PyDLT Documentation

Release 0.0.6+b018f75

Demetris Marnerides

Jul 27, 2018

Quickstart

PyDLT is a PyTorch based Deep Learning Toolbox.

CHAPTER 1

Workflow Quickstart

```
import dlt
```

1.1 Logging

dlt.util.Logger can be used to create CSV logs.

```
log = dlt.util.Logger('losses', ['training', 'validation'])
log({'training': 10, 'validation':20})
```

1.2 Checkpointing

dlt.util.Checkpointer can be used to create checkpoints for any (torch serializable) objects.

```
data_chkp = Checkpointer('data')
a = np.array([1,2,3])
data_chkp.save(a)
a = None
a = data_chkp.load()
```

Checkpointers automatically save and load a network's state dict.

```
net_chkp = Checkpointer('model')
net = nn.Linear(10, 10)
data_chkp.save(net)
net = None
# net needs to be passed to the checkpointer to set the state dict.
data_chkp.load(net)
```

1.3 Built-in Trainers

The trainers (*dlt.train*) provide an easy to use interface for quick experimentation.

```
# Define a trainer
if use_gan:
    trainer = dlt.train.VanillaGANTrainer(generator, discriminator, g_optim, d_optim)
    log = dlt.util.Logger('gan_training', trainer.loss_names_training())
else:
    trainer = dlt.train.VanillaTrainer(generator, nn.MSE(), g_optim)
    log = dlt.util.Logger('normal_training', trainer.loss_names_training())

# Training is just iterating the trainer with a data loader.
# See the trainers documentation for more detail
for batch, (prediction, losses) in trainer(loader):
    log(losses)
```

1.4 Configuration

The package provides a parser, with some built-in command line arguments. These can be used to quickly configure experiments without much boilerplate code. Extra arguments can be added.

```
# file main.py
# Add some extra command line options to the built-in ones
dlt.config.add_extras([{'flag': '--extra_opt', 'default': 'foo'}])
# Parse
opt = dlt.config.parse(verbose=False)
# Can access built in options as well as the added ones
print('Some Settings: ', opt.experiment_name, opt.batch_size, opt.lr, opt.extra_opt)
```

Using configuration files makes bookkeeping a lot easier.

```
# file settings.cfg
# Can use commented lines
--experiment_name config_test
--lr 1e-4
--batch_size 64
# Can also set any extra settings
--extra_opt bar
```

Invoke the *main.py* script from above using these settings:

```
$ python main.py @settings.cfg
Some Settings:  config_test 64 0.0001 bar
```

The functions in *dlt.config* use the built-in arguments and can be configured from the command line.

Arguments belong to categories. Each category can be split into subsets.

```
# file main.py
# Split dataset settings
dlt.config.make_subsets({'dataset': ['set1', 'set2']})
# Training set
set1_data = dlt.config.torchvision_dataset(preprocess=cv2torch, subset='set1')
set1_loader = dlt.config.loader(set1_data)
```

(continues on next page)

(continued from previous page)

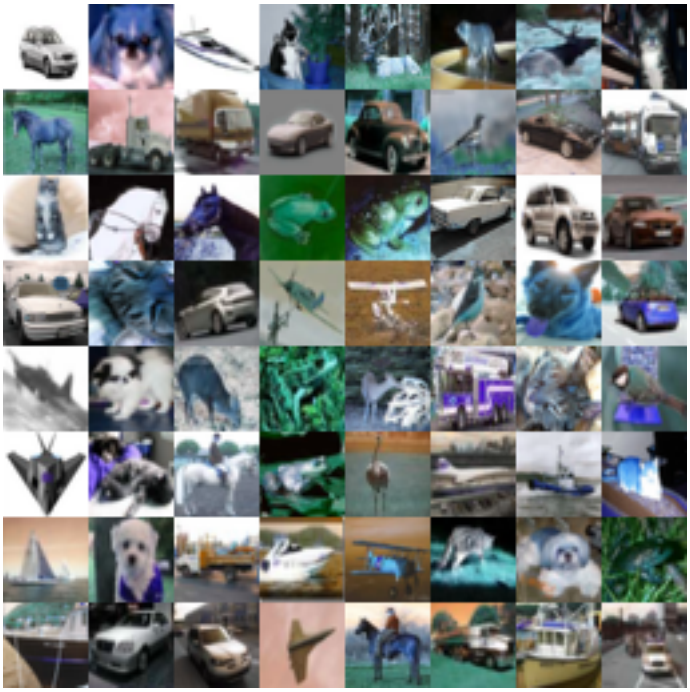
```
# Validation set
set2_data = dlt.config.torchvision_dataset(preprocess=cv2torch, subset='set2')
set2_loader = dlt.config.loader(set2_data)

for (img, lbl) in set1_loader:
    dlt.viz.imshow(dlt.util.make_grid(img, color=False), pause=2.0)
    break

for (img, lbl) in set2_loader:
    dlt.viz.imshow(dlt.util.make_grid(img, color=True), pause=2.0)
    break
```

```
# file settings.cfg
--batch_size 64
# We can set the split arguments like so:
# set1
--named_dataset_set1 mnist
--data_set1 ~/data/mnist
# set2
--named_dataset_set2 cifar10
--data_set2 ~/data/cifar10
```

```
$ python main.py @settings.cfg
```



The following is a configuration file template with the default values:

```
# general
--experiment_name experiment
--save_path .
--seed None
--max_epochs 100000
# dataset
--data .
--load_all False
--torchvision_dataset None
--extensions ['jpg']
# dataloader
--num_threads 4
--batch_size 1
--shuffle True
--pin_memory True
--drop_last False
# model
--overwrite_weights True
# optimizer
--optimizer adam
--lr 0.001
--momentum 0.9
--dampening 0.0
--beta1 0.9
--beta2 0.99
--rho 0.9
--alpha 0.99
--centered False
--lr_decay 0.0
--optim_eps 1e-08
--weight_decay 0.0
# scheduler
```

(continues on next page)

(continued from previous page)

```
--lr_schedule step
--lr_step_size 100
--lr_patience 10
--lr_cooldown 0
--lr_min 1e-07
--lr_ratio 0.5
# gpu
--use_gpu True
--device 0
--cudnn_benchmark True
# samples
--save_samples False
--sample_freq 1
```

1.5 Command Line Plotting

dlt-plot can be used from the command line to plot CSV files.

- Live update using the '-r'/'–refresh' flag (seconds).

```
dlt-plot -f training.csv -r 5
```

- Select columns using their number or name after the file.

```
dlt-plot -f averages.csv train_loss val_loss
```

- Plot multiple files/columns. This will plot columns 0 and 3 from file_1.csv and column 4 from file_2.csv

```
dlt-plot -f file_1.csv 0 3 -f file_2.csv 4
```

- Select the first/last points to plot using '–head'/'–tail' (or remove using '–rhead'/'–rtail).

```
dlt-plot -f data.csv --head 100
```

- Average/Variance every <N> points using '–sub_avg'/'–sub_var'.

```
dlt-plot -f training.csv --sub_avg 10
```

- Moving Average/Variance every <N> points using '–mov_avg'/'–mov_var'.

```
dlt-plot -f training.csv --mov_avg 10
```

- If multiple columns/files are used then the settings can be set for each individual column/file respectively. E.g. this will sub_avg training by 1000 and validation by 100:

```
dlt-plot -f training.csv -f validation.csv --sub_avg 1000 100
```

- NOTE: Transformations are applied in the order they are encountered. E.g. this will plot points 5-10:

```
dlt-plot -f training.csv --head 10 --tail 5
```

  whereas this will plot points end-10 to end-5:

```
dlt-plot -f training.csv --tail 10 --head 5
```

There are more available flags (logarithmic axes, setting titles, labels etc). For more information:

```
dlt-plot --help
```

1.6 Dispatching

dlt-dispatch can be used from the command line to create a self contained experiment in a directory. Useful for saving snapshots of code that is likely to change.

```
dlt-dispatch experiment_name -d ~/experiments -m main.py -c settings.cfg
```

The above code will create a directory ~/experiments/experiment_name containg main.py, settings.cfg (along with a run.sh script).

CHAPTER 2

Full Example

The following example implements a configurable training of GANs. It includes multiple GAN training types (Vanilla, WGAN-GP, FisherGAN, BEGAN) and multiple datasets (MNIST, FashionMNIST, CIFAR10/100). It can be extended relatively straightforwardly.

The code consists of two files, *main.py* and *models.py* along with a configuration file *settings.cfg*. To run:

```
$ python main.py @settings.cfg
```

It is worth noting:

- The networks are saved every epoch. Restarting continues from the previous checkpoint.
- Changing –experiment_save creates a new directory for the experiment with all the checkpoints and log.

main.py:

```
import torch
import dlt
from dlt.train import VanillaGANTrainer as GAN
from dlt.train import WGANGPTrainer as WGANGP
from dlt.train import WGANCTTrainer as WGANCT
from dlt.train import BEGANTrainer as BEGAN
from dlt.train import FisherGANTrainer as FisherGAN
from models import *

# Settings
dlt.config.make_subsets({'model': ['generator', 'discriminator'],
                         'optimizer': ['generator', 'discriminator']})
dlt.config.add_extras([
    dict(flag='--gan_type', choices=['vanilla', 'wgan-gp', 'wgan-ct', 'began',
↪'fishergan'],
                           default='vanilla', help='Gan type'),
    dict(flag='--num_hidden', type=int, default=64, help='Number of hidden units'),
    dict(flag='--z_dim', type=int, default=128, help='Input noise dimensionality'),
    dict(flag='--lambda_gp', type=float, default=10, help='Gradient penalty magnitude
↪'),
```

(continues on next page)

