things clean.

Finally create a file `ROOT/meta.yaml`.

Our folder should look something like this:



```

ROOT/
├ labels/
│   ├── app_view--{M}--int64.npy
│   └── seq_view--{M}x5--int64.npy
└ meta.yaml

```

Now let us fill the `meta.yaml`. All we need to do is specify the base dataset and how we want to use our views:

```

# meta.yaml

description: |
    This is our very own View on the data.
    Let's have fun with it!

base_dset: import.path.to.dset_object
base_kwargs:
    stuff: needed_for_construction

views:
    appearance: app_view
    frames: seq_view

```

Now we are ready to construct our view on the base dataset! Use `.show()` to see how the dataset looks like. This works especially nice in a jupyter notebook.

```

ViewDset = MetaViewDataset('ROOT')

print(ViewDset.labels.keys()) # ['appearance', 'frames']
print(len(ViewDset))         # {M}

ViewDset.show() # prints the labels and the first example

```

`__init__(root)`

Parameters **root** (*str*) – Where to look for all the data.

`get_example(idx)`

Get the examples from the base dataset at defined at `view[idx]`. Load loaders if applicable.

## edflow.data.believers.sequence module

### Summary

Classes:

<i>SequenceDataset</i>	Wraps around a dataset and returns sequences of examples.
<i>UnSequenceDataset</i>	Flattened version of a <i>SequenceDataset</i> .

Functions:

<code>getSeqDataset</code>	This allows to not define a dataset class, but use a base-class and a <i>length</i> and <i>step</i> parameter in the supplied <i>config</i> to load and sequentialize a dataset.
<code>get_sequence_view</code>	Generates a view on some base dataset given its sequence indices <code>seq_indices</code> .

## Reference

`edflow.data.believers.sequence.get_sequence_view(frame_ids, length, step=1, strategy='raise', base_step=1)`

Generates a view on some base dataset given its sequence indices `seq_indices`.

### Parameters

- **seq\_indices** (`np.ndarray`) – An array of *sorted* frame indices. Must be of type *int*.
- **length** (`int`) – Length of the returned sequences in frames.
- **step** (`int`) – Step between returned frames. Must be  $\geq 1$ .
- **strategy** (`str`) – How to handle bad sequences, i.e. sequences starting with a `fid_key`  $> 0$ . - `raise`: Raise a `ValueError` - `remove`: remove the sequence - `reset`: remove the sequence
- **base\_step** (`int`) – Step between base frames of returned sequences. Must be  $\geq 1$ .
- **view will have `len(dataset) - length * step` entries and `shape(This)` –**
- **`- length * step, length]` (`[len(dataset))` –**

**class** `edflow.data.believers.sequence.SequenceDataset(dataset, length, step=1, fid_key='fid', strategy='raise', base_step=1)`

Bases: `edflow.data.dataset_mixin.DatasetMixin`

Wraps around a dataset and returns sequences of examples. Given the length of those sequences the number of available examples is reduced by this length times the step taken. Additionally each example must have a frame id `fid_key` specified in the labels, by which it can be filtered. This is to ensure that each frame is taken from the same video.

This class assumes that examples come sequentially with `fid_key` and that frame id 0 exists.

The `SequenceDataset` also exposes the Attribute `self.base_indices`, which holds at each index *i* the indices of the elements contained in the example from the sequentialized dataset.

**\_\_init\_\_** (`dataset, length, step=1, fid_key='fid', strategy='raise', base_step=1`)

### Parameters

- **dataset** (`DatasetMixin`) – Dataset from which single frame examples are taken.
- **length** (`int`) – Length of the returned sequences in frames.
- **step** (`int`) – Step between returned frames. Must be  $\geq 1$ .
- **fid\_key** (`str`) – Key in labels, at which the frame indices can be found.
- **strategy** (`str`) – How to handle bad sequences, i.e. sequences starting with a `fid_key`  $> 0$ . - `raise`: Raise a `ValueError` - `remove`: remove the sequence - `reset`: remove the sequence

- **base\_step** (*int*) – Step between base frames of returned sequences. Must be  $\geq 1$ .
- **dataset will have  $\text{len}(\text{dataset}) - \text{length} * \text{step}$  examples.**  
(*This*) –

**class** edflow.data.believers.sequence.**UnSequenceDataset** (*seq\_dataset*)

Bases: *edflow.data.dataset\_mixin.DatasetMixin*

Flattened version of a *SequenceDataset*. Adds a new key *seq\_idx* to each example, corresponding to the sequence index and a key *example\_idx* corresponding to the original index. The ordering of the dataset is kept and sequence examples are ordered as in the sequence they are taken from.

**Warning:** This will not create the original non-sequence dataset! The new dataset contains  $\text{sequence-length} \times \text{len}(\text{SequenceDataset})$  examples.

If the original dataset would be represented as a 2d numpy array the UnSequence version of it would be the concatenation of all its rows:

```
a = np.arange(12)
seq_dataset = a.reshape([3, 4])
unseq_dataset = np.concatenate(seq_dataset, axis=-1)

np.all(a == unseq_dataset) # True
```

**\_\_init\_\_** (*seq\_dataset*)

**Parameters** *seq\_dataset* (*SequenceDataset*) – A *SequenceDataset* with attributes *length*.

**get\_example** (*i*)

Examples are gathered with the index  $i' = i // \text{seq\_len} + i \% \text{seq\_len}$

edflow.data.believers.sequence.**getSeqDataSet** (*config*)

This allows to not define a dataset class, but use a baseclass and a *length* and *step* parameter in the supplied *config* to load and sequentialize a dataset.

A config passed to edflow would look like this:

```
dataset: edflow.data.dataset.getSeqDataSet
model: Some Model
iterator: Some Iterator

seqdataset:
    dataset: import.path.to.your.basedataset
    length: 3
    step: 1
    fid_key: fid
    base_step: 1
```

*getSeqDataSet* will import the base dataset and pass it to *SequenceDataset* together with *length* and *step* to make the actually used dataset.

**Parameters** *config* (*dict*) –

An edflow config, with at least the keys *seqdataset* and nested inside it *dataset*, *seq\_length* and *seq\_step*.

**Returns** A Sequence Dataset based on the basedataset.

**Return type** *SequenceDataset*

## edflow.data.processing package

Submodules:

## edflow.data.processing.labels module

### Summary

Classes:

---

<i>ExtraLabelsDataset</i>	A dataset with extra labels added.
<i>LabelDataset</i>	A label only dataset to avoid loading unnecessary data.

---

### Reference

**class** edflow.data.processing.labels.**LabelDataset** (*data*)

Bases: *edflow.data.dataset\_mixin.DatasetMixin*

A label only dataset to avoid loading unnecessary data.

**\_\_init\_\_** (*data*)

**Parameters** **data** (*DatasetMixin*) – Some dataset where we are only interested in the labels.

**get\_example** (*i*)

Return only labels of example.

**class** edflow.data.processing.labels.**ExtraLabelsDataset** (*data, labeler*)

Bases: *edflow.data.dataset\_mixin.DatasetMixin*

A dataset with extra labels added.

**\_\_init\_\_** (*data, labeler*)

**Parameters**

- **data** (*DatasetMixin*) – Some Base dataset you want to add labels to
- **labeler** (*Callable*) – Must accept two arguments: a *Dataset* and an index *i* and return a dictionary of labels to add or overwrite. For all indices the keys in the returned *dict* must be the same and the type and shape of the values at those keys must be the same per key.

**property labels**

Add default behaviour for datasets defining an attribute *data*, which in turn is a dataset. This happens often when stacking several datasets on top of each other.

The default behaviour is to return `self.data.labels` if possible, and otherwise revert to the original behaviour.

## edflow.data.processing.processed module

### Summary

Classes:

<i>ProcessedDataset</i>	A dataset with data processing applied.
-------------------------	---

### Reference

**class** edflow.data.processing.processed.**ProcessedDataset** (*data*, *process*, *update=True*)

Bases: *edflow.data.dataset\_mixin.DatasetMixin*

A dataset with data processing applied.

**\_\_init\_\_** (*data*, *process*, *update=True*)

Applies *process* to the examples in *data* everytime an example is requested.

#### Parameters

- **data** (*DatasetMixin*) – The dataset to be processed.
- **process** (*Callable*) – A function which expects all entries in the examples of *data* as keyword arguments and returns a dictionary.

```
D = SomeDataset()
print(D[42]) # {'a': 1, 'b': 2, 'index_': 42, 'foo': 'bar'}

def process(a, b, **kwargs):
    return {'a': a+1, 'b': b**2}

PD = ProcessedDataset(D, process)
print(PD[42]) # {'a': 2, 'b': 4, 'index_': 42, 'foo', 'bar'}
```

- **update** (*bool*) – If *True* (which is default), takes the original example and does an update call on it with the dict returned by *process*. Otherwise simply returns the dict generated by *process*.

**get\_example** (*i*)

Get example and process.

## edflow.data.util package

Submodules:

## edflow.data.util.cached\_dset module

## Summary

Classes:

<i>CachedDataset</i>	Using a Dataset of single examples creates a cached (saved to memory) version, which can be accessed way faster at runtime.
<i>ExamplesFolder</i>	Contains all examples and labels of a cached dataset.
<i>PathCachedDataset</i>	Used for simplified decorator interface to dataset caching.

Functions:

<i>cachable</i>	Decorator to cache datasets.
<i>make_client_manager</i>	
<i>make_server_manager</i>	
<i>pickle_and_queue</i>	Parallelizable function to retrieve and queue examples from a Dataset.

## Reference

edflow.data.util.cached\_dset.**make\_server\_manager** (*port=63127, authkey=b'edcache'*)

edflow.data.util.cached\_dset.**make\_client\_manager** (*ip, port=63127, authkey=b'edcache'*)

edflow.data.util.cached\_dset.**pickle\_and\_queue** (*dataset\_factory, inqueue, outqueue, naming\_template='example\_{}.p'*)

Parallelizable function to retrieve and queue examples from a Dataset.

**Parameters**

- **dataset\_factory** (*chainer.DatasetMixin*) – A dataset factory, with methods described in *CachedDataset*.
- **indices** (*list*) – List of indices, used to retrieve samples from dataset.
- **queue** (*mp.Queue*) – Queue to put the samples in.
- **naming\_template** (*str*) – Formatable string, which defines the name of the stored file given its index.

**class** edflow.data.util.cached\_dset.**ExamplesFolder** (*root*)

Bases: object

Contains all examples and labels of a cached dataset.

**\_\_init\_\_** (*root*)

Initialize self. See help(type(self)) for accurate signature.

**read** (*name*)

**class** edflow.data.util.cached\_dset.**CachedDataset** (*dataset, force\_cache=False, keep\_existing=True, \_legacy=True, chunk\_size=64*)

Bases: `edflow.data.dataset_mixin.DatasetMixin`

Using a Dataset of single examples creates a cached (saved to memory) version, which can be accessed way faster at runtime.

To avoid creating the dataset multiple times, it is checked if the cached version already exists.

Calling `__getitem__` on this class will try to retrieve the samples from the cached dataset to reduce the preprocessing overhead.

The cached dataset will be stored in the root directory of the base dataset in the subfolder *cached* with name *name.zip*.

Besides the usual DatasetMixin interface, datasets to be cached must also implement

`root # (str) root folder to cache into`  
`name # (str) unique name`

Optionally but highly recommended, they should provide

`in_memory_keys # list(str) keys which will be collected from examples`

The collected values are stored in a dict of list, mapping an `in_memory_key` to a list containing the *i*-th value at the *i*-th place. This data structure is then exposed via the attribute *labels* and enables rapid iteration over useful labels without loading each example separately. That way, downstream datasets can filter the indices of the cached dataset efficiently, e.g. filtering based on train/eval splits.

Caching proceeds as follows: Expose a method which returns the dataset to be cached, e.g.

**def DataToCache():** `path = "/path/to/data"` return `MyCachableDataset(path)`

Start caching server on host `<server_ip_or_hostname>`:

`edcache -server -dataset import.path.to.DataToCache`

Wake up a worker bee on same or different hosts:

`edcache -address <server_ip_or_hostname> -dataset import.path.to.DataCache # noqa`

Start a cacherhive!

`__init__ (dataset, force_cache=False, keep_existing=True, _legacy=True, chunk_size=64)`

Given a dataset class, stores all examples in the dataset, if this has not yet happened.

#### Parameters

- **dataset** (*object*) – Dataset class which defines the following methods:
  - *root*: returns the path to the raw data
  - *name*: returns the name of the dataset -> best be unique
  - *\_\_len\_\_*: number of examples in the dataset
  - *\_\_getitem\_\_*: returns a single datum
  - *in\_memory\_keys*: returns all keys, that are stored

alongside the dataset, in a *labels.p* file. This allows to retrieve labels more quickly and can be used to filter the data more easily.
- **force\_cache** (*bool*) – If True the dataset is cached even if an existing, cached version is overwritten.
- **keep\_existing** (*bool*) – If True, existing entries in cache will not be recomputed and only non existing examples are appended to the cache. Useful if caching was interrupted.
- **\_legacy** (*bool*) – Read from the cached Zip file. Deprecated mode. Future Datasets should not write into zips as read times are very long.

- **chunksize** (*int*) – Length of the index list that is sent to the worker.

**classmethod from\_cache** (*root, name, \_legacy=True*)

Use this constructor to avoid initialization of original dataset which can be useful if only the cached zip file is available or to avoid expensive constructors of datasets.

**property fork\_safe\_zip**

**cache\_dataset** ()

Checks if a dataset is stored. If not iterates over all possible indices and stores the examples in a file, as well as the labels.

**property labels**

Returns the labels associated with the base dataset, but from the cached source.

**property root**

Returns the root to the base dataset.

**get\_example** (*i*)

Given an index *i*, returns a example.

**class** edflow.data.util.cached\_dset.**PathCachedDataset** (*dataset, path*)

Bases: *edflow.data.util.cached\_dset.CachedDataset*

Used for simplified decorator interface to dataset caching.

**\_\_init\_\_** (*dataset, path*)

Given a dataset class, stores all examples in the dataset, if this has not yet happened.

#### Parameters

- **dataset** (*object*) – Dataset class which defines the following methods:
  - *root*: returns the path to the raw data
  - *name*: returns the name of the dataset -> best be unique
  - *\_\_len\_\_*: number of examples in the dataset
  - *\_\_getitem\_\_*: returns a single datum
  - *in\_memory\_keys*: returns all keys, that are stored alongside the dataset, in a *labels.p* file. This allows to retrieve labels more quickly and can be used to filter the data more easily.
- **force\_cache** (*bool*) – If True the dataset is cached even if an existing, cached version is overwritten.
- **keep\_existing** (*bool*) – If True, existing entries in cache will not be recomputed and only non existing examples are appended to the cache. Useful if caching was interrupted.
- **\_legacy** (*bool*) – Read from the cached Zip file. Deprecated mode. Future Datasets should not write into zips as read times are very long.
- **chunksize** (*int*) – Length of the index list that is sent to the worker.

edflow.data.util.cached\_dset.**cachable** (*path*)

Decorator to cache datasets. If not cached, will start a caching server, subsequent calls will just load from cache. Currently all worker must be able to see the path. Be careful, function parameters are ignored on future calls. Can be used on any callable that returns a dataset. Currently the path should be the path to a zip file to cache into - i.e. it should end in zip.



**edflow.data.util.util\_dsets module****Summary**

Classes:

<i>DataFolder</i>	Given the root of a possibly nested folder containing datafiles and a Callable that generates the labels to the datafile from its full name, this class creates a labeled dataset.
<i>RandomlyJoinedDataset</i>	Load multiple examples which have the same label.

Functions:

<i>JoinedDataset</i>	Concat <code>n_joins</code> random samples based on the condition that <code>example_i[key] == example_j[key]</code> for all <code>i,j</code> .
<i>getDebugDataset</i>	Loads a dataset from the config and makes it reasonably small.

**Reference**

`edflow.data.util.util_dsets.JoinedDataset` (*dataset*, *key*, *n\_joins*)

Concat `n_joins` random samples based on the condition that `example_i[key] == example_j[key]` for all `i,j`. Key must be in labels of dataset.

`edflow.data.util.util_dsets.getDebugDataset` (*config*)

Loads a dataset from the config and makes it reasonably small. The config syntax works as in `getSeqDataset()`. See there for more extensive documentation.

**Parameters** *config* (*dict*) –

An edflow config, with at least the keys `debugdataset` and nested inside it `dataset`, `debug_length`, defining the basedataset and its size.

**Returns** A dataset based on the basedataset of the specified length.

**Return type** `SubDataset`

**class** `edflow.data.util.util_dsets.RandomlyJoinedDataset` (*config*)

Bases: `edflow.data.dataset_mixin.DatasetMixin`, `edflow.util.util.PRNGMixin`

Load multiple examples which have the same label.

**Required config parameters:**

**RandomlyJoinedDataset/dataset** The dataset from which to load examples.

**RandomlyJoinedDataset/key** The key of the label to join on.

**Optional config parameters:**

**test\_mode=False** If True, behaves deterministic.

**RandomlyJoinedDataset/n\_joins=2** How many examples to load.

**RandomlyJoinedDataset/balance=False** If True and not in `test_mode`, sample join labels uniformly.

**RandomlyJoinedDataset/avoid\_identity=True** If True and not in test\_mode, never return a pair containing the same image if possible.

**The i-th example returns:**

**‘examples’** A list of examples, where each example has the same label as specified by key. If data\_balancing is *False*, the first element of the list will be the *i*-th example of the dataset.

The dataset’s labels are the same as that of dataset. Be careful, *examples[j]* of the i-th example does not correspond to the i-th entry of the labels but to the *examples[j][“index\_”]*-th entry.

**\_\_init\_\_** (*config*)

Initialize self. See help(type(self)) for accurate signature.

**property labels**

Careful this can only give labels of the original item, not the joined ones. Use ‘examples[j][“index\_”]’ to get the correct label index.

**get\_example** (*i*)

---

**Note:** Please the documentation of DatasetMixin to not be confused.

---

Add default behaviour for datasets defining an attribute data, which in turn is a dataset. This happens often when stacking several datasets on top of each other.

The default behaviour now is to return `self.data.get_example(idx)` if possible, and otherwise revert to the original behaviour.

```
class edflow.data.util.util_dsets.DataFolder (image_root,      read_fn,      label_fn,
                                              sort_keys=None,  in_memory_keys=None,
                                              legacy=True, show_bar=False)
```

Bases: `edflow.data.dataset_mixin.DatasetMixin`

Given the root of a possibly nested folder containing datafiles and a Callable that generates the labels to the datafile from its full name, this class creates a labeled dataset.

A filtering of unwanted Data can be achieved by having the `label_fn` return `None` for those specific files. The actual files are only read when `__getitem__` is called.

If for example `label_fn` returns a dict with the keys ['a', 'b', 'c'] and `read_fn` returns one with keys ['d', 'e'] then the dict returned by `__getitem__` will contain the keys ['a', 'b', 'c', 'd', 'e', 'file\_path', 'index\_'].

```
__init__ (image_root, read_fn, label_fn, sort_keys=None, in_memory_keys=None, legacy=True,
          show_bar=False)
```

**Parameters**

- **image\_root** (*str*) – Root containing the files of interest.
- **read\_fn** (*Callable*) – Given the path to a file, returns the datum as a dict.
- **label\_fn** (*Callable*) – Given the path to a file, returns a dict of labels. If `label_fn` returns `None`, this file is ignored.
- **sort\_keys** (*list*) – A hierarchy of keys by which the data in this Dataset are sorted.
- **in\_memory\_keys** (*list*) – keys which will be collected from examples when the dataset is cached.
- **legacy** (*bool*) – Use the old read ethod, where only the path to the current file is passed to the reader. The new version will see all labels, that have been previously collected.

- **show\_bar** (*bool*) – Show a loading bar when loading labels.

**get\_example** (*i*)

Load the files specified in example *i*.

## Summary

## Reference

`edflow.data.util.flow2hsv` (*flow*)

Given a Flowmap of shape  $[W, H, 2]$  calculates an hsv image, showing the relative magnitude and direction of the optical flow.

**Parameters** **flow** (*np.array*) – Optical flow with shape  $[W, H, 2]$ .

**Returns** Containing the hsv data.

**Return type** `np.array`

`edflow.data.util.cart2polar` (*x, y*)

Takes two array as x and y coordinates and returns the magnitude and angle.

`edflow.data.util.hsv2rgb` (*hsv*)

color space conversion hsv -> rgb. simple wrapper for nice name.

`edflow.data.util.flow2rgb` (*flow*)

converts a flow field to an rgb color image.

**Parameters** **flow** (*np.array*) – optical flow with shape  $[W, H, 2]$ .

**Returns** Containing the rgb data. Color indicates orientation, intensity indicates magnitude.

**Return type** `np.array`

`edflow.data.util.get_support` (*image*)

... warning: This function makes a lot of assumptions that need not be met!

Assuming that there are three categories of images and that the `image_array` has been properly constructed, this function will estimate the support of the given `image`.

**Parameters** **image** (*np.ndarray*) – Some properly constructed image like array. No assumptions need to be made about the shape of the image, we simply assume each value is some color value.

**Returns** The support. Either '0->1', '-1->1' or '0->255'

**Return type** `str`

`edflow.data.util.sup_str_to_num` (*support\_str*)

Converts a support string into usable numbers.

`edflow.data.util.adjust_support` (*image, future\_support, current\_support=None, clip=False*)

Will adjust the support of all color values in `image`.

**Parameters**

- **image** (*np.ndarray*) – Array containing color values. Make sure this is properly constructed.
- **future\_support** (*str*) – The support this array is supposed to have after the transformation. Must be one of '-1->1', '0->1', or '0->255'.

