crumpets

Folz et al.

Jul 15, 2021

CONTENTS

Pre-process and augment your data, use training primitives for deep learning.

Below you can find the full documentation and a useful Quick Start Guide.

CHAPTER

ONE

QUICK START GUIDE

You are new to crumpets? Don't worry. We have prepared an easy to follow guide for first steps. Learn about how to use crumpet's efficient data loading utilities and training framework.

1.1 Quick Start Guide

This tutorial introduces the basic concepts of how to use crumpets for efficient data processing in Deep Learning.

Crumpets has two important main functionalities. The first one is providing a data processing pipeline using crumpets' *TurboDataLoader*. The other is offering a *Trainer* class, that can be used to train a given network. These will be described in separate. But first at all, how to install it?

1.1.1 1. Installation

Crumpets is pip-installable! Go to the root directory and execute:

```
>>> pip install .
```

It might also be useful to install crumpets.torch, which defines torch dependent packages, like a fast *TorchTurboDataLoader* and DeepLearning Stuff.

```
>>> python setup-torch.py install
```

1.1.2 2. Data Processing

This part of the tutorial uses the both examples dataloader_simple.py and dataloader_datadings.py, which can be found in crumpets.examples. A crumpets TurboDataLoader (TDL) loads a given dataset and processes it efficiently. A set of parameters allows modifying its behavior according to one's individual requirements. Setting up a TDL at least requires an iterable, a batch size, a worker tempalte and the number of workers. Batch size and number of workers are self explaining. So let's first focus on the iterable and the worker template.

The iterable is what actually defines the dataset. As the name states, this in general can be of any type, but must be something the Loader is able to iterate over, i.e. a set, list, tuple, or such. In specific, the type of worker that is used defines what type the elements in the iterable have to be of, since the workers implement the handling of those. Crumpets predefined workers all expect msgpack packed dictionaries. So, for instance, having the dataset stored in an ImageNet style folder structure (see below), we first need to preprocess that.

Folder Structure:

```
`– tinyset/
      `– cat/
            |– 1.jpg
            |– 2.jpg
            |– 3.jpg
            |– 4.jpg
      `– elephant/
            |– 1.jpg
            |– 2.jpg
            |– 3.jpg
            |– 4.jpg
```

Preprocessing Code:

```
def prepare_dataset():
    dsdir = 'tinyset'
    iterable = []
    # walk over all subdirectories containing the classes
    for cls_id, (cls_dir, _, imgs) in enumerate(list(os.walk(dsdir))[1:]):
        # inside a subdirectory specifying a class, walk over all images
        for img_path in imgs:
            # read the image
            with io.FileIO(pt.join(cls_dir, img_path)) as f:
                img = f.read()
            # put it inside a dictionary together with some class id
            dic = {'image': img, 'label': cls_id}
            # pack the dict using msgpack and append it to the result
            iterable.append(msgpack.packb(
               dic, use_bin_type=True,
               default=msgpack_numpy.encode
            ))
    return iterable
```

But for most datasets this code is unnecessary, because we also offer a python project named datadings. With that one the usual datasets comfortably can be downloaded and preprocessed, using just one command.

Now, having the dataset in the correct format, we have to define a `worker_template`, which the loader can use to generate worker instances from. The loader later reads from the iterable and sends the elements to its workers, which process them. In parallel, it receives the processed results in a consuming Thread and returns them, if asked for. The worker template can be you own custom implementation, but should at least inherit `worker`. Usually it is a good idea to directly inherit *BufferWorker*, as that one already implements the most basic stuff. The most default predefined worker probably is the *ClassificationWorker*. This one can be used, as the name states, for the standard Deep Learning Task, i.e. Classification. The worker at least requires 2 parameters, image and label. Both are 3-tuples defining the shape, dtype and fill_value of the corresponding input. Shape and dtype are self-explaining. Fill_value is optional. For instance for Imagenet, we might want to define our template like this:

```
>>> w = ClassificationWorker(
   ((3, 224, 224), np.uint8),
   ((1, ), np.int)
)
```

Having the worker defined and the data preprocessed, we finally can set up our TDL:

```
batch_size = 2
epochs = 3
nworkers = 2
sample_size = (3, 224, 224)

# prepare iterable
iterable = prepare_dataset()
num_samples = len(iterable)
cycler = cycle(iterable)  # necessary for multiple epochs

# create loader
loader = TurboDataLoader(
    cycler, batch_size,
    ClassificationWorker(  # Other workers are available, such as SaliencyWorkers
        (sample_size, np.uint8), ((1, ), np.int),
        # this actually means using default augmentations, found in ~crumpets.
↪randomization.randomize_args
        image_rng=AUGMENTATION_TRAIN
    ),
    nworkers,
    length=num_samples,
)
```

This loader can now easily be used within a with-statement. Note that iterating over the loader returns mini_batches. The default value for that is 1, thus it returns a list of size 1. But one can modify the parameters of the TDL to increase that number to overcome RAM limitations.

```
with loader:
  for epoch in range(epochs):
     for iteration, mini_batch in loader:
         for sample in mini_batch:
             image = sample['image']
             label = sample['label']
```

1.1.3 3. Training

The training tutorial refers to both examples pytorch_cifar10.py and pytorch_resnet.py found in crumpets.examples. At the current state of crumpets the only Deep Learning framework that is supported is pytorch. This step of the tutorial thus uses that one and therefore it is necessary to install crumpets-torch in addition to the standard crumpets version. Have a look into the installation section to see how that can be accomplished.

As a first step to get a network trained, we first need to actually define the net. The major part of its implementation is skipped, as this guide is not intended to explain pytorch mechanics. But crumpets requires just a bit of attention when using nets, because of the multi-gpu support and design of the TDL. Loader's return type is a minibatch of dictionaries. Thus the network must be able to process dictionaries and also return such:

```
class Net(torch.nn.Module):
    def forward(self, sample):
        x = sample['image'].float()
        x = foo(x)
        sample['output'] = x
        return sample
```

(continues on next page)

(continued from previous page)

```
net = Net()
```

Crumpets also offers an Unpacker Module for this, therefore equivalent:

```
class Net(torch.nn.Module):
   def forward(self, sample):
      return foo(sample)
net = Unpacker(Net(), output_key, input_key)
```

Also, when using pytorch, a slightly modified DataLoader is required, the *TorchTurboDataLoader*. It basically returns torch tensors instead of numpy arrays and, as said, enables cuda and thus gpu support. The loader can be used in either single or multi-gpu mode, which can be controlled using the `devices` parameter. If this parameter is just a single string/int/torch.device like `'cuda:0'`, single mode is used and thus the loader can be used exactly as its more simple ancestor discussed previously. But if the parameter is iterable and potentially contains several cuda devices, it is crucial to wrap a `ParallelApply` module around the net, since the return type of the loader changes. The ParallelApply module will take care of that and run the net in parallel on multiple gpus if such are available. At default it will also merge the results obtained in the forward passes to be given on the main device. Note that, if the loader shall use cpu exclusively, e.g. if no gpus are available, one can set the `devices` parameter to `'cpu:0'`. There are helper methods in *torch*, namely *is_single_torch_device()* and `is_gpu_only()`, that can be used to check the devices parameter. Setting up a network and loader might look like this:

```
if not is_cpu_only(torch_devices):
    if is_single_torch_device(torch_devices):
        Unpacker(Net().cuda())
    else:
        network = ParallelApply(Unpacker(Net()))
else:
    network = Unpacker(Net())

# abstract methods, implementation can be found in previous section
train = make_loader(
    train_set, batch_size, devices=devices
)
val = make_loader(
    val_set, batch_size, devices=devices
)
```

Note that some well known network architectures are reimplemented in crumpets which can be imported and used without having the need of unpackers or additional care. Have a look at crumpets.torch.models.

As usually, training of networks requires an optimization methodology and perhaps a scheduler for varying learning rates and parameters. Again, this is not further explained, as in crumpets this does not differ from standard pytorch. Have a look at pytorch tutorials.

```
optimizer = SGD([
    {'params': network.parameters(), 'lr': lr},
], momentum=momentum, weight_decay=1e-4)
scheduler = PolyPolicy(optimizer, epochs, 1)
```

Instead, a special handling again is required when it comes to losses. As often stated, crumpets loaders all return dictionaries. This dictionary may contain different variables depending on the worker's design. In general and for classification, it consists out of an image input and target label. The default worker for those uses the most common keys, i.e. 'image' and 'label'. Also, if the sample is forwarded through a network, a third value is added to the dictionary. The output of the network. Usually its key is called 'output', but that depends on the implementation of the network

itself. Anyway, crumpets offers its own loss methods in crumpets.torch.loss, which are minor modifications of the standard torch ones. They are able to handle dictionaries, but require to know the keys:

```
loss = CrossEntropyLoss(target_key='label', output_key='output')
if cuda:
   loss = loss.cuda()
```

It is helpful to define further metrics measuring networks quality. Implementations of those can be found in crumpets.torch.metrics. Similar to losses, they need to get the keys passed:

```
metric = AccuracyMetric(target_key='label', output_key='output')
```

Finally, all that is left to do, is constructing and running a crumpets Trainer instance, which will take care of the complete training:

```
trainer = Trainer(
     network=network,
     optimizer=optimizer,
     loss=loss,
     metric=metric,
     train_policy=policy,
     val_policy=None,
     train_iter=train,
     val_iter=val,
     outdir=outdir
 )
 with train:
     with val:
         trainer.train(epochs)
```

Snapshots, outputs and further logging information can be found in outdir.