(continued from previous page)

```
    dict(flag='--m_ct', type=float, default=0.001, help='Constant bound for␣
↪consistency term for WGAN-CT'),
    dict(flag='--lambda_ct', type=float, default=0.001, help='Weight of consistency␣
↪term for WGAN-CT'),
    dict(flag='--lambda_k', type=float, default=0.001, help='Learning rate for k for␣
↪BEGAN'),
    dict(flag='--gamma', type=float, default=0.5, help='Gamma for BEGAN (diversity␣
↪ratio)'),
    dict(flag='--rho', type=float, default=1e-6, help='rho for FisherGAN'),
    dict(flag='--d_iter', type=int, default=2, help='Number of discriminator steps␣
↪per generator')
])
opt = dlt.config.parse(verbose=True)

# Configure seeds
if opt.seed is not None:
    torch.manual_seed(opt.seed)

# Data
sizes = {'mnist': (1, 28), 'fashionmnist': (1, 28),
         'cifar10': (3, 32), 'cifar100': (3, 32)}
if opt.torchvision_dataset not in sizes:
    raise ValueError('--torchvision_dataset must be one of {0}'.format(','.join(sizes.
↪keys())))
size = sizes[opt.torchvision_dataset]
def preprocess(datum):
    noise = torch.Tensor(opt.z_dim).uniform_(-1, 1)
    real_image = (dlt.util.cv2torch(datum[0]).float()/255.0) * 1.8 - 0.9
    # By convention, the trainer accepts the first point as the generator
    # input and the second as the real input for the discriminator
    return noise, real_image

dataset = dlt.config.torchvision_dataset()
loader = dlt.config.loader(dataset, preprocess)

# Models
generator = Generator(opt.num_hidden, opt.z_dim, size[0], size[1])
gen_chkp = dlt.config.model_checkpointer(generator, subset='generator')

if opt.gan_type == 'began':
    discriminator = DiscriminatorBEGAN(opt.num_hidden, size[0], size[1])
else:
    discriminator = Discriminator(opt.num_hidden, size[0], size[1])
disc_chkp = dlt.config.model_checkpointer(discriminator, subset='discriminator')

# Cudafy
if opt.use_gpu:
    torch.cuda.set_device(opt.device)
    torch.backends.cudnn.benchmark = opt.cudnn_benchmark
    generator.cuda()
    discriminator.cuda()

# Optimizers
g_optim = dlt.config.optimizer(generator, subset='generator')
g_optim_chkp = dlt.config.optimizer_checkpointer(g_optim, subset='generator')
d_optim = dlt.config.optimizer(discriminator, subset='discriminator')
d_optim_chkp = dlt.config.optimizer_checkpointer(d_optim, subset='discriminator')
```

(continues on next page)

(continued from previous page)

```
# Trainer
if opt.gan_type == 'wgan-gp':
    trainer = WGANGP(generator, discriminator, g_optim, d_optim, opt.lambda_gp, opt.d_
↪iter)
elif opt.gan_type == 'began':
    trainer = BEGAN(generator, discriminator, g_optim, d_optim, opt.lambda_k, opt.
↪gamma, opt.d_iter)
elif opt.gan_type == 'fishergan':
    trainer = FisherGAN(generator, discriminator, g_optim, d_optim, opt.rho, opt.d_
↪iter)
elif opt.gan_type == 'wgan-ct':
    trainer = WGANCT(generator, discriminator, g_optim, d_optim, opt.lambda_gp, opt.m_
↪ct, opt.lambda_ct, opt.d_iter)
else:
    trainer = GAN(generator, discriminator, g_optim, d_optim, opt.d_iter)

trainer_chkp = dlt.config.trainer_checkpointer(trainer)

if opt.use_gpu:
    trainer.cuda() # Trainers might have buffers that need to be transferred to GPU

# Logging
log = dlt.util.Logger('training', trainer.loss_names_training(), opt.save_path)

# Training loop
for epoch in range(trainer.epoch, opt.max_epochs):
    tag = 'epoch-{0}'.format(epoch)
    print('-'*79 + '\nEpoch {0}:'.format(epoch))
    # Set to training mode
    trainer.train()
    # The trainer iterator performs the optimization and gives predictions and
    # losses at each iteration
    for i, (batch, (prediction, losses)) in enumerate(trainer(loader)):
        # Show progress of each iteration and log the losses
        dlt.config.sample_images([batch[1], prediction], color=size[0] == 3,
                                 preprocess=dlt.util.map_range, tag=tag)
        log(losses)

    # Checkpoint everything
    gen_chkp(generator, tag=tag)
    disc_chkp(discriminator, tag=tag)
    g_optim_chkp(g_optim, tag=tag)
    d_optim_chkp(d_optim, tag=tag)
    trainer_chkp(trainer, tag=tag)
```

models.py:

```
from torch import nn

def selu_init(model):
    for m in model.modules():
        if any([isinstance(m, x) for x in [nn.Conv2d, nn.ConvTranspose2d, nn.
↪Linear]]):
            nn.init.kaiming_normal_(m.weight, 1)
            if m.bias is not None:
```

(continues on next page)

(continued from previous page)

```
                nn.init.constant_(m.bias, val=0)

class Generator(nn.Module):
    def __init__(self, num_hidden, z_dim, num_chan, num_pix):
        super(Generator, self).__init__()
        self.num_pix = num_pix
        self.num_chan = num_chan
        self.main = nn.Sequential(
            nn.Linear(z_dim, num_hidden),
            nn.SELU(),
            nn.Linear(num_hidden, num_hidden),
            nn.SELU(),
            nn.Linear(num_hidden, num_chan*num_pix*num_pix),
            nn.Tanh()
        )
        selu_init(self)

    def forward(self, v_input):
        return self.main(v_input).view(v_input.size(0), self.num_chan, self.num_pix,␣
↪self.num_pix)

class Discriminator(nn.Module):
    def __init__(self, num_hidden, num_chan, num_pix):
        super(Discriminator, self).__init__()
        self.num_pix = num_pix
        self.num_chan = num_chan
        self.main = nn.Sequential(
            nn.Linear(num_chan*num_pix*num_pix, num_hidden),
            nn.SELU(),
            nn.Linear(num_hidden, num_hidden),
            nn.SELU()
        )
        self.last_layer = nn.Linear(num_hidden, 1)
        selu_init(self)

    # The correction term is for WGAN-CT
    def forward(self, v_input, correction_term=False):
        if correction_term:
            main = self.main(v_input.view(v_input.size(0), -1))
            noisy_main = nn.functional.dropout(main, p=0.1)
            return main, self.last_layer(noisy_main)
        else:
            return self.last_layer(self.main(v_input.view(v_input.size(0), -1)))

# BEGAN needs an autoencoding discriminator
class DiscriminatorBEGAN(nn.Module):
    def __init__(self, num_hidden, num_chan, num_pix):
        super(DiscriminatorBEGAN, self).__init__()
        self.num_pix = num_pix
        self.num_chan = num_chan
        self.main = nn.Sequential(
            nn.Linear(num_chan*num_pix*num_pix, num_hidden), nn.SELU(),
            nn.Linear(num_hidden, num_hidden), nn.SELU(),
            nn.Linear(num_hidden, num_chan*num_pix*num_pix),
        )
        selu_init(self)
```

(continues on next page)

(continued from previous page)

```
    def forward(self, v_input):
        res = self.main(v_input.view(v_input.size(0), -1))
        return res.view(v_input.size(0), self.num_chan, self.num_pix, self.num_pix)
```

settings.cfg:

```
# general
--experiment_name mnist_wgan-gp
--save_path .
--seed 10
# dataset
--data ~/data/mnist
--torchvision_dataset mnist
# dataloader
--num_threads 0
--batch_size 128
--shuffle True
--pin_memory True
--drop_last False
# model
--overwrite_model_chkp_generator True
--overwrite_model_chkp_discriminator True
# optimizer
# Generator
--optimizer_generator adam
--lr_generator 1e-3
--beta1_generator 0.5
--beta2_generator 0.99
--weight_decay_generator 0.0
# Discriminator
--optimizer_discriminator adam
--lr_discriminator 5e-4
--beta1_discriminator 0.5
--beta2_discriminator 0.99
--weight_decay_discriminator 0.0
# gpu
--use_gpu True
--device 0
--cudnn_benchmark True
# samples
--save_samples True
--display_samples True
--sample_freq 50
# extras
--num_hidden 512
--z_dim 32
--gan_type vanilla
--d_iter 1
```

CHAPTER 3

Tensors and Arrays for Imaging

3.1 Views

We refer to an image's **view** as the order of the dimensions for its channels-width-height, e.g. **chw or hwc**, along with the ordering of the channel dimension, e.g. **RGB or BGR**.

The following commonly used packages have their own default **view** for images:

Package	Data Type	View
PyTorch	Torch Tensors/Variables	chw-RGB
OpenCV[1]	Numpy Arrays	hwc-BGR
PyPlot	Numpy Arrays	hwc-RGB

3.1.1 Type and View Conversions

dlt.util.change_view() can be used to convert between views. Please see the documentation for available views.

```
# Load image using OpenCV
img = cv2.imread('kodim16.png')
# Change the view and display with pyplot
plt_img = change_view(img, 'cv', 'plt')
plt.imshow(plt_img)
plt.show()
```

[1] OpenCV provides its own functionality for changing the channel ordering.

Functions such as *dlt.util.cv2torch()*, *dlt.util.torch2cv()* and *dlt.util.torch2plt()*, change the view as well as the data type. They also accept Variables or GPU Tensors as inputs and always return a Tensor or Array (copy) in main memory.

```
# Load image using OpenCV
img = cv2.imread('kodim16.png').astype('float32')/255.0
# Model to processes the image
net = nn.MaxPool2d(5,stride=2).cuda()
# Create input using cv2torch and get result
result = net(Variable(util.cv2torch(img).unsqueeze(0).cuda()))
```

3.2 Displaying

3.2.1 imshow

dlt.viz.imshow() can be used to display images. Accepts Arrays and Tensors as well as different views. The view must be provided as an argument.

```
# Load image using OpenCV
img = cv2.imread('kodim16.png')
# Display image for 5 seconds
dlt.viz.imshow(img, view='cv')
```

3.2.2 make_grid

dlt.util.make_grid() can be used to create grids. It can accept mixed lists of Arrays, Tensors and Variables, as well as different sized images (as long as they have the same view).

```
# Load image using OpenCV
img = cv2.imread('kodim16.png').astype('float32')/255.0
# Model to processes the image
net = nn.MaxPool2d(5,stride=2).cuda()
result = net(Variable(util.cv2torch(img).unsqueeze(0).cuda()))
# Make a grid with the images
input_img = dlt.util.change_view(img, 'cv', 'torch') # all must have the same view
viz.imshow(dlt.util.make_grid([input_img, result, net(result), net(net(result))]))
```

3.3 HDR

The functions *dlt.hdr.imread()* and *dlt.hdr.imwrite()* support all the OpenCV formats (including '.hdr' and '.exr') as well as '.pfm'.