- **current\_support** (*str*) – The support of the colors currentl in image. If not given it will be estimated by `get_support()`.
- **clip** (*bool*) – By default the return values in image are simply coming from a linear transform, thus the actual support might be larger than the requested interval. If set to `True` the returned array will be clipped to `future_support`.

**Returns** The given image with transformed support.

**Return type** same type as image

`edflow.data.util.clip_to_support (image, supp_str)`

`edflow.data.util.add_im_info (image, ax)`

Adds some interesting facts about the image.

`edflow.data.util.im_fn (key, im, ax)`

Plot an image. Used by `plot_datum()`.

`edflow.data.util.heatmap_fn (key, im, ax)`

Assumes that heatmap shape is [H, W, N]. Used by `plot_datum()`.

`edflow.data.util.keypoints_fn (key, keypoints, ax)`

Plots a list of keypoints as a dot plot.

`edflow.data.util.flow_fn (key, im, ax)`

Plot an flow. Used by `plot_datum()`.

`edflow.data.util.other_fn (key, obj, ax)`

Print some text about the object. Used by `plot_datum()`.

`edflow.data.util.default_heuristic (key, obj)`

Determines the kind of an object. Used by `plot_datum()`.

`edflow.data.util.plot_datum (nested_thing, savename='datum.png', heuristics=<function default_heuristic>, plt_functions={'flow': <function flow_fn>, 'heat': <function heatmap_fn>, 'image': <function im_fn>, 'keypoints': <function keypoints_fn>, 'other': <function other_fn>})`

Plots all data in the nested\_thing as best as can.

If heuristics is given, this determines how each leaf datum is converted to something plottable.

#### Parameters

- **nested\_thing** (*dict or list*) – Some nested object.
- **savename** (*str*) – Path/to/the/plot.png.
- **heuristics** (*Callable*) – If given this should produce a string specifying the kind of data of the leaf. If `None` determinde automatically. See `default_heuristic()`.
- **plt\_functions** (*dict of Callables*) – Maps a kind to a function which can plot it. Each callable must be able to receive a the key, the leaf object and the Axes to plot it in.

### 4.11.11 edflow.datasets package

Submodules:

#### edflow.datasets.celeba module

#### Summary

Classes:

---

*CelebA*

---

*CelebATest*

---

*CelebATrain*

---

*CelebAVal*

---

#### Reference

```
class edflow.datasets.celeba.CelebA (config=None)
    Bases: edflow.data.dataset_mixin.DatasetMixin

    NAME = 'CelebA'

    URL = 'http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html'

    FILES = ['img_align_celeba.zip', 'list_eval_partition.txt', 'identity_CelebA.txt', 'li

    __init__ (config=None)
        Initialize self. See help(type(self)) for accurate signature.

    get_example (i)
```

---

**Note:** Please the documentation of DatasetMixin to not be confused.

---

Add default behaviour for datasets defining an attribute data, which in turn is a dataset. This happens often when stacking several datasets on top of each other.

The default behaviour now is to return `self.data.get_example(idx)` if possible, and otherwise revert to the original behaviour.

```
class edflow.datasets.celeba.CelebATrain (config=None)
    Bases: edflow.datasets.celeba.CelebA

class edflow.datasets.celeba.CelebAVal (config=None)
    Bases: edflow.datasets.celeba.CelebA

class edflow.datasets.celeba.CelebATest (config=None)
    Bases: edflow.datasets.celeba.CelebA
```

## edflow.datasets.cifar module

### Summary

Classes:

---

*CIFAR10*

---

*CIFAR10Test*

---

*CIFAR10Train*

---

### Reference

```
class edflow.datasets.cifar.CIFAR10 (config=None)
    Bases: edflow.data.dataset_mixin.DatasetMixin

    NAME = 'CIFAR10'

    URL = 'https://www.cs.toronto.edu/~kriz/'

    FILES = {'DATA': 'cifar-10-python.tar.gz'}

    __init__ (config=None)
        Initialize self. See help(type(self)) for accurate signature.

    get_split ()

    get_example (i)
```

---

**Note:** Please the documentation of `DatasetMixin` to not be confused.

---

Add default behaviour for datasets defining an attribute `data`, which in turn is a dataset. This happens often when stacking several datasets on top of each other.

The default behaviour now is to return `self.data.get_example(idx)` if possible, and otherwise revert to the original behaviour.

```
class edflow.datasets.cifar.CIFAR10Train (config=None)
    Bases: edflow.datasets.cifar.CIFAR10

class edflow.datasets.cifar.CIFAR10Test (config=None)
    Bases: edflow.datasets.cifar.CIFAR10
```

## edflow.datasets.fashionmnist module

### Summary

Classes:

---

*FashionMNIST*

---

*FashionMNISTTest*

---

*FashionMNISTTrain*

---

Functions:

---

`read_mnist_file`


---

## Reference

`edflow.datasets.fashionmnist.read_mnist_file(path)`

**class** `edflow.datasets.fashionmnist.FashionMNIST` (*config=None*)

Bases: `edflow.data.dataset_mixin.DatasetMixin`

**NAME** = 'FashionMNIST'

**URL** = 'http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/'

**FILES** = {'TEST\_DATA': 't10k-images-idx3-ubyte.gz', 'TEST\_LABELS': 't10k-labels-idx1-ubyte.gz'}

**\_\_init\_\_** (*config=None*)

Initialize self. See help(type(self)) for accurate signature.

**get\_example** (*i*)

---

**Note:** Please the documentation of `DatasetMixin` to not be confused.

---

Add default behaviour for datasets defining an attribute `data`, which in turn is a dataset. This happens often when stacking several datasets on top of each other.

The default behaviour now is to return `self.data.get_example(idx)` if possible, and otherwise revert to the original behaviour.

**class** `edflow.datasets.fashionmnist.FashionMNISTTrain` (*config=None*)

Bases: `edflow.datasets.fashionmnist.FashionMNIST`

**class** `edflow.datasets.fashionmnist.FashionMNISTTest` (*config=None*)

Bases: `edflow.datasets.fashionmnist.FashionMNIST`

## edflow.datasets.mnist module

### Summary

Classes:

---

`MNIST`


---

`MNISTTest`


---

`MNISTTrain`


---

Functions:

---

`read_mnist_file`


---

## Reference

`edflow.datasets.mnist.read_mnist_file(path)`

**class** `edflow.datasets.mnist.MNIST` (*config=None*)

Bases: `edflow.data.dataset_mixin.DatasetMixin`

**NAME** = 'MNIST'

**URL** = 'https://storage.googleapis.com/cvdf-datasets/mnist/'

**FILES** = {'TEST\_DATA': 't10k-images-idx3-ubyte.gz', 'TEST\_LABELS': 't10k-labels-idx1-ubyte.gz'}

**\_\_init\_\_** (*config=None*)

Initialize self. See help(type(self)) for accurate signature.

**get\_example** (*i*)

---

**Note:** Please the documentation of `DatasetMixin` to not be confused.

---

Add default behaviour for datasets defining an attribute `data`, which in turn is a dataset. This happens often when stacking several datasets on top of each other.

The default behaviour now is to return `self.data.get_example(idx)` if possible, and otherwise revert to the original behaviour.

**class** `edflow.datasets.mnist.MNISTTrain` (*config=None*)

Bases: `edflow.datasets.mnist.MNIST`

**class** `edflow.datasets.mnist.MNISTTest` (*config=None*)

Bases: `edflow.datasets.mnist.MNIST`

## edflow.datasets.utils module

### Summary

Functions:

---

`download_url`

---

`download_urls`

---

`get_root`

---

`is_prepared`

---

`mark_prepared`

---

`prompt_download`

---

`quadratic_crop`

bbox is xmin, ymin, xmax, ymax

---

`reporhook`

tqdm progress bar for downloads.

---

`unpack`

---



## Reference

```
edflow.datasets.utils.unpack(path)
edflow.datasets.utils.reporthook(bar)
    tqdm progress bar for downloads.
edflow.datasets.utils.get_root(name)
edflow.datasets.utils.is_prepared(root)
edflow.datasets.utils.mark_prepared(root)
edflow.datasets.utils.prompt_download(file_, source, target_dir, content_dir=None)
edflow.datasets.utils.download_url(file_, url, target_dir)
edflow.datasets.utils.download_urls(urls, target_dir)
edflow.datasets.utils.quadratic_crop(x, bbox, alpha=1.0)
    bbox is xmin, ymin, xmax, ymax
```

### 4.11.12 edflow.edsetup\_files package

Submodules:

#### edflow.edsetup\_files.dataset module

## Summary

Classes:

---

*Dataset*

---

## Reference

```
class edflow.edsetup_files.dataset.Dataset(config)
    Bases: edflow.data.dataset_mixin.DatasetMixin, edflow.util.util.PRNGMixin
    __init__(config)
        A pure dataset initialisation with random inputs and labels.
        Args: config (dict): The config for the training.
    get_example(idx)
        Return a dictionary you're going to work with in the iterator.
        Parameters (int) (idx) –
        Returns example (dict)
        Return type These will be retrieved by their respective keys in the step_op method of the iterator.
```

## edflow.edsetup\_files.iterator module

### Summary

Classes:

---

*Iterator*

---

Clean iterator skeleton for initialization.

---

### Reference

**class** edflow.edsetup\_files.iterator.**Iterator**(\*args, \*\*kwargs)  
Bases: *edflow.iterators.template\_iterator.TemplateIterator*

Clean iterator skeleton for initialization.

**\_\_init\_\_**(\*args, \*\*kwargs)  
Constructor.

#### Parameters

- **model** (*object*) – Model class.
- **num\_epochs** (*int*) – Number of times to iterate over the data.
- **hooks** (*list*) – List containing Hook instances.
- **hook\_freq** (*int*) – Frequency at which hooks are evaluated.
- **bar\_position** (*int*) – Used by tqdm to place bars at the right position when using multiple Iterators in parallel.

**save** (*checkpoint\_path*)

Function for saving the model at a given state :param checkpoint\_path: :type checkpoint\_path: The path where the saved checkpoint should lie.

**restore** (*checkpoint\_path*)

Function for model restoration from a given checkpoint. :param checkpoint\_path: :type checkpoint\_path: The path where the checkpoint for restoring lies.

#### Returns

**Return type** The restored model from the given checkpoint.

**step\_op** (*model*, \*\*kwargs)

The main method to be called for training by the iterator. Calculating the loss, optimizer step etc. :param model: :type model: The given model class.

#### Returns

**Return type** A dictionary with *train\_op*, *log\_op* and *eval\_op* keys and their returns as their values.

## edflow.edsetup\_files.model module

### Summary

Classes:

<i>Model</i>	Clean model skeleton for initialization.
--------------	--

### Reference

```
class edflow.edsetup_files.model.Model (config)
    Bases: object

    Clean model skeleton for initialization.

    __init__ (config)
        Initialize self. See help(type(self)) for accurate signature.
```

## 4.11.13 edflow.eval package

Submodules:

### edflow.eval.pipeline module

To produce consistent results we adopt the following pipeline:

**Step 1:** Evaluate model on a test dataset and write out all data of interest:

- generated image
- latent representations

**Step 2:** Load the generated data in a Datafolder using the EvalDataset

**Step 3:** Pass both the test Dataset and the Datafolder to the evaluation scripts

Sometime in the future: **(Step 4):** Generate a report:

- latex tables
- paths to videos
- plots

### Usage

The pipeline is easily setup: In you Iterator (Trainer or Evaluator) add the EvalHook and as many callbacks as you like. You can also pass no callback at all.

**Warning:** To use the output with `edeval` you must set `config=config`.

```

from edflow.eval.pipeline import EvalHook

def my_callback(root, data_in, data_out, config):
    # Do somethin fancy with the data
    results = ...

    return results

class MyIterator(PyHookedModelIterator):

    def __init__(self, config, root, model, **kwargs):

        self.model = model

        self.hooks += [EvalHook(self.dataset,
                                callbacks={'cool_cb': my_callback},
                                config=config, # Must be specified for edeval
                                step_getter=self.get_global_step)]

    def eval_op(self, inputs):
        return {'generated': self.model(inputs)}

self.step_ops(self):
    return self.eval_op

```

Next you run your evaluation on your data using your favourite edflow command.

```
edflow -n myexperiment -e the_config.yaml -p path_to_project
```

This will create a new evaluation folder inside your project's eval directory. Inside this folder everything returned by your step ops is stored. In the case above this would mean your outputs would be stored as `generated:index.something`. But you don't need to concern yourself with that, as the outputs can now be loaded using the `EvalDataFolder`.

All you need to do is pass the `EvalDataFolder` the root folder in which the data has been saved, which is the folder where you can find the `model_outputs.csv`. Now you have all the generated data easily usable at hand. The indices of the data in the `EvalDataFolder` correspond to the indices of the data in the dataset, which was used to create the model outputs. So you can directly compare inputs, targets etc, with the outputs of your model!

If you specified a callback, this all happens automatically. Each callback receives at least 4 parameters: The `root`, where the data lives, the two datasets `data_in`, which was fed into the model and `data_out`, which was generated by the model, and the `config`. You can specify additional keyword arguments by defining them in the config under `eval_pipeline/callback_kwargs`.

Should you want to run evaluations on the generated data after it has been generated, you can run the `edeval` command while specifying the path to the model outputs csv and the callbacks you want to run.

```
edeval -c path/to/model_outputs.csv -cb name1:callback1 name2:callback2
```

The callbacks must be supplied using `name:callback` pairs. Names must be unique as `edeval` will construct a dictionary from these inputs.

If at some point you need to specify new parameters in your config or change existing ones, you can do so exactly like you would when running the `edflow` command. Simply pass the parameters you want to add/change via the commandline like this:

```
edeval -c path/to/model_outputs.csv -cb name1:callback1 --key1 val1 --key/path/2 val2
```

**Warning:** Changing config parameters from the commandline adds some dangers to the eval workflow: E.g. you can change parameters which determine the construction of the generating dataset, which potentially breaks the mapping between inputs and outputs.

## Summary

Classes:

<i>EvalHook</i>	Stores all outputs in a reusable fashion.
<i>TemplateEvalHook</i>	EvalHook that disables itself when the eval op returns None.

Functions:

<i>add_meta_data</i>	Prepends kwargs of interest to a csv file as comments (#)
<i>apply_callbacks</i>	Runs all given callbacks on the datasets <i>in_data</i> and <i>out_data</i> .
<i>cbargs2cbdict</i>	Turns a list of <code>name:callback</code> into a dict <code>{name:callback}</code>
<i>config2cbdict</i>	Extracts the callbacks inside a config and returns them as dict.
<i>decompose_name</i>	<b>param name</b>
<i>determine_loader</i>	Returns a loader name for a given file extension
<i>determine_saver</i>	Applies some heuristics to save an object.
<i>image_saver</i>	<b>param savepath</b>
<i>is_loadable</i>	<b>param filename</b>
<i>isimage</i>	<b>param np_arr</b>
<i>load_callbacks</i>	Loads all callbacks, i.e.
<i>main</i>	
<i>np_saver</i>	<b>param savepath</b>
<i>save_example</i>	Manages the writing process of a single datum: (1) Determine type, (2) Choose saver, (3) save.
<i>save_output</i>	Saves the output of some model contained in example in a reusable manner.
<i>standalone_eval_meta_dset</i>	Runs all given callbacks on the data in the EvalDataFolder constructed from the given csv.abs

## Reference

```
class edflow.eval.pipeline.EvalHook(datasets, sub_dir_keys=[], labels_key=None, callbacks={}, config=None, step_getter=None, keypath='step_ops', clean_after_callbacks=False)
```

Bases: `edflow.hooks.hook.Hook`

Stores all outputs in a reusable fashion.

```
__init__(datasets, sub_dir_keys=[], labels_key=None, callbacks={}, config=None, step_getter=None, keypath='step_ops', clean_after_callbacks=False)
```

**Warning:** To work with edeval you **must** specify `config=config` when instantiating the EvalHook.

### Parameters

- **datasets** (*dict* (*split*: *DatasetMixin*)) – The Datasets used for creating the new data.
- **sub\_dir\_keys** (*list* (*str*)) – Keys found in `example`, which will be used to make a subdirectory for the stored example. Subdirectories are made in a nested fashion in the order of the list. The keys will be removed from the example dict and not be stored explicitly.
- **labels\_key** (*str*) – All data behind the key found in the `example`s`, will be stored in large arrays and later loaded as labels. This should be small data types like `int` or str or small numpy arrays`.
- **callbacks** (*dict* (*name*: *str or Callable*)) – All callbacks are called at the end of the epoch. Must accept `root` as argument as well as the generating dataset and the generated dataset and a `config` (in that order). Additional keyword arguments found at `eval_pipeline/callback_kwargs` will also be passed to the callbacks. You can also leave this empty and supply import paths via `config`.
- **config** (*object, dict*) – An object containing metadata. Must be dumpable by `yaml`. Usually the `edflow config`. You can define callbacks here as well. These must be under the keypath `eval_pipeline/callbacks`. Also you can define additional keyword arguments passed to the callbacks as described in `callbacks`.
- **step\_getter** (*Callable*) – Function which returns the global step as `int`.
- **keypath** (*str*) – Path in result which will be stored.
- **clean\_after\_callbacks** (*bool*) – Remove model outputs after all callbacks finished.

**before\_epoch** (*epoch*)

Sets up the dataset for the current epoch.

**before\_step** (*step, fetches, feeds, batch*)

Get dataset indices from batch.

**after\_step** (*step, last\_results*)

Save examples and store label values.

**at\_exception** (*\*args, \*\*kwargs*)

Save all meta data. The already written data is not lost in any even if this fails.

**after\_epoch** (*epoch*)

Save meta data for reuse and then start the evaluation callbacks

**save\_meta** ()

**class** edflow.eval.pipeline.**TemplateEvalHook** (\*args, \*\*kwargs)

Bases: `edflow.eval.pipeline.EvalHook`

EvalHook that disables itself when the eval op returns None.

**\_\_init\_\_** (\*args, \*\*kwargs)

**Warning:** To work with `edeval` you **must** specify `config=config` when instantiating the EvalHook.

### Parameters

- **datasets** (*dict (split: DatasetMixin)*) – The Datasets used for creating the new data.
- **sub\_dir\_keys** (*list (str)*) – Keys found in `example`, which will be used to make a subdirectory for the stored example. Subdirectories are made in a nested fashion in the order of the list. The keys will be removed from the example dict and not be stored explicitly.
- **labels\_key** (*str*) – All data behind the key found in the `example`s`, will be stored in large arrays and later loaded as labels. This should be small data types like `int`` or `str` or small numpy arrays.
- **callbacks** (*dict (name: str or Callable)*) – All callbacks are called at the end of the epoch. Must accept `root` as argument as well as the generating dataset and the generated dataset and a `config` (in that order). Additional keyword arguments found at `eval_pipeline/callback_kwargs` will also be passed to the callbacks. You can also leave this empty and supply import paths via `config`.
- **config** (*object, dict*) – An object containing metadata. Must be dumpable by `yaml`. Usually the `edflow config`. You can define callbacks here as well. These must be under the keypath `eval_pipeline/callbacks`. Also you can define additional keyword arguments passed to the callbacks as described in `callbacks`.
- **step\_getter** (*Callable*) – Function which returns the global step as `int`.
- **keypath** (*str*) – Path in result which will be stored.
- **clean\_after\_callbacks** (*bool*) – Remove model outputs after all callbacks finished.

**before\_epoch** (\*args, \*\*kwargs)

Sets up the dataset for the current epoch.

**before\_step** (\*args, \*\*kwargs)

Get dataset indices from batch.

**after\_step** (*step, last\_results*)

Save examples and store label values.

**after\_epoch** (\*args, \*\*kwargs)

Save meta data for reuse and then start the evaluation callbacks

**at\_exception** (\*args, \*\*kwargs)

Save all meta data. The already written data is not lost in any even if this fails.

`edflow.eval.pipeline.save_output (root, example, index, sub_dir_keys=[], keypath='step_ops')`

Saves the output of some model contained in `example` in a reusable manner.

#### Parameters

- **root** (*str*) – Storage directory
- **example** (*dict*) – name: datum pairs of outputs.
- **index** (*list (int)*) – dataset index corresponding to example.
- **sub\_dir\_keys** (*list (str)*) – Keys found in `example`, which will be used to make a subdirectory for the stored example. Subdirectories are made in a nested fashion in the order of the list. The keys will be removed from the example dict and not be stored. Directories are name `key:val` to be able to completely recover the keys. (Default value = [])

#### Returns

**path\_dics** – Name: path pairs of the saved outputs.

**Warning:** Make sure the values behind the `sub_dir_keys` are compatible with the file system you are saving data on.

#### Return type dict

`edflow.eval.pipeline.add_meta_data (eval_root, metadata, description=None)`

Prepends kwargs of interest to a csv file as comments (#)

#### Parameters

- **eval\_root** (*str*) – Where the `meta.yaml` will be written.
- **metadata** (*dict*) – config like object, which will be written in the `meta.yaml`.
- **description** (*str*) – Optional description string. Will be added unformatted as yaml multiline literal.

**Returns meta\_path** – Full path of the `meta.yaml`.

#### Return type str

`edflow.eval.pipeline.save_example (savepath, datum)`

Manages the writing process of a single datum: (1) Determine type, (2) Choose saver, (3) save.

#### Parameters

- **savepath** (*str*) – Where to save. Must end with `.{}` to put in the file ending via `.format()`.
- **datum** (*object*) – Some python object to save.

#### Returns

- **savepath** (*str*) – Where the example has been saved. This string has been formatted and can be used to load the file at the described location.
- **loader\_name** (*str*) – The name of a loader, which can be passed to the `meta.yaml` 's `loaders` entry.