CHAPTER

TWO

DATA AUGMENTATION

Crumpets offers a great variety of data augmentation operations. All those are integrated in crumpet's data loading pipeline, thus efficient and easy to use. Here we present an overview of available augmentations and their usage. On top of it you get the unique chance of seeing multiple pictures of sweet baby elephants!

2.1 Augmentation Guide

This tutorial introduces augmentations and data randomization.

A common problem of Deep Neural Network is to overfit, i.e. the network fits well to the exact training data but does not generalize for other data. There are multiple approaches on how to deal with this, known as regularization. One method is to randomly transform the training data in each iteration. This enhances the overall dataset, as augmented images are kind of unseen and force the network to generalize for better adaptation.

Crumpets offers a fast, efficient and reliable way of doing this.

2.1.1 1. Usage

As stated, using augmentations with crumpets is easy, as it is integrated in the TurboDataLoader. More precise: in the workers. This guide will not explain how to use the data loader in general. Have a look at the Quick Start Guide.

All current augmentations work for images, thus workers handling images need to be used. Crumpets offers an *ImageWorker*. This worker and all its descendants can be used to work with images. Random numbers generated to augment images are controlled by *RNG* objects given as parameter 'image_rng'. Later the worker will, per iteration and image of the data loader, pick random values in each range. And lastly apply augmentations according to the chosen values either using cpu or gpu, depending on your configuration.

In other words, all you need to do for using augmentations in crumpets, is defining a range for each operation and pass those to the worker template you need to define for the TurboDataLoader.

For instance, to create a custom *RNG* object:

```
from crumpets.presets import MixtureRNG
rng = MixtureRNG(
    blur_range=(0.002, 0.0025),
    brightness_range=(-0.4, 0.4),
    contrast_range=(-0.5, 0.5),
    noise_range=(0.01, 1.0),
    aspect_sigma=2/48.0,
    shift_range=(-1, 1),
    scale_range=(0.5, 1.5),
```

(continues on next page)

(continued from previous page)

```
    vmirror=0.5,
    rotation_sigma=18,
    color_range=(-0.25, 0.25),
)
```

That object can then be given to the worker:

```
worker_template = ClassificationWorker(
    (sample_size, np.uint8),
    ((1, ), np.int),
    image_rng=rng
)
```

Note that by default no augmentation is applied. **:module:`~crumpets.presets`** provides presets like *AUGMENTATION_TRAIN* that contain sensible values for image augmentation during training.

2.1.2 2. List of Available Augmentations

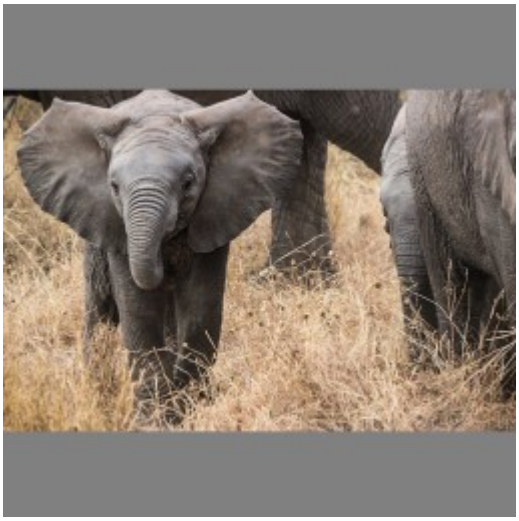
To get an intuition for the impact of different augmentations, we will pick one example image and compare it's unaugmented version to the augmented ones.

And, as promised, sweet baby elephants will serve as an example:

1. No Augmentation

   Let's have a look at what a TurboDataLoader returns without using augmentations, but with a smaller target size and scaling using its longest edge:

```
worker_template = ClassificationWorker(
    ((3, 256, 256), np.uint8, (128, 128, 128)),
    ((1, ), np.int),
    image_params=dict(scale_mode='longest'),
)
```



The image of course is smaller. But also some grey background appeared. This is due to the fact that neural networks usually work with quadratic sample sizes, thus if the original image is scaled down matching it's longest edge to the target size, some pixels are left undefined. This is filled up with custom background, default is black.

2. Aspect

   For applying only one specific augmentation with an exact intensity, we have to use the no_augmentation() dictionary as a starting point. We slightly modify it s.t. just the desired range is set. In this case we set aspect ratio to 0.3 sigma.

```
from crumpets.presets import MixtureRNG
rng = MixtureRNG(prob=1.0, aspect_sigma=0.3)
worker_template = ClassificationWorker(
    ((3, 256, 256), np.uint8, (128, 128, 128)),
    image_params=dict(scale_mode='longest'),
    image_rng=rng
)
```

3. Blur

   To keep it simple, from now on only the update dictionary is presented:

```
>>> prob=1.0, blur_range=(1.50/448, 1.50/448)
```

4. Brightness

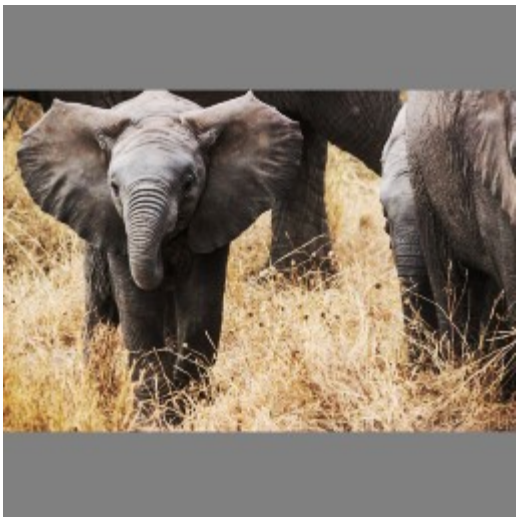
```
>>> prob=1.0, brightness_range=(0.45, 0.45)
```

5. Color

```
>>> prob=1.0, color_range=(-0.3, 0.3)
```

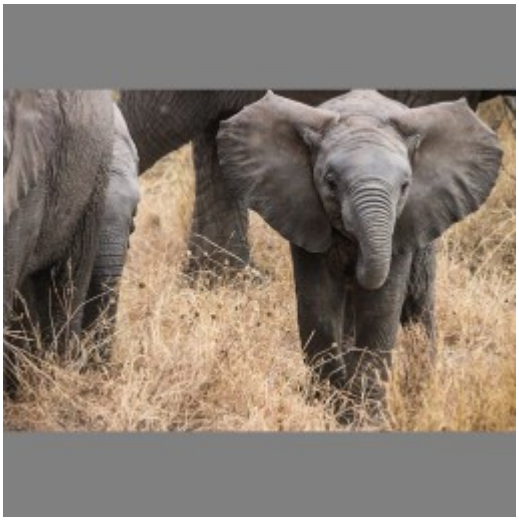
6. Contrast

```
>>> prob=1.0, contrast_range=(0.35, 0.35)
```



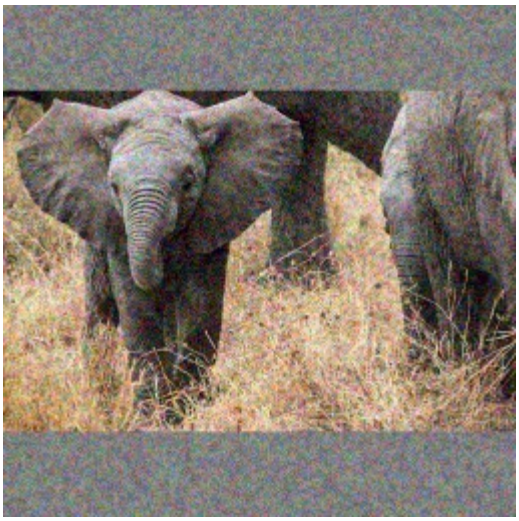
7. Horizontal Mirror

```
>>> prob=1.0, hmirror=1
```



8. Noise

```
>>> prob=1.0, noise_range=(0.2, 0.2)
```



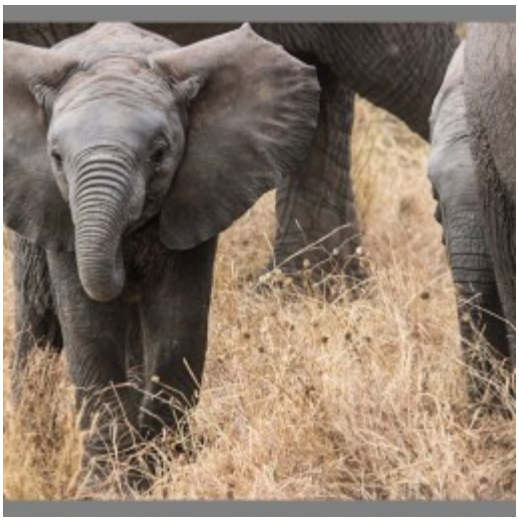
9. Rotation

```
>>> prob=1.0, rotation_sigma=24
```