CHAPTER 4

dlt.util

4.1 Functionals

4.1.1 compose

dlt.util.**compose**(*transforms*)

Composes list of transforms (each accept and return one item).

(From PyTorchNet)

Parameters **transforms** (*list*) – List of callables, each accepts and returns one item.

Returns The composed transforms.

Return type callable

4.1.2 applier

dlt.util.**applier**(*f*)

Returns a function that applies *f* to a collection of inputs (or just one).

Useful to use in conjuction with *dlt.util.compose()*

Parameters **f** (*function*) – Function to be applied.

Returns A function that applies 'f' to collections

Return type callable

Example

```
>>> pow2 = dlt.util.applier(lambda x: x**2)
>>> pow2(42)
1764
>>> pow2([1, 2, 3])
[1, 4, 9]
>>> pow2({'a': 1, 'b': 2, 'c': 3})
{'a': 1, 'b': 4, 'c': 9}
```

4.2 Quality of Life

4.2.1 Logger

class dlt.util.**Logger**(*name*, *fields*, *directory='.'*, *delimiter=','*, *resume=True*)

Logs values in a csv file.

Parameters

- **name** (*str*) – Filename without extension.
- **fields** (*list or tuple*) – Field names (column headers).
- **directory** (*str, optional*) – Directory to save file (default '.').
- **delimiter** (*str, optional*) – Delimiter for values (default ',').
- **resume** (*bool, optional*) – If True it appends to an already existing file (default True).

__call__(*values*)

Same as *log()*

log(*values*)

Logs a row of values.

Parameters **values** (*dict*) – Dictionary containing the names and values.

4.2.2 Checkpointer

class dlt.util.**Checkpointer**(*name*, *directory='.'*, *overwrite=True*, *verbose=True*, *timestamp=False*, *add_count=True*)

Checkpointer for objects using torch serialization.

Parameters

- **name** (*str*) – Name of the checkpointer. This will also be used for the checkpoint filename.
- **directory** (*str, optional*) – Parent directory of where the checkpoints will happen. A new sub-directory called checkpoints will be created (default '.').
- **overwrite** (*bool, optional*) – Overwrite/remove the previous checkpoint (default True).
- **verbose** (*bool, optional*) – Print a statement when loading a checkpoint (default True).
- **timestamp** (*bool, optional*) – Add a timestamp to checkpoint filenames (default False).

- **add_count** (*bool, optional*) – Add (zero-padded) counter to checkpoint filenames (default True).

Example

```
>>> a = {'data': 5}
>>> a_chkp = dlt.util.Checkpointer('a_saved')
>>> a_chkp.save(a)
>>> b = a_chkp.load()
{'data': 5}
```

It automatically saves and loads the *state_dict* of objects:

```
>>> net = nn.Sequential(nn.Linear(1,1), nn.Sigmoid())
>>> net_chkp = dlt.util.Checkpointer('net_state_dict')
>>> net_chkp.save(net)
>>> net_chkp.load(net)
Sequential(
    (0): Linear(in_features=1, out_features=1)
    (1): Sigmoid()
)
>>> state_dict = net_chkp.load()
OrderedDict([('0.weight',
-0.2286
[torch.FloatTensor of size 1x1]
), ('0.bias',
0.8495
[torch.FloatTensor of size 1]
)])
```

Warning: If a model is wrapped in *nn.DataParallel* then the wrapped model.module (state_dict) is saved. Thus applying *nn.DataParallel* must be done after using *Checkpointer.load()*.

__call__(*obj*, *tag=None*, **args*, ***kwargs*)
Same as *save()*

load(*obj=None*, *preprocess=None*, **args*, ***kwargs*)
Loads a checkpoint from disk.

Parameters

- **obj** (*optional*) – Needed if we load the *state_dict* of an *nn.Module*.
- **preprocess** (*optional*) – Callable to preprocess the loaded object.
- **args** – Arguments to pass to *torch.load*.
- **kwargs** – Keyword arguments to pass to *torch.load*.

Returns The loaded file.

save(*obj*, *tag=None*, **args*, ***kwargs*)
Saves a checkpoint of an object.

Parameters

- **obj** – Object to save (must be serializable by torch).

- **tag** (*str, optional*) – Tag to add to saved filename (default None).
- **args** – Arguments to pass to *torch.save*.
- **kwargs** – Keyword arguments to pass to *torch.save*.

4.2.3 ImageSampler

class dlt.util.**ImageSampler**(*name*, *directory='.'*, *overwrite=False*, *view='torch'*, *ext='.jpg'*, *color=True*, *size=None*, *inter_pad=None*, *fill_value=0*, *preprocess=None*, *display=False*, *save=True*, *sample_freq=1*)

Saves and/or displays image samples.

Parameters

- **name** (*str*) – Name of the checkpointer. This will also be used for the checkpoint filename.
- **directory** (*str, optional*) – Parent directory of where the samples will be saved. A new sub-directory called samples will be created (default '.').
- **overwrite** (*bool, optional*) – Overwrite/remove the previous checkpoint (default False).
- **view** (*str, optional*) – The image view e.g. 'hwc-bgr' or 'torch' (default 'torch').
- **ext** (*str, optional*) – The image format for the saved samples (default '.jpg').
- **color** (*bool, optional*) – Treat images as colored or not (default True).
- **size** (*list or tuple, optional*) – Grid dimensions, rows x columns. (default None).
- **inter_pad** (*python:int, optional*) – Padding separating the images (default None).
- **fill_value** (*python:int, optional*) – Fill value for inter-padding (default 0).
- **preprocess** (*callable, optional*) – Pre processing to apply to the image samples (default None).
- **display** (*bool, optional*) – Display images (default False).
- **save** (*bool, optional*) – Save images to disk (default True).
- **sample_freq** (*python:int, optional*) – Frequency of samples (per sampler call) (default 1).

__call__(*imgs*)

Same as *sample()*

sample(*imgs*)

Saves and/or displays functions depending on the configuration.

Parameters **imgs** (*Tensor, Array, list or tuple*) – Image samples. Will automatically be put in a grid. Must be in the [0,1] range.

4.2.4 Averages

class dlt.util.**Averages**(*names*)

Keeps multiple named averages.

Parameters **names** (*collection*) – Collection of strings to be used as names for the averages.

add(*values*, *count=1*)
Adds new values

Parameters

- **values** (*dict or list*) – Collection of values to be added. Could be given as a dict or a list. Order is preserved.
- **count** (*python:int, optional*) – Number of summed values that make the total values given. Can be used to register multiple (summed) values at once (default 1).

get(*names=None*, *ret_dict=False*)
Returns the current averages

Parameters

- **names** (*str or list, optional*) – Names of averages to be returned.
- **ret_dict** (*bool, optional*) – If true return the results in a dictionary, otherwise a list.

Returns The averages.

Return type dict or list

names()
Returns the names of the values held.

reset(*names=None*)
Resets averages to 0.

Parameters **names** (*collection, optional*) – Collection of the names to be reset. If None is given, all the values are reset (default None).

4.2.5 barit

dlt.util.**barit**(*iterable*, *start=None*, *end=None*, *time_it=True*, *length=20*, *leave=True*, *filler='='*)
Minimal progress bar for iterables.

Parameters

- **iterable** (*list or tuple etc*) – An iterable.
- **start** (*str, optional*) – String to place infront of the progress bar (default None).
- **end** (*str, optional*) – String to add at the end of the progress bar (default None).
- **timeit** (*bool, optional*) – Print elapsed time and ETA (default True).
- **length** (*python:int, optional*) – Length of the progress bar not including ends (default 20).
- **leave** (*bool, optional*) – If False, it deletes the progress bar once it ends (default True).
- **filler** (*str, optional*) – Filler character for the progress bar (default '=').

Example

```
>>> for _ in dlt.util.barit(range(100), start='Count'):
>>>     time.sleep(0.02)
Count: [====================] 100.0%, (10/10), Total: 0.2s, ETA: 0.0s
```

Note:

barit can be put to silent mode using:

```
>>> dlt.util.silent = True
```

For a progress bar with more functionality have a look at tqdm.

4.2.6 make_grid

dlt.util.**make_grid**(*images*, *view='torch'*, *color=True*, *size=None*, *inter_pad=None*, *fill_value=0*, *scale_each=False*)

Creates a single image grid from a set of images.

Parameters

- **images** (*Tensor, Array, list or tuple*) – Torch Tensor(s) and/or Numpy Array(s).
- **view** (*str, optional*) – The image view e.g. 'hwc-bgr' or 'torch' (default 'torch').
- **color** (*bool, optional*) – Treat images as colored or not (default True).
- **size** (*list or tuple, optional*) – Grid dimensions, rows x columns. (default None).
- **inter_pad** (*python:int or list/tuple, optional*) – Padding separating the images (default None).
- **fill_value** (*python:int, optional*) – Fill value for inter-padding (default 0).
- **scale_each** (*bool, optional*) – Scale each image to [0-1] (default False).

Returns The resulting grid. If any of the inputs is an Array then the result is an Array, otherwise a Tensor.