`edflow.eval.pipeline.determine_saver (py_obj)`

Applies some heuristics to save an object.

**Parameters py\_obj** (*object*) – Some python object to be saved.



**Raises `NotImplementedError`** – If `py_obj` is of unrecognized type. Feel free to implement your own savers and publish them to edflow.

`edflow.eval.pipeline.determine_loader(ext)`

Returns a loader name for a given file extension

**Parameters** `ext (str)` – File ending excluding the `..`. Same as what would be returned by `os.path.splitext()`

**Returns** `name` – Name of the meta loader (see `meta_loaders`).

**Return type** `str`

**Raises `ValueError`** – If the file extension cannot be handled by the implemented loaders. Feel free to implement you own and publish them to edflow.

`edflow.eval.pipeline.decompose_name(name)`

**Parameters** `name` –

`edflow.eval.pipeline.is_loadable(filename)`

**Parameters** `filename` –

`edflow.eval.pipeline.isimage(np_arr)`

**Parameters** `np_arr` –

`edflow.eval.pipeline.image_saver(savepath, image)`

**Parameters**

- `savepath` –
- `image` –

`edflow.eval.pipeline.np_saver(savepath, np_arr)`

**Parameters**

- `savepath` –
- `np_arr` –

`edflow.eval.pipeline.standalone_eval_meta_dset(path_to_meta_dir, callbacks, additional_kwargs={}, other_config=None)`

Runs all given callbacks on the data in the `EvalDataFolder` constructed from the given `csv.abs`

**Parameters**

- `path_to_csv (str)` – Path to the csv file.
- `callbacks (dict(name: str or Callable))` – Import commands used to construct the functions applied to the Data extracted from `path_to_csv`.
- `additional_kwargs (dict)` – Keypath-value pairs added to the config, which is extracted from the `model_outputs.csv`. These will overwrite parameters in the original config extracted from the csv.
- `other_config (str)` – Path to additional config used to update the existing one as taken from the `model_outputs.csv`. Cannot overwrite the dataset. Only used for callbacks. Parameters in this other config will overwrite the parameters in the original config and those of the commandline arguments.

**Returns** `outputs` – The collected outputs of the callbacks.

**Return type** `dict`

`edflow.eval.pipeline.load_callbacks(callbacks)`

Loads all callbacks, i.e. if the callback is given as str, will load the module behind the import path, otherwise will do nothing.

`edflow.eval.pipeline.apply_callbacks(callbacks, root, in_data, out_data, config, callback_kwargs={})`

Runs all given callbacks on the datasets `in_data` and `out_data`.

#### Parameters

- **callbacks** (*dict* (*name*: *Callable*)) – List of all callbacks to apply. All callbacks must accept at least the signature `callback(root, data_in, data_out, config)`. If supplied via the config, additional keyword arguments are passed to the callback. These are expected under the keypath `eval_pipeline/callback_kwargs`.
- **in\_data** (*DatasetMixin*) – Dataset used to generate the content in `out_data`.
- **out\_data** (*DatasetMixin*) – Generated data. Example `i` is expected to be generated using `in_data[i]`.
- **config** (*dict*) – edflow config dictionary.
- **callback\_kwargs** (*dict*) – Keyword Arguments for the callbacks.

**Returns** `outputs` – All results generated by the callbacks at the corresponding key.

**Return type** `dict`(*name*: callback output)

`edflow.eval.pipeline.cbargs2cbdict(arglist)`

Turns a list of `name:callback` into a `dict {name: callback}`

`edflow.eval.pipeline.config2cbdict(config)`

Extracts the callbacks inside a config and returns them as dict. Callbacks must be defined at `eval_pipeline/callback_kwargs`.

**Parameters** **config** (*dict*) – A config dictionary.

**Returns** **callbacks** – All `name:callback` pairs as `dict {name: callback}`

**Return type** `dict`

`edflow.eval.pipeline.main()`

## 4.11.14 edflow.hooks package

Submodules:

**edflow.hooks.hook module**

### Summary

Classes:

---

*Hook*

Base Hook to be inherited from.

---

## Reference

**class** edflow.hooks.hook.Hook

Bases: object

Base Hook to be inherited from. Hooks can be passed to `HookedModelIterator` and will be called at fixed intervals.

The inheriting class only needs to overwrite those methods below, which are of interest.

In principle a hook can be used to do anything during its execution. It is intended to be used as an update mechanism for the standard fetches and feeds, passed to the session managed e.g. by a `HookedModelIterator` and then working with the results of the run call to the session.

Assuming there is one hook that is passed to a `HookedModelIterator` its methods will be called in the following fashion:

```
for epoch in epochs:
    hook.before_epoch(epoch)
    for i, batch in enumerate(batches):
        fetches, feeds = some_function(batch)
        hook.before_step(i, fetches, feeds) # change fetches & feeds

        results = session.run(fetches, feed_dict=feeds)

        hook.after_step(i, results)
    hook.after_epoch(epoch)
```

**before\_epoch** (*epoch*)

Called before each epoch.

**Parameters** *epoch* (*int*) – Index of epoch that just started.

**before\_step** (*step*, *fetches*, *feeds*, *batch*)

Called before each step. Can update any feeds and fetches.

**Parameters**

- **step** (*int*) – Current training step.
- **fetches** (*list or dict*) – Fetches for the next session.run call.
- **feeds** (*dict*) – Data used at this step.
- **batch** (*list or dict*) – All data available at this step.

**after\_step** (*step*, *last\_results*)

Called after each step.

**Parameters**

- **step** (*int*) – Current training step.
- **last\_results** (*list*) – Results from last time this hook was called.

**after\_epoch** (*epoch*)

Called after each epoch.

**Parameters** *epoch* (*int*) – Index of epoch that just ended.

**at\_exception** (*exception*)

Called when an exception is raised.

**Parameters** *exception* –

**Raises**

- **be** – raised again after all
- **been** – handled

**edflow.hooks.pytorch\_hooks module****Summary**

Classes:

<i>DataPrepHook</i>	The hook is needed in order to convert the input appropriately.
<i>PyCheckpointHook</i>	Does that checkpoint thingy where it stores everything in a checkpoint.
<i>PyLoggingHook</i>	Supply and evaluate logging ops at an intervall of training steps.
<i>ToFromTorchHook</i>	
<i>ToNumpyHook</i>	Converts all pytorch Variables and Tensors in the results to numpy arrays and leaves the rest as is.
<i>ToTorchHook</i>	Converts all numpy arrays in the batch to torch.Tensor arrays and leaves the rest as is.

**Reference**

**class** edflow.hooks.pytorch\_hooks.**PyCheckpointHook** (*root\_path*, *model*, *model-name*='model', *interval*=None)

Bases: *edflow.hooks.hook.Hook*

Does that checkpoint thingy where it stores everything in a checkpoint.

**\_\_init\_\_** (*root\_path*, *model*, *modelname*='model', *interval*=None)

**Parameters**

- **root\_path** (*str*) – Path to where the checkpoints are stored.
- **model** (*nn.Module*) – Model to checkpoint.
- **modelname** (*str*) – Prefix for checkpoint files.
- **interval** (*int*) – Number of iterations after which a checkpoint is saved. In any case a checkpoint is savead after each epoch.

**before\_epoch** (*epoch*)

Called before each epoch.

**Parameters** **epoch** (*int*) – Index of epoch that just started.

**after\_epoch** (*epoch*)

Called after each epoch.

**Parameters** **epoch** (*int*) – Index of epoch that just ended.

**after\_step** (*step*, *last\_results*)

Called after each step.

**Parameters**

- **step** (*int*) – Current training step.
- **last\_results** (*list*) – Results from last time this hook was called.

**at\_exception** (*\*args, \*\*kwargs*)  
Called when an exception is raised.

**Parameters exception** –

**Raises**

- **be** – raised again after all
- **been** – handled

**save** ()

```
class edflow.hooks.pytorch_hooks.PyLoggingHook (log_ops=[], scalar_keys=[], histogram_keys=[], image_keys=[], log_keys=[], graph=None, interval=100, root_path='logs')
```

Bases: `edflow.hooks.hook.Hook`

Supply and evaluate logging ops at an intervall of training steps.

```
__init__ (log_ops=[], scalar_keys=[], histogram_keys=[], image_keys=[], log_keys=[], graph=None, interval=100, root_path='logs')
```

**Parameters**

- **log\_ops** (*list*) – Ops to run at logging time.
- **scalars** (*dict*) – Scalar ops.
- **histograms** (*dict*) – Histogram ops.
- **images** (*dict*) – Image ops. Note that for these no tensorboard logging ist used but a custom image saver.
- **logs** (*dict*) – Logs to std out via logger.
- **graph** (*tf.Graph*) – Current graph.
- **interval** (*int*) – Intervall of training steps before logging.
- **root\_path** (*str*) – Path at which the logs are stored.

**before\_step** (*batch\_index, fetches, feeds, batch*)  
Called before each step. Can update any feeds and fetches.

**Parameters**

- **step** (*int*) – Current training step.
- **fetches** (*list or dict*) – Fetches for the next session.run call.
- **feeds** (*dict*) – Data used at this step.
- **batch** (*list or dict*) – All data available at this step.

**after\_step** (*batch\_index, last\_results*)  
Called after each step.

**Parameters**

- **step** (*int*) – Current training step.
- **last\_results** (*list*) – Results from last time this hook was called.

```
class edflow.hooks.pytorch_hooks.ToNumpyHook
```

Bases: `edflow.hooks.hook.Hook`

Converts all pytorch Variables and Tensors in the results to numpy arrays and leaves the rest as is.

**after\_step** (*step, results*)

Called after each step.

**Parameters**

- **step** (*int*) – Current training step.
- **last\_results** (*list*) – Results from last time this hook was called.

```
class edflow.hooks.pytorch_hooks.ToTorchHook (push_to_gpu=True,  
                                              dtype=<Mock          name='mock.float'  
                                              id='140169497937184'>)
```

Bases: `edflow.hooks.hook.Hook`

Converts all numpy arrays in the batch to torch.Tensor arrays and leaves the rest as is.

**\_\_init\_\_** (*push\_to\_gpu=True, dtype=<Mock name='mock.float' id='140169497937184'>*)

Initialize self. See help(type(self)) for accurate signature.

**before\_step** (*step, fetches, feeds, batch*)

Called before each step. Can update any feeds and fetches.

**Parameters**

- **step** (*int*) – Current training step.
- **fetches** (*list or dict*) – Fetches for the next session.run call.
- **feeds** (*dict*) – Data used at this step.
- **batch** (*list or dict*) – All data available at this step.

```
class edflow.hooks.pytorch_hooks.ToFromTorchHook (*args, **kwargs)  
Bases: edflow.hooks.pytorch_hooks.ToNumpyHook, edflow.hooks.pytorch_hooks.  
ToTorchHook
```

**\_\_init\_\_** (*\*args, \*\*kwargs*)

Initialize self. See help(type(self)) for accurate signature.

```
class edflow.hooks.pytorch_hooks.DataPrepHook (*args, **kwargs)
```

Bases: `edflow.hooks.pytorch_hooks.ToFromTorchHook`

The hook is needed in order to convert the input appropriately. Here, we have to reshape the input i.e. append 1 to the shape (for the number of channels of the image). Plus, it converts to data to Pytorch tensors, and back.

**before\_step** (*step, fetches, feeds, batch*)

Steps taken before the training step. :param step: Training step. :param fetches: Fetches for the next session.run call. :param feeds: Feeds for the next session.run call. :param batch: The batch to be iterated over.

**after\_step** (*step, results*)

Steps taken after the training step. :param step: Training step. :param results: Result of the session.

## edflow.hooks.runtime\_input module

### Summary

Classes:

<i>RuntimeInputHook</i>	Given a textfile reads that at each step and passes the results to a callback function.
-------------------------	---

### Reference

**class** edflow.hooks.runtime\_input.**RuntimeInputHook** (*update\_file, callback*)

Bases: *edflow.hooks.hook.Hook*

Given a textfile reads that at each step and passes the results to a callback function.

**\_\_init\_\_** (*update\_file, callback*)

#### Parameters

- **update\_file** (*str*) – path/to/yaml-file containing the parameters of interest.
- **callback** (*Callable*) – Each time something changes in the update\_file this function is called with the content of the file as argument.

**before\_step** (*\*args, \*\*kwargs*)

Checks if something changed and if yes runs the callback.

## edflow.hooks.util\_hooks module

### Summary

Classes:

<i>ExpandHook</i>	Retrieve paths.
<i>IntervalHook</i>	This hook manages a set of hooks, which it will run each time its interval flag is set to True.

### Reference

**class** edflow.hooks.util\_hooks.**ExpandHook** (*paths, interval, default=None*)

Bases: *edflow.hooks.hook.Hook*

Retrieve paths.

**\_\_init\_\_** (*paths, interval, default=None*)

#### Parameters

- **paths** (*list of keypaths to expand.*) –
- **interval** (*int*) – The interval in which expansion is performed.

**after\_step** (*step, last\_results*)

Called after each step.

```
class edflow.hooks.util_hooks.IntervalHook(hooks, interval, start=None, stop=None,
                                             modify_each=None, modifier=<function IntervalHook.<lambda>>, max_interval=None,
                                             get_step=None)
```

Bases: `edflow.hooks.hook.Hook`

This hook manages a set of hooks, which it will run each time its interval flag is set to True.

```
__init__(hooks, interval, start=None, stop=None, modify_each=None, modifier=<function IntervalHook.<lambda>>, max_interval=None, get_step=None)
```

#### Parameters

- **hook** (*list of Hook*) – The set of managed hooks. Each must implement the methods of a Hook.
- **interval** (*int*) – The number of steps after which the managed hooks are run.
- **start** (*int*) – If *start* is not None, the first time the hooks are run is after *start* number of steps have been made.
- **stop** (*int*) – If given, this hook is not evaluated anymore after *stop* steps.
- **modify\_each** (*int*) – If given, *modifier* is called on the interval after this many executions of this hook. If *None* it is set to *interval*. In case you do not want any modification you can either set *max\_interval* to *interval* or choose the modifier to be *lambda x: x* or set *modify\_each* to *float: inf*.
- **modifier** (*Callable*) – See *modify\_each*.
- **max\_interval** (*int*) – If given, the modifier can only increase the interval up to this number of steps.
- **get\_step** (*Callable*) – If given, prefer over the use of batch index to determine run condition, e.g. to run based on global step.

```
run_condition(step, is_before=False)
```

```
maybe_modify(step)
```

```
before_epoch(*args, **kwargs)
    Called before each epoch.
```

```
before_step(step, *args, **kwargs)
    Called before each step. Can update any feeds and fetches.
```

```
after_step(step, *args, **kwargs)
    Called after each step.
```

```
after_epoch(*args, **kwargs)
    Called after each epoch.
```

Subpackages:



## edflow.hooks.checkpoint\_hooks package

Submodules:

### edflow.hooks.checkpoint\_hooks.common module

## Summary

Classes:

<i>CollectorHook</i>	Collects data.
<i>KeepBestCheckpoints</i>	Tries to find a metric for all checkpoints and keeps the n_keep best checkpoints and the latest checkpoint.
<i>MetricTuple</i>	
<i>StoreArraysHook</i>	Collects lots of data, stacks them and then stores them.
<i>WaitForCheckpointHook</i>	Waits until a new checkpoint is created, then lets the Iterator continue.

Functions:

<i>dict_repr</i>	Makes a nice representation of a nested dict.
<i>get_checkpoint_files</i>	Return {global_step: [files,...]}.
<i>get_latest_checkpoint</i>	Return path to name of latest checkpoint in checkpoint_root dir.
<i>make_iterator</i>	Make an iterator that yields key value pairs.
<i>strenumerate</i>	Same as enumerate, but yields str(index).
<i>test_valid_metricktuple</i>	Checks if all inputs are correct.
<i>tf_parse_global_step</i>	
<i>torch_parse_global_step</i>	

## Reference

`edflow.hooks.checkpoint_hooks.common.get_latest_checkpoint` (*checkpoint\_root*, *filter\_cond*=<function <lambda>>)

Return path to name of latest checkpoint in checkpoint\_root dir.

### Parameters

- **checkpoint\_root** (*str*) – Path to where the checkpoints live.
- **filter\_cond** (*Callable*) – A function used to filter files, to only get the checkpoints that are wanted.

**Returns** path of the latest checkpoint. Note that for tensorflow checkpoints this is not an existing file, but path{.index,.meta,data\*} should be

**Return type** str

```
class edflow.hooks.checkpoint_hooks.common.WaitForCheckpointHook (checkpoint_root,
                                                                    fil-
                                                                    ter_cond=<function
                                                                    Wait-
                                                                    ForCheck-
                                                                    pointHook.<lambda>>,
                                                                    interval=5,
                                                                    add_sec=5,
                                                                    call-
                                                                    back=None,
                                                                    eval_all=False)
```

Bases: `edflow.hooks.hook.Hook`

Waits until a new checkpoint is created, then lets the Iterator continue.

```
__init__ (checkpoint_root, filter_cond=<function WaitForCheckpointHook.<lambda>>, interval=5,
          add_sec=5, callback=None, eval_all=False)
```

#### Parameters

- **checkpoint\_root** (*str*) – Path to look for checkpoints.
- **filter\_cond** (*Callable*) – A function used to filter files, to only get the checkpoints that are wanted.
- **interval** (*float*) – Number of seconds after which to check for a new checkpoint again.
- **add\_sec** (*float*) – Number of seconds to wait, after a checkpoint is found, to avoid race conditions, if the checkpoint is still being written at the time it's meant to be read.
- **callback** (*Callable*) – Callback called with path of found checkpoint.
- **eval\_all** (*bool*) – Accept all instead of just latest checkpoint.

**fcond** (*c*)

**look** ()

Loop until a new checkpoint is found.

**before\_epoch** (*ep*)

Called before each epoch.

**Parameters epoch** (*int*) – Index of epoch that just started.

`edflow.hooks.checkpoint_hooks.common.strenumerate` (*\*args, \*\*kwargs*)

Same as `enumerate`, but yields `str(index)`.

`edflow.hooks.checkpoint_hooks.common.make_iterator` (*list\_or\_dict*)

Make an iterator that yields key value pairs.

`edflow.hooks.checkpoint_hooks.common.dict_repr` (*some\_dict, pre="", level=0*)

Makes a nice representation of a nested dict.

**class** `edflow.hooks.checkpoint_hooks.common.CollectorHook`

Bases: `edflow.hooks.hook.Hook`

Collects data. Supposed to be used as base class.

**\_\_init\_\_** ()

Initialize self. See `help(type(self))` for accurate signature.

**after\_step** (*step, results*)

Called after each step.

**Parameters**

- **step** (*int*) – Current training step.
- **last\_results** (*list*) – Results from last time this hook was called.

**stack\_results** (*new\_data, all\_data*)

Given the current collected data append the new results along the batch dimension.

**Parameters**

- **new\_data** (*list or dict*) – data to append.
- **all\_data** (*list or dict*) – data to append to.

**class** edflow.hooks.checkpoint\_hooks.common.**StoreArraysHook** (*save\_root*)

Bases: `edflow.hooks.checkpoint_hooks.common.CollectorHook`

Collects lots of data, stacks them and then stores them.

**\_\_init\_\_** (*save\_root*)

Collect all outputs of step op and store them as npz.

**after\_epoch** (*epoch*)

Called after each epoch.

**Parameters** **epoch** (*int*) – Index of epoch that just ended.

**flatten\_results** (*results, prefix, store\_dict*)

Recursively walk over the results dictionary and stack the data.

**Parameters**

- **results** (*dict or list*) – Containing results.
- **prefix** (*str*) – Prepend to name when storing.
- **store\_dict** (*dict*) – Flat storage dictionary.

**class** edflow.hooks.checkpoint\_hooks.common.**MetricTuple** (*input\_names, output\_names, metric, name*)

Bases: `tuple`

**input\_names**

Alias for field number 0

**metric**

Alias for field number 2

**name**

Alias for field number 3

**output\_names**

Alias for field number 1

edflow.hooks.checkpoint\_hooks.common.**test\_valid\_metrixtuple** (*metric\_tuple*)

Checks if all inputs are correct.

edflow.hooks.checkpoint\_hooks.common.**torch\_parse\_global\_step** (*checkpoint*)

edflow.hooks.checkpoint\_hooks.common.**tf\_parse\_global\_step** (*checkpoint*)

edflow.hooks.checkpoint\_hooks.common.**get\_checkpoint\_files** (*checkpoint\_root*)

Return {global\_step: [files,...]}.

**Parameters** **checkpoint\_root** (*str*) – Path to where the checkpoints live.

```
class edflow.hooks.checkpoint_hooks.common.KeepBestCheckpoints (checkpoint_root,  
metric_template,  
metric_key,  
n_keep=5,  
lower_is_better=True)
```

Bases: *edflow.hooks.hook.Hook*

Tries to find a metric for all checkpoints and keeps the `n_keep` best checkpoints and the latest checkpoint.

```
__init__ (checkpoint_root, metric_template, metric_key, n_keep=5, lower_is_better=True)
```

#### Parameters

- **checkpoint\_root** (*str*) – Path to look for checkpoints.
- **metric\_template** (*str*) – Format string to find metric file.
- **metric\_key** (*str*) – Key to use from metric file.
- **n\_keep** (*int*) – Maximum number of checkpoints to keep.

```
get_loss (step)
```

```
after_epoch (ep)
```

Called after each epoch.