10. Scale

    Zoom in:

```
>>> prob=1.0, scale_range=(1.4, 1.4)
```

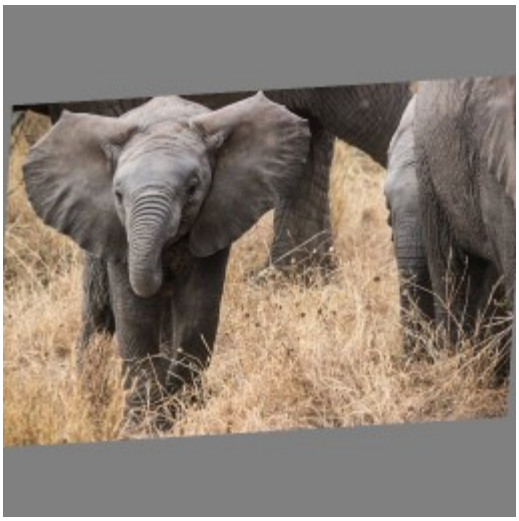


Zoom out:

```
>>> prob=1.0, scale_range=(0.6, 0.6)
```

11. Shear

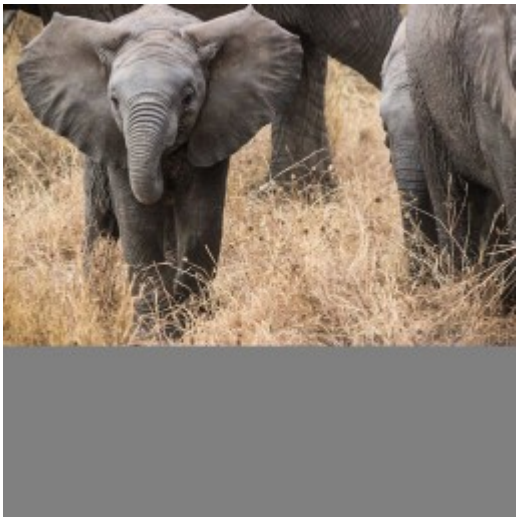
```
>>> prob=1.0,  shear_range=(0.06, 0.06)
```



12. Shift

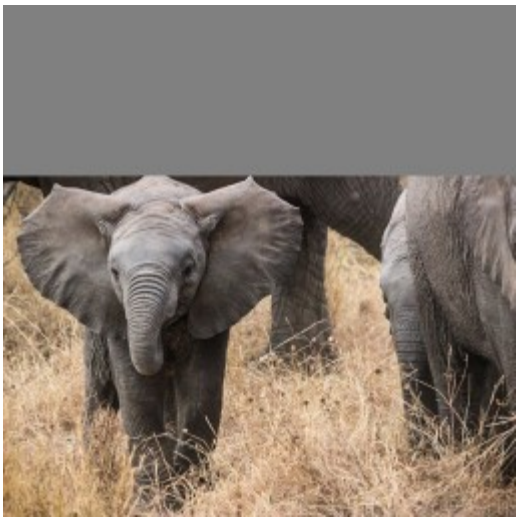
    Shift up:

```
>>> prob=1.0, shift_range=(1, 1)
```



    Shift down:

```
>>> prob=1.0, shift_range=(-1, -1)
```



13. Vertical Mirror

```
>>> prob=1.0, vmirror=1
```

CHAPTER

THREE

DOCUMENTATION

3.1 crumpets package

3.1.1 Subpackages

crumpets.torch package

crumpets.torch.**is_cpu_only**(*val*)
checks if val is a value determining cpu-only cuda devices

crumpets.torch.**is_single_torch_device**(*val*)
checks if val is a value determining a single cuda device

Submodules

crumpets.torch.augmentation_cuda module

crumpets.torch.augmentation_cuda.**add_blur**(*im*, *augs*)
A Function that takes a tensor that contains a batch of images and a list of dictionaries that contain information about the desired blur and takes this information to blur the image.

This function is hardware accelerated, so be sure that the im is located on the GPU.

Parameters

- **im** – the Tensor that contains the image data
- **augs** – a list of dictionaries. Each dict should contain a 'blur' value. This blur indicates the sigma value of the normal distribution filter that is used to blur the image. Also note that the blur value should be relative to the image size, to achieve the same optical blur effect on different image sizes. For further information see *randomize_image()*

crumpets.torch.augmentation_cuda.**add_gamma**(*im_tensor*, *augs*, *maxv=None*)
A Function that takes a tensor that contains a Batch of Images and a list of dictionaries that contain information about the desired gamma values and takes those gamma values to apply gamma correction to the images. This function is hardware accelerated, so be sure that the im_tensor is located on the GPU.

Parameters

- **im_tensor** – the Tensor that contains the Image data
- **augs** – a list of dictionaries. Each dict should contain a 'color', a 'gamma_gray', a 'gamma_color', and a 'contrast' value to specify the behaviour of the gamma augmentation. For further information see *randomize_image()*

- **maxv** – Maximum value of the entries. This value is data type dependent, so be careful with it. It defaults to "None". None indicates that the value is taken according to the data type of the tensor.

crumpets.torch.augmentation_cuda.**add_noise_other**(*im*, *augs*, *minv=None*, *maxv=None*, *internal_ftype=None*)

A Function that takes a tensor that contains a batch of images and a list of dictionaries that contain information about the desired noise and adds noise according to that to the images.

This function is Hardware accelerated, so be sure that the im tensor is located on the GPU.

Parameters

- **im** – the Tensor that contains the image data
- **augs** – a list of dictionaries. Each dict should contain a 'noise' value to specify the behaviour of the noise augmentation. For further information see *randomize_image()*
- **minv** – Minimum value of the entries. This value is data type dependent, so be careful with it. It defaults to "None". None indicates that the value is taken according to the data type of the tensor.
- **maxv** – Maximum value of the entries. This value is data type dependent, so be careful with it. It defaults to "None". None indicates that the value is taken according to the data type of the tensor.
- **internal_ftype** – The type that is used internally to compute the noise. For most types the internal type is float32. The type defaults to None, what indicates that a fitting type is chosen according to the input type.

crumpets.torch.augmentation_cuda.**add_noise_rgb**(*im*, *augs*, *minv=None*, *maxv=None*, *internal_ftype=None*)

A Function that takes a tensor that contains a batch of images and a list of dictionaries that contain information about the desired noise and takes this information to add noise according to the that to the images.

This noise function tries to mimic the rgb noise of a camera sensor, what means that the green value has a lower noise.

This function is hardware accelerated, so be sure that the im is located on the GPU.

Parameters

- **im** – the Tensor that contains the Image data
- **augs** – a list of dictionaries. Each dict should contain a 'noise' value to specify the behaviour of the noise augmentation. For further information see *randomize_image()*
- **minv** – Minimum value of the entries. This value is data type dependent, so be careful with it. It defaults to "None". None indicates that the value is taken according to the data type of the tensor.
- **maxv** – Maximum value of the entries. This value is data type dependent, so be careful with it. It defaults to "None". None indicates that the value is taken according to the data type of the tensor.
- **internal_ftype** – The type that is used internally to compute the noise. The type defaults to None, what indicates that a fitting type is chosen according to the input type. For most types the internal type is float32.

crumpets.torch.dataloader module

class crumpets.torch.dataloader.**TorchTurboDataLoader**(*iterable*, *batch_size*, *worker_template*, *nworkers*, *length=None*, *num_mini_batches=1*, *start_iteration=0*, *device='cuda:0'*, *gpu_augmentation=False*, *shared_memory=True*)

Bases: *crumpets.dataloader.TurboDataLoader*

TorchTurboDataLoader is a subclass of *TurboDataLoader* intended for use with the Pytorch framework. It produces torch tensors instead of numpy arrays.

See *TurboDataLoader* for more details on its operation.

Parameters

- **iterable** – An iterable providing a sample per iteration.
- **batch_size** – The amount of samples per batch.
- **worker_template** – An actual worker instance, determines the kind of processing. Has to inherit crumpets.broker.Worker.
- **nworkers** – Number of workers processing the samples simultaneously. worker_template is copied to create them.
- **length** – Specifies the length of the dataset. Defaults to the actual length of iterable (if available). If given differs from default, the number of iterations per epoch is modified accordingly.
- **num_mini_batches** – Number of mini_batches per batch.
- **start_iteration** – Start the iteration counter from this number. Useful when resuming training.
- **shared_memory** – Whether to use shared memory to transfer data from workers. If 0 or *False*, shared memory is disabled. If *True*, *2*nworkers* shared buffers will be used. If any number > 0, that number of buffers will be used. A value of 1 is strongly discouraged to prevent deadlocks. Permanently storing values returned by a loader may also cause deadlocks.
- **device** – torch device to use, Defaults to 'cuda:0'.
- **gpu_augmentation** – Use a *Randomizer* to calculate certain data augmentation operations on GPU. This disables said operations on the CPU side.

crumpets.torch.loss module

class crumpets.torch.loss.**CrossEntropyLoss**(**args: Any*, ***kwargs: Any*)

Bases: torch.nn.