Return type Tensor or Array

Notes

- Images of **different sizes are padded** to match the largest.
- Works for **color** (3 channels) or **grey** (1 channel/0 channel) images.
- Images must have the same view (e.g. chw-rgb (torch))
- The Tensors/Arrays can be of **any dimension >= 2**. The last 2 (grey) or last 3 (color) dimensions are the images and all other dimensions are stacked. E.g. a 4x5x3x256x256 (torch view) input will be treated:
  - As 20 3x256x256 color images if color is True.
  - As 60 256x256 grey images if color is False.
- If color is False, then only the last two channels are considered (as hw) thus any colored images will be split into their channels.
- The image list can contain both **Torch Tensors and Numpy Arrays**. at the same time as long as they have the same view.

- If size is not given, the resulting grid will be the smallest square in which all the images fit. If the images are more than the given size then the default smallest square is used.

Raises

- `TypeError` – If images are not Arrays, Tensors, a list or a tuple
- `ValueError` – If channels or dimensions are wrong.

4.2.7 count_parameters

`dlt.util.`**`count_parameters`**(*net*)
Counts the parameters of a given PyTorch model.

4.2.8 accuracy

`dlt.util.`**`accuracy`**(*output*, *target*, *topk=1*)
Computes the precision@k for the specified values of k.

(From ImageNet example)

Args: output (Tensor): The class labels. target (Tensor): The predictions from the model. topk (int or collection): The specified values of k.

Returns: list: The top-k accuracy values.

4.3 Tensor/Array Operations

4.3.1 slide_window_

`dlt.util.`**`slide_window_`**(*a*, *kernel*, *stride=None*)
Expands last dimension to help compute sliding windows.

Parameters

- **`a`** (*`Tensor or Array`*) – The Tensor or Array to view as a sliding window.
- **`kernel`** (*`python:int`*) – The size of the sliding window.
- **`stride`** (*`tuple or python:int, optional`*) – Strides for viewing the expanded dimension (default 1)

The new dimension is added at the end of the Tensor or Array.

Returns The expanded Tensor or Array.

Running Sum Example:

```
>>> a = torch.Tensor([1, 2, 3, 4, 5, 6])
 1
 2
 3
 4
 5
 6
[torch.FloatTensor of size 6]
```

(continues on next page)

(continued from previous page)

```
>>> a_slided = dlt.util.slide_window_(a.clone(), kernel=3, stride=1)
 1  2  3
 2  3  4
 3  4  5
 4  5  6
[torch.FloatTensor of size 4x3]
>>> running_total = (a_slided*torch.Tensor([1,1,1])).sum(-1)
  6
  9
 12
 15
[torch.FloatTensor of size 4]
```

Averaging Example:

```
>>> a = torch.Tensor([1, 2, 3, 4, 5, 6])
 1
 2
 3
 4
 5
 6
[torch.FloatTensor of size 6]
>>> a_sub_slide = dlt.util.slide_window_(a.clone(), kernel=3, stride=3)
 1  2  3
 4  5  6
[torch.FloatTensor of size 2x3]
>>> a_sub_avg = (a_sub_slide*torch.Tensor([1,1,1])).sum(-1) / 3.0
 2
 5
[torch.FloatTensor of size 2]
```

4.3.2 re_stride

dlt.util.**re_stride**(*a*, *kernel*, *stride=None*)

Returns a re-shaped and re-strided Rensor given a kernel (uses as_strided).

Parameters

- **a** (*Tensor*) – The Tensor to re-stride.
- **kernel** (*tuple or python:int*) – The size of the new dimension(s).
- **stride** (*tuple or python:int, optional*) – Strides for viewing the expanded dimension(s) (default 1)

4.3.3 replicate

dlt.util.**replicate**(*x*, *dim=-3*, *nrep=3*)

Replicates Tensor/Array in a new dimension.

Parameters

- **x** (*Tensor or Array*) – Tensor to replicate.
- **dim** (*python:int, optional*) – New dimension where replication happens.

- **nrep** (*python:int, optional*) – Number of replications.

4.3.4 is_tensor

dlt.util.**is_tensor**(*x*)
Checks if input is a Tensor

4.3.5 is_cuda

dlt.util.**is_cuda**(*x*)
Checks if input is a cuda Tensor.

4.3.6 is_array

dlt.util.**is_array**(*x*)
Checks if input is a numpy array or a pandas Series.

4.3.7 to_array

dlt.util.**to_array**(*x*)
Converts x to a Numpy Array. Returns a copy of the data.

Parameters **x** (*Tensor or Array*) – Input to be converted. Can also be on the GPU.

Automatically gets the data from torch Tensors and casts GPU Tensors to CPU.

4.3.8 to_tensor

dlt.util.**to_tensor**(*x*)
Converts x to a Torch Tensor (CPU). Returns a copy of the data if x is a Tensor.

Parameters **x** (*Tensor or Array*) – Input to be converted. Can also be on the GPU.

Automatically casts GPU Tensors to CPU.

4.4 Image Conversions

4.4.1 permute

dlt.util.**permute**(*x*, *perm*)
Permutes the last three dimensions of the input Tensor or Array.

Parameters

- **x** (*Tensor or Array*) – Input to be permuted.
- **perm** (*tuple or list*) – Permutation.

Note: If the input has less than three dimensions a copy is returned.

4.4.2 hwc2chw

dlt.util.**hwc2chw**(*x*)
Permutes the last three dimensions of the hwc input to become chw.

Parameters **x** (*Tensor or Array*) – Input to be permuted.

4.4.3 chw2hwc

dlt.util.**chw2hwc**(*x*)
Permutes the last three dimensions of the chw input to become hwc.

Parameters **x** (*Tensor or Array*) – Input to be permuted.

4.4.4 channel_flip

dlt.util.**channel_flip**(*x*, *dim=-3*)
Reverses the channel dimension.

Parameters

- **x** (*Tensor or Array*) – Input to have its channels flipped.
- **dim** (*python:int, optional*) – Channels dimension (default -3).

Note: If the input has less than three dimensions a copy is returned.

4.4.5 rgb2bgr

dlt.util.**rgb2bgr**(*x*, *dim=-3*)
Reverses the channel dimension. See *channel_flip()*

4.4.6 bgr2rgb

dlt.util.**bgr2rgb**(*x*, *dim=-3*)
Reverses the channel dimension. See *channel_flip()*

4.4.7 change_view

dlt.util.**change_view**(*x*, *current*, *new*)
Changes the view of the input. Returns a copy.

Parameters

- **x** (*Tensor or Array*) – Input whose view is to be changed.
- **current** (*str*) – Current view.
- **new** (*str*) – New view.

Possible views:

View	Aliases
opencv	hwcbgr, hwc-bgr, bgrhwc, bgr-hwc, opencv, open-cv, cv, cv2
torch	chwrgb, chw-rgb, rgbchw, rgb-chw, torch, pytorch
plt	hwcrgb, hwc-rgb, rgbhwc, rgb-hwc, plt, pyplot, matplotlib
other	chwbgr, chw-bgr, bgrchw, bgr-chw

Note: If the input has less than three dimensions a copy is returned.

4.4.8 cv2torch

dlt.util.**cv2torch**(*x*)
Converts input to Tensor and changes view from cv (hwc-bgr) to torch (chw-rgb).

For more detail see *change_view()*

4.4.9 torch2cv

dlt.util.**torch2cv**(*x*)
Converts input to Array and changes view from torch (chw-rgb) to cv (hwc-bgr).

For more detail see *change_view()*

4.4.10 cv2plt

dlt.util.**cv2plt**(*x*)
Changes view from cv (hwc-bgr) to plt (hwc-rgb).

For more detail see *change_view()*

4.4.11 plt2cv

dlt.util.**plt2cv**(*x*)
Changes view from plt (hwc-rgb) to cv (hwc-bgr).

For more detail see *change_view()*

4.4.12 plt2torch

dlt.util.**plt2torch**(*x*)
Converts input to Tensor and changes view from plt (hwc-rgb) to torch (chw-rgb).

For more detail see *change_view()*

4.4.13 torch2plt

dlt.util.**torch2plt**(*x*)
Converts input to Array and changes view from torch (chw-rgb) to plt (hwc-rgb) .

For more detail see *change_view()*

4.5 Math

4.5.1 moving_avg

dlt.util.**moving_avg**(*x*, *width=5*)
Performes moving average of a one dimensional Tensor or Array

Parameters

- **x** (*Tensor or Array*) – 1D Tensor or array.
- **width** (*python:int, optional*) – Width of the kernel.

4.5.2 moving_var

dlt.util.**moving_var**(*x*, *width=5*)
Performes moving variance of a one dimensional Tensor or Array

Parameters

- **x** (*Tensor or Array*) – 1D Tensor or array.
- **width** (*python:int, optional*) – Width of the kernel.

4.5.3 sub_avg

dlt.util.**sub_avg**(*x*, *width=5*)
Performes averaging of a one dimensional Tensor or Array every *width* elements.

Parameters

- **x** (*Tensor or Array*) – 1D Tensor or array.
- **width** (*python:int, optional*) – Width of the kernel.

4.5.4 sub_var

dlt.util.**sub_var**(*x*, *width=5*)
Calculates variance of a one dimensional Tensor or Array every *width* elements.

Parameters

- **x** (*Tensor or Array*) – 1D Tensor or array.
- **width** (*python:int, optional*) – Width of the kernel.

4.5.5 has

4.5.6 has_nan

dlt.util.**has_nan**(*x*)
Checks if a Tensor/Array has NaNs.

4.5.7 has_inf

dlt.util.**has_inf**(*x*)
Checks if a Tensor/Array array has Infs.

4.5.8 replace_nan_

dlt.util.**replace_nan_**(*x*, *val=0*)
Replaces NaNs from a Numpy Array.

Parameters

- **x** (*Array*) – The Array (gets replaced in place).
- **val** (*python:int, optional*) – Value to replace Infs with (default 0).

4.5.9 replace_inf_

dlt.util.**replace_inf_**(*x*, *val=0*)
Replaces Infs from a Numpy Array.