**Parameters** **epoch** (*int*) – Index of epoch that just ended.

## edflow.hooks.checkpoint\_hooks.lambda\_checkpoint\_hook module

### Summary

Classes:

---

*LambdaCheckpointHook*

---

### Reference

```
class edflow.hooks.checkpoint_hooks.lambda_checkpoint_hook.LambdaCheckpointHook (root_path,  
global_step_getter,  
global_step_setter,  
save,  
restore,  
interval=None,  
ckpt_zero=False,  
modelname='model')
```

Bases: *edflow.hooks.hook.Hook*

```
__init__ (root_path, global_step_getter, global_step_setter, save, restore, interval=None,  
ckpt_zero=False, modelname='model')
```

```
before_epoch (epoch)
```

Parameters `epoch` –

`after_epoch` (*epoch*)

Parameters `epoch` –

`after_step` (*step*, *last\_results*)

Parameters

- `step` –

- `last_results` –

`at_exception` (*\*args*, *\*\*kwargs*)

Parameters

- `*args` –

- `**kwargs` –

`save` (*force\_active=False*)

`static parse_global_step` (*checkpoint*)

Parameters `checkpoint` –

## edflow.hooks.checkpoint\_hooks.tf\_checkpoint\_hook module

### Summary

Classes:

<i>CheckpointHook</i>	Does that checkpoint thingy where it stores everything in a checkpoint.
<i>RestoreCurrentCheckpointHook</i>	Restores a TensorFlow model from a checkpoint at each epoch.
<i>RestoreModelHook</i>	Restores a TensorFlow model from a checkpoint at each epoch.
<i>RestoreTFModelHook</i>	alias of <code>edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.RestoreModelHook</code>
<i>RetrainHook</i>	Restes the global step at the beginning of training.
<i>WaitForManager</i>	Wait to make sure checkpoints are not overflowing.

## Reference

```
class edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.RestoreModelHook(variables,
                                                                    check-
                                                                    point_path,
                                                                    fil-
                                                                    ter_cond=<function
                                                                    Re-
                                                                    store-
                                                                    Mod-
                                                                    el-
                                                                    Hook.<lambda>>,
                                                                    global_step_setter=None)
```

Bases: `edflow.hooks.hook.Hook`

Restores a TensorFlow model from a checkpoint at each epoch. Can also be used as a functor.

```
__init__(variables, checkpoint_path, filter_cond=<function RestoreModelHook.<lambda>>,
          global_step_setter=None)
```

### Parameters

- **variables** (*list*) – tf.Variable to be loaded from the checkpoint.
- **checkpoint\_path** (*str*) – Directory in which the checkpoints are stored or explicit checkpoint. Ignored if used as functor.
- **filter\_cond** (*Callable*) – A function used to filter files, to only get the checkpoints that are wanted. Ignored if used as functor.
- **global\_step\_setter** (*Callable*) – Callback to set global\_step.

**property session**

**before\_epoch** (*ep*)

Parameters **ep** –

**static parse\_global\_step** (*checkpoint*)

Parameters **checkpoint** –

```
edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.RestoreTFModelHook
    alias of edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.RestoreModelHook
```

```
class edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.CheckpointHook(root_path,
                                                                    vari-
                                                                    ables,
                                                                    mod-
                                                                    el-
                                                                    name='model',
                                                                    ses-
                                                                    sion=None,
                                                                    step=None,
                                                                    inter-
                                                                    val=None,
                                                                    max_to_keep=5)
```

Bases: `edflow.hooks.hook.Hook`

Does that checkpoint thingy where it stores everything in a checkpoint.

```
__init__(root_path, variables, modelname='model', session=None, step=None, interval=None,
        max_to_keep=5)
```

**Parameters**

- **root\_path** (*str*) – Path to where the checkpoints are stored.
- **variables** (*list*) – List of all variables to keep track of.
- **session** (*tf.Session*) – Session instance for saver.
- **modelname** (*str*) – Used to name the checkpoint.
- **step** (*tf.Tensor or callable*) – Step op, that can be evaluated: i.e. a *tf.Tensor* or a python callable returning the step as an integer).
- **interval** (*int*) – Number of iterations after which a checkpoint is saved. If None, a checkpoint is saved after each epoch.
- **max\_to\_keep** (*int*) – Maximum number of checkpoints to keep on disk. Use 0 or None to never delete any checkpoints.

```
before_epoch (ep)
```

**Parameters** *ep* –

```
after_epoch (epoch)
```

**Parameters** *epoch* –

```
after_step (step, last_results)
```

**Parameters**

- **step** –
- **last\_results** –

```
at_exception (*args, **kwargs)
```

**Parameters**

- **\*args** –
- **\*\*kwargs** –

```
save ()
```

```
global_step ()
```

```
class edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.RetrainHook (global_step=None)
```

Bases: [edflow.hooks.hook.Hook](#)

Restes the global step at the beginning of training.

```
__init__ (global_step=None)
```

**Parameters** **global\_step** (*tf.Variable*) – Variable tracking the training step.

```
before_epoch (epoch)
```

**Parameters** *epoch* –

```
before_step (batch_index, fetches, feeds, batch)
```

**Parameters**

- **batch\_index** –
- **fetches** –

- **feeds** –
- **batch** –

**after\_step** (*step*, \**args*, \*\**kwargs*)

**Parameters**

- **step** –
- **\*args** –
- **\*\*kwargs** –

```
class edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.WaitForManager (checkpoint_root,  
                                                                    max_n,  
                                                                    inter-  
                                                                    val=5)
```

Bases: `edflow.hooks.hook.Hook`

Wait to make sure checkpoints are not overflowing.

**\_\_init\_\_** (*checkpoint\_root*, *max\_n*, *interval*=5)

**Parameters**

- **checkpoint\_root** (*str*) – Path to look for checkpoints.
- **max\_n** (*int*) – Wait as long as there are more than max\_n ckpts.
- **interval** (*float*) – Number of seconds after which to check for number of checkpoints again.

**wait** ()

Loop until the number of checkpoints got reduced.

**before\_epoch** (*ep*)

**Parameters ep** –

```
class edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.RestoreCurrentCheckpointHook (variables,  
                                                                    check-  
                                                                    point_pat  
                                                                    fil-  
                                                                    ter_cond  
                                                                    Re-  
                                                                    store-  
                                                                    Mod-  
                                                                    el-  
                                                                    Hook.<la  
                                                                    global_st
```

Bases: `edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.RestoreModelHook`

Restores a TensorFlow model from a checkpoint at each epoch. Can also be used as a functor.

**before\_epoch** (*ep*)

**Parameters ep** –



## edflow.hooks.checkpoint\_hooks.torch\_checkpoint\_hook module

### Summary

Classes:

<code>RestorePytorchModelHook</code>	Restores a PyTorch model from a checkpoint at each epoch.
--------------------------------------	---

### Reference

**class** `edflow.hooks.checkpoint_hooks.torch_checkpoint_hook.RestorePytorchModelHook` (*model*, *checkpoint\_path*, *filter\_cond*=<function `RestorePytorchModelHook.<lambda>`>, *Re-storePytorch-Model-Hook.<lambda>*, *global\_step\_setter*=None)

Bases: `edflow.hooks.hook.Hook`

Restores a PyTorch model from a checkpoint at each epoch. Can also be used as a functor.

**\_\_init\_\_** (*model*, *checkpoint\_path*, *filter\_cond*=<function `RestorePytorchModelHook.<lambda>`>, *global\_step\_setter*=None)

#### Parameters

- **model** (*torch.nn.Module*) – Model to initialize
- **checkpoint\_path** (*str*) – Directory in which the checkpoints are stored or explicit checkpoint. Ignored if used as functor.
- **filter\_cond** (*Callable*) – A function used to filter files, to only get the checkpoints that are wanted. Ignored if used as functor.
- **global\_step\_setter** (*Callable*) – Function, that the retrieved global step can be passed to.

**before\_epoch** (*ep*)

Called before each epoch.

**Parameters** *epoch* (*int*) – Index of epoch that just started.

**static** `parse_global_step` (*checkpoint*)

**static** `parse_checkpoint` (*checkpoint*)

## edflow.hooks.logging\_hooks package

Submodules:

### edflow.hooks.logging\_hooks.minimal\_logging\_hook module

## Summary

Classes:

---

*LoggingHook*

---

Minimal implementation of a logging hook.

---

## Reference

```
class edflow.hooks.logging_hooks.minimal_logging_hook.LoggingHook(paths,  
                                                                interval,  
                                                                root_path,  
                                                                im-  
                                                                age_format='png',  
                                                                name=None)
```

Bases: *edflow.hooks.hook.Hook*

Minimal implementation of a logging hook. Can be easily extended by adding handlers.

```
__init__(paths, interval, root_path, image_format='png', name=None)
```

### Parameters

- **paths** (*list(str)*) – List of key-paths to logging outputs. Will be expanded so they can be evaluated lazily.
- **interval** (*int*) – Intervall of training steps before logging.
- **root\_path** (*str*) – Path at which the logs are stored.
- **image\_format** (*str*) – Optional image format to use when saving images as files to disk. Default *png*
- **name** (*str*) – Optional name to recognize logging output.

```
after_step(batch_index, last_results)
```

Called after each step.

### Parameters

- **step** (*int*) – Current training step.
- **last\_results** (*list*) – Results from last time this hook was called.

```
log_scalars(results, step, path)
```

```
log_figures(results, step, path)
```

```
log_images(results, step, path)
```

## edflow.hooks.logging\_hooks.tensorboard\_handler module

### Summary

Functions:

<i>log_tensorboard_config</i>
<i>log_tensorboard_figures</i>
<i>log_tensorboard_images</i>
<i>log_tensorboard_scalars</i>

### Reference

edflow.hooks.logging\_hooks.tensorboard\_handler.**log\_tensorboard\_scalars**(writer, re-sults, step, path)

edflow.hooks.logging\_hooks.tensorboard\_handler.**log\_tensorboard\_images**(writer, re-sults, step, path)

edflow.hooks.logging\_hooks.tensorboard\_handler.**log\_tensorboard\_figures**(writer, re-sults, step, path)

edflow.hooks.logging\_hooks.tensorboard\_handler.**log\_tensorboard\_config**(writer, con-fig, step)

## edflow.hooks.logging\_hooks.tf\_logging\_hook module

### Summary

Classes:

<i>ImageOverviewHook</i>	
<i>LoggingHook</i>	Supply and evaluate logging ops at an intervall of training steps.

## Reference

```
class edflow.hooks.logging_hooks.tf_logging_hook.LoggingHook (scalars={}, histograms={}, images={}, logs={}, graph=None, interval=100, root_path='logs', log_images_to_tensorboard=False)
```

Bases: `edflow.hooks.hook.Hook`

Supply and evaluate logging ops at an interval of training steps.

```
__init__ (scalars={}, histograms={}, images={}, logs={}, graph=None, interval=100, root_path='logs', log_images_to_tensorboard=False)
```

### Parameters

- **scalars** (*dict*) – Scalar ops.
- **histograms** (*dict*) – Histogram ops.
- **images** (*dict*) – Image ops. Note that for these no tensorboard logging is used but a custom image saver.
- **logs** (*dict*) – Logs to std out via logger.
- **graph** (*tf.Graph*) – Current graph.
- **interval** (*int*) – Interval of training steps before logging.
- **root\_path** (*str*) – Path at which the logs are stored.

**before\_epoch** (*ep*)

Called before each epoch.

**Parameters** **epoch** (*int*) – Index of epoch that just started.

**before\_step** (*batch\_index, fetches, feeds, batch*)

Called before each step. Can update any feeds and fetches.

### Parameters

- **step** (*int*) – Current training step.
- **fetches** (*list or dict*) – Fetches for the next session.run call.
- **feeds** (*dict*) – Data used at this step.
- **batch** (*list or dict*) – All data available at this step.

**after\_step** (*batch\_index, last\_results*)

Called after each step.

### Parameters

- **step** (*int*) – Current training step.
- **last\_results** (*list*) – Results from last time this hook was called.

```
class edflow.hooks.logging_hooks.tf_logging_hook.ImageOverviewHook (images={}, interval=100, root_path='logs')
```

Bases: `edflow.hooks.hook.Hook`

---

`__init__` (*images*=*{}*, *interval*=100, *root\_path*='logs')

Logs an overview of all image outputs at an interval of training steps.

#### Parameters

- **scalars** (*dict*) – Scalar ops.
- **histograms** (*dict*) – Histogram ops.
- **images** (*dict*) – Image ops. Note that for these no tensorboard logging is used but a custom image saver.
- **logs** (*dict*) – Logs to std out via logger.
- **graph** (*tf.Graph*) – Current graph.
- **interval** (*int*) – Interval of training steps before logging.
- **root\_path** (*str*) – Path at which the logs are stored.

`after_step` (*batch\_index*, *last\_results*)

Called after each step.

#### Parameters

- **step** (*int*) – Current training step.
- **last\_results** (*list*) – Results from last time this hook was called.

### edflow.hooks.logging\_hooks.wandb\_handler module

#### Summary

Functions:

---

`log_wandb`

---

`log_wandb_images`

---

#### Reference

`edflow.hooks.logging_hooks.wandb_handler.log_wandb` (*results*, *step*, *path*)

`edflow.hooks.logging_hooks.wandb_handler.log_wandb_images` (*results*, *step*, *path*)

## edflow.hooks.metric\_hooks package

Submodules:

### edflow.hooks.metric\_hooks.tf\_metric\_hook module

## Summary

Classes:

---

<i>MetricHook</i>	Applies a set of given metrics to the calculated data.
-------------------	--

---

## Reference

**class** edflow.hooks.metric\_hooks.tf\_metric\_hook.**MetricHook** (*metrics, save\_root, consider\_only\_first=None*)

Bases: *edflow.hooks.hook.Hook*

Applies a set of given metrics to the calculated data.

**\_\_init\_\_** (*metrics, save\_root, consider\_only\_first=None*)

### Parameters

- **metrics** (*list*) – List of MetricTuple``s of the form ``: input names, output names, metric, name).
  - input names are the keys corresponding to the feeds of interest, e.g. an original image.
  - output names are the keys corresponding to the values in the results dict.
  - metric is a Callable that accepts all inputs and outputs keys as keyword arguments
  - name is a

If nested feeds or results are expected the names can be passed as “path” like 'key1\_key2' returning dict[key1][key2].

- **save\_root** (*str*) – Path to where the results are stored.
- **consider\_only\_first** (*int*) – Metric is only evaluated on the first *consider\_only\_first* examples.

**before\_epoch** (*epoch*)

Called before each epoch.

**Parameters** **epoch** (*int*) – Index of epoch that just started.

**before\_step** (*step, fetches, feeds, batch*)

Called before each step. Can update any feeds and fetches.

### Parameters

- **step** (*int*) – Current training step.
- **fetches** (*list or dict*) – Fetches for the next session.run call.

- **feeds** (*dict*) – Data used at this step.
- **batch** (*list or dict*) – All data available at this step.

**after\_step** (*step, results*)  
Called after each step.

#### Parameters

- **step** (*int*) – Current training step.
- **last\_results** (*list*) – Results from last time this hook was called.

**after\_epoch** (*epoch*)  
Called after each epoch.

**Parameters** **epoch** (*int*) – Index of epoch that just ended.

## 4.11.15 edflow.iterators package

Submodules:

### edflow.iterators.batches module

#### Summary

Functions:

<i>batch_to_canvas</i>	convert batch of images to canvas
<i>deep_lod2dol</i>	Turns a list of nested dictionaries into a nested dictionary of lists.
<i>load_image</i>	
<i>make_batches</i>	
<i>plot_batch</i>	Save batch of images tiled.
<i>save_image</i>	Save image.
<i>tile</i>	Tile images for display.

#### Reference

`edflow.iterators.batches.load_image` (*path*)

`edflow.iterators.batches.save_image` (*x, path*)  
Save image.

`edflow.iterators.batches.tile` (*X, rows, cols*)  
Tile images for display.

`edflow.iterators.batches.plot_batch` (*X, out\_path, cols=None*)  
Save batch of images tiled.

`edflow.iterators.batches.batch_to_canvas` (*X, cols=None*)  
convert batch of images to canvas

`edflow.iterators.batches.deep_lod2dol` (*list\_of\_nested\_things*)  
Turns a list of nested dictionaries into a nested dictionary of lists. This function takes care that all leafs of the nested dictionaries are considered as full keys, not only the top level keys.

---

**Note:** The difference to `deep_lod2dol()` is, that the correct type is always checked not only at exceptions.

---

**Parameters** `list_of_nested_things` (*list*) – A list of deep dictionaries

**Returns** `out` – A dict containing lists of leaf entries.

**Return type** dict

**Raises** **ValueError** – Raised if the passed object is not a `list` or if its values are not `dict`s.

```
edflow.iterators.batches.make_batches(dataset, batch_size, shuffle, n_processes=8,
                                       n_prefetch=1, error_on_timeout=False)
```

## edflow.iterators.model\_iterator module

### Summary

Exceptions:

---

<code>ShutdownRequest</code>	Raised when we receive a SIGTERM signal to shut down.
------------------------------	---

---

Classes:

---

<code>PyHookedModelIterator</code>	Implements a similar interface as the <code>HookedModelIterator</code> to train framework independent models.
------------------------------------	---

---

### Reference

**exception** `edflow.iterators.model_iterator.ShutdownRequest`

Bases: `Exception`

Raised when we receive a SIGTERM signal to shut down. Allows hooks to perform final actions such as writing a last checkpoint.

```
class edflow.iterators.model_iterator.PyHookedModelIterator(config, root,
                                                           model, datasets,
                                                           hook_freq=100,
                                                           num_epochs=100,
                                                           hooks=[],
                                                           bar_position=0,
                                                           nogpu=False,
                                                           desc=")
```

Bases: `object`

Implements a similar interface as the `HookedModelIterator` to train framework independent models.

```
__init__(config, root, model, datasets, hook_freq=100, num_epochs=100, hooks=[],
         bar_position=0, nogpu=False, desc=")
```

Constructor.

#### Parameters



- **model** (*object*) – Model class.
- **num\_epochs** (*int*) – Number of times to iterate over the data.
- **hooks** (*list*) – List containing `Hook` instances.
- **hook\_freq** (*int*) – Frequency at which hooks are evaluated.
- **bar\_position** (*int*) – Used by `tqdm` to place bars at the right position when using multiple Iterators in parallel.

**get\_split** (*\*args, \*\*kwargs*)

Get the current split that is processed.

**get\_global\_step** (*\*args, \*\*kwargs*)

Get the global step. The global step corresponds to the number of steps the model was trained for. It is updated in each step during training but not during evaluation.

**set\_global\_step** (*step*)

Set the global step. Should be done when restoring a model from a checkpoint.

**get\_batch\_step** (*\*args, \*\*kwargs*)

Batch index of current run.

**get\_epoch\_step** (*\*args, \*\*kwargs*)

Epoch index of current run.

**reset\_global\_step** ()

**increment\_global\_step** (*\*args, \*\*kwargs*)

**make\_feeds** (*batch*)

**iterate** (*batches*)

Iterates over the data supplied and feeds it to the model.

#### Parameters

- **batch\_iterator** (*Iterable*) – Iterable returning training data.
- **batch\_iterator\_validation** (*Iterable*) – Iterable returning validation data or None

**run** (*fetches, feed\_dict*)

Runs all fetch ops and stores the results.

#### Parameters

- **fetches** (*dict*) – name: Callable pairs.
- **feed\_dict** (*dict*) – Passed as kwargs to all fetch ops

**Returns** name: results pairs.

**Return type** dict

**run\_hooks** (*index, fetches=None, feeds=None, batch=None, results=None, before=True, epoch\_hooks=False*)

Run all hooks and manage their stuff. The passed arguments determine which method of the hooks is called.

#### Parameters

- **index** (*int*) – Current epoch or batch index. This is not necessarily the global training step.
- **fetches** (*list or dict*) – Fetches for the next session.run call.

- **feeds** (*dict*) – Feeds for the next session.run call.
- **results** (*same as fetches*) – Results from the last session.run call.
- **before** (*bool*) – If not obvious determines if the before or after methods of the hooks should be called.

**Returns**

- **test** (*same as fetches*) – Updated fetches.
- **test** (*dict*) – Updated feeds

**step\_ops()**

Defines ops that are called at each step.

**Returns**

**Return type** The operation run at each step.

**initialize** (*checkpoint\_path=None*)

**edflow.iterators.resize module****Summary**

Functions:

<code>resize_float32</code>	
<code>resize_hfloat32</code>	
<code>resize_image</code>	size is expanded if necessary and swapped to Pillow
<code>resize_uint8</code>	x: np.ndarray of shape (height, width) or (height, width, channels) and dtype uint8 size: int or (int, int) for target height, width

**Reference**

`edflow.iterators.resize.resize_image(x, size)`  
size is expanded if necessary and swapped to Pillow

`edflow.iterators.resize.resize_uint8(x, size)`  
x: np.ndarray of shape (height, width) or (height, width, channels) and dtype uint8 size: int or (int, int) for target height, width

`edflow.iterators.resize.resize_float32(x, size)`

`edflow.iterators.resize.resize_hfloat32(x, size)`

## edflow.iterators.template\_iterator module

### Summary

Classes:

---

<code>TemplateIterator</code>	A specialization of <code>PyHookedModelIterator</code> which adds reasonable default behaviour.
-------------------------------	---

---

### Reference

**class** `edflow.iterators.template_iterator.TemplateIterator` (\*args, \*\*kwargs)

Bases: `edflow.iterators.model_iterator.PyHookedModelIterator`

A specialization of `PyHookedModelIterator` which adds reasonable default behaviour. Subclasses should implement `save`, `restore` and `step_op`.

`__init__` (\*args, \*\*kwargs)

Constructor.