Wrapper for *torch.nn.CrossEntropyLoss* that accepts dictionaries as input.

Parameters

- **output_key** – key in given sample dict which maps to the output tensor
- **target_key** – key in given sample dict which maps to the target tensor
- **weight** – a manual rescaling weight given to each class. If given, it has to be a Tensor of size *C*. Otherwise, it is treated as if having all ones.

- **reduction** – Specifies the reduction to apply to the output: `'none' | 'mean' | 'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Default: `'mean'`
- **ignore_index** – Specifies a target value that is ignored and does not contribute to the input gradient. When `size_average` is `True`, the loss is averaged over non-ignored targets.

forward(*sample*)

class crumpets.torch.loss.**L1Loss**(**args: Any*, ***kwargs: Any*)
Bases: torch.nn.

Wrapper for *torch.nn.L1Loss* that accepts dictionaries as input.

Parameters

- **output_key** – key in given sample dict which maps to the output tensor
- **target_key** – key in given sample dict which maps to the target tensor
- **reduction** – Specifies the reduction to apply to the output: `'none' | 'mean' | 'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Default: `'mean'`

forward(*sample*)

class crumpets.torch.loss.**LabelSmoothing**(**args: Any*, ***kwargs: Any*)
Bases: torch.nn.

Loss for LabelSmoothing based on NLL-Loss

Parameters

- **smoothing** – label smoothing factor
- **output_key** – key in given sample dict which maps to the output tensor
- **target_key** – key in given sample dict which maps to the target tensor

forward(*sample*)

class crumpets.torch.loss.**MSELoss**(**args: Any*, ***kwargs: Any*)
Bases: torch.nn.

Wrapper for *torch.nn.MSELoss* that accepts dictionaries as input.

Parameters

- **output_key** – key in given sample dict which maps to the output tensor
- **target_key** – key in given sample dict which maps to the target tensor
- **reduction** – Specifies the reduction to apply to the output: `'none' | 'mean' | 'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Default: `'mean'`

forward(*sample*)

class crumpets.torch.loss.**NLLLoss**(**args: Any*, ***kwargs: Any*)
Bases: torch.nn.

Wrapper for *torch.nn.NLLLoss* that accepts dictionaries as input.

Parameters

- **output_key** – key in given sample dict which maps to the output tensor
- **target_key** – key in given sample dict which maps to the target tensor

- **weight** – a manual rescaling weight given to each class. If given, it has to be a Tensor of size *C*. Otherwise, it is treated as if having all ones.
- **reduction** – Specifies the reduction to apply to the output: `'none' | 'mean' | 'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Default: `'mean'`
- **ignore_index** – Specifies a target value that is ignored and does not contribute to the input gradient. When `size_average` is `True`, the loss is averaged over non-ignored targets.

forward(*sample*)

class crumpets.torch.loss.**NSSLoss**(**args: Any*, ***kwargs: Any*)
Bases: `torch.nn.`

Loss for saliency applications that optimizes the normalized scanpath saliency (NSS) metric.

The output of the network is normalized to zero-mean and unit standard deviation. Then the values at gaze locations given by the target image tensor are maximized.

Since with NSS higher values are better and it does not have an upper bound, the output is simply negated. This means the loss will become negative at some point if your network is learning.

Parameters

- **output_key** – key in given sample dict which maps to the output tensor
- **target_key** – key in given sample dict which maps to the target tensor

forward(*sample*)

crumpets.torch.metrics module

class crumpets.torch.metrics.**AccuracyMetric**(*top_k=1*, *output_key='output'*, *target_key='label'*)
Bases: *crumpets.torch.metrics.Metric*

Computes the top-k accuracy metric for given classification scores, i.e. predicted class probabilities. The metric is computed as {1 if target_i in top_k_predicted_classes_i else 0 for all i in n} / n

Parameters

- **output_key** – the key with which the output is found in the input dictionary
- **target_key** – the key with which the target is found in the input dictionary

reset()

value()
implement to return the currently stored metric. :return: current metric

class crumpets.torch.metrics.**AverageMetric**(*output_key='output'*, *metric_key='average_metric'*)
Bases: *crumpets.torch.metrics.Metric*

Computes a simple average metric for given values inside the output.

Parameters

- **output_key** – the key with which the output is found in the input dictionary
- **metric_key** – the key with which the metric is to be stored in the output dictionary

value()
implement to return the currently stored metric. :return: current metric

class crumpets.torch.metrics.**AverageValue**
Bases: object

value()

class crumpets.torch.metrics.**CombinedMetric**(*children*)
Bases: object

A simple meta metric. Given metric instances, returns a collection of them.

Parameters **children** – list of metric class instances

class crumpets.torch.metrics.**ConfusionMatrix**(*nclasses=10*, *output_key='output'*, *target_key='target_image'*, *metric_key='confusion_matrix'*)
Bases: *crumpets.torch.metrics.Metric*

Computes the confusion matrix for given classification scores, i.e. predicted class probabilities.

Parameters

- **output_key** – the key with which the output is found in the input dictionary
- **target_key** – the key with which the target is found in the input dictionary
- **metric_key** – the key with which the metric is to be stored in the output dictionary

get_true_false_positives()
Calculate the true positive and false positive rates per class :return: 2d-array. Cx3 array where the first column corresponds

to the true positives per class, the second column, to the false positives per class and the last one, the number of samples per class in total that have been seen.

reset()

value()
implement to return the currently stored metric. :return: current metric

class crumpets.torch.metrics.**MSELossMetric**(*output_key='output'*, *target_key='target_image'*, *metric_key='mse'*)
Bases: *crumpets.torch.metrics.Metric*

Computes the mean squared error

Parameters

- **output_key** – the key with which the output is found in the input dictionary
- **target_key** – the key with which the target is found in the input dictionary
- **metric_key** – the key with which the metric is to be stored in the output dictionary

value()
implement to return the currently stored metric. :return: current metric

class crumpets.torch.metrics.**Metric**(*output_key='output'*, *target_key='target_image'*, *metric_key='metric'*)
Bases: object

Abstract class which is to be inherited by every metric. As usual, this class is designed to handle crumpets dictionaries.

Parameters

- **output_key** – the key with which the output is found in the input dictionary
- **target_key** – the key with which the target is found in the imput dictionary

- **metric_key** – the key with which the metric is to be stored in the output dictionary

reset()

abstract value()
implement to return the currently stored metric. :return: current metric

class crumpets.torch.metrics.**NSSMetric**(*output_key='output'*, *target_key='target_image'*, *metric_key='nss'*)

Bases: *crumpets.torch.metrics.Metric*

Computes the Normalized Scanpath Saliency (NSS) by Bylinskii et. al. (https://arxiv.org/pdf/1604.03605.pdf)

Parameters

- **output_key** – the key with which the output is found in the input dictionary
- **target_key** – the key with which the target is found in the input dictionary
- **metric_key** – the key with which the metric is to be stored in the output dictionary

value()
implement to return the currently stored metric. :return: current metric

class crumpets.torch.metrics.**NoopMetric**(*output_key='output'*, *target_key='target_image'*, *metric_key='metric'*)

Bases: *crumpets.torch.metrics.Metric*

Provides the same API as a real metric but does nothing. Can be used where some metric-like object is required, but no actual metrics should be calculated.

value()
implement to return the currently stored metric. :return: current metric

crumpets.torch.policy module

class crumpets.torch.policy.**NoopPolicy**

Bases: object

Just a noop Policy. Use it when you don't want to modify the lr

step(**args*, ***kwargs*)

class crumpets.torch.policy.**PolyPolicy**(**args: Any*, ***kwargs: Any*)

Bases: torch.optim.lr_scheduler.

A policy that can be described as a polynomial.

Parameters

- **optimizer** – an optimizer object
- **num_epochs** – the number of epochs that this policy is defined for. Don't use it longer than that, because this might cause unexpected behaviour
- **power** – power value
- **last_epoch** – The current state of the policy. This can be used to set the initial state of the policy for instance to change the policy during training.

get_lr()

class crumpets.torch.policy.**RampPolicy**(**args: Any*, ***kwargs: Any*)
Bases: torch.optim.lr_scheduler.

This Policy increases the learning rate step by step

Parameters

- **optimizer** – an optimizer object
- **ramp_epochs** – the value where the plateau is reached
- **last_epoch** – The current state of the policy. This can be used to set the initial state of the policy for instance to change the policy during training.

get_lr()

step(*epoch=None*, *metrics=None*)

class crumpets.torch.policy.**ReduceLROnPlateau**(**args: Any*, ***kwargs: Any*)
Bases: torch.optim.lr_scheduler.