Parameters

- **x** (*Array*) – The Array (gets replaced in place).
- **val** (*python:int, optional*) – Value to replace Infs with (default 0).

4.5.10 replace_specials_

dlt.util.**replace_specials_**(*x*, *val=0*)
Replaces NaNs and Infs from a Tensor/Array.

Parameters

- **x** (*Tensor or Array*) – The Tensor/Array (gets replaced in place).
- **val** (*python:int, optional*) – Value to replace NaNs and Infs with (default 0).

4.5.11 map_range

dlt.util.**map_range**(*x*, *low=0*, *high=1*)
Maps the range of a Numpy Array to [low, high] globally.

4.6 Convolution Layer Math

4.6.1 out_size

dlt.util.**out_size**(*dim_in*, *k*, *s*, *p*, *d*)

Calculates the resulting size after a convolutional layer.

Parameters

- **dim_in** (*python:int*) – Input dimension size.
- **k** (*python:int*) – Kernel size.
- **s** (*python:int*) – Stride of convolution.
- **p** (*python:int*) – Padding (of input).
- **d** (*python:int*) – Dilation

4.6.2 in_size

dlt.util.**in_size**(*dim_out*, *k*, *s*, *p*, *d*)

Calculates the input size before a convolutional layer.

Parameters

- **dim_out** (*python:int*) – Output dimension size.
- **k** (*python:int*) – Kernel size.
- **s** (*python:int*) – Stride of convolution.
- **p** (*python:int*) – Padding (of input).
- **d** (*python:int*) – Dilation

4.6.3 kernel_size

dlt.util.**kernel_size**(*dim_in*, *dim_out*, *s*, *p*, *d*)

Calculates the possible kernel size(s) of a convolutional layer given input and output.

Parameters

- **dim_in** (*python:int*) – Input dimension size.
- **dim_out** (*python:int*) – Output dimension size.
- **s** (*python:int*) – Stride of convolution.
- **p** (*python:int*) – Padding (of input).
- **d** (*python:int*) – Dilation

4.6.4 stride_size

dlt.util.**stride_size**(*dim_in*, *dim_out*, *k*, *p*, *d*)

Calculates the possible stride size(s) of a convolutional layer given input and output.

Parameters

- **dim_in** (*python:int*) – Input dimension size.
- **dim_out** (*python:int*) – Output dimension size.
- **k** (*python:int*) – Kernel size.
- **p** (*python:int*) – Padding (of input).
- **d** (*python:int*) – Dilation

4.6.5 padding_size

dlt.util.**padding_size**(*dim_in*, *dim_out*, *k*, *s*, *d*)

Calculates the possible padding size(s) of a convolutional layer given input and output.

Parameters

- **dim_in** (*python:int*) – Input dimension size.
- **dim_out** (*python:int*) – Output dimension size.
- **k** (*python:int*) – Kernel size.
- **s** (*python:int*) – Stride of convolution.
- **d** (*python:int*) – Dilation

4.6.6 dilation_size

dlt.util.**dilation_size**(*dim_in*, *dim_out*, *k*, *s*, *p*)

Calculates the possible dilation size(s) of a convolutional layer given input and output.

Parameters

- **dim_in** (*python:int*) – Input dimension size.
- **dim_out** (*python:int*) – Output dimension size.
- **k** (*python:int*) – Kernel size.
- **s** (*python:int*) – Stride of convolution.
- **p** (*python:int*) – Padding (of input).

4.6.7 find_layers

dlt.util.**find_layers**(*dims_in=None*, *dims_out=None*, *ks=None*, *ss=None*, *ps=None*, *ds=None*)

Calculates all the possible convolutional layer size(s) and parameters.

Parameters

- **dim_in** (*list*) – Input dimension sizes.
- **dim_out** (*list*) – Output dimension sizes.
- **k** (*list*) – Kernel sizes.
- **s** (*list*) – Strides of convolutions.
- **p** (*list*) – Paddings (of inputs).

4.7 Datasets

4.7.1 LoadedDataset

class dlt.util.**LoadedDataset**(*dataset*, *preprocess=None*)

Create a torch Dataset from data in memory with on the fly pre-processing.

Useful when to use with torch DataLoader.

Parameters

- **dataset** (*sequence or collection*) – A sequence or collection of data points that can be indexed.
- **preprocess** (*callable, optional*) – A function that takes a single data point from the dataset to preprocess on the fly (default None).

Example

```
>>> a = [1.0, 2.0, 3.0]
>>> a_dataset = dlt.util.LoadedDataset(a, lambda x: x**2)
>>> loader = torch.utils.data.DataLoader(a_dataset, batch_size=3)
>>> for val in loader:
>>>     print(val)
1
4
9
[torch.DoubleTensor of size 3]
```

4.7.2 DirectoryDataset

class dlt.util.**DirectoryDataset**(*data_root*, *extensions*, *load_fn*, *preprocess=None*)

Creates a dataset of images (no label) recursively (no structure requirement).

Similar to *torchvision.datasets.FolderDataset*, however there is no need for a specific directory structure, or data format.

Parameters

- **data_root** (*string*) – Path to root directory of data.
- **extensions** (*list or tuple*) – Extensions/ending patterns of data files.
- **loader** (*callable*) – Function that loads the data files.
- **preprocess** (*callable, optional*) – A function that takes a single data point from the dataset to preprocess on the fly (default None).

4.8 Sampling

4.8.1 index_gauss

dlt.util.**index_gauss**(*img*, *precision=None*, *crop_size=None*, *random_size=True*, *ratio=None*, *seed=None*)

Returns indices (Numpy slice) of an image crop sampled spatially using a gaussian distribution.

Parameters

- **img** (*Array*) – Image as a Numpy array (OpenCV view, hwc-BGR).
- **precision** (*list or tuple, optional*) – Floats representing the precision of the Gaussians (default [1, 4])
- **crop_size** (*list or tuple, optional*) – Ints representing the crop size (default [img_width/4, img_height/4]).
- **random_size** (*bool, optional*) – If true, randomizes the crop size with a minimum of crop_size. It uses an exponential distribution such that smaller crops are more likely (default True).
- **ratio** (*python:float, optional*) – Keep a constant crop ratio width/height (default None).
- **seed** (*python:float, optional*) – Set a seed for np.random.seed() (default None)

Note:

- If *ratio* is None then the resulting ratio can be anything.
- If *random_size* is False and *ratio* is not None, the largest dimension dictated by the ratio is adjusted accordingly:
  - *crop_size* is (w=100, h=10) and *ratio* = 9 ==> (w=90, h=10)
  - *crop_size* is (w=100, h=10) and *ratio* = 0.2 ==> (w=100, h=20)

4.8.2 slice_gauss

dlt.util.**slice_gauss**(*img*, *precision=None*, *crop_size=None*, *random_size=True*, *ratio=None*, *seed=None*)

Returns a cropped sample from an image array using *index_gauss()*

4.8.3 index_uniform

dlt.util.**index_uniform**(*img*, *crop_size=None*, *random_size=True*, *ratio=None*, *seed=None*)

Returns indices (Numpy slice) of an image crop sampled spatially using a uniform distribution.

Parameters

- **img** (*Array*) – Image as a Numpy array (OpenCV view, hwc-BGR).
- **crop_size** (*list or tuple, optional*) – Ints representing the crop size (default [img_width/4, img_height/4]).
- **random_size** (*bool, optional*) – If true, randomizes the crop size with a minimum of crop_size. It uses an exponential distribution such that smaller crops are more likely (default True).
- **ratio** (*python:float, optional*) – Keep a constant crop ratio width/height (default None).
- **seed** (*python:float, optional*) – Set a seed for np.random.seed() (default None)

Note:

- If *ratio* is None then the resulting ratio can be anything.
- If *random_size* is False and *ratio* is not None, the largest dimension dictated by the ratio is adjusted accordingly:
  - *crop_size* is (w=100, h=10) and *ratio* = 9 ==> (w=90, h=10)
  - *crop_size* is (w=100, h=10) and *ratio* = 0.2 ==> (w=100, h=20)

4.8.4 slice_uniform

dlt.util.**slice_uniform**(*img*, *crop_size=None*, *random_size=True*, *ratio=None*, *seed=None*)
Returns a cropped sample from an image array using *index_uniform()*

4.9 HPC

4.9.1 slurm

dlt.util.**slurm**(*code*, *directory='.'*, *name='job'*, *directives=None*)
Creates a script for the Slurm Scheduler.

Parameters

- **code** (*str*) – The code that is to be run from the script
- **directory** (*str, optional*) – The directory where the script is created (defult '.').
- **name** (*str, optional*) – Script filename (default 'job').
- **directives** (*dict*) – Set of directives to use (default None).

Available directives:

key	Default
–job-name	job
–time	48:00:00
–nodes	1
–ntasks-per-node	1
–mem-per-cpu	None
–mem	None
–partition	None
–gres	None
–exclude	None
–nodelist	None
–output	None
–mail-type	None
–mail-user	None

4.10 Misc

4.10.1 str2bool

dlt.util.**str2bool**(*x*)
Converts a string to boolean type.

If the string is any of ['no', 'false', 'f', '0'], or any capitalization, e.g. 'fAlSe' then returns False. All other strings are True.

4.10.2 str_is_int

dlt.util.**str_is_int**(*s*)
Checks if a string can be converted to int.

4.10.3 paths

dlt.util.paths.**split**(*directory*)
Splits a full filename path into its directory path, name and extension

Parameters **directory** (*str*) – Directory to split.

Returns (Directory name, filename, extension)

Return type tuple

dlt.util.paths.**make**(*directory*)
Make a new directory

Parameters **directory** (*str*) – Directory to make.

dlt.util.paths.**copy_to_dir**(*file*, *directory*)
Copies a file to a directory

Parameters

- **file** (*str*) – File to copy.
- **directory** (*str*) – Directory to copy file to.

dlt.util.paths.**process**(*directory*, *create=False*)
Expands home path, finds absolute path and creates directory (if create is True).