#### Parameters

- **model** (*object*) – Model class.
- **num\_epochs** (*int*) – Number of times to iterate over the data.
- **hooks** (*list*) – List containing `Hook` instances.
- **hook\_freq** (*int*) – Frequency at which hooks are evaluated.
- **bar\_position** (*int*) – Used by `tqdm` to place bars at the right position when using multiple Iterators in parallel.

**initialize** (*checkpoint\_path=None*)

**step\_ops** ()

Defines ops that are called at each step.

#### Returns

**Return type** The operation run at each step.

**save** (*checkpoint\_path*)

Save state to checkpoint path.

**restore** (*checkpoint\_path*)

Restore state from checkpoint path.

**step\_op** (*model, \*\*kwargs*)

Actual step logic. By default, a dictionary with keys ‘train\_op’, ‘log\_op’, ‘eval\_op’ and callable values is expected. ‘train\_op’ should update the model’s state as a side-effect, ‘log\_op’ will be logged to the project’s train folder. It should be a dictionary with keys ‘images’ and ‘scalars’. Images are written as png’s, scalars are written to the log file and stdout. Outputs of ‘eval\_op’ are written into the project’s eval folder to be evaluated with *edeval*.

**get\_fixed\_examples** (*which\_set='train'*)

collect fixed examples from dataset with names.

**edflow.iterators.tf\_batches module****Summary**

Functions:

---

<code>image_grid</code>	Arrange a minibatch of images into a grid to form a single image.
<code>tf_batch_to_canvas</code>	reshape a batch of images into a grid canvas to form a single image.

---

**Reference**

`edflow.iterators.tf_batches.tf_batch_to_canvas` (*X*, *cols*: *int* = *None*)  
reshape a batch of images into a grid canvas to form a single image.

**Parameters**

- **X** (*Tensor*) – Batch of images to format. [N, H, W, C]-shaped
- **cols** (*int* :) –
- **cols** – (Default value = *None*)

**Returns** `image_grid` – Tensor representing the image grid. [1, HH, WW, C]-shaped

**Return type** Tensor

**Examples**

```
x = np.ones((9, 100, 100, 3)) x = tf.convert_to_tensor(x) canvas = batches.tf_batch_to_canvas(x) assert canvas.shape == (1, 300, 300, 3)
```

```
canvas = batches.tf_batch_to_canvas(x, cols=5) assert canvas.shape == (1, 200, 500, 3)
```

```
edflow.iterators.tf_batches.image_grid(input_tensor, grid_shape, image_shape=32, 32,  
                                       num_channels=3)
```

Arrange a minibatch of images into a grid to form a single image.

**Parameters**

- **input\_tensor** – Tensor. Minibatch of images to format, either 4D ([batch size, height, width, num\_channels]) or flattened ([batch size, height \* width \* num\_channels]).
- **grid\_shape** – Sequence of int. The shape of the image grid, formatted as [grid\_height, grid\_width].
- **image\_shape** – Sequence of int. The shape of a single image, formatted as [image\_height, image\_width]. (Default value = (32))
- **32**) –
- **num\_channels** – (Default value = 3)

**Returns**

**Return type** Tensor representing a single image in which the input images have been

**Raises** **ValueError** – The grid shape and minibatch size don't match, or the image shape and number of channels are incompatible with the input tensor.

## edflow.iterators.tf\_evaluator module

### Summary

Classes:

---

*TFBaseEvaluator*

---

### Reference

```
class edflow.iterators.tf_evaluator.TFBaseEvaluator(*args, desc='Eval',
                                                    hook_freq=1, num_epochs=1,
                                                    **kwargs)
```

Bases: *edflow.iterators.tf\_iterator.TFHookedModelIterator*

```
__init__(*args, desc='Eval', hook_freq=1, num_epochs=1, **kwargs)
```

New Base evaluator restores given checkpoint path if provided, else scans checkpoint directory for latest checkpoint and uses that

#### Parameters

- **desc** (*str*) – a description for the evaluator. This description will be used during the logging.
- **hook\_freq** (*int*) – Frequency at which hooks are evaluated.
- **num\_epochs** (*int*) – Number of times to iterate over the data.

```
initialize(checkpoint_path=None)
```

```
define_graph()
```

```
step_ops()
```

Defines ops that are called at each step.

#### Returns

**Return type** The operation run at each step.

## edflow.iterators.tf\_iterator module

### Summary

Classes:

---

*TFHookedModelIterator*

---

## Reference

```
class edflow.iterators.tf_iterator.TFHookedModelIterator(config, root, model,  
                                                    datasets, hook_freq=100,  
                                                    num_epochs=100,  
                                                    hooks=[],  
                                                    bar_position=0,  
                                                    nogpu=False, desc="")
```

Bases: `edflow.iterators.model_iterator.PyHookedModelIterator`

**make\_feeds** (*batch*)

**run** (*fetches, feed\_dict*)

Runs all fetch ops and stores the results.

### Parameters

- **fetches** (*dict*) – name: Callable pairs.
- **feed\_dict** (*dict*) – Passed as kwargs to all fetch ops

**Returns** name: results pairs.

**Return type** dict

**iterate** (*batch\_iterator, validation\_batch\_iterator=None*)

Iterates over the data supplied and feeds it to the model.

### Parameters

- **batch\_iterator** (*Iterable*) – Iterable returning training data.
- **batch\_iterator\_validation** (*Iterable*) – Iterable returning validation data or None

**property** session

## edflow.iterators.tf\_trainer module

## Summary

Classes:

<code>TFBaseTrainer</code>	Same but based on TFHookedModelIterator.
<code>TFFrequencyTrainer</code>	
<code>TFListTrainer</code>	
<code>TFMultiStageModel</code>	
<code>TFMultiStageTrainer</code>	Adds multistage training to Edflow Trainer

Functions:

<code>make_linear_var</code>	Linear from $(a, \alpha)$ to $(b, \beta)$ , i.e.
------------------------------	--

## Reference

`edflow.iterators.tf_trainer.make_linear_var`(*step*, *start*, *end*, *start\_value*, *end\_value*,  
 $\text{clip\_min}=\text{None}$ ,  $\text{clip\_max}=\text{None}$ ,  
 $\text{**kwargs}$ )

Linear from  $(a, \alpha)$  to  $(b, \beta)$ , i.e.  $y = (\beta - \alpha) / (b - a) * (x - a) + \alpha$

### Parameters

- **step** (*tf.Tensor*) –  $x$
- **start** (*int*) –  $a$
- **end** (*int*) –  $b$
- **start\_value** (*float*) –  $\alpha$
- **end\_value** (*float*) –  $\beta$
- **clip\_min** (*int*) – Minimal value returned.
- **clip\_max** (*int*) – Maximum value returned.

**Returns** :math:`y`

**Return type** `tf.Tensor`

**class** `edflow.iterators.tf_trainer.TFBaseTrainer`(*config*, *root*, *model*,  $\text{**kwargs}$ )

Bases: `edflow.iterators.tf_iterator.TFHookedModelIterator`

Same but based on `TFHookedModelIterator`.

**\_\_init\_\_**(*config*, *root*, *model*,  $\text{**kwargs}$ )

Constructor.

### Parameters

- **model** (*object*) – Model class.
- **num\_epochs** (*int*) – Number of times to iterate over the data.
- **hooks** (*list*) – List containing `Hook` instances.
- **hook\_freq** (*int*) – Frequency at which hooks are evaluated.
- **bar\_position** (*int*) – Used by `tqdm` to place bars at the right position when using multiple `Iterators` in parallel.

**initialize**(*checkpoint\_path*=*None*)

Initialize from scratch or restore and keep restorer around.

**step\_ops**()

Defines ops that are called at each step.

### Returns

**Return type** The operation run at each step.

**make\_feeds**(*batch*)

Put global step into batches and add all extra required placeholders from batches.

**setup**()

Init `train_placeholders`, `log_ops` and `img_ops` which can be added to.

**create\_train\_op**()

Default optimizer + optimize each submodule

**make\_loss\_ops()**  
Return per submodule loss. Can add tensors to log\_ops and img\_ops

**make\_run\_once\_op()**  
Return op to be run at step zero. Used for custom initialization etc.

**get\_trainable\_variables(submodule)**

**get\_init\_variables()**

**get\_restore\_variables()**

**get\_checkpoint\_variables()**

**run(fetches, feed\_dict)**  
Runs all fetch ops and stores the results.

**Parameters**

- **fetches** (*dict*) – name: Callable pairs.
- **feed\_dict** (*dict*) – Passed as kwargs to all fetch ops

**Returns** name: results pairs.

**Return type** dict

**class** edflow.iterators.tf\_trainer.**TFFrequencyTrainer** (*config, root, model, \*\*kwargs*)  
Bases: *edflow.iterators.tf\_trainer.TFBaseTrainer*

**create\_train\_op()**  
Default optimizer + optimize each submodule

**run(fetches, feed\_dict)**  
Runs all fetch ops and stores the results.

**Parameters**

- **fetches** (*dict*) – name: Callable pairs.
- **feed\_dict** (*dict*) – Passed as kwargs to all fetch ops

**Returns** name: results pairs.

**Return type** dict

**class** edflow.iterators.tf\_trainer.**TFListTrainer** (*config, root, model, \*\*kwargs*)  
Bases: *edflow.iterators.tf\_trainer.TFBaseTrainer*

**create\_train\_op()**  
Default optimizer + optimize each submodule

**run(fetches, feed\_dict)**  
Runs all fetch ops and stores the results.

**Parameters**

- **fetches** (*dict*) – name: Callable pairs.
- **feed\_dict** (*dict*) – Passed as kwargs to all fetch ops

**Returns** name: results pairs.

**Return type** dict

**get\_learning\_rate\_multiplier(i)**



---

```
class edflow.iterators.tf_trainer.TFMultiStageTrainer(config, root, model,
                                                    **kwargs)
```

Bases: `edflow.iterators.tf_trainer.TFBaseTrainer`

Adds multistage training to Edflow Trainer

Stages are defined through the config. For example

**stages:**

- 1:** name: pretrain end: 10 losses: []
- 2:** name: retrain end: 30 losses: ["model"]
- 3:** name: train losses: ["model"]

The stages are sorted by their key. It is recommended to keep the simple numeric ordering. In each stage, a set of losses can be specified through the *losses* : [ "loss1", "loss2", ...] syntax. The duration of each stage is given by the *end* : *num\_steps* value. Note that the end of a stage is determined in the order of the stages. A later stage has to have a higher end value then the previous one.

The model has to implement the `edflowiterators.tf_trainer.TFMultiStageModel` interface. Look at the `multi-stage_trainer` example.

```
__init__(config, root, model, **kwargs)
    Constructor.
```

**Parameters**

- **model** (*object*) – Model class.
- **num\_epochs** (*int*) – Number of times to iterate over the data.
- **hooks** (*list*) – List containing Hook instances.
- **hook\_freq** (*int*) – Frequency at which hooks are evaluated.
- **bar\_position** (*int*) – Used by tqdm to place bars at the right position when using multiple Iterators in parallel.

```
create_train_op()
```

Default optimizer + optimize each submodule

```
run(fetches, feed_dict)
```

Runs all fetch ops and stores the results.

**Parameters**

- **fetches** (*dict*) – name: Callable pairs.
- **feed\_dict** (*dict*) – Passed as kwargs to all fetch ops

**Returns** name: results pairs.

**Return type** dict

```
get_current_train_op(current_stage)
```

```
determine_current_stage()
```

```
class edflow.iterators.tf_trainer.TFMultiStageModel
```

Bases: object

```
__init__()
```

Initialize self. See help(type(self)) for accurate signature.

```
property stage_update_op
```

```
property stage_placeholder
property stage
```

## edflow.iterators.torch\_iterator module

### Summary

Classes:

---

<i>TorchHookedModelIterator</i>	Iterator class for framework PyTorch, inherited from PyHookedModelIterator.
---------------------------------	---

---

### Reference

```
class edflow.iterators.torch_iterator.TorchHookedModelIterator(*args, transform=True,
                                                                **kwargs)
```

Bases: *edflow.iterators.model\_iterator.PyHookedModelIterator*

Iterator class for framework PyTorch, inherited from PyHookedModelIterator.

**Parameters** **transform** (*bool*) – If the batches are to be transformed to pytorch tensors. Should be true even if your input is already pytorch tensors!

**\_\_init\_\_** (*\*args, transform=True, \*\*kwargs*)  
Constructor.

#### Parameters

- **model** (*object*) – Model class.
- **num\_epochs** (*int*) – Number of times to iterate over the data.
- **hooks** (*list*) – List containing Hook instances.
- **hook\_freq** (*int*) – Frequency at which hooks are evaluated.
- **bar\_position** (*int*) – Used by tqdm to place bars at the right position when using multiple Iterators in parallel.

## 4.11.16 edflow.metrics package

Submodules:

## edflow.metrics.image\_metrics module

### Summary

Functions:

---

<i>l2_metric</i>	Pixelwise l2 distance mean.
<i>ssim_metric</i>	Compute the structural similarity score.

---

## Reference

`edflow.metrics.image_metrics.ssim_metric(batch1, batch2)`

Compute the structural similarity score.

`edflow.metrics.image_metrics.l2_metric(batch1, batch2)`

Pixelwise l2 distance mean.

### 4.11.17 edflow.util package

Submodules:

#### edflow.util.edexplore module

#### Summary

Functions:

<code>display_flow</code>	Displays flow colored image and histogram in streamlit
<code>display_flow_on_image</code>	Display flow vectors on image in streamlit
<code>first_index</code>	Find first item in iterable containing part of the string
<code>isflow</code>	Heuristic if item could be displayed as an optical flow field
<code>isimage</code>	Heuristic if item could be displayed as an image
<code>issegmentation</code>	Heuristic if item could be displayed as a segmentation mask
<code>istext</code>	Heuristic if item could be displayed as text
<code>st_get_list_or_dict_item</code>	Displays streamlit selectbox for selecting a value by key from list or dict

## Reference

`edflow.util.edexplore.isimage(obj)`

Heuristic if item could be displayed as an image

**Parameters** `obj` (*Any*) – item

**Returns** Returns True for rank-three numpy arrays where third axis is one, three or, four. And for rank-two arrays if both sides are bigger than 4.

**Return type** bool

`edflow.util.edexplore.isflow(obj)`

Heuristic if item could be displayed as an optical flow field

**Parameters** `obj` (*Any*) – item

**Returns** Returns True for rank-three numpy arrays where third axis is two.

**Return type** bool

`edflow.util.edexplore.issegmentation(obj)`

Heuristic if item could be displayed as a segmentation mask

**Parameters** `obj` (*Any*) – item

**Returns** Returns True for rank-three numpy arrays with values boolean or in [0,1]

**Return type** bool

**Warning:** This heuristic catches quite a lot of cases, which could include non-segmentation masks. Only use it for interactive display where users can manually select other display types.

`edflow.util.edexplore.istext(obj)`

Heuristic if item could be displayed as text

**Parameters** `obj` (*Any*) – item

**Returns** Returns True for items of type int, float, str, np.integer, np.float

**Return type** bool

`edflow.util.edexplore.display_flow(obj, key='flow values')`

Displays flow colored image and histogram in streamlit

**Parameters**

- `obj` (*np.ndarray*) – Optical flow field
- `key` (*str, optional*) – Title of the histogram, by default “flow values”

`edflow.util.edexplore.first_index(keys, key_part)`

Find first item in iterable containing part of the string

**Parameters**

- `keys` (*Iterable[str]*) – Iterable with strings to search through
- `key_part` (*str*) – String to look for

**Returns** Returns index of first element in keys containing key\_part, 0 if not found.

**Return type** int

`edflow.util.edexplore.st_get_list_or_dict_item(list_or_dict, item_key, description=None, filter_fn=<function <lambda>>, config=None, config_key=None, selectbox_key=None)`

Displays streamlit selectbox for selecting a value by key from list or dict

**Parameters**

- `list_or_dict` (*Union[list, dict]*) – List or dict to find item in
- `item_key` (*str*) – Key how item too look for is most likely called
- `description` (*str, optional*) – Description for streamlit selectbox, by default last part of config\_key or item key
- `filter_fn` (*callable, optional*) – Function to check if item is desired, by default *lambda ex\_item: True*
- `config` (*dict, optional*) – Config for default values, by default None
- `config_key` (*str, optional*) – How to find default value in config, by default None
- `selectbox_key` (*str, optional*) – Key passed to streamlit.selectbox

**Returns** item, item\_key for item found in dict or list

**Return type** Tuple[Any, str]

`edflow.util.edexplore.display_flow_on_image(ex, config)`

Display flow vectors on image in streamlit

Add config for this visualization to your config file to enable this visualization by default.

## Examples

Add visualizations to the text box with their import path. For example:

```
edexplore:
  visualizations:
    optical_flow_on_image:
      image_key: "images/0/image"
      flow_key: "forward_flow"
      vector_frequency: 5
      flow_downsample_method: max_magnitude
```

## Parameters

- **ex** (*dict*) – Example dict from dataset
- **config** (*dict*) – Config dict

## edflow.util.util module

Some Utility functions, that make yur life easier but don't fit in any better catorgory than util.

## Summary

Exceptions:

---

*KeyNotFoundError*

---

Classes:

---

*NoModel*

*PRNGMixin*

Adds a prng property which is a numpy RandomState which gets reinitialized whenever the pid changes to avoid synchronized sampling behavior when used in conjunction with multiprocessing.

---

*Printer*

For usage with walk: collects strings for printing

---

*TablePrinter*

For usage with walk: Collects string to put in a table.

---

Functions:

---

*cached\_function*

a very rough cache for function calls.

---

*contains\_key*

Tests if the path like key can find an object in the nested\_thing.

---

continues on next page

Table 71 – continued from previous page

<code>edprint</code>	Prints every leaf variable in <code>nested_thing</code> in the form of a table.
<code>get_leaf_names</code>	
<code>get_obj_from_str</code>	
<code>get_str_from_obj</code>	
<code>get_value_from_key</code>	Get value from collection given key
<code>linear_var</code>	Linear from $(a, \alpha)$ to $(b, \beta)$ , i.e.
<code>pop_keypath</code>	Given a nested list or dict structure, pop the desired value at key expanding callable nodes if necessary and <code>expand</code> is <code>True</code> .
<code>pop_value_from_key</code>	Pop item from collection given key
<code>pp2mkdtable</code>	Turns a formatted string into a markdown table.
<code>pprint</code>	Prints nested objects and tries to give relevant information.
<code>pprint_str</code>	Formats nested objects as string and tries to give relevant information.
<code>retrieve</code>	Given a nested list or dict return the desired value at key expanding callable nodes if necessary and <code>expand</code> is <code>True</code> .
<code>set_default</code>	Combines <code>retrieve()</code> and <code>set_value()</code> to create the behaviour of python's <code>dict.setdefault</code> : If key is found in <code>list_or_dict</code> , return its value, otherwise return default and add it to <code>list_or_dict</code> at key.
<code>set_value</code>	Sets a value in a possibly nested list or dict object.
<code>strenumerate</code>	Works just as <code>enumerate</code> , but the returned index is a string.
<code>update</code>	
<code>walk</code>	Walk a nested list and/or dict recursively and call <code>fn</code> on all non list or dict objects.

## Reference

`edflow.util.util.get_str_from_obj(obj)`

`edflow.util.util.get_obj_from_str(string, reload=False)`

`edflow.util.util.linear_var(step, start, end, start_value, end_value, clip_min=0.0, clip_max=1.0)`

Linear from  $(a, \alpha)$  to  $(b, \beta)$ , i.e.  $y = (\beta - \alpha) / (b - a) * (x - a) + \alpha$

### Parameters

- `step(int)` –  $x$
- `start(float)` –  $a$
- `end(float)` –  $b$
- `start_value(float)` –  $\alpha$
- `end_value(float)` –  $\beta$
- `clip_min(float)` – Minimal value returned.
- `clip_max(float)` – Maximum value returned.

**Returns**  $y$

**Return type** float

`edflow.util.util.walk(dict_or_list, fn, inplace=False, pass_key=False, prev_key='', splitval='/', walk_np_arrays=False)`

Walk a nested list and/or dict recursively and call fn on all non list or dict objects.

### Example

```
dol = {'a': [1, 2], 'b': {'c': 3, 'd': 4}}

def fn(val):
    return val**2

result = walk(dol, fn)
print(result) # {'a': [1, 4], 'b': {'c': 9, 'd': 16}}
print(dol)   # {'a': [1, 2], 'b': {'c': 3, 'd': 4}}

result = walk(dol, fn, inplace=True)
print(result) # {'a': [1, 4], 'b': {'c': 9, 'd': 16}}
print(dol)   # {'a': [1, 4], 'b': {'c': 9, 'd': 16}}
```

### Parameters

- **dict\_or\_list** (*dict or list*) – Possibly nested list or dictionary.
- **fn** (*Callable*) – Applied to each leave of the nested list\_dict-object.
- **inplace** (*bool*) – If False, a new object with the same structure and the results of fn at the leaves is created. If True the leaves are replaced by the results of fn.
- **pass\_key** (*bool*) – Also passes the key or index of the leave element to fn.
- **prev\_key** (*str*) – If pass\_key == True keys of parent nodes are passed to calls of walk on child nodes to accumulate the keys.
- **splitval** (*str*) – String used to join keys if pass\_key is True.
- **walk\_np\_arrays** (*bool*) – If True, numpy arrays are interpreted as list, ie not as leaves.

### Returns

- *The resulting nested list-dict-object with the results of*
- **fn at its leaves.** (*dict or list*)

**exception** `edflow.util.util.KeyNotFoundError` (*cause, keys=None, visited=None*)

Bases: Exception

**\_\_init\_\_** (*cause, keys=None, visited=None*)

Initialize self. See help(type(self)) for accurate signature.