A policy that reduces the learning rate when the training progress reaches a plateau. It inherits from torch.optim.lr_scheduler.ReduceLROnPlateau and because of that shares the same interface

step(*epoch=None*, *metrics=None*)

class crumpets.torch.policy.**SigmoidPolicy**(**args: Any*, ***kwargs: Any*)
Bases: torch.optim.lr_scheduler.

A policy that can be described as a sigmoid. It can be described using the formula base_lr / (1 + math.exp(self.q * x), where x is last_epoch/num_epochs - 1

Parameters

- **optimizer** – an optimizer object
- **num_epochs** – the number of epochs that this policy is defined for. Don't use it longer than that, because this might cause unexpected behaviour
- **q** – q value to describe the behaviour of the policy.
- **last_epoch** – The current state of the policy. This can be used to set the initial state of the policy for instance to change the policy during training.

get_lr()

crumpets.torch.randomizer module

class crumpets.torch.randomizer.**Randomizer**(**args: Any*, ***kwargs: Any*)
Bases: torch.nn.

Given a network (or in general, some pytorch module), it is wrapped around the nets forward pass. If the randomizer's forward function is invoked, it first randomizes the image in the sample dictionary. That means it basically works like *randomize_image()*, which is usually applied to the image in one of the workers. The major difference here is that all augmentations are gpu powered, and thus faster. Also not all augmentation operations are supported. The randomizer does not rotate or resize. The values used for augmenting are picked out of the dictionary. Therefore the sample dictionary must contain these. Usually crumpets worker take care of that.

Parameters **net** – some network the randomizer shall be wrapped around

cpu()

cuda(*device_id=None*)

forward(*sample*, **args*, ***kwargs*)
Applies different randomizing augmentations to input images and then forwards result through net, if given.

Parameters **sample** – dictonary with {"image": Tensor of shape n,c,h,w,

"augmentation": list of augmentation parameters per image in batch}

Returns modified dictionary with randomized image and network modified entries

crumpets.torch.shm module

class crumpets.torch.shm.**DummyTensorManager**(*device='cuda:0'*)
Bases: *crumpets.shm.DummyBufferManager*

Torch replacement for DummyBufferManager. Returns torch tensors instead of numpy arrays when unpacking.

Parameters **device** – output device; buffers are copied here when ready

next()

unpack(*data*)
Unpack an msgpack message. :param data: msgpack message bytes :return: packed objects

class crumpets.torch.shm.**SharedTensorManager**(*num_buffers*, *batch_size*, *buffer_specs*, *device='cuda:0'*, *_queueclass=<bound method BaseContext.Queue of <multiprocessing.context.DefaultContext object>>*)
Bases: *crumpets.shm.SharedBufferManager*

crumpets.torch.shm.**shared_tensor**(*shape*, *dtype=<class 'numpy.float32'>*, *device_type='cuda'*)
Create a torch tensor that resides in shared memory.

Parameters

- **shape** – array shape
- **dtype** – numpy dtype
- **device_type** – tensor.pin_memory() if 'cuda'

Returns np.ndarray

crumpets.torch.trainer module

class crumpets.torch.trainer.**Trainer**(*network*, *optimizer*, *loss*, *metric*, *train_policy*, *val_policy*, *train_iter*, *val_iter*, *outdir*, *val_loss=None*, *val_metric=None*, *snapshot_interval=1*, *quiet=False*)
Bases: object

The Trainer can be used to train a given network. It alternately trains one epoch and validates the resulting net one epoch. Given loss is evaluated each batch, gradients are computed and optimizer used to updated weights. The loss is also passed to the policy, which might update the learning rate. Useful information about the training flow is regularly printed to the console, including an estimated time of arrival. Loss, metric and snapshots per epoch are also logged in outdir, for later investigation. outdir is created if either quiet is *False* or *snapshot_interval* > *0*.

Parameters

- **network** – Some network that is to be trained. If multiple gpus are used (i.e. multiple devices passed to the data loader) a ParallelApply module has to be wrapped around.
- **optimizer** – some torch optimzer, e.g. SGD or ADAM, given the network's parameters.

- **loss** – some loss function, e.g. CEL or MSE. Make sure to use crumpets.torch.loss or implement your own ones, but do not use torch losses directly, since they are not capable of handling crumpets sample style (i.e dictionaries).
- **metric** – some metric to further measure network's quality. Similar to losses, use crumpets.torch.metrics
- **train_policy** – some policy to maintain learning rates and such, in torch usually called lr_schedulers. After each iteration it, given the current loss, updates learning rates and potentially other hyperparameters.
- **val_policy** – same as train_policy, but updates after validation epoch.
- **train_iter** – iterator for receiving training samples, usually this means a `TorchTurboDataLoader` instance.
- **val_iter** – same as train_iter, but for retrieving validation samples.
- **outdir** – Output directory for logfiles and snapshots. Is created including all parent directories if it does not exist.
- **val_loss** – same as loss, but applied during validation. Default is None, which results in using loss again for validation.
- **val_metric** – same as metric, but applied during validation. Default is None, which results in using metric again for validation.
- **snapshot_interval** – Number of epochs between snapshots. Set to 0 or *None* to disable snapshots. Default is 1, which means taking a snapshot after every epoch.
- **quiet** – If True, trainer will not print to console and will not attempt to create a logfile.

add_hook(*name*, *fun*)

Add a function hook for the given event. Function must accept trainer *state* dictionary as first positional argument the current, as well as further keyword arguments depending on the type of hook.

The following events are available during training:

- *'train_begin'*: run at the beginning of a training epoch
- *'train_end'*: run after a training epoch has ended
- *'train_pre_forward'*: run before the forward step; receives kwarg *sample*
- *'train_forward'*: run after the forward step; receives kwargs *metric*, *loss*, and *output*
- *'train_backward'*: run after the backward step; receives kwargs *metric*, *loss*, and *output*

During validation the following hooks are available:

- *'val_begin'*: run at the beginning of a training epoch
- *'val_end'*: run after a training epoch has ended
- *'val_pre_forward'*: run before the forward step; receives kwarg *sample*
- *'val_forward'*: run after the forward step; receives kwargs *metric*, *loss*, and *output*

Parameters

- **name** – The event name. See above for available hook names and when they are executed.
- **fun** – A function that is to be invoked when given event occurs. See above for method signature.

print_info(*epoch*)
prints and logs current learning rates as well as the epoch.

Parameters **epoch** – the current epoch.

remove_hook(*name*, *fun*)
Remove the function hook with the given name.

Parameters

- **name** – type of hook to remove
- **fun** – hook function object to remove

Returns

snapshot(*epoch*)
stores snapshot of current model (including optimizer state), uses epoch for naming convention (but does always store current model).

Parameters **epoch** – epoch for naming output file

train(*num_epochs*, *start_epoch=0*)
starts the training, logs loss and metrics in logging file and prints progress in the console, including an ETA. Also stores snapshots of current model each epoch.

Parameters

- **num_epochs** – number of epochs to train
- **start_epoch** – the first epoch, default to 0. Can be set higher for finetuning, etc.

train_epoch()
trains one epoch, is invoked by train function. Usually not necessary to be called outside.

Returns train metric result

validate_epoch(*epoch*)
Validate once. Invoked by train function. Usually not necessary to be called outside.

Returns val metric result

crumpets.torch.utils module

class crumpets.torch.utils.**Normalize**(**args: Any*, ***kwargs: Any*)
Bases: torch.nn.

forward(*x*)

class crumpets.torch.utils.**Unpacker**(**args: Any*, ***kwargs: Any*)
Bases: torch.nn.

forward(*sample*, **_*, ***__*)

crumpets.torch.utils.**filter_state**(*own_state*, *state_dict*)

crumpets.torch.utils.**other_type**(*s*)

crumpets.torch.utils.**resume**(*path*, *model*, *optimizer*)
Given parameters, extracts a training state, i.e. initializes a network and optimizer.

Parameters

- **path** – path to a pytorch snapshot (including model and optimizer states)

- **model** – a network architecture for that the extracted weights are applied to
- **optimizer** – an optimizer for which the extracted optimizer parameters are applied to

Returns the loaded snapshot

crumpets.torch.utils.**save**(*path*, *iteration*, *model*, *optimizer*, ***kwargs*)

crumpets.torch.utils.**try_dicts**(*k*, **ds*)

crumpets.torch.utils.**try_types**(*k*, **ds*)

crumpets.workers package

class crumpets.workers.**ClassificationWorker**(*image*, *label*, *image_params=None*, *image_rng=None*, ***kwargs*)

Bases: *crumpets.workers.ImageWorker*

Worker for processing (Image, Label)-pairs for classification.