Parameters

- **directory** (*str*) – Directory to process.
- **create** (*bool, optional*) – If True, it creates the directory.

Returns The processed directory.

Return type str

dlt.util.paths.**write_file**(*contents*, *filename*, *directory='.'*, *append=False*)
Writes contents to file.

Parameters

- **contents** (*str*) – Contents to write to file.
- **filename** (*str*) – File to write contents to.

- **directory** (*str, optional*) – Directory to put file in.
- **append** (*bool, optional*) – If True and file exists, it appends contents.

Returns Full path to file.

Return type str

CHAPTER 5

dlt.config

5.1 Options

5.1.1 parse

dlt.config.**parse**(*verbose=False*)

Parses Command Line Arguments using the built-in settings (and any added extra).

Parameters **verbose** (*bool, optional*) – Print the parsed settings (default False).

Comes with built-in commonly used settings. To add extra settings use *add_extras()*.

For a comprehensive list of available settings, their categories and use of configuration files please see the *Workflow Quickstart* example.

5.1.2 add_extras

dlt.config.**add_extras**(*extras*)

Adds extra options for the parser, in addition to the built-in ones.

Args:

extras (list or dict, optional): Extra command line arguments to parse. If a list is given, a new category *extras* is added with the listed arguments. If a dict is given, the keys contain the category names and the elements are lists of arguments (default None).

Note: The *extras* parameter must have one of the following structures:

```
# List of arguments as dicts (this will add extras to )
extras = [dict(flag='--arg1', type=float),
          dict(flag='--arg2', type=int),
          dict(flag='--other_arg', type=int)]
# OR dictionary with category names in keys
# and lists of dicts for as values for each category
```

(continues on next page)

(continued from previous page)

```
extras = {'my_category': [dict(flag='--arg1', type=float),
                          dict(flag='--arg2', type=int)]
          'other_category': [dict(flag='--other_arg', type=int)]}
```

The keys accepted in the argument dictionaries are the ones used as arguments in the argparse package *add_argument* function.

DuplStdOut

Warning: The parser takes strings as inputs, so passing 'False' at the command for a bool argument, it will be converted to *True*. Instead of using type=bool, use type=dlt.util.str2bool.

5.1.3 make_subsets

dlt.config.**make_subsets**(*subsets*)

Splits command line argument categories into subsets.

The subset names are appended at the end of each of the categories options after an underscore. For example the dataset category can be split into training and validation subsets by passing the following as *subsets*:

```
subsets = {dataset=['training', 'validation']}
```

This will cause:

```
- `--data` to split into `--data_training` and `--data_validation`.
- `--load_all` to split into `--load_all_training` and `--load_all_validation`
- etc ...
```

Parameters **subsets** (*dict, optional*) – Dictionary containing parameter categories as keys and its subsets as a list of strings (default None).

5.1.4 print_opts

dlt.config.**print_opts**(*opt*)

Prints the parsed command line options.

5.2 Data

5.2.1 torchvision_dataset

dlt.config.**torchvision_dataset**(*transform=None*, *target_transform=None*, *train=True*, *subset=None*)

Creates a dataset from torchvision, configured using Command Line Arguments.

Parameters

- **transform** (*callable, optional*) – A function that transforms an image (default None).
- **target_transform** (*callable, optional*) – A function that transforms a label (default None).
- **train** (*bool, optional*) – Training set or validation - if applicable (default True).

- **subset** (*string, optional*) – Specifies the subset of the relevant categories, if any of them was split (default, None).

Relevant Command Line Arguments:

- **dataset**: *–data*, *–torchvision_dataset*.

Note: Settings are automatically acquired from a call to *dlt.config.parse()* from the built-in ones. If *dlt.config.parse()* was not called in the main script, this function will call it.

Warning: Unlike the torchvision datasets, this function returns a dataset that uses NumPy Arrays instead of a PIL Images.

5.2.2 directory_dataset

dlt.config.**directory_dataset**(*load_fn=<function imread>*, *preprocess=None*, *subset=None*)
Creates a *dlt.util.DirectoryDataset*, configured using Command Line Arguments.

Parameters

- **load_fn** (*callable, optional*) – Function that loads the data files (default *dlt.hdr.imread()*).
- **preprocess** (*callable, optional*) – A function that takes a single data point from the dataset to preprocess on the fly (default None).
- **subset** (*string, optional*) – Specifies the subset of the relevant categories, if any of them was split (default, None).

Relevant Command Line Arguments:

- **dataset**: *–data*, *–load_all*, *–extensions*.
- **dataloader**: *–num_threads*.

Note: Settings are automatically acquired from a call to *dlt.config.parse()* from the built-in ones. If *dlt.config.parse()* was not called in the main script, this function will call it.

5.2.3 loader

dlt.config.**loader**(*dataset*, *preprocess=None*, *subset=None*, *worker_init_fn=None*)
Creates a torch DataLoader using the dataset, configured using Command Line Arguments.

Parameters

- **dataset** (*Dataset*) – A torch compatible dataset.
- **preprocess** (*callable, optional*) – A function that takes a single data point from the dataset to preprocess on the fly (default None).
- **subset** (*string, optional*) – Specifies the subset of the relevant categories, if any of them was split (default, None).

Relevant Command Line Arguments:

- **dataloader**: *–batch_size*, *–num_threads*, *–pin_memory*, *–shuffle*, *–drop_last*.

Note: Settings are automatically acquired from a call to `dlt.config.parse()` from the built-in ones. If `dlt.config.parse()` was not called in the main script, this function will call it.

5.3 Models

5.3.1 model_checkpointer

dlt.config.**model_checkpointer**(*model*, *preprocess=None*, *subset=None*)

Returns the model checkpointer. Configurable using command line arguments.

The function also loads any previous checkpoints if present.

Parameters

- **model** (*nn.Module*) – The network for the checkpointer.
- **preprocess** (*callable, optional*) – Callable to change the loaded state dict before assigning it to the network.
- **subset** (*string, optional*) – Specifies the subset of the relevant categories, if any of them was split (default, None).

Relevant Command Line Arguments:

- **general**: *–experiment_name*, *–save_path*.
- **model**: *–overwrite_model_chkp*, *–timestamp_model_chkp*, *–count_model_chkp*.

Note: Settings are automatically acquired from a call to `dlt.config.parse()` from the built-in ones. If `dlt.config.parse()` was not called in the main script, this function will call it.

5.4 Optimization

5.4.1 optimizer

dlt.config.**optimizer**(*model*, *extra_params=None*, *subset=None*)

Returns the optimizer for the given model.

Parameters

- **model** (*nn.Module*) – The network for the optimizer.
- **extra_params** (*generator, optional*) – Extra parameters to pass to the optimizer.
- **subset** (*string, optional*) – Specifies the subset of the relevant categories, if any of them was split (default, None).

Relevant Command Line Arguments:

- **optimizer: *–optimizer*, *–lr*, *–momentum***, *–dampening*, *–beta1*, *–beta2*, *–weight_decay*.

Note: Settings are automatically acquired from a call to `dlt.config.parse()` from the built-in ones. If `dlt.config.parse()` was not called in the main script, this function will call it.

5.4.2 optimizer_checkpointer

dlt.config.**optimizer_checkpointer**(*optimizer*, *subset=None*)

Returns the optimizer checkpointer. Configurable using command line arguments.

The function also loads any previous checkpoints if present.

Parameters

- **optimizer** (*torch.optim.Optimizer*) – The optimizer.
- **subset** (*string, optional*) – Specifies the subset of the relevant categories, if any of them was split (default, None).

Relevant Command Line Arguments:

- **general**: *–experiment_name*, *–save_path*.
- **model: *–overwrite_optimizer_chkp*, *–timestamp_optimizer_chkp***, *–count_optimizer_chkp*.

Note: Settings are automatically acquired from a call to `dlt.config.parse()` from the built-in ones. If `dlt.config.parse()` was not called in the main script, this function will call it.

5.4.3 scheduler

dlt.config.**scheduler**(*optimizer*, *subset=None*)

Returns a scheduler callable closure which accepts one argument.

Configurable using command line arguments.

Parameters

- **optimizer** (*torch.optim.Optimizer*) – The optimizer for the scheduler.
- **subset** (*string, optional*) – Specifies the subset of the relevant categories, if any of them was split (default, None).

Relevant Command Line Arguments:

- **scheduler: *–lr_schedule*, *–lr_step_size*, *–lr_patience***, *–lr_cooldown*, *–lr_ratio*, *–lr_min*,

Note: Settings are automatically acquired from a call to `dlt.config.parse()` from the built-in ones. If `dlt.config.parse()` was not called in the main script, this function will call it.

5.5 Training

5.5.1 trainer_checkpointer

dlt.config.**trainer_checkpointer**(*trainer*, *subset=None*)

Returns the trainer checkpointer. Configurable using command line arguments.

The function also loads any previous checkpoints if present.

Parameters

- **trainer** (BaseTrainer) – The trainer for the checkpointer.
- **subset** (*string, optional*) – Specifies the subset of the relevant categories, if any of them was split (default, None).

Relevant Command Line Arguments:

- **general**: *–experiment_name*, *–save_path*.
- **trainer**: *–overwrite_trainer_chkp*, *–timestamp_trainer_chkp*, *–count_trainer_chkp*.

Note: Settings are automatically acquired from a call to *dlt.config.parse()* from the built-in ones. If *dlt.config.parse()* was not called in the main script, this function will call it.

CHAPTER 6

dlt.train

6.1 Base Classes

class dlt.train.**BaseTrainer**

Generic Base trainer object to inherit functionality from.

__call__(*loader*)

Performs an epoch of training or validation.

Parameters **loader** (*iterable*) – The data loader.

cpu(*device=0*)

Sets the trainer to CPU mode

cuda(*device=0*)

Sets the trainer to GPU mode.