`edflow.util.util.retrieve(list_or_dict, key, splitval='/', default=None, expand=True, pass_success=False)`

Given a nested list or dict return the desired value at key expanding callable nodes if necessary and expand is True. The expansion is done in-place.

### Parameters

- **list\_or\_dict** (*list or dict*) – Possibly nested list or dictionary.

- **key** (*str*) – key/to/value, path like string describing all keys necessary to consider to get to the desired value. List indices can also be passed here.
- **splitval** (*str*) – String that defines the delimiter between keys of the different depth levels in *key*.
- **default** (*obj*) – Value returned if *key* is not found.
- **expand** (*bool*) – Whether to expand callable nodes on the path or not.

#### Returns

- The desired value or if *default* is not *None* and the
- *key* is not found returns *default*.

:raises Exception if *key* not in *list\_or\_dict* and *default* is: :raises *None*..

`edflow.util.util.pop_keypath` (*current\_item*: *Union[callable, list, dict]*, *key*: *str*, *splitval*: *str* = '/', *default*: *object* = *None*, *expand*: *bool* = *True*, *pass\_success*: *bool* = *False*)

Given a nested list or dict structure, pop the desired value at *key* expanding callable nodes if necessary and *expand* is *True*. The expansion is done in-place.

#### Parameters

- **current\_item** (*list* or *dict*) – Possibly nested list or dictionary.
- **key** (*str*) – key/to/value, path like string describing all keys necessary to consider to get to the desired value. List indices can also be passed here.
- **splitval** (*str*) – String that defines the delimiter between keys of the different depth levels in *key*.
- **default** (*obj*) – Value returned if *key* is not found.
- **expand** (*bool*) – Whether to expand callable nodes on the path or not.

#### Returns

- The desired value or if *default* is not *None* and the
- *key* is not found returns *default*.

:raises Exception if *key* not in *list\_or\_dict* and *default* is: :raises *None*..

`edflow.util.util.get_value_from_key` (*collection*: *Union[list, dict]*, *key*: *str*)  
Get value from collection given *key*

`edflow.util.util.pop_value_from_key` (*collection*: *Union[list, dict]*, *key*: *str*)  
Pop item from collection given *key*

#### Parameters

- **collection** (*Union[list, dict]*) –
- **key** –

`edflow.util.util.set_default` (*list\_or\_dict*, *key*, *default*, *splitval*='/')

Combines `retrieve()` and `set_value()` to create the behaviour of `python's dict.setdefault`: If *key* is found in *list\_or\_dict*, return its value, otherwise return *default* and add it to *list\_or\_dict* at *key*.

#### Parameters

- **list\_or\_dict** (*list* or *dict*) – Possibly nested list or dictionary. *splitval* (*str*): String that defines the delimiter between keys of the different depth levels in *key*.



- **key** (*str*) – key/to/value, path like string describing all keys necessary to consider to get to the desired value. List indices can also be passed here.
- **default** (*object*) – Value to be returned if key not in list\_or\_dict and set to be at key in this case.
- **splitval** (*str*) – String that defines the delimiter between keys of the different depth levels in key.

### Returns

- The retrieved value or if the key is not found returns
- default.

`edflow.util.util.set_value(list_or_dict, key, val, splitval='/')`

Sets a value in a possibly nested list or dict object.

### Parameters

- **key** (*str*) – key/to/value, path like string describing all keys necessary to consider to get to the desired value. List indices can also be passed here.
- **value** (*object*) – Anything you want to put behind key
- **list\_or\_dict** (*list or dict*) – Possibly nested list or dictionary.
- **splitval** (*str*) – String that defines the delimiter between keys of the different depth levels in key.

### Examples

```
dol = {"a": [1, 2], "b": {"c": {"d": 1}, "e": 2}}

# Change existing entry
set_value(dol, "a/0", 3)
# {'a': [3, 2], 'b': {'c': {'d': 1}, 'e': 2}}
```

```
set_value(dol, "b/e", 3)
# {"a": [3, 2], "b": {"c": {"d": 1}, "e": 3}}
```

```
set_value(dol, "a/1/f", 3)
# {"a": [3, {"f": 3}], "b": {"c": {"d": 1}, "e": 3}}
```

```
# Append to list
dol = {"a": [1, 2], "b": {"c": {"d": 1}, "e": 2}}
```

```
set_value(dol, "a/2", 3)
# {"a": [1, 2, 3], "b": {"c": {"d": 1}, "e": 2}}
```

```
set_value(dol, "a/5", 6)
# {"a": [1, 2, 3, None, None, 6], "b": {"c": {"d": 1}, "e": 2}}
```

```
# Add key
dol = {"a": [1, 2], "b": {"c": {"d": 1}, "e": 2}}
```

```
set_value(dol, "f", 3)
# {"a": [1, 2], "b": {"c": {"d": 1}, "e": 2}, "f": 3}
```

```
set_value(dol, "b/1", 3)
# {"a": [1, 2], "b": {"c": {"d": 1}, "e": 2, 1: 3}, "f": 3}
```

(continues on next page)

(continued from previous page)

```

# Raises Error:
# Appending key to list
# set_value(dol, 'a/g', 3) # should raise

# Fancy Overwriting
dol = {"a": [1, 2], "b": {"c": {"d": 1}}, "e": 2}

set_value(dol, "e/f", 3)
# {"a": [1, 2], "b": {"c": {"d": 1}}, "e": {"f": 3}}

set_value(dol, "e/f/1/g", 3)
# {"a": [1, 2], "b": {"c": {"d": 1}}, "e": {"f": [None, {"g": 3}]}}

# Toplevel new key
dol = {"a": [1, 2], "b": {"c": {"d": 1}}, "e": 2}
set_value(dol, "h", 4)
# {"a": [1, 2], "b": {"c": {"d": 1}}, "e": 2, "h": 4}

set_value(dol, "i/j/k", 4)
# {"a": [1, 2], "b": {"c": {"d": 1}}, "e": 2, "h": 4, "i": {"j": {"k": 4}}}

set_value(dol, "j/0/k", 4)
# {"a": [1, 2], "b": {"c": {"d": 1}}, "e": 2, "h": 4, "i": {"j": {"k": 4}}, "j": [
  ↳ {"k": 4}], }

# Toplevel is list new key
dol = [{"a": [1, 2], "b": {"c": {"d": 1}}, "e": 2}, 2, 3]

set_value(dol, "0/k", 4)
# [{"a": [1, 2], "b": {"c": {"d": 1}}, "e": 2, "k": 4}, 2, 3]

set_value(dol, "0", 1)
# [1, 2, 3]

```

`edflow.util.util.contains_key` (*nested\_thing*, *key*, *splitval*='/', *expand*=True)

Tests if the path like key can find an object in the *nested\_thing*.

`edflow.util.util.update` (*to\_update*, *to\_update\_with*, *splitval*='/', *expand*=True)

`edflow.util.util.get_leaf_names` (*nested\_thing*)

`edflow.util.util.strenumerate` (*iterable*)

Works just as `enumerate`, but the returned index is a string.

**Parameters** `iterable` (*Iterable*) – An (guess what) iterable object.

`edflow.util.util.cached_function` (*fn*)

a very rough cache for function calls. Highly experimental. Only active if activated with environment variable.

**class** `edflow.util.util.PRNGMixin`

Bases: `object`

Adds a `prng` property which is a `numpy RandomState` which gets reinitialized whenever the `pid` changes to avoid synchronized sampling behavior when used in conjunction with multiprocessing.

**property** `prng`

**class** `edflow.util.util.Printer` (*string\_fn*)

Bases: `object`

For usage with walk: collects strings for printing

```
__init__ (string_fn)
    Initialize self. See help(type(self)) for accurate signature.
```

```
class edflow.util.util.TablePrinter (string_fn, names=None, jupyter_style=False)
    Bases: object
```

For usage with walk: Collects string to put in a table.

```
__init__ (string_fn, names=None, jupyter_style=False)
    Initialize self. See help(type(self)) for accurate signature.
```

```
edflow.util.util.pprint_str (nested_thing, heuristics=None)
    Formats nested objects as string and tries to give relevant information.
```

#### Parameters

- **nested\_thing** (*dict or list*) – Some nested object.
- **heuristics** (*Callable*) – If given this should produce the string, which is printed as description of a leaf object.

```
edflow.util.util.pprint (nested_thing, heuristics=None)
    Prints nested objects and tries to give relevant information.
```

#### Parameters

- **nested\_thing** (*dict or list*) – Some nested object.
- **heuristics** (*Callable*) – If given this should produce the string, which is printed as description of a leaf object.

```
edflow.util.util.pp2mkdtable (nested_thing, jupyter_style=False)
    Turns a formatted string into a markdown table.
```

```
edflow.util.util.edprint (nested_thing)
    Prints every leaf variable in nested_thing in the form of a table.
```

**Parameters** **nested\_thing** (*dict or list*) – Some nested object.

```
class edflow.util.util.NoModel (config)
    Bases: object
```

```
__init__ (config)
    Initialize self. See help(type(self)) for accurate signature.
```

## Summary

## Reference



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### e

- edflow, 30
- edflow.applications, 38
- edflow.applications.tf\_perceptual\_loss, 38
- edflow.args, 30
- edflow.config, 39
- edflow.config.commandline\_kwargs, 39
- edflow.custom\_logging, 30
- edflow.data, 40
- edflow.data.agnostics, 44
- edflow.data.agnostics.concatenated, 44
- edflow.data.agnostics.csv\_dset, 45
- edflow.data.agnostics.late\_loading, 46
- edflow.data.agnostics.subdataset, 47
- edflow.data.believers, 47
- edflow.data.believers.meta, 47
- edflow.data.believers.meta\_loaders, 49
- edflow.data.believers.meta\_util, 51
- edflow.data.believers.meta\_view, 51
- edflow.data.believers.sequence, 53
- edflow.data.dataset, 40
- edflow.data.dataset\_mixin, 40
- edflow.data.processing, 56
- edflow.data.processing.labels, 56
- edflow.data.processing.processed, 57
- edflow.data.util, 57
- edflow.data.util.cached\_dset, 58
- edflow.data.util.util\_dsets, 61
- edflow.datasets, 65
- edflow.datasets.celeba, 65
- edflow.datasets.cifar, 66
- edflow.datasets.fashionmnist, 66
- edflow.datasets.mnist, 67
- edflow.datasets.utils, 68
- edflow.debug, 34
- edflow.edsetup\_files, 69
- edflow.edsetup\_files.dataset, 69
- edflow.edsetup\_files.iterator, 70
- edflow.edsetup\_files.model, 71
- edflow.eval, 71
- edflow.eval.pipeline, 71
- edflow.explore, 35
- edflow.fpdb, 37
- edflow.hooks, 78
- edflow.hooks.checkpoint\_hooks, 85
- edflow.hooks.checkpoint\_hooks.common, 85
- edflow.hooks.checkpoint\_hooks.lambda\_checkpoint\_hook, 88
- edflow.hooks.checkpoint\_hooks.tf\_checkpoint\_hook, 89
- edflow.hooks.checkpoint\_hooks.torch\_checkpoint\_hook, 93
- edflow.hooks.hook, 78
- edflow.hooks.logging\_hooks, 94
- edflow.hooks.logging\_hooks.minimal\_logging\_hook, 94
- edflow.hooks.logging\_hooks.tensorboard\_handler, 95
- edflow.hooks.logging\_hooks.tf\_logging\_hook, 95
- edflow.hooks.logging\_hooks.wandb\_handler, 97
- edflow.hooks.metric\_hooks, 98
- edflow.hooks.metric\_hooks.tf\_metric\_hook, 98
- edflow.hooks.pytorch\_hooks, 80
- edflow.hooks.runtime\_input, 83
- edflow.hooks.util\_hooks, 83
- edflow.iterators, 99
- edflow.iterators.batches, 99
- edflow.iterators.model\_iterator, 100
- edflow.iterators.resize, 102
- edflow.iterators.template\_iterator, 103
- edflow.iterators.tf\_batches, 104
- edflow.iterators.tf\_evaluator, 105
- edflow.iterators.tf\_iterator, 105
- edflow.iterators.tf\_trainer, 106
- edflow.iterators.torch\_iterator, 110
- edflow.main, 37
- edflow.metrics, 110
- edflow.metrics.image\_metrics, 110
- edflow.project\_manager, 38

`edflow.util`, [111](#)  
`edflow.util.edexplore`, [111](#)  
`edflow.util.util`, [113](#)



## Symbols

<code>__init__()</code> ( <i>edflow.applications.tf_perceptual_loss.VGG19Features</i> method), 38	<code>__init__()</code> ( <i>edflow.datasets.cifar.CIFAR10</i> method), 65
<code>__init__()</code> ( <i>edflow.custom_logging.ColorLineFormatter</i> method), 33	<code>__init__()</code> ( <i>edflow.datasets.fashionmnist.FashionMNIST</i> method), 66
<code>__init__()</code> ( <i>edflow.data.agnostics.concatenated.DisjunctiveExampleConcatenatedDataset</i> method), 45	<code>__init__()</code> ( <i>edflow.datasets.mnist.MNIST</i> method), 67
<code>__init__()</code> ( <i>edflow.data.agnostics.concatenated.ExampleConcatenatedDataset</i> method), 44	<code>__init__()</code> ( <i>edflow.debug.ConfigDebugDataset</i> method), 35
<code>__init__()</code> ( <i>edflow.data.agnostics.csv_dset.CsvDataset</i> method), 45	<code>__init__()</code> ( <i>edflow.debug.DebugDataset</i> method), 34
<code>__init__()</code> ( <i>edflow.data.agnostics.late_loading.LateLoadingDataset</i> method), 47	<code>__init__()</code> ( <i>edflow.debug.DebugIterator</i> method), 34
<code>__init__()</code> ( <i>edflow.data.believers.meta.MetaDataset</i> method), 48	<code>__init__()</code> ( <i>edflow.debug.DebugModel</i> method), 34
<code>__init__()</code> ( <i>edflow.data.believers.meta_view.MetaViewDataset</i> method), 53	<code>__init__()</code> ( <i>edflow.edsetup_files.dataset.Dataset</i> method), 69
<code>__init__()</code> ( <i>edflow.data.believers.sequence.SequenceDataset</i> method), 54	<code>__init__()</code> ( <i>edflow.edsetup_files.iterator.Iterator</i> method), 70
<code>__init__()</code> ( <i>edflow.data.believers.sequence.UnSequenceDataset</i> method), 55	<code>__init__()</code> ( <i>edflow.edsetup_files.model.Model</i> method), 71
<code>__init__()</code> ( <i>edflow.data.dataset_mixin.ConcatenatedDataset</i> method), 43	<code>__init__()</code> ( <i>edflow.eval.pipeline.EvalHook</i> method), 74
<code>__init__()</code> ( <i>edflow.data.dataset_mixin.SubDataset</i> method), 43	<code>__init__()</code> ( <i>edflow.eval.pipeline.TemplateEvalHook</i> method), 75
<code>__init__()</code> ( <i>edflow.data.processing.labels.ExtraLabelsDataset</i> method), 56	<code>__init__()</code> ( <i>edflow.hooks.checkpoint_hooks.common.CollectorHook</i> method), 86
<code>__init__()</code> ( <i>edflow.data.processing.labels.LabelDataset</i> method), 56	<code>__init__()</code> ( <i>edflow.hooks.checkpoint_hooks.common.KeepBestCheckpointHook</i> method), 88
<code>__init__()</code> ( <i>edflow.data.processing.processed.ProcessedDataset</i> method), 57	<code>__init__()</code> ( <i>edflow.hooks.checkpoint_hooks.common.StoreArraysHook</i> method), 87
<code>__init__()</code> ( <i>edflow.data.util.cached_dset.CachedDataset</i> method), 59	<code>__init__()</code> ( <i>edflow.hooks.checkpoint_hooks.common.WaitForCheckpointHook</i> method), 86
<code>__init__()</code> ( <i>edflow.data.util.cached_dset.ExamplesFolder</i> method), 58	<code>__init__()</code> ( <i>edflow.hooks.checkpoint_hooks.lambda_checkpoint_hook.LambdaCheckpointHook</i> method), 88
<code>__init__()</code> ( <i>edflow.data.util.cached_dset.PathCachedDataset</i> method), 60	<code>__init__()</code> ( <i>edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.CheckpointHook</i> method), 90
<code>__init__()</code> ( <i>edflow.data.util.util_dsets.DataFolder</i> method), 62	<code>__init__()</code> ( <i>edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.RestoreCheckpointHook</i> method), 90
<code>__init__()</code> ( <i>edflow.data.util.util_dsets.RandomlyJoinedDataset</i> method), 62	<code>__init__()</code> ( <i>edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.RetrainCheckpointHook</i> method), 91
<code>__init__()</code> ( <i>edflow.datasets.celeba.CelebA</i> method), 65	<code>__init__()</code> ( <i>edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.WaitForCheckpointHook</i> method), 92
	<code>__init__()</code> ( <i>edflow.hooks.checkpoint_hooks.torch_checkpoint_hook.RestoreCheckpointHook</i> method), 93

`__init__()` (`edflow.hooks.logging_hooks.minimal_logging_hook.LoggingHook` `method`), 94  
`__init__()` (`edflow.hooks.logging_hooks.tf_logging_hook.ImageOverviewHook` `method`), 96  
`__init__()` (`edflow.hooks.logging_hooks.tf_logging_hook.LoggingHook` `method`), 96  
`__init__()` (`edflow.hooks.metric_hooks.tf_metric_hook.MetricHook` `method`), 98  
`__init__()` (`edflow.hooks.pytorch_hooks.PyCheckpointHook` `method`), 80  
`__init__()` (`edflow.hooks.pytorch_hooks.PyLoggingHook` `method`), 81  
`__init__()` (`edflow.hooks.pytorch_hooks.ToFromTorchHook` `method`), 82  
`__init__()` (`edflow.hooks.pytorch_hooks.ToTorchHook` `method`), 82  
`__init__()` (`edflow.hooks.runtime_input.RuntimeInputHook` `method`), 83  
`__init__()` (`edflow.hooks.util_hooks.ExpandHook` `method`), 83  
`__init__()` (`edflow.hooks.util_hooks.IntervalHook` `method`), 84  
`__init__()` (`edflow.iterators.model_iterator.PyHookedModelIterator` `method`), 100  
`__init__()` (`edflow.iterators.template_iterator.TemplateIterator` `method`), 103  
`__init__()` (`edflow.iterators.tf_evaluator.TFBaseEvaluator` `method`), 105  
`__init__()` (`edflow.iterators.tf_trainer.TFBaseTrainer` `method`), 107  
`__init__()` (`edflow.iterators.tf_trainer.TFMultiStageModel` `method`), 109  
`__init__()` (`edflow.iterators.tf_trainer.TFMultiStageTrainer` `method`), 109  
`__init__()` (`edflow.iterators.torch_iterator.TorchHookedModelIterator` `method`), 110  
`__init__()` (`edflow.util.util.KeyNotFoundError` `method`), 115  
`__init__()` (`edflow.util.util.NoModel` `method`), 119  
`__init__()` (`edflow.util.util.Printer` `method`), 119  
`__init__()` (`edflow.util.util.TablePrinter` `method`), 119

**A**  
`add_im_info()` (in module `edflow.data.util`), 64  
`add_meta_data()` (in module `edflow.eval.pipeline`), 76  
`adjust_support()` (in module `edflow.data.util`), 63  
`after_epoch()` (`edflow.eval.pipeline.EvalHook` `method`), 74  
`after_epoch()` (`edflow.eval.pipeline.TemplateEvalHook` `method`), 75  
`after_epoch()` (`edflow.hooks.checkpoint_hooks.common.KeepBestCheckpoints` `method`), 88  
`after_epoch()` (`edflow.hooks.checkpoint_hooks.common.StoreArraysHook` `method`), 87  
`after_epoch()` (`edflow.hooks.checkpoint_hooks.lambda_checkpoint_hook.LambdaCheckpointHook` `method`), 89  
`after_epoch()` (`edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.CheckpointHook` `method`), 91  
`after_epoch()` (`edflow.hooks.hook.Hook` `method`), 79  
`after_epoch()` (`edflow.hooks.metric_hooks.tf_metric_hook.MetricHook` `method`), 99  
`after_epoch()` (`edflow.hooks.pytorch_hooks.PyCheckpointHook` `method`), 80  
`after_epoch()` (`edflow.hooks.pytorch_hooks.PyLoggingHook` `method`), 81  
`after_epoch()` (`edflow.hooks.pytorch_hooks.ToNumpyHook` `method`), 82  
`after_epoch()` (`edflow.hooks.util_hooks.ExpandHook` `method`), 83  
`after_step()` (`edflow.eval.pipeline.EvalHook` `method`), 74  
`after_step()` (`edflow.eval.pipeline.TemplateEvalHook` `method`), 75  
`after_step()` (`edflow.hooks.checkpoint_hooks.common.CollectorHook` `method`), 86  
`after_step()` (`edflow.hooks.checkpoint_hooks.lambda_checkpoint_hook.LambdaCheckpointHook` `method`), 89  
`after_step()` (`edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.CheckpointHook` `method`), 91  
`after_step()` (`edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.RetrieveHook` `method`), 92  
`after_step()` (`edflow.hooks.hook.Hook` `method`), 79  
`after_step()` (`edflow.hooks.logging_hooks.minimal_logging_hook.LoggingHook` `method`), 94  
`after_step()` (`edflow.hooks.logging_hooks.tf_logging_hook.ImageOverviewHook` `method`), 97  
`after_step()` (`edflow.hooks.logging_hooks.tf_logging_hook.LoggingHook` `method`), 96  
`after_step()` (`edflow.hooks.metric_hooks.tf_metric_hook.MetricHook` `method`), 99  
`after_step()` (`edflow.hooks.pytorch_hooks.DataPrepHook` `method`), 82  
`after_step()` (`edflow.hooks.pytorch_hooks.PyCheckpointHook` `method`), 80  
`after_step()` (`edflow.hooks.pytorch_hooks.PyLoggingHook` `method`), 81  
`after_step()` (`edflow.hooks.pytorch_hooks.ToNumpyHook` `method`), 82  
`after_step()` (`edflow.hooks.util_hooks.ExpandHook` `method`), 83