Parameters

- **image** – tuple of image information (shape, dtype, fill_value); fill_value is optional, defaults to 0
- **label** – tuple of label information (shape, dtype, fill_value); fill_value is optional, defaults to 0
- **image_params** – dict of fixed image parameters; overwrites random augmentation values
- **image_rng** – RNG object used for image augmentation, see *RNG* and `randomize_args()`

prepare(*sample*, *batch*, *buffers*)

Implement this method to define the behavior of the BufferWorker subclass. Results must be written to buffers and/or batch object.

Parameters

- **sample** – individual sample object to process
- **batch** – the object the sample belongs to; append values to lists as necessary
- **buffers** – output buffers to use for this sample

class crumpets.workers.**FCNWorker**(*image*, *target_image*, *image_params=None*, *target_image_params=None*, *image_rng=None*, ***kwargs*)

Bases: *crumpets.workers.ImageWorker*

Worker for fully convolutional networks (FCN). Produces *image-target_image*-pairs.

Parameters

- **image** – tuple of image information (shape, dtype, fill_value); fill_value is optional, defaults to 0
- **target_image** – tuple of target image information (shape, dtype, fill_value); fill_value is optional, defaults to 0
- **image_params** – dict of fixed image parameters; overwrites random augmentation values
- **target_image_params** – dict of fixed target image parameters; overwrites random augmentation values
- **image_rng** – RNG object used for image augmentation, see *RNG* and `randomize_args()`

prepare(*sample*, *batch*, *buffers*)
Implement this method to define the behavior of the BufferWorker subclass. Results must be written to buffers and/or batch object.

Parameters

- **sample** – individual sample object to process
- **batch** – the object the sample belongs to; append values to lists as necessary
- **buffers** – output buffers to use for this sample

class crumpets.workers.**ImageWorker**(*image*, *image_params=None*, *image_rng=None*, ***kwargs*)
Bases: *crumpets.broker.BufferWorker*

Worker for processing images of any kind.

Parameters

- **image** – tuple of image information (shape, dtype, fill_value); fill_value is optional, defaults to 0
- **image_params** – dict of fixed image parameters; overwrites random augmentation values
- **image_rng** – RNG object used for image augmentation, see *RNG* and `randomize_args()`
- **gpu_augmentation** – disables augmentations for which gpu versions are available (*randomizer*)

prepare(*sample*, *batch*, *buffers*)
Implement this method to define the behavior of the BufferWorker subclass. Results must be written to buffers and/or batch object.

Parameters

- **sample** – individual sample object to process
- **batch** – the object the sample belongs to; append values to lists as necessary
- **buffers** – output buffers to use for this sample

prepare_image(*im*, *buffers*, *params*, *key*)

Subpackages

crumpets.workers.saliency package

class crumpets.workers.saliency.**SaliencyWorker**(*image*, *target_image*, *image_params=None*, *target_image_params=None*, *image_rng=None*, ***kwargs*)
Bases: *crumpets.workers.ImageWorker*

Worker that outputs images and saliency maps created from raw gaze locations. Expects the following keys present in each sample:

{"image": encoded image data "experiments": [experiment, ...]}

Each experiment is first checked for fixations points under key "fixations". Falls back to key "locations" of raw gaze data if no fixations are found.

The following parameters can be configured:

- image_params: see ImageWorker

- **target_image_params:**
  - **"sample_ratio" (default: 1):** float in [0, 1]; percentage of experiments sampled from the list of all experiments
  - **"jitter" (default: 0):** add noise to the individual gaze locations; sigma of a Gaussian distribution, scaled by the size of the target_images: noise ~ N(jitter * target_image_size)
  - **"interpolate" (default: False):** use linear interpolation to map gaze locations to the target_image
  - **"blur" (default: 0):** apply Gaussian blur with sigma blur * target_image_size to target_image
  - **"maxnorm" (default: False):** apply maximum norm to target_image

prepare(*sample*, *batch*, *buffers*)

Implement this method to define the behavior of the BufferWorker subclass. Results must be written to buffers and/or batch object.

Parameters

- **sample** – individual sample object to process
- **batch** – the object the sample belongs to; append values to lists as necessary
- **buffers** – output buffers to use for this sample

crumpets.workers.saliency.**check_range**(*points*, *h*, *w*)

crumpets.workers.saliency.**discretize_points**(*points*, *h*, *w*)

crumpets.workers.saliency.**interpolate_points**(*points*, *h*, *w*)

crumpets.workers.saliency.**multivariate_normal**(*mean*, *cov*, *size=None*, *check_valid='warn'*, *tol=1e-08*)

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

Note: New code should use the `multivariate_normal` method of a `default_rng()` instance instead; please see the random-quick-start.

---

mean [1-D array_like, of length N] Mean of the N-dimensional distribution.

cov [2-D array_like, of shape (N, N)] Covariance matrix of the distribution. It must be symmetric and positive-semidefinite for proper sampling.

size [int or tuple of ints, optional] Given a shape of, for example, `(m,n,k)`, `m*n*k` samples are generated, and packed in an *m*-by-*n*-by-*k* arrangement. Because each sample is *N*-dimensional, the output shape is `(m,n,k,N)`. If no shape is specified, a single (*N*-D) sample is returned.

check_valid [{ 'warn', 'raise', 'ignore' }, optional] Behavior when the covariance matrix is not positive semidefinite.

tol [float, optional] Tolerance when checking the singular values in covariance matrix. cov is cast to double before the check.

out [ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

Generator.multivariate_normal: which should be used for new code.

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- Spherical covariance (*cov* is a multiple of the identity matrix)
- Diagonal covariance (*cov* has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0, 0]
>>> cov = [[1, 0], [0, 100]]  # diagonal covariance
```

Diagonal covariance means that points are oriented along x or y-axis:

```
>>> import matplotlib.pyplot as plt
>>> x, y = np.random.multivariate_normal(mean, cov, 5000).T
>>> plt.plot(x, y, 'x')
>>> plt.axis('equal')
>>> plt.show()
```

Note that the covariance matrix must be positive semidefinite (a.k.a. nonnegative-definite). Otherwise, the behavior of this method is undefined and backwards compatibility is not guaranteed.