If flagged, the data is cast to GPU before every iteration after being retrieved from the loader.

eval()

Sets the trainer and models to inference mode

iterate(*loader*)

Performs an epoch of training or validation.

Parameters **loader** (*iterable*) – The data loader.

load_state_dict(*state_dict*)

Loads the trainers state.

Parameters **state_dict** (*dict*) – scheduler state. Should be an object returned from a call to *state_dict()*.

loss_names(*training=None*)

Returns the name(s)/key(s) of the training or validation loss(es).

Parameters **training** (*bool, optional*) – If provided then the training or validation losses are returned if True or False respectively. If not provided the current mode loss is returned.

loss_names_training()
Returns the name(s)/key(s) of the training loss(es).

loss_names_validation()
Returns the name(s)/key(s) of the validation loss(es).

state_dict()
Returns the state of the trainer as a `dict`.

It contains an entry for every variable in self.__dict__ which is not the one of the models or optimizers.

train()
Sets the trainer and models to training mode

class dlt.train.**GANBaseTrainer**(*generator*, *discriminator*, *g_optimizer*, *d_optimizer*, *d_iter*)
Base Trainer to inherit functionality from for training Generative Adversarial Networks.

Parameters

- **generator** (*nn.Module*) – The generator network.
- **discriminator** (*nn.Module*) – The discriminator network.
- **g_optimizer** (*torch.optim.Optimizer*) – Generator Optimizer.
- **d_optimizer** (*torch.optim.Optimizer*) – Discriminator Optimizer.
- **d_iter** (*python:int*) – Number of discriminator steps per generator step.

Inherits from *dlt.train.BaseTrainer*

6.2 Vanilla

class dlt.train.**VanillaTrainer**(*model*, *criterion*, *optimizer*)
Training of a network using a criterion/loss function.

Parameters

- **model** (*nn.Module*) – The network to train.
- **criterion** (*callable*) – The function to optimize.
- **optimizer** (*torch.optim.Optimizer*) – A torch Optimizer.

Each iteration returns the mini-batch and a tuple containing:

- The model prediction.
- A dictionary with the *training_loss* or *validation_loss* (along with the partial losses, if criterion returns a dictionary).

Example

```
>>> trainer = dlt.train.VanillaTrainer(my_model, nn.L1Loss(), my_optimizer)
>>> # Training mode
>>> trainer.train()
```

(continues on next page)

(continued from previous page)

```
>>> for batch, (prediction, loss) in trainer(train_data_loader):
>>>     print(loss['training_loss'])
>>> # Validation mode
>>> trainer.eval()
>>> for batch, (prediction, loss) in trainer(valid_data_loader):
>>>     print(loss['validation_loss'])
```

Note: If the criterion returns a dict of (named) losses, then they are added together to backpropagate. The total is returned along with all the partial losses.

6.3 Vanilla GAN

class dlt.train.**VanillaGANTrainer**(*generator*, *discriminator*, *g_optimizer*, *d_optimizer*, *d_iter=1*)

Generative Adversarial Networks trainer.

Parameters

- **generator** (*nn.Module*) – The generator network.
- **discriminator** (*nn.Module*) – The discriminator network.
- **g_optimizer** (*torch.optim.Optimizer*) – Generator Optimizer.
- **d_optimizer** (*torch.optim.Optimizer*) – Discriminator Optimizer.
- **d_iter** (*python:int, optional*) – Number of discriminator steps per generator step (default 1).

Each iteration returns the mini-batch and a tuple containing:

- The generator prediction.
- A dictionary containing a *d_loss* (not when validating) and a *g_loss* dictionary (only if a generator step is performed):
  - *d_loss contains*: *d_loss*, *real_loss*, and *fake_loss*.
  - *g_loss* contains: *g_loss*.

Example

```
>>> trainer = dlt.train.VanillaGANTrainer(gen, disc, g_optim, d_optim)
>>> # Training mode
>>> trainer.train()
>>> for batch, (prediction, loss) in trainer(train_data_loader):
>>>     print(loss['d_loss']['d_loss'])
```

Warning: This trainer uses BCEWithLogitsLoss, which means that the discriminator must NOT have a sigmoid at the end.

6.4 WGAN-GP

class dlt.train.**WGANGPTrainer**(*generator*, *discriminator*, *g_optimizer*, *d_optimizer*, *lambda_gp*, *d_iter=1*)

Wasserstein GAN Trainer with gradient penalty.

Parameters

- **generator** (*nn.Module*) – The generator network.
- **discriminator** (*nn.Module*) – The discriminator network.
- **g_optimizer** (*torch.optim.Optimizer*) – Generator Optimizer.
- **d_optimizer** (*torch.optim.Optimizer*) – Discriminator Optimizer.
- **lambda_gp** (*python:float*) – Weight of gradient penalty.
- **d_iter** (*python:int, optional*) – Number of discriminator steps per generator step (default 1).

Each iteration returns the mini-batch and a tuple containing:

- The generator prediction.
- A dictionary containing a *d_loss* (not when validating) and a *g_loss* dictionary (only if a generator step is performed):
  - *d_loss contains*: *d_loss*, *w_loss*, and *gp*.
  - *g_loss* contains: *g_loss*.

Example

```
>>> trainer = dlt.train.WGANGPTrainer(gen, disc, g_optim, d_optim, lambda_gp)
>>> # Training mode
>>> trainer.train()
>>> for batch, (prediction, loss) in trainer(train_data_loader):
>>>     print(loss['d_loss']['w_loss'])
```

6.5 WGAN-CT

class dlt.train.**WGANCTTrainer**(*generator*, *discriminator*, *g_optimizer*, *d_optimizer*, *lambda_gp*, *m_ct*, *lambda_ct*, *d_iter=1*)

Wasserstein GAN Trainer with gradient penalty and correction term.

From Improving the Improved Training of Wasserstein GANs: A Consistency Term and Its Dual Effect.

https://openreview.net/forum?id=SJx9GQb0-

Parameters

- **generator** (*nn.Module*) – The generator network.
- **discriminator** (*nn.Module*) – The discriminator network.
- **g_optimizer** (*torch.optim.Optimizer*) – Generator Optimizer.
- **d_optimizer** (*torch.optim.Optimizer*) – Discriminator Optimizer.

- **lambda_gp** (*python:float*) – Weight of gradient penalty.
- **m_ct** (*python:float*) – Constant bound for consistency term.
- **lambda_ct** (*python:float*) – Weight of consistency term.
- **d_iter** (*python:int, optional*) – Number of discriminator steps per generator step (default 1).

Each iteration returns the mini-batch and a tuple containing:

- The generator prediction.
- A dictionary containing a *d_loss* (not when validating) and a *g_loss* dictionary (only if a generator step is performed):
  - *d_loss contains*: *d_loss*, *w_loss*, *gp* and *ct*.
  - *g_loss* contains: *g_loss*.

Warning: The discriminator forward function needs to be able to accept an optional bool argument *correction_term*. When set to true, the forward function must add dropout noise to the model and return a tuple containing the second to last output of the discriminator along with the final output.

Example

```
>>> trainer = dlt.train.WGANCTTrainer(gen, disc, g_optim, d_optim, lambda_gp, m_
↪ct, lambda_ct)
>>> # Training mode
>>> trainer.train()
>>> for batch, (prediction, loss) in trainer(train_data_loader):
>>>     print(loss['d_loss']['w_loss'])
```

6.6 BEGAN

class dlt.train.**BEGANTrainer**(*generator*, *discriminator*, *g_optimizer*, *d_optimizer*, *lambda_k*, *gamma*, *d_iter=1*)

Boundary Equilibrium GAN trainer.

Parameters

- **generator** (*nn.Module*) – The generator network.
- **discriminator** (*nn.Module*) – The discriminator network.
- **g_optimizer** (*torch.optim.Optimizer*) – Generator Optimizer.
- **d_optimizer** (*torch.optim.Optimizer*) – Discriminator Optimizer.
- **lambda_k** (*python:float*) – Learning rate of k parameter.
- **gamma** (*python:float*) – Diversity ratio.
- **d_iter** (*python:int*) – Number of discriminator steps per generator step.

Each iteration returns the mini-batch and a tuple containing:

- The generator prediction.

- A dictionary containing a *d_loss* (not when validating) and a *g_loss* dictionary (only if a generator step is performed):
  - *d_loss contains*: *d_loss*, *real_loss*, *fake_loss*, *k*, *balance*, and *measure*.
  - *g_loss* contains: *g_loss*.

Example:

```
>>> trainer = dlt.train.BEGANTrainer(gen, disc, g_optim, d_optim, lambda_k, gamma)
>>> # Training mode
>>> trainer.train()
>>> for batch, (prediction, loss) in trainer(train_data_loader):
>>>     print(loss['d_loss']['measure'])
```

6.7 Fisher-GAN

class dlt.train.**FisherGANTrainer**(*generator*, *discriminator*, *g_optimizer*, *d_optimizer*, *rho*, *d_iter=1*)

Fisher GAN trainer.

Parameters

- **generator** (*nn.Module*) – The generator network.
- **discriminator** (*nn.Module*) – The discriminator network.
- **g_optimizer** (*torch.optim.Optimizer*) – Generator Optimizer.
- **d_optimizer** (*torch.optim.Optimizer*) – Discriminator Optimizer.
- **rho** (*python:float*) – Quadratic penalty weight.
- **d_iter** (*python:int, optional*) – Number of discriminator steps per generator step (default 1).

Each iteration returns the mini-batch and a tuple containing:

- The generator prediction.
- A dictionary containing a *d_loss* (not when validating) and a *g_loss* dictionary (only if a generator step is performed):
  - *d_loss contains*: *ipm_enum*, *ipm_denom*, *ipm_ratio*, *d_loss*, *constraint*, *epf*, *eqf*, *epf2*, *eqf2* and *lagrange*.
  - *g_loss* contains: *g_loss*.