`after_step()` (*edflow.hooks.util\_hooks.IntervalHook* method), 84  
`append_labels()` (*edflow.data.dataset\_mixin.DatasetMixin* property), 43  
`apply_callbacks()` (in module *edflow.eval.pipeline*), 78  
`at_exception()` (*edflow.eval.pipeline.EvalHook* method), 74  
`at_exception()` (*edflow.eval.pipeline.TemplateEvalHook* method), 75  
`at_exception()` (*edflow.hooks.checkpoint\_hooks.lambda\_checkpoint\_hook.LambdaCheckpointHook* method), 89  
`at_exception()` (*edflow.hooks.checkpoint\_hooks.tf\_checkpoint\_hook.CheckpointHook* method), 91  
`at_exception()` (*edflow.hooks.hook.Hook* method), 79  
`at_exception()` (*edflow.hooks.pytorch\_hooks.PyCheckpointHook* method), 81

## B

`batch_to_canvas()` (in module *edflow.iterators.batches*), 99  
`before_epoch()` (*edflow.eval.pipeline.EvalHook* method), 74  
`before_epoch()` (*edflow.eval.pipeline.TemplateEvalHook* method), 75  
`before_epoch()` (*edflow.hooks.checkpoint\_hooks.common.WaitForCheckpointHook* method), 86  
`before_epoch()` (*edflow.hooks.checkpoint\_hooks.lambda\_checkpoint\_hook.LambdaCheckpointHook* method), 88  
`before_epoch()` (*edflow.hooks.checkpoint\_hooks.tf\_checkpoint\_hook.CheckpointHook* method), 91  
`before_epoch()` (*edflow.hooks.checkpoint\_hooks.tf\_checkpoint\_hook.RestoreCustomCheckpointHook* method), 92  
`before_epoch()` (*edflow.hooks.checkpoint\_hooks.tf\_checkpoint\_hook.RestoreModelHook* method), 90  
`before_epoch()` (*edflow.hooks.checkpoint\_hooks.tf\_checkpoint\_hook.RetrainHook* method), 91  
`before_epoch()` (*edflow.hooks.checkpoint\_hooks.tf\_checkpoint\_hook.WaitForManager* method), 92  
`before_epoch()` (*edflow.hooks.hook.Hook* method), 79  
`before_epoch()` (*edflow.hooks.logging\_hooks.tf\_logging\_hook.LoggingHook* method), 96  
`before_epoch()` (*edflow.hooks.metric\_hooks.tf\_metric\_hook.MetricHook* method), 98  
`before_epoch()` (*edflow.hooks.pytorch\_hooks.PyCheckpointHook* method), 81  
`before_epoch()` (*edflow.hooks.pytorch\_hooks.DataPrepHook* method), 82  
`before_epoch()` (*edflow.hooks.pytorch\_hooks.LambdaCheckpointHook* method), 88  
`before_epoch()` (*edflow.hooks.pytorch\_hooks.PyLoggingHook* method), 81  
`before_epoch()` (*edflow.hooks.pytorch\_hooks.ToTorchHook* method), 82  
`before_epoch()` (*edflow.hooks.pytorch\_hooks.RestoreCustomCheckpointHook* method), 92  
`before_epoch()` (*edflow.hooks.runtime\_input.RuntimeInputHook* method), 83  
`before_epoch()` (*edflow.hooks.util\_hooks.IntervalHook* method), 84

## C

`cacheable()` (in module *edflow.data.util.cached\_dset*), 60  
`cache_dataset()` (*edflow.data.util.cached\_dset.CachedDataset* method), 60

method), 60

cached\_function() (in module *edflow.util.util*), 118

CachedDataset (class in *edflow.data.util.cached\_dset*), 58

cart2polar() (in module *edflow.data.util*), 63

category() (in module *edflow.data.believers.meta\_loaders*), 50

cbargs2cbdict() (in module *edflow.eval.pipeline*), 78

CelebA (class in *edflow.datasets.celeba*), 65

CelebATest (class in *edflow.datasets.celeba*), 65

CelebATrain (class in *edflow.datasets.celeba*), 65

CelebAVal (class in *edflow.datasets.celeba*), 65

CheckpointHook (class in *edflow.hooks.checkpoint\_hooks.tf\_checkpoint\_hook*), 90

checkpoints (*edflow.custom\_logging.run* attribute), 31

CIFAR10 (class in *edflow.datasets.cifar*), 66

CIFAR10Test (class in *edflow.datasets.cifar*), 66

CIFAR10Train (class in *edflow.datasets.cifar*), 66

clean\_keys() (in module *edflow.data.believers.meta*), 49

clip\_to\_support() (in module *edflow.data.util*), 64

code (*edflow.custom\_logging.run* attribute), 31

code\_root (*edflow.custom\_logging.run* attribute), 31

CollectorHook (class in *edflow.hooks.checkpoint\_hooks.common*), 86

ColorLineFormatter (class in *edflow.custom\_logging*), 33

ConcatenatedDataset (class in *edflow.data.dataset\_mixin*), 43

config2cbdict() (in module *edflow.eval.pipeline*), 78

ConfigDebugDataset (class in *edflow.debug*), 35

configs (*edflow.custom\_logging.run* attribute), 31

contains\_key() (in module *edflow.util.util*), 118

create\_train\_op() (in module *edflow.iterators.tf\_trainer.TFBaseTrainer* method), 107

create\_train\_op() (in module *edflow.iterators.tf\_trainer.TFFrequencyTrainer* method), 108

create\_train\_op() (in module *edflow.iterators.tf\_trainer.TFListTrainer* method), 108

create\_train\_op() (in module *edflow.iterators.tf\_trainer.TFMultiStageTrainer* method), 109

CsvDataset (class in *edflow.data.agnostics.csv\_dset*), 45

custom\_visualizations() (in module *edflow.explore*), 36

## D

DataFolder (class in *edflow.data.util.util\_dsets*), 62

DataPrepHook (class in *edflow.hooks.pytorch\_hooks*), 82

Dataset (class in *edflow.edsetup\_files.dataset*), 69

DatasetMixin (class in *edflow.data.dataset\_mixin*), 41

debug\_step\_op() (in module *edflow.debug*), 34

DebugDataset (class in *edflow.debug*), 34

DebugIterator (class in *edflow.debug*), 34

DebugModel (class in *edflow.debug*), 34

decompose\_name() (in module *edflow.eval.pipeline*), 77

deep\_lod2dol() (in module *edflow.iterators.batches*), 99

default\_heuristic() (in module *edflow.data.util*), 64

define\_graph() (in module *edflow.iterators.tf\_evaluator.TFBaseEvaluator* method), 105

determine\_current\_stage() (in module *edflow.iterators.tf\_trainer.TFMultiStageTrainer* method), 109

determine\_loader() (in module *edflow.eval.pipeline*), 77

determine\_saver() (in module *edflow.eval.pipeline*), 76

dict\_repr() (in module *edflow.hooks.checkpoint\_hooks.common*), 86

DisjunctExampleConcatenatedDataset (class in *edflow.data.agnostics.concatenated*), 44

display() (in module *edflow.explore*), 35

display\_default() (in module *edflow.explore*), 35

display\_flow() (in module *edflow.util.edexplore*), 112

display\_flow\_on\_image() (in module *edflow.util.edexplore*), 113

download\_url() (in module *edflow.datasets.utils*), 69

download\_urls() (in module *edflow.datasets.utils*), 69

## E

edflow  
module, 30

edflow.applications  
module, 38

edflow.applications.tf\_perceptual\_loss  
module, 38

edflow.args  
module, 30

edflow.config  
module, 39

edflow.config.commandline\_kwargs  
module, 39

---

edflow.custom_logging	edflow.debug
module, 30	module, 34
edflow.data	edflow.edsetup_files
module, 40	module, 69
edflow.data.agnostics	edflow.edsetup_files.dataset
module, 44	module, 69
edflow.data.agnostics.concatenated	edflow.edsetup_files.iterator
module, 44	module, 70
edflow.data.agnostics.csv_dset	edflow.edsetup_files.model
module, 45	module, 71
edflow.data.agnostics.late_loading	edflow.eval
module, 46	module, 71
edflow.data.agnostics.subdataset	edflow.eval.pipeline
module, 47	module, 71
edflow.data.believers	edflow.explore
module, 47	module, 35
edflow.data.believers.meta	edflow.fpdb
module, 47	module, 37
edflow.data.believers.meta_loaders	edflow.hooks
module, 49	module, 78
edflow.data.believers.meta_util	edflow.hooks.checkpoint_hooks
module, 51	module, 85
edflow.data.believers.meta_view	edflow.hooks.checkpoint_hooks.common
module, 51	module, 85
edflow.data.believers.sequence	edflow.hooks.checkpoint_hooks.lambda_checkpoint_hook
module, 53	module, 88
edflow.data.dataset	edflow.hooks.checkpoint_hooks.tf_checkpoint_hook
module, 40	module, 89
edflow.data.dataset_mixin	edflow.hooks.checkpoint_hooks.torch_checkpoint_hook
module, 40	module, 93
edflow.data.processing	edflow.hooks.hook
module, 56	module, 78
edflow.data.processing.labels	edflow.hooks.logging_hooks
module, 56	module, 94
edflow.data.processing.processed	edflow.hooks.logging_hooks.minimal_logging_hook
module, 57	module, 94
edflow.data.util	edflow.hooks.logging_hooks.tensorboard_handler
module, 57	module, 95
edflow.data.util.cached_dset	edflow.hooks.logging_hooks.tf_logging_hook
module, 58	module, 95
edflow.data.util.util_dsets	edflow.hooks.logging_hooks.wandb_handler
module, 61	module, 97
edflow.datasets	edflow.hooks.metric_hooks
module, 65	module, 98
edflow.datasets.celeba	edflow.hooks.metric_hooks.tf_metric_hook
module, 65	module, 98
edflow.datasets.cifar	edflow.hooks.pytorch_hooks
module, 66	module, 80
edflow.datasets.fashionmnist	edflow.hooks.runtime_input
module, 66	module, 83
edflow.datasets.mnist	edflow.hooks.util_hooks
module, 67	module, 83
edflow.datasets.utils	edflow.iterators
module, 68	module, 99



edflow.iterators.batches  
     module, 99  
 edflow.iterators.model\_iterator  
     module, 100  
 edflow.iterators.resize  
     module, 102  
 edflow.iterators.template\_iterator  
     module, 103  
 edflow.iterators.tf\_batches  
     module, 104  
 edflow.iterators.tf\_evaluator  
     module, 105  
 edflow.iterators.tf\_iterator  
     module, 105  
 edflow.iterators.tf\_trainer  
     module, 106  
 edflow.iterators.torch\_iterator  
     module, 110  
 edflow.main  
     module, 37  
 edflow.metrics  
     module, 110  
 edflow.metrics.image\_metrics  
     module, 110  
 edflow.project\_manager  
     module, 38  
 edflow.util  
     module, 111  
 edflow.util.edexplore  
     module, 111  
 edflow.util.util  
     module, 113  
 edprint() (in module *edflow.util.util*), 119  
 emit() (*edflow.custom\_logging.TqdmHandler* method), 32  
 eval (*edflow.custom\_logging.run* attribute), 31  
 EvalHook (class in *edflow.eval.pipeline*), 74  
 ExampleConcatenatedDataset (class in *edflow.data.agnostics.concatenated*), 44  
 ExamplesFolder (class in *edflow.data.util.cached\_dset*), 58  
 exists (*edflow.custom\_logging.run* attribute), 32  
 expand() (*edflow.data.dataset\_mixin.DatasetMixin* property), 43  
 expand() (in module *edflow.data.agnostics.late\_loading*), 47  
 ExpandHook (class in *edflow.hooks.util\_hooks*), 83  
 explore() (in module *edflow.explore*), 37  
 extract\_features() (*edflow.applications.tf\_perceptual\_loss.VGG19Features* method), 38  
 ExtraLabelsDataset (class in *edflow.data.processing.labels*), 56

## F

FashionMNIST (class in *edflow.datasets.fashionmnist*), 67  
 FashionMNISTTest (class in *edflow.datasets.fashionmnist*), 67  
 FashionMNISTTrain (class in *edflow.datasets.fashionmnist*), 67  
 fcond() (*edflow.hooks.checkpoint\_hooks.common.WaitForCheckpointHook* method), 86  
 FILES (*edflow.datasets.celeba.CelebA* attribute), 65  
 FILES (*edflow.datasets.cifar.CIFAR10* attribute), 66  
 FILES (*edflow.datasets.fashionmnist.FashionMNIST* attribute), 67  
 FILES (*edflow.datasets.mnist.MNIST* attribute), 68  
 first\_index() (in module *edflow.util.edexplore*), 112  
 flatten\_results() (*edflow.hooks.checkpoint\_hooks.common.StoreArraysHook* method), 87  
 flow2hsv() (in module *edflow.data.util*), 63  
 flow2rgb() (in module *edflow.data.util*), 63  
 flow\_fn() (in module *edflow.data.util*), 64  
 fork\_safe\_zip() (*edflow.data.util.cached\_dset.CachedDataset* property), 60  
 ForkedPdb (class in *edflow.fpdb*), 37  
 format() (*edflow.custom\_logging.ColorLineFormatter* method), 33  
 from\_cache() (*edflow.data.util.cached\_dset.CachedDataset* class method), 60

## G

get\_batch\_step() (*edflow.iterators.model\_iterator.PyHookedModelIterator* method), 101  
 get\_checkpoint\_files() (in module *edflow.hooks.checkpoint\_hooks.common*), 87  
 get\_checkpoint\_variables() (*edflow.iterators.tf\_trainer.TFBaseTrainer* method), 108  
 get\_current\_train\_op() (*edflow.iterators.tf\_trainer.TFMultiStageTrainer* method), 109  
 get\_epoch\_step() (*edflow.iterators.model\_iterator.PyHookedModelIterator* method), 101  
 get\_example() (*edflow.data.agnostics.concatenated.DisjunctExampleConcatenatedDataset* method), 45  
 get\_example() (*edflow.data.agnostics.concatenated.ExampleConcatenatedDataset* method), 44  
 get\_example() (*edflow.data.agnostics.csv\_dset.CsvDataset* method), 46

[get\\_example\(\)](#) (*ed-flow.data.agnostics.late\_loading.LateLoadingDataset* method), 47  
[get\\_example\(\)](#) (*ed-flow.data.believers.meta.MetaDataset* method), 48  
[get\\_example\(\)](#) (*ed-flow.data.believers.meta\_view.MetaViewDataset* method), 53  
[get\\_example\(\)](#) (*ed-flow.data.believers.sequence.UnSequenceDataset* method), 55  
[get\\_example\(\)](#) (*ed-flow.data.dataset\_mixin.ConcatenatedDataset* method), 43  
[get\\_example\(\)](#) (*ed-flow.data.dataset\_mixin.DatasetMixin* method), 42  
[get\\_example\(\)](#) (*ed-flow.data.dataset\_mixin.SubDataset* method), 43  
[get\\_example\(\)](#) (*ed-flow.data.processing.labels.LabelDataset* method), 56  
[get\\_example\(\)](#) (*ed-flow.data.processing.processed.ProcessedDataset* method), 57  
[get\\_example\(\)](#) (*ed-flow.data.util.cached\_dset.CachedDataset* method), 60  
[get\\_example\(\)](#) (*ed-flow.data.util.util\_dsets.DataFolder* method), 63  
[get\\_example\(\)](#) (*ed-flow.data.util.util\_dsets.RandomlyJoinedDataset* method), 62  
[get\\_example\(\)](#) (*edflow.datasets.celeba.CeleBA* method), 65  
[get\\_example\(\)](#) (*edflow.datasets.cifar.CIFAR10* method), 66  
[get\\_example\(\)](#) (*edflow.datasets.fashionmnist.FashionMNIST* method), 67  
[get\\_example\(\)](#) (*edflow.datasets.mnist.MNIST* method), 68  
[get\\_example\(\)](#) (*edflow.debug.DebugDataset* method), 34  
[get\\_example\(\)](#) (*edflow.edsetup\_files.dataset.Dataset* method), 69  
[get\\_fixed\\_examples\(\)](#) (*ed-flow.iterators.template\_iterator.TemplateIterator* method), 103  
[get\\_global\\_step\(\)](#) (*ed-flow.iterators.model\_iterator.PyHookedModelIterator* method), 101  
[get\\_init\\_variables\(\)](#) (*ed-flow.iterators.tf\_trainer.TFBaseTrainer* method), 108  
[get\\_latest\\_checkpoint\(\)](#) (in module *ed-flow.hooks.checkpoint\_hooks.common*), 85  
[get\\_leaf\\_names\(\)](#) (in module *edflow.util.util*), 118  
[get\\_learning\\_rate\\_multiplier\(\)](#) (*ed-flow.iterators.tf\_trainer.TFListTrainer* method), 108  
[get\\_logger\(\)](#) (*edflow.custom\_logging.log* class method), 32  
[get\\_logger\(\)](#) (in module *edflow.custom\_logging*), 33  
[get\\_loss\(\)](#) (*edflow.hooks.checkpoint\_hooks.common.KeepBestCheckpoint* method), 88  
[get\\_obj\\_from\\_str\(\)](#) (in module *edflow.util.util*), 114  
[get\\_parser\(\)](#) (in module *edflow.args*), 30  
[get\\_restore\\_variables\(\)](#) (*ed-flow.iterators.tf\_trainer.TFBaseTrainer* method), 108  
[get\\_root\(\)](#) (in module *edflow.datasets.utils*), 69  
[get\\_sequence\\_view\(\)](#) (in module *ed-flow.data.believers.sequence*), 54  
[get\\_split\(\)](#) (*edflow.datasets.cifar.CIFAR10* method), 66  
[get\\_split\(\)](#) (*edflow.iterators.model\_iterator.PyHookedModelIterator* method), 101  
[get\\_str\\_from\\_obj\(\)](#) (in module *edflow.util.util*), 114  
[get\\_support\(\)](#) (in module *edflow.data.util*), 63  
[get\\_trainable\\_variables\(\)](#) (*ed-flow.iterators.tf\_trainer.TFBaseTrainer* method), 108  
[get\\_value\\_from\\_key\(\)](#) (in module *edflow.util.util*), 116  
[getDebugDataset\(\)](#) (in module *ed-flow.data.util.util\_dsets*), 61  
[getSeqDataset\(\)](#) (in module *ed-flow.data.believers.sequence*), 55  
[git\\_tag](#) (*edflow.custom\_logging.run* attribute), 31  
[global\\_step\(\)](#) (*ed-flow.hooks.checkpoint\_hooks.tf\_checkpoint\_hook.CheckpointHook* method), 91  
[grams\(\)](#) (*edflow.applications.tf\_perceptual\_loss.VGG19Features* method), 39  

## H

[heatmap\\_fn\(\)](#) (in module *edflow.data.util*), 64  
[Hook](#) (class in *edflow.hooks.hook*), 79  
[hsv2rgb\(\)](#) (in module *edflow.data.util*), 63  

## I

[im\\_fn\(\)](#) (in module *edflow.data.util*), 64

`image_grid()` (in module `edflow.iterators.tf_batches`), 104  
`image_loader()` (in module `edflow.data.believers.meta_loaders`), 50  
`image_saver()` (in module `edflow.eval.pipeline`), 77  
`ImageOverviewHook` (class in `edflow.hooks.logging_hooks.tf_logging_hook`), 96  
`increment_global_step()` (`edflow.iterators.model_iterator.PyHookedModelIterator` method), 101  
`init()` (`edflow.custom_logging.run` class method), 32  
`initialize()` (`edflow.iterators.model_iterator.PyHookedModelIterator` method), 102  
`initialize()` (`edflow.iterators.template_iterator.TemplateIterator` method), 103  
`initialize()` (`edflow.iterators.tf_evaluator.TFBaseEvaluator` method), 105  
`initialize()` (`edflow.iterators.tf_trainer.TFBaseTrainer` method), 107  
`input_names` (`edflow.hooks.checkpoint_hooks.common.MetricHook` attribute), 87  
`interaction()` (`edflow.fpdb.ForkedPdb` method), 37  
`IntervalHook` (class in `edflow.hooks.util_hooks`), 83  
`is_loadable()` (in module `edflow.eval.pipeline`), 77  
`is_prepared()` (in module `edflow.datasets.utils`), 69  
`isflow()` (in module `edflow.util.edexplore`), 111  
`isimage()` (in module `edflow.eval.pipeline`), 77  
`isimage()` (in module `edflow.util.edexplore`), 111  
`issegmentation()` (in module `edflow.util.edexplore`), 111  
`istext()` (in module `edflow.util.edexplore`), 112  
`iterate()` (`edflow.iterators.model_iterator.PyHookedModelIterator` method), 101  
`iterate()` (`edflow.iterators.tf_iterator.TFHookedModelIterator` method), 106  
`Iterator` (class in `edflow.edsetup_files.iterator`), 70  
**J**  
`JoinedDataset()` (in module `edflow.data.util.util_dsets`), 61  
**K**  
`KeepBestCheckpoints` (class in `edflow.hooks.checkpoint_hooks.common`), 87  
`KeyNotFoundError`, 115  
`keypoints_fn()` (in module `edflow.data.util`), 64  
**L**  
`l2_metric()` (in module `edflow.metrics.image_metrics`), 111  
`LabelDataset` (class in `edflow.data.processing.labels`), 56  
`labels()` (`edflow.data.agnostics.concatenated.ExampleConcatenatedDataset` property), 44  
`labels()` (`edflow.data.dataset_mixin.ConcatenatedDataset` property), 43  
`labels()` (`edflow.data.dataset_mixin.DatasetMixin` property), 43  
`labels()` (`edflow.data.dataset_mixin.SubDataset` property), 43  
`labels()` (`edflow.data.processing.labels.ExtraLabelsDataset` property), 56  
`labels()` (`edflow.data.util.cached_dset.CachedDataset` property), 60  
`labels()` (`edflow.data.util.util_dsets.RandomlyJoinedDataset` property), 62  
`labels()` (`edflow.debug.DebugDataset` property), 35  
`LambdaCheckpointHook` (class in `edflow.hooks.checkpoint_hooks.lambda_checkpoint_hook`), 88  
`LateLoadingDataset` (class in `edflow.data.agnostics.late_loading`), 46  
`LevelHook` (class in `edflow.hooks.checkpoint_hooks.level_hook`), 31  
`level` (`edflow.custom_logging.log` attribute), 32  
`linear_var()` (in module `edflow.util.util`), 114  
`load_callbacks()` (in module `edflow.eval.pipeline`), 77  
`load_image()` (in module `edflow.iterators.batches`), 99  
`load_labels()` (in module `edflow.data.believers.meta`), 49  
`loader_from_key()` (in module `edflow.data.believers.meta`), 49  
`log` (class in `edflow.custom_logging`), 32  
`log_figures()` (`edflow.hooks.logging_hooks.minimal_logging_hook.LoggingHook` method), 94  
`log_images()` (`edflow.hooks.logging_hooks.minimal_logging_hook.LoggingHook` method), 94  
`log_scalars()` (`edflow.hooks.logging_hooks.minimal_logging_hook.LoggingHook` method), 94  
`log_tensorboard_config()` (in module `edflow.hooks.logging_hooks.tensorboard_handler`), 95  
`log_tensorboard_figures()` (in module `edflow.hooks.logging_hooks.tensorboard_handler`), 95  
`log_tensorboard_images()` (in module `edflow.hooks.logging_hooks.tensorboard_handler`), 95  
`log_tensorboard_scalars()` (in module `edflow.hooks.logging_hooks.tensorboard_handler`), 95  
`log_wandb()` (in module `edflow.hooks.logging_hooks.wandb_handler`), 95



`flow.hooks.logging_hooks.wandb_handler`), 97  
`log_wandb_images()` (in module `ed-flow.hooks.logging_hooks.wandb_handler`), 97  
`loggers` (`edflow.custom_logging.log` attribute), 32  
`LoggingHook` (class in `ed-flow.hooks.logging_hooks.minimal_logging_hook`), 94  
`LoggingHook` (class in `ed-flow.hooks.logging_hooks.tf_logging_hook`), 96  
`LogSingleton` (in module `edflow.custom_logging`), 33  
`look()` (`edflow.hooks.checkpoint_hooks.common.WaitForCheckpointHook` method), 86

## M

`main()` (in module `edflow.eval.pipeline`), 78  
`make_batches()` (in module `ed-flow.iterators.batches`), 100  
`make_client_manager()` (in module `ed-flow.data.util.cached_dset`), 58  
`make_feature_ops()` (`ed-flow.applications.tf_perceptual_loss.VGG19Features` method), 39  
`make_feeds()` (`edflow.iterators.model_iterator.PyHookedModelIterator` method), 101  
`make_feeds()` (`edflow.iterators.tf_iterator.TFHookedModelIterator` method), 106  
`make_feeds()` (`edflow.iterators.tf_trainer.TFBaseTrainer` method), 107  
`make_iterator()` (in module `ed-flow.hooks.checkpoint_hooks.common`), 86  
`make_ll_nll_op()` (`ed-flow.applications.tf_perceptual_loss.VGG19Features` method), 39  
`make_linear_var()` (in module `ed-flow.iterators.tf_trainer`), 107  
`make_loss_op()` (`ed-flow.applications.tf_perceptual_loss.VGG19Features` method), 39  
`make_loss_ops()` (`ed-flow.iterators.tf_trainer.TFBaseTrainer` method), 107  
`make_nll_op()` (`ed-flow.applications.tf_perceptual_loss.VGG19Features` method), 39  
`make_run_once_op()` (`ed-flow.iterators.tf_trainer.TFBaseTrainer` method), 108  
`make_server_manager()` (in module `ed-flow.data.util.cached_dset`), 58  
`make_style_op()` (`ed-flow.applications.tf_perceptual_loss.VGG19Features` method), 39  
`mark_prepared()` (in module `edflow.datasets.utils`), 69  
`maybe_modify()` (`ed-flow.hooks.util_hooks.IntervalHook` method), 84  
`MetaDataset` (class in `edflow.data.believers.meta`), 48  
`MetaViewDataset` (class in `ed-flow.data.believers.meta_view`), 52  
`metric` (`edflow.hooks.checkpoint_hooks.common.MetricTuple` attribute), 87  
`MetricHook` (class in `ed-flow.hooks.metric_hooks.tf_metric_hook`), 87  
`MetricTuple` (class in `ed-flow.hooks.checkpoint_hooks.common`), 87  
`MNIST` (class in `edflow.datasets.mnist`), 68  
`MNISTTest` (class in `edflow.datasets.mnist`), 68  
`MNISTTrain` (class in `edflow.datasets.mnist`), 68  
`Model` (class in `edflow.edsetup_files.model`), 71

## module

`edflow`, 30  
`edflow.applications`, 38  
`edflow.applications.tf_perceptual_loss`, 38  
`edflow.args`, 30  
`edflow.config`, 39  
`edflow.config.commandline_kwarg`, 39  
`edflow.custom_logging`, 30  
`edflow.data`, 40  
`edflow.data.agnostics`, 44  
`edflow.data.agnostics.concatenated`, 44  
`edflow.data.agnostics.csv_dset`, 45  
`edflow.data.agnostics.late_loading`, 46  
`edflow.data.agnostics.subdataset`, 47  
`edflow.data.believers`, 47  
`edflow.data.believers.meta`, 47  
`edflow.data.believers.meta_loaders`, 49  
`edflow.data.believers.meta_util`, 51  
`edflow.data.believers.meta_view`, 51  
`edflow.data.believers.sequence`, 53  
`edflow.data.dataset`, 40  
`edflow.data.dataset_mixin`, 40  
`edflow.data.processing`, 56  
`edflow.data.processing.labels`, 56  
`edflow.data.processing.processed`, 57  
`edflow.data.util`, 57  
`edflow.data.util.cached_dset`, 58  
`edflow.data.util.util_dsets`, 61  
`edflow.datasets`, 65  
`edflow.datasets.celeba`, 65

edflow.datasets.cifar, 66  
 edflow.datasets.fashionmnist, 66  
 edflow.datasets.mnist, 67  
 edflow.datasets.utils, 68  
 edflow.debug, 34  
 edflow.edsetup\_files, 69  
 edflow.edsetup\_files.dataset, 69  
 edflow.edsetup\_files.iterator, 70  
 edflow.edsetup\_files.model, 71  
 edflow.eval, 71  
 edflow.eval.pipeline, 71  
 edflow.explore, 35  
 edflow.fpdb, 37  
 edflow.hooks, 78  
 edflow.hooks.checkpoint\_hooks, 85  
 edflow.hooks.checkpoint\_hooks.common, 85  
 edflow.hooks.checkpoint\_hooks.lambda\_checkpoint\_hook, 88  
 edflow.hooks.checkpoint\_hooks.tf\_checkpoint\_hook, 89  
 edflow.hooks.checkpoint\_hooks.torch\_checkpoint\_hook, 93  
 edflow.hooks.hook, 78  
 edflow.hooks.logging\_hooks, 94  
 edflow.hooks.logging\_hooks.minimal\_logging\_hook, 94  
 edflow.hooks.logging\_hooks.tensorboard\_handler, 95  
 edflow.hooks.logging\_hooks.tf\_logging\_hook, 95  
 edflow.hooks.logging\_hooks.wandb\_handler, 97  
 edflow.hooks.metric\_hooks, 98  
 edflow.hooks.metric\_hooks.tf\_metric\_hook, 98  
 edflow.hooks.pytorch\_hooks, 80  
 edflow.hooks.runtime\_input, 83  
 edflow.hooks.util\_hooks, 83  
 edflow.iterators, 99  
 edflow.iterators.batches, 99  
 edflow.iterators.model\_iterator, 100  
 edflow.iterators.resize, 102  
 edflow.iterators.template\_iterator, 103  
 edflow.iterators.tf\_batches, 104  
 edflow.iterators.tf\_evaluator, 105  
 edflow.iterators.tf\_iterator, 105  
 edflow.iterators.tf\_trainer, 106  
 edflow.iterators.torch\_iterator, 110  
 edflow.main, 37  
 edflow.metrics, 110  
 edflow.metrics.image\_metrics, 110  
 edflow.project\_manager, 38  
 edflow.util, 111  
 edflow.util.edexplore, 111  
 edflow.util.util, 113

## N

name (*edflow.custom\_logging.run attribute*), 31  
 NAME (*edflow.datasets.celeba.CelebA attribute*), 65  
 NAME (*edflow.datasets.cifar.CIFAR10 attribute*), 66  
 NAME (*edflow.datasets.fashionmnist.FashionMNIST attribute*), 67  
 NAME (*edflow.datasets.mnist.MNIST attribute*), 68  
 name (*edflow.hooks.checkpoint\_hooks.common.MetricTuple attribute*), 87  
 NoModel (*class in edflow.util.util*), 119  
 now (*edflow.custom\_logging.run attribute*), 31  
 np\_saver() (*in module edflow.eval.pipeline*), 77  
 numpy\_loader() (*in module edflow.data.util*), 50  
 other\_fn() (*in module edflow.data.util*), 64

## O

output\_names (*edflow.hooks.checkpoint\_hooks.common.MetricTuple attribute*), 87

## P

parse\_checkpoint() (*edflow.hooks.checkpoint\_hooks.torch\_checkpoint\_hook.RestorePytorchCheckpointHook static method*), 93  
 parse\_global\_step() (*edflow.hooks.checkpoint\_hooks.lambda\_checkpoint\_hook.LambdaCheckpointHook static method*), 89  
 parse\_global\_step() (*edflow.hooks.checkpoint\_hooks.tf\_checkpoint\_hook.RestoreModelHook static method*), 90  
 parse\_global\_step() (*edflow.hooks.checkpoint\_hooks.torch\_checkpoint\_hook.RestorePytorchCheckpointHook static method*), 93  
 parse\_unknown\_args() (*in module edflow.config.commandline\_kwargs*), 39  
 PathCachedDataset (*class in edflow.data.util.cached\_dset*), 60  
 pickle\_and\_queue() (*in module edflow.data.util.cached\_dset*), 58  
 plot\_batch() (*in module edflow.iterators.batches*), 99  
 plot\_datum() (*in module edflow.data.util*), 64  
 pop\_keypath() (*in module edflow.util.util*), 116  
 pop\_value\_from\_key() (*in module edflow.util.util*), 116  
 postfix (*edflow.custom\_logging.run attribute*), 31  
 pp2mktable() (*in module edflow.util.util*), 119  
 pprint() (*in module edflow.util.util*), 119  
 pprint\_str() (*in module edflow.util.util*), 119

```

preprocess_input() (in module ed-
    flow.applications.tf_perceptual_loss), 38
Printer (class in edflow.util.util), 118
prng() (edflow.util.util.PRNGMixin property), 118
PRNGMixin (class in edflow.util.util), 118
ProcessedDataset (class in ed-
    flow.data.processing.processed), 57
prompt_download() (in module ed-
    flow.datasets.utils), 69
PyCheckpointHook (class in ed-
    flow.hooks.pytorch_hooks), 80
PyHookedModelIterator (class in ed-
    flow.iterators.model_iterator), 100
PyLoggingHook (class in ed-
    flow.hooks.pytorch_hooks), 81

```

## Q

```

quadratic_crop() (in module edflow.datasets.utils),
    69

```

## R

```

RandomlyJoinedDataset (class in ed-
    flow.data.util.util_dsets), 61
read() (edflow.data.util.cached_dset.ExamplesFolder
    method), 58
read_mnist_file() (in module ed-
    flow.datasets.fashionmnist), 67
read_mnist_file() (in module ed-
    flow.datasets.mnist), 68
reporthook() (in module edflow.datasets.utils), 69
reset_global_step() (ed-
    flow.iterators.model_iterator.PyHookedModelIterator
    method), 101
resize_float32() (in module ed-
    flow.iterators.resize), 102
resize_hfloat32() (in module ed-
    flow.iterators.resize), 102
resize_image() (in module edflow.iterators.resize),
    102
resize_uint8() (in module edflow.iterators.resize),
    102
restore() (edflow.edsetup_files.iterator.Iterator
    method), 70
restore() (edflow.iterators.template_iterator.TemplateIter-
    ator method), 103
RestoreCurrentCheckpointHook (class in ed-
    flow.hooks.checkpoint_hooks.tf_checkpoint_hook), 92
RestoreModelHook (class in ed-
    flow.hooks.checkpoint_hooks.tf_checkpoint_hook), 90
RestorePytorchModelHook (class in ed-
    flow.hooks.checkpoint_hooks.torch_checkpoint_hook), 93
RestoreTFModelHook (in module ed-
    flow.hooks.checkpoint_hooks.tf_checkpoint_hook),
    90
resumed (edflow.custom_logging.run attribute), 31
RetrainHook (class in ed-
    flow.hooks.checkpoint_hooks.tf_checkpoint_hook),
    91
retrieve() (in module edflow.util.util), 115
root (edflow.custom_logging.run attribute), 31
root() (edflow.data.util.cached_dset.CachedDataset
    property), 60
run (class in edflow.custom_logging), 30
run() (edflow.iterators.model_iterator.PyHookedModelIterator
    method), 101
run() (edflow.iterators.tf_iterator.TFHookedModelIterator
    method), 106
run() (edflow.iterators.tf_trainer.TFBaseTrainer
    method), 108
run() (edflow.iterators.tf_trainer.TFFrequencyTrainer
    method), 108
run() (edflow.iterators.tf_trainer.TFListTrainer
    method), 108
run() (edflow.iterators.tf_trainer.TFMultiStageTrainer
    method), 109
run_condition() (ed-
    flow.hooks.util_hooks.IntervalHook method),
    84
run_hooks() (edflow.iterators.model_iterator.PyHookedModelIterator
    method), 101
RuntimeInputHook (class in ed-
    flow.hooks.runtime_input), 83

```

## S

```

save() (edflow.edsetup_files.iterator.Iterator method),
    70
save() (edflow.hooks.checkpoint_hooks.lambda_checkpoint_hook.Lambda
    method), 89
save() (edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.CheckpointH
    method), 91
save() (edflow.hooks.pytorch_hooks.PyCheckpointHook
    method), 81
save() (edflow.iterators.template_iterator.TemplateIterator
    method), 103
save_example() (in module edflow.eval.pipeline), 76
save_image() (in module edflow.iterators.batches),
    99
save_meta() (edflow.eval.pipeline.EvalHook
    method), 75
save_output() (in module edflow.eval.pipeline), 76
selector() (in module edflow.explore), 35
SequenceDataset (class in ed-
    flow.data.believers.sequence), 54
session() (edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.RestoreM
    property), 90

```

`session()` (*edflow.iterators.tf\_iterator.TFHookedModelIterator* attribute), 106  
`set_default()` (*in module edflow.util.util*), 116  
`set_example_pars()` (*ed-flow.hooks.checkpoint\_hooks.common*), 86  
`flow.data.agnostics.concatenated.ExampleConcatenatedDataset` (*in module edflow.util.util*), 118  
`method`, 44  
`SubDataset` (*class in edflow.data.dataset\_mixin*), 43  
`set_global_step()` (*ed-flow.iterators.model\_iterator.PyHookedModelIterator* method), 101  
`sup_str_to_num()` (*in module edflow.data.util*), 63  
  
**T**  
`set_log_level()` (*edflow.custom\_logging.log* class method), 33  
`set_log_target()` (*edflow.custom\_logging.log* class method), 32  
`set_value()` (*in module edflow.util.util*), 117  
`setup()` (*edflow.iterators.tf\_trainer.TFBaseTrainer* method), 107  
`setup_loaders()` (*in module ed-flow.data.believers.meta*), 48  
`show()` (*edflow.data.believers.meta.MetaDataset* method), 48  
`show_example()` (*in module edflow.explore*), 36  
`ShutdownRequest`, 100  
`ssim_metric()` (*in module ed-flow.metrics.image\_metrics*), 111  
`st_get_list_or_dict_item()` (*in module ed-flow.util.edexplore*), 112  
`stack_results()` (*ed-flow.hooks.checkpoint\_hooks.common.CollectorHook* method), 87  
`stage()` (*edflow.iterators.tf\_trainer.TFMultiStageModel* property), 110  
`stage_placeholder()` (*ed-flow.iterators.tf\_trainer.TFMultiStageModel* property), 109  
`stage_update_op()` (*ed-flow.iterators.tf\_trainer.TFMultiStageModel* property), 109  
`standalone_eval_meta_dset()` (*in module ed-flow.eval.pipeline*), 77  
`step_op()` (*edflow.edsetup\_files.iterator.Iterator* method), 70  
`step_op()` (*edflow.iterators.template\_iterator.TemplateIterator* method), 103  
`step_ops()` (*edflow.debug.DebugIterator* method), 34  
`step_ops()` (*edflow.iterators.model\_iterator.PyHookedModelIterator* method), 102  
`step_ops()` (*edflow.iterators.template\_iterator.TemplateIterator* method), 103  
`step_ops()` (*edflow.iterators.tf\_evaluator.TFBaseEvaluator* method), 105  
`step_ops()` (*edflow.iterators.tf\_trainer.TFBaseTrainer* method), 107  
`store_label_mmap()` (*in module ed-flow.data.believers.meta\_util*), 51  
  
`strenumerate()` (*in module ed-flow.hooks.checkpoint\_hooks.common*), 87  
`statefulDataset` (*in module edflow.util.util*), 118  
`SubDataset` (*class in edflow.data.dataset\_mixin*), 43  
`sup_str_to_num()` (*in module edflow.data.util*), 63  
  
`TablePrinter` (*class in edflow.util.util*), 119  
`target` (*edflow.custom\_logging.log* attribute), 32  
`TemplateEvalHook` (*class in edflow.eval.pipeline*), 75  
`TemplateIterator` (*class in ed-flow.iterators.template\_iterator*), 103  
`test()` (*in module edflow.main*), 38  
`test_valid_metrictuple()` (*in module ed-flow.hooks.checkpoint\_hooks.common*), 87  
`tf_batch_to_canvas()` (*in module ed-flow.iterators.tf\_batches*), 104  
`tf_parse_global_step()` (*in module ed-flow.hooks.checkpoint\_hooks.common*), 87  
`TFBaseEvaluator` (*class in ed-flow.iterators.tf\_evaluator*), 105  
`TFBaseTrainer` (*class in edflow.iterators.tf\_trainer*), 107  
`TFFrequencyTrainer` (*class in ed-flow.iterators.tf\_trainer*), 108  
`TFHookedModelIterator` (*class in ed-flow.iterators.tf\_iterator*), 106  
`TFListTrainer` (*class in edflow.iterators.tf\_trainer*), 108  
`TFMultiStageModel` (*class in ed-flow.iterators.tf\_trainer*), 109  
`TFMultiStageTrainer` (*class in ed-flow.iterators.tf\_trainer*), 108  
`tile()` (*in module edflow.iterators.batches*), 99  
`ToFromTorchHook` (*class in ed-flow.hooks.pytorch\_hooks*), 82  
`ToNumpyHook` (*class in edflow.hooks.pytorch\_hooks*), 81  
`torch_parse_global_step()` (*in module ed-flow.hooks.checkpoint\_hooks.common*), 87  
`TorchHookedModelIterator` (*class in ed-flow.iterators.torch\_iterator*), 110  
`TorchHook` (*class in edflow.hooks.pytorch\_hooks*), 82  
`TeamHandler` (*class in edflow.custom\_logging*), 32  
`train` (*edflow.custom\_logging.run* attribute), 31  
`train()` (*in module edflow.main*), 38  
  
**U**  
`unpack()` (*in module edflow.datasets.utils*), 69  
`UnSequenceDataset` (*class in ed-flow.data.believers.sequence*), 55

`update()` (in module `edflow.util.util`), 118  
`update_config()` (in module `edflow.config.commandline_kwargs`), 39  
`URL` (`edflow.datasets.celeba.CelebA` attribute), 65  
`URL` (`edflow.datasets.cifar.CIFAR10` attribute), 66  
`URL` (`edflow.datasets.fashionmnist.FashionMNIST` attribute), 67  
`URL` (`edflow.datasets.mnist.MNIST` attribute), 68

## V

`VGG19Features` (class in `edflow.applications.tf_perceptual_loss`), 38

## W

`wait()` (`edflow.hooks.checkpoint_hooks.tf_checkpoint_hook.WaitForManager` method), 92  
`WaitForCheckpointHook` (class in `edflow.hooks.checkpoint_hooks.common`), 85  
`WaitForManager` (class in `edflow.hooks.checkpoint_hooks.tf_checkpoint_hook`), 92  
`walk()` (in module `edflow.util.util`), 115