```
>>> mean = (1, 2)
>>> cov = [[1, 0], [0, 1]]
>>> x = np.random.multivariate_normal(mean, cov, (3, 3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> list((x[0,0,:] - mean) < 0.6)
[True, True] # random
```

crumpets.workers.saliency.**show**(*name*, *mat*)

crumpets.workers.segmentation package

class crumpets.workers.segmentation.**SegmentationWorker**(*image*, *target_image*, *image_params=None*, *target_image_params=None*, *image_rng=None*, ***kwargs*)

Bases: *crumpets.workers.FCNWorker*

Worker for image segmentation tasks. *target_image_params* defaults nearest neighbor interpolation, no supersampling, and to disable all pixel-based augmentations like brightness and color.

prepare(*sample*, *batch*, *buffers*)

Implement this method to define the behavior of the BufferWorker subclass. Results must be written to buffers and/or batch object.

Parameters

- **sample** – individual sample object to process
- **batch** – the object the sample belongs to; append values to lists as necessary
- **buffers** – output buffers to use for this sample

3.1.2 Submodules

crumpets.augmentation module

crumpets.augmentation.**calc_scale_ratio**(*source_size*, *target_size*, *scale*, *scale_mode*)

crumpets.augmentation.**decode_image**(*data*, *color*, *min_height=0*, *min_width=0*, *min_factor=2*)

crumpets.augmentation.**decode_opencv**(*data*, *color*)

crumpets.augmentation.**make_transform**(*source_size*, *target_size*, *angle=0*, *scale=1*, *aspect=1*, *shift=None*, *hmirror=False*, *vmirror=False*, *shear=None*, *scale_mode='shortest'*, *__identity__=array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])*)

crumpets.augmentation.**randomize_image**(*im*, *size*, *background=None*, *color=True*, *angle=0*, *scale=1*, *shift=None*, *aspect=1*, *hmirror=False*, *vmirror=False*, *interp_method=1*, *gamma_gray=None*, *gamma_color=None*, *contrast=None*, *noise=None*, *blur=None*, *shear=None*, *is_rgb=True*, *scale_mode='shortest'*, *supersampling=0*, *gpu_augmentation=False*, *do_rotate_and_resize=True*)

Randomizes image according to given parameters.

Parameters

- **im** – image to be transformed.
- **size** – target size of resulting image.
- **background** – background color that fills areas in the output where there is no pixel data; can be number or tuple with same number of elements as channel
- **color** – Boolean that flags if image is black-white or colored.
- **angle** – degrees of rotation
- **scale** – Scales the image with respect to its target size. *scale=1.0* scales the image to fit perfectly within the target size. Based on *scale_mode* either the shorter or longer edge is used as reference. *scale=2.0* doubles the length of the sides, *scale=0.5* halves it.
- **shift** – tuple of int (x,y) defining a shift of the picture, may create undefined space, if source image is moved out of target image, filled up with background color.
- **aspect** – float of aspect ratio change
- **hmirror** – boolean flag for horizontal mirror
- **vmirror** – boolean flag for vertical mirror
- **interp_method** – some interpolation method. At the moment one of: INTERP_NEAREST INTERP_LINEAR INTERP_CUBIC INTERP_LANCZOS4 INTERP_AERA
- **gamma_gray** – float defining a black-white gamma
- **gamma_color** – tuple of floats defining a rgb gamma
- **contrast** – float between -1 and 1 defining a contrast change

- **noise** – float defining a noise strength
- **blur** – float defining a blur intensity, i.e. the standard deviation of a gaussian filter relative to image width
- **shear** – float defining shear intensity, i.e. the gradient of the horizontal edges. A shear of 0.0 therefore creates a rectangular image.
- **is_rgb** – boolean that flags if rgb color encoding is used
- **scale_mode** – Either *'shortest'* or *'longest'*. Scale the image using either shortest or longest edge as reference. *'shortest'* crops part of the image if the aspect ratio of image and target size do not match. *'longest'* ensures that the whole image can be fit into target size. A scale > 1.0 makes it bigger than target image, thus parts of it get cut out. A scale < 1.0 makes it smaller than target image, thus parts of the target image are undefined and filled up with background.
- **supersampling** – supersampling factor, 1 turns off supersampling, 2 means 4 samples per pixel, 3 means 9 samples and so on; default of 0 means choose best based on true image size, output size and scale factor
- **gpu_augmentation** – boolean that flags if gpu augmentations is used elsewhere and thus disables cpu augmentations in this method for all augmentations where gpu versions are available.
- **do_rotate_and_resize** – boolean that flags if rotation and resize operations are used. Mostly used for test cases. Should usually not be changed.

Returns randomized image

crumpets.augmentation.**rotate_and_resize**(*im*, *angle*, *target_size*, *scale*, *aspect*, *shift*, *method*, *background*, *hmirror*, *vmirror*, *shear=None*, *scale_mode='shortest'*, *supersampling=0*)

crumpets.augmentation_cpu module

crumpets.augmentation_cpu.**add_blur**(*im*, *sigma*)

A Function that takes a numpy array that contains an Image and information about the desired blur and blurs the image. It uses cv2 to blur the image, for more information about the sigma parameter have a look into the cv2 documentation. cv.GaussianBlur

Parameters

- **im** – the numpy array that contains the Image data
- **sigma** – the sigma of the gaussian blur

crumpets.augmentation_cpu.**add_gamma**(*im*, *color*, *gamma_gray*, *gamma_color*, *contrast*, *_base_lut=None*, *_pos_contrast_lut=None*, *_neg_contrast_lut=None*)

A Function that takes a numpy array that contains an Image and information about the desired gamma values and takes those gamma values to apply gamma correction to the images.

Parameters

- **im** – the numpy array that contains the Image data
- **color** – flag that indicates if gamma_color should be used
- **gamma_gray** – gray parameter of the gamma correction
- **gamma_color** – color parameter of the gamma correction
- **contrast** – contrast parameter of the gamma correction

- **_base_lut** – a lookup table that can be precomputed. Defaults to None. None indicates that the default lookup

table should be used. The default lookup table is computed only once and then cached. :param _pos_contrast_lut: similar to base_lut, just for the positive part of the contrast :param _neg_contrast_lut: see positive… contrast is treated asymmetrically to give better results

crumpets.augmentation_cpu.**add_noise_other**(*im*, *strength*)

A Function that takes a numpy array that contains an Image and information about the desired noise and takes those values to add noise to the images.

Parameters

- **im** – the numpy array that contains the Image data
- **strength** – strength of the noise

crumpets.augmentation_cpu.**add_noise_rgb**(*im*, *strength*)

A Function that takes a numpy array that contains an Image and information about the desired rgb noise and takes those values to add noise to the images. This function adds rgb noise, that mimics the noise of a camera sensor, what means that green has less noise.

Parameters

- **im** – the numpy array that contains the Image data
- **strength** – strength of the noise

crumpets.broker module

class crumpets.broker.**BufferManager**(*batch_size*, *buffer_specs*)

Bases: `object`

BufferManager is a compatibility class that replaces the SharedDictManager for cases where shared memory is not used by the pipeline. It creates buffers from buffer specs for use with the BufferWorker.

next()

Return the dictionary of buffers as defined by buffer specs.

Returns buffer dictionary

static pack(*obj*)

Pack an object using msgpack. Any shared object are replaced by references.

Parameters **obj** – object to pack

Returns msgpack message bytes

static unpack(*data*)

Unpack an msgpack message. Any shared object references are replaced with the object.

Parameters **data** – msgpack message bytes

Returns packed objects

class crumpets.broker.**BufferWorker**(*buffer_manager=None*, ***kwargs*)

Bases: *crumpets.broker.Worker*

Base class for workers that use constant-size buffers.

Parameters

- **buffer_manager** – Dict of buffer specs (shape, dtype, fill_value). fill_value is optional and defaults to 0. It must be either a scalar or iterable of length equal to the number of channels in the respective image.
- **param_groups** – Dict of fixed parameter dicts. To be used in conjunction with buffers of the same key.
- **kwargs** – Passed to broker.Worker.

add_buffer(*key*, *buf*)
Register a new buffer with the worker.

Parameters

- **key** – name of the buffer
- **buf** – buffer spec or array to use as template

add_params(*key*, *params*, *default=None*)
Add a parameter group to the worker.

Parameters

- **key** – name of the parameters
- **params** – parameter object, usually dictionary
- **default** – default value to use if params is None

get_buffer_manager()
Returns the current buffer manager. May be None. :return: *BufferManager* or *SharedBufferManager* object

abstract prepare(*sample*, *batch*, *buffers*)
Implement this method to define the behavior of the BufferWorker subclass. Results must be written to buffers and/or batch object.

Parameters

- **sample** – individual sample object to process
- **batch** – the object the sample belongs to; append values to lists as necessary
- **buffers** – output buffers to use for this sample

process(*request*)

Implement this method to define worker behavior. Can return an iterable to create several batches from one input. This method can return an iterable or define a generator with the yield keyword. For instance: *process()*

.

param data multipart zmq message from Producer to process

return iterable of zmq messages to send to Consumer

set_buffer_manager(*buffer_manager*)
Set the buffer manager to be used by this worker. Can be None, in which case a *BufferManager* will be created as necessary.

Parameters **buffer_manager** – a *BufferManager* or *SharedBufferManager* object, or None

class crumpets.broker.**Consumer**(*result_address*, *recv_timeout=2000*, *queue_length=3*, *bind=True*, *io_threads=1*)

Bases: *crumpets.broker.ThreadedConsumerBase*

Basic threaded Consumer that receives und unpacks msgpack messages.

class crumpets.broker.**ConsumerBase**(*result_address*, *recv_timeout=1000*, *queue_length=3*, *bind=True*, *io_threads=1*)

Bases: object

Abstract base class for Consumers, the final pipeline stage. Implement the _transform method to define subclass behavior.

retrieve()

retrieve_data()

stop()

class crumpets.broker.**Dispatcher**(*worker_template*, *nworkers*, *work_addresses*, *result_addresses*, *control_address*, *daemon=None*, *gpu_augmentation=None*)

Bases: object

The Dispatcher creates worker processes from a worker template, can starts and stops them and monitor their status.

Parameters

- **worker_template** – instance of Worker subclass to use as template for workers; copy.copy is used to create as many objects as needed
- **nworkers** – number of worker processes to start
- **work_addresses** – list of work addresses to use; cycled through
- **result_addresses** – list of result addresses to use; cycles through
- **control_address** – control address workers can send status updates on
- **daemon** – daemon flag for processes, see multiprocessing.Process
- **gpu_augmentation** – bool passed to workers, true disables cpu augmentations where gpu versions are available in *randomizer*; if None worker_template.gpu_augmentation is used

active()

True if any workers are alive.

start()

stop()

terminate()

class crumpets.broker.**Pipeline**(*worker_template*, *nworkers*, *iterable*, *batch_size*, *work_addresses*, *result_addresses*, *producer_kwargs=None*, *control_address=None*, *gpu_augmentation=None*)

Bases: object

start()

stop()

class crumpets.broker.**Producer**(*work_addresses*, *iterable*, *batch*, *queue_length=8*, *io_threads=1*)

Bases: *crumpets.broker.ProducerBase*

Producer implementation that reads sequentially from arbitrary iterable objects. Items must be a msgpack messages that are understood by the workers. :param iterable:

iterable of msgpack messages

Parameters **batch** – batch size for workers

yield_requests()

class crumpets.broker.**ProducerBase**(*work_addresses*, *daemon=True*, *queue_length=8*, *io_threads=1*)
Bases: multiprocessing.context.Process

Abstract base class for producer processes. Producers are the first stage of the pre-processing pipeline that load data into memory and supply it to workers. Implement the yield_requests method to customize its behavior.

Parameters

- **work_addresses** – List of worker addresses the producer pushes work to; cycled through for load balancing
- **daemon** – Flag whether this Producer is a daemon process; see multiprocessing.Process
- **queue_length** – Length of send queue per worker socket
- **io_threads** – Number of IO threads to use; 1 is fine for almost all cases

run()
Method to be run in sub-process; can be overridden in sub-class

stop()

yield_requests()

class crumpets.broker.**Proxy**(*in_address*, *out_address*, *queue_length=1*, *daemon=True*)
Bases: multiprocessing.context.Process

Utility class that receives and redirects zmq PULL/PUSH streams.

run()
Method to be run in sub-process; can be overridden in sub-class

class crumpets.broker.**ThreadedConsumerBase**(*result_address*, *recv_timeout=2000*, *queue_length=3*, *bind=True*, *io_threads=1*)
Bases: *crumpets.broker.ConsumerBase*

Abstract base class for Consumers that receive and transform data on a separate thread. Implement the _transform method to define subclass behavior.

retrieve()

stop()

class crumpets.broker.**Value**(*_, **__)
Bases: object

class crumpets.broker.**Worker**(*timeout=1000*, *daemon=True*, *gpu_augmentation=False*)
Bases: object

Abstract base class for workers. Implement the process method to define the behavior of subclasses.

Note: set_addresses must be called before starting a worker. The *TurboDataLoader* does this for you.

Parameters

- **timeout** – zmq socket timeout in milliseconds
- **daemon** – set daemon flag - used in process
- **gpu_augmentation** – set GPU augmentation flag

inner()

abstract process(*data*)

Implement this method to define worker behavior. Can return an iterable to create several batches from one input. This method can return an iterable or define a generator with the yield keyword. For instance: *process()*

.

param data multipart zmq message from Producer to process

return iterable of zmq messages to send to Consumer

run()
Starts the worker process.

set_addresses(*work*, *result*, *control*)
Set all required zmq addresses. Required before run can be invoked.

Parameters

- **work** – address where work is received on
- **result** – results are pushed to this address
- **control** – control message are sent here, e.g., exceptions that occurred while processing

set_gpu_augmentation(*val*)
Sets the gpu_augmentation flag to given value, true disables all cpu_augmentations for which a gpu version is available. Note that this does not directly activate usage of gpu augmentation, as for that a *randomizer* module is used, which usually the *TurboDataLoader* takes care of.

Parameters **val** – boolean flag

stop()
Stops the worker process.

crumpets.broker.**make_buffer**(*batchsize*, *shape*, *dtype*, *fill_value*)
Create an array for a given batch size and buffer spec. Resulting array has shape = (batchsize,) + shape.

Parameters

- **batchsize** – size of the first dimension
- **shape** – remaining shape of the array
- **dtype** – numpy dtype of the array
- **fill_value** – array comes pre-filled with this value

Returns array

crumpets.broker.**make_bufferspec**(*buf*)
Turn numpy.ndarray into buffer specification: :param buf:

np.ndarray or buffer spec

Returns tuple(shape, dtype, fill_value)

crumpets.broker.**make_fill_value**(*shape*, *dtype*, *fill_value: Union[int, float, Iterable] = 0*)

Create a numpy array for a given fill value. This array can be used to fill any array of the given shape and dtype, e.g., arr[:] = make_fill_value(arr.shape, arr.dtype, 17) will set all elements of arr to 17.

Note: An implicit first dimension for the batch size is added.

fill_value can be a scalar or iterable. Iterables are padded ith unit dimensions until they match the number of dimensions of the given shape, e.g.:

```
>>> make_fill_value((3, 224, 224), np.uint8, (1, 2, 3))
array([[[[1]], [[2]], [[3]]]], dtype=uint8)
```

The resulting fill value array has shape (1, 3, 1, 1).

Parameters

- **shape** – array shape
- **dtype** – array dtype
- **fill_value** – optional fill value(s)

Returns fill value array

crumpets.broker.**unpack**(*obj*)

crumpets.dataloader module

class crumpets.dataloader.**Consumer**(*result_address*, *control_address*, *recv_timeout=1000*, *bind=True*)

Bases: object

A Consumer retrieves and forward processed samples from workers.

Parameters

- **result_address** – address to retrieve processed samples from, workers send their results to it
- **control_address** – address to retrieve control messages from, such as exceptions raised in other processes
- **recv_timeout** – time to wait in ms until another receiving attempt is made
- **bind** – bind addresses instead of connecting to them

retrieve()

retrieve_data()

set_buffer_manager(*buffer_manager*)

start()

Starts the sample retriever thread and listen on the control stream.

stop()

Stops all threads opened by this consumer.

class crumpets.dataloader.**Slicer**(*iterable*)

Bases: object

class crumpets.dataloader.**TurboDataLoader**(*iterable*, *batch_size*, *worker_template*, *nworkers*, *length=None*, *num_mini_batches=1*, *start_iteration=0*, *shared_memory=True*)

Bases: object

TurboDataLoader provides fast parallel loading and processing of input data. Use *TorchTurboDataLoader* for a version supporting gpu and pytorch tensors.

Always use the loader inside of a with statement, otherwise workers and consumer won't start and stop.

`TurboDataLoader`s are intended to be used as iterators. Each iteration yields the following data structure:

By default *iteration* starts at 0 and counts the number of batches that the loader has yielded. The list contains as many mini-batches as specified by *num_mini_batches*. Note that the number of samples across all mini-batches is equal to *batch_size*, i.e., *batch_size* must be divisible by *num_mini_batches*. Finally each mini-batch is a dictionary that contains key-value-pairs produced by the workers. E.g., a *ClassificationWorker* produces keys *'image'*, *'label'*, and *'augmentation'*. Image and label are arrays and augmentation contains a list of one dictionary per sample in the batch with parameters used to create said sample.

Example usage:

```
model = make_some_model()
with loader:
    for epoch in range(epochs):
        for iteration, mini_batch in loader:
            for sample in mini_batch:
                sample = model(sample)
                images = sample['image']
                ...
```

Depending on parameters, the TurboDataLoaders starts several processes, some of which cannot be started with the standard "fork" method that Python uses in *nix systems. This can result in crashing with an obscure error message. Thus loaders need to be guarded against starting in non-main modules, i.e.:

```
if __name__ == "__main__":
    # stuff
    with loader:
        # other stuff
```

Parameters

- **iterable** – An iterable providing a sample per iteration.
- **batch_size** – The amount of samples per batch.
- **worker_template** – An actual worker instance, determines the kind of processing. Has to inherit crumpets.broker.Worker.
- **nworkers** – Number of workers processing the samples simultaneously. worker_template is copied to create them.
- **length** – Specifies the length of the dataset. Defaults to the actual length of iterable (if available). If given differs from default, the number of iterations per epoch is modified accordingly.
- **num_mini_batches** – Number of mini_batches per batch.
- **start_iteration** – Start the iteration counter from this number. Useful when resuming training.

- **shared_memory** – Whether to use shared memory to transfer data from workers. If 0 or *False*, shared memory is disabled. If *True*, *2*nworkers* shared buffers will be used. If any number > 0, that number of buffers will be used. A value of 1 is strongly discouraged to prevent deadlocks. Permanently storing values returned by a loader may also cause deadlocks.

set_epoch_iterations(*iterations*)
Set number of iterations in one epoch. Does not modify length. :param iterations: number of iterations per epoch

set_length(*length*)
Set the length of enclosed iterable. Modifies epoch_iterations accordingly. :param length: len(iterable)

start()
Start the processing pipeline.

stop()
Stop the processing pipeline.

crumpets.dataloader.**make_addresses**(*uid*, *pipeline*, *numbers=(('work', 1), ('consume', 1))*)

crumpets.dataloader.**remove_files**(*files*)

crumpets.dataloader.**remove_ipc_handles**(*handles*)

crumpets.logging module

class crumpets.logging.**JSONLogger**(*name*, *filename*, *level=0*)
Bases: logging.Logger

A subclass of the default Python Logger that uses the JSONLines output format.

critical(***kwargs*)
Log 'msg % args' with severity 'CRITICAL'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.critical("Houston, we have a %s", "major disaster", exc_info=1)

debug(***kwargs*)
Log 'msg % args' with severity 'DEBUG'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.debug("Houston, we have a %s", "thorny problem", exc_info=1)

error(***kwargs*)
Log 'msg % args' with severity 'ERROR'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.error("Houston, we have a %s", "major problem", exc_info=1)

exception(*exc_info=True*, ***kwargs*)
Convenience method for logging an ERROR with exception information.

fatal(***kwargs*)
Log 'msg % args' with severity 'CRITICAL'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.critical("Houston, we have a %s", "major disaster", exc_info=1)

info(***kwargs*)
Log 'msg % args' with severity 'INFO'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.info("Houston, we have a %s", "interesting problem", exc_info=1)

log(*level*, ***kwargs*)
Log 'msg % args' with the integer severity 'level'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.log(level, "We have a %s", "mysterious problem", exc_info=1)

warning(***kwargs