Example

```
>>> trainer = dlt.train.FisherGANTrainer(gen, disc, g_optim, d_optim, rho)
>>> # Training mode
>>> trainer.train()
>>> for batch, (prediction, loss) in trainer(train_data_loader):
>>>     print(loss['d_loss']['constraint'])
```

CHAPTER 7

dlt.viz

7.1 Images

7.1.1 imshow

dlt.viz.**imshow**(*img*, *view='torch'*, *figure=None*, *pause=0*, *title=None*, *interactive=False*, **args*, ***kwargs*)

Displays a Tensor or Array image to screen.

Parameters

- **img** (*Tensor or Array*) – Image to display.
- **view** (*str, optional*) – View of image. For more details see *dlt.util.change_view()* (default 'torch').
- **figure** (*python:int, optional*) – Use selected figure (default None).
- **pause** (*python:float, optional*) – Number of seconds to pause execution for displaying when interactive is True. If a value less than 1e-2 is given then it defaults to 1e-2 (default 1e-2).
- **title** (*str, optional*) – Title for figure (default None).
- **interactive** (*bool, optional*) – If the image will be updated; uses plt.ion() (default False).
- ***args** (*optional*) – Extra arguments to be passed to plt.imshow().
- ****kwargs** (*optional*) – Extra keyword arguments to be passed to plt.imshow().

Example

```
>>> for video_1_frame, video_2_frame in two_videos_frames:
>>>     dlt.viz.imshow(video_1_frame, view='cv', figure=1, interactive=True,␣
↪title='Video 1')
>>>     dlt.viz.imshow(video_2_frame, view='cv', figure=2, interactive=True,␣
↪title='Video 2')
```

7.2 Models

7.2.1 forward

dlt.viz.modules.**forward_hook**(*net*, *modules=None*, *match_names=None*, *do_input=False*, *do_output=True*, *tag=''*, *save_path='.'*, *replace=True*, *histogram=True*, *bins=100*)

Registers a forward hook to a network's modules for vizualization of the inputs and outputs.

When net.forward() is called, the hook saves an image grid or a histogram of input/output of the specified modules.

Parameters

- **net** (*nn.Module*) – The network whose modules are to be visualized.
- **modules** (*list or tuple, optional*) – List of class definitions for the modules where the hook is attached e.g. nn.Conv2d (default None).
- **match_names** (*list or tuple, optional*) – List of strings. If any modules contain one of the strings then the hook is attached (default None).
- **do_input** (*bool, optional*) – If True the input of the module is visualized (default False).
- **do_output** (*bool, optional*) – If True the output of the module is visualized (default True).
- **tag** (*str, optional*) – String tag to attach to saved images (default None).
- **save_path** (*str, optional*) – Path to save visualisation results (default '.').
- **replace** (*bool, optional*) – If True, the images (from the same module) are replaced whenever the hook is called (default True).
- **histogram** (*bool, optional*) – If True then the visualization is a histrogram, otherwise it's an image grid.
- **bins** (*bool, optional*) – Number of bins for histogram, if *histogram* is True (default 100).

Note:

- If modules or match_names are not provided then no hooks will be attached.

7.2.2 backward

dlt.viz.modules.**backward_hook**(*net*, *modules=None*, *match_names=None*, *do_grad_input=False*, *do_grad_output=True*, *tag=''*, *save_path='.'*, *replace=True*, *histogram=True*, *bins=100*)

Registers a backward hook to a network's modules for vizualization of the gradients.

When net.backward() is called, the hook saves an image grid or a histogram of grad_input/grad_output of the specified modules.

Parameters

- **net** (*nn.Module*) – The network whose gradients are to be visualized.
- **modules** (*list or tuple, optional*) – List of class definitions for the modules where the hook is attached e.g. nn.Conv2d (default None).
- **match_names** (*list or tuple, optional*) – List of strings. If any modules contain one of the strings then the hook is attached (default None).
- **do_grad_input** (*bool, optional*) – If True the grad_input of the module is visualized (default False).
- **do_grad_output** (*bool, optional*) – If True the grad_output of the module is visualized (default True).
- **tag** (*str, optional*) – String tag to attach to saved images (default None).
- **save_path** (*str, optional*) – Path to save visualisation results (default '.').
- **replace** (*bool, optional*) – If True, the images (from the same module) are replaced whenever the hook is called (default True).
- **histogram** (*bool, optional*) – If True then the visualization is a histrogram, otherwise it's an image grid.
- **bins** (*bool, optional*) – Number of bins for histogram, if *histogram* is True (default 100).

Note:

- If modules or match_names are not provided then no hooks will be attached.

7.2.3 parameters_hooked

dlt.viz.modules.**parameters_hook**(*net*, *modules=None*, *match_names=None*, *param_names=None*, *tag=''*, *save_path='.'*, *replace=True*, *histogram=True*, *bins=100*)

Registers a forward hook to a network's modules for vizualization of its parameters.

When net.forward() is called, the hook saves an image grid or a histogram of the parameters of the specified modules.

Parameters

- **net** (*nn.Module*) – The network whose parameters are to be visualized.
- **modules** (*list or tuple, optional*) – List of class definitions for the modules where the hook is attached e.g. nn.Conv2d (default None).

- **match_names** (*list or tuple, optional*) – List of strings. If any modules contain one of the strings then the hook is attached (default None).
- **param_names** (*list or tuple, optional*) – List of strings. If any parameters of the module contain one of the strings then they are visualized (default None).
- **tag** (*str, optional*) – String tag to attach to saved images (default None).
- **save_path** (*str, optional*) – Path to save visualisation results (default '.').
- **replace** (*bool, optional*) – If True, the images (from the same module) are replaced whenever the hook is called (default True).
- **histogram** (*bool, optional*) – If True then the visualization is a histrogram, otherwise it's an image grid.
- **bins** (*bool, optional*) – Number of bins for histogram, if *histogram* is True (default 100).

Note:

- If modules or match_names are not provided then no hooks will be attached.
- If param_names are not provided then no parameters will be visualized.

7.2.4 parameters_once

dlt.viz.modules.**parameters**(*net*, *modules=None*, *match_names=None*, *param_names=None*, *tag=''*, *save_path='.'*, *histogram=True*, *bins=100*)

Visualizes a network's parameters on an image grid or histogram.

Parameters

- **net** (*nn.Module*) – The network whose parameters are to be visualized.
- **modules** (*list or tuple, optional*) – List of class definitions for the modules where the hook is attached e.g. nn.Conv2d (default None).
- **match_names** (*list or tuple, optional*) – List of strings. If any modules contain one of the strings then the hook is attached (default None).
- **param_names** (*list or tuple, optional*) – List of strings. If any parameters of the module contain one of the strings then they are visualized (default None).
- **tag** (*str, optional*) – String tag to attach to saved images (default None).
- **save_path** (*str, optional*) – Path to save visualisation results (default '.').
- **histogram** (*bool, optional*) – If True then the visualization is a histrogram, otherwise it's an image grid.
- **bins** (*bool, optional*) – Number of bins for histogram, if *histogram* is True (default 100).

Note:

- If modules or match_names are not provided then no parameters will be visualized.
- If param_names are not provided then no parameters will be visualized.

CHAPTER 8

dlt.hdr

8.1 imread

dlt.hdr.**imread**(*filename*)

Reads an image file from disk into a Numpy Array (OpenCV view).

Parameters **filename** (*str*) – Name of pfm image file.

8.2 imwrite

dlt.hdr.**imwrite**(*filename*, *img*, **args*, ***kwargs*)

Writes an image to disk. Supports HDR and LDR image formats.

Parameters

- **filename** (*str*) – Name of image file.
- **img** (*Array*) – Numpy Array containing the image (OpenCV view hwc-BGR).
- ***args** – Extra arguments to pass to cv2.imwrite or write_pfm if saving a .pfm image.
- ****kwargs** – Extra keyword arguments to pass to cv2.imwrite or write_pfm if saving a .pfm image.

8.3 load_pfm

dlt.hdr.**load_pfm**(*filename*)

Loads a pfm image file from disk into a Numpy Array (OpenCV view).

Supports HDR and LDR image formats.

Parameters **filename** (*str*) – Name of pfm image file.

8.4 write_pfm

dlt.hdr.**write_pfm**(*filename*, *img*, *scale=1*)
Writes an OpenCV image into pfm format on disk.

Parameters

- **filename** (*str*) – Name of the image file. The .pfm extension is not added.
- **img** (*Array*) – Numpy Array containing the image (OpenCV view hwc-BGR)
- **scale** (*python:float*) – Scale factor for file. Positive for big endian, otherwise little endian. The number tells the units of the samples in the raster (default 1)

8.5 load_encoded

dlt.hdr.**load_encoded**(*filename*)
Loads a file as a Numpy Byte (uint8) Array.

Parameters **filename** (*str*) – Name of file.

8.6 decode_loaded

dlt.hdr.**decode_loaded**(*x*)
Decodes an image stored in a Numpy Byte (uint8) Array using OpenCV.

Parameters **x** – The Numpy Byte (uint8) Array.

CHAPTER 9

Command Line Tools

9.1 dlt-plot

dlt.viz.**plot_csv**(*use_args=None*)
Plots data from csv files using pandas.

Also usable as a command line program *dlt-plot*.

Parameters **use_args** (*dict, optional*) – Arguments to use instead of (command line) args.

Example

Use with command line:

```
$ dlt-plot -f training.csv --sub_avg 500
```

From inside a script:

```
>>> dlt.viz.plot_csv(['--file', 'training.csv', '--sub_avg', '500'])
```

Note: For information on available functionality use:

```
$ dlt-plot --help
```

9.2 dlt-dispatch

dlt.util.**dispatch**()
Creates a self contained experiment in a directory

Also usable as a command line program *dlt-dispatch*.

Example

Use with command line:

```
$ dlt-dispatch test_low_lr -d ~/experiments -m main.py -e models.py data.py -c␣
↪settings_low_lr.cfg
```

Note: For information on available functionality use:

```
$ dlt-dispatch --help
```

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

V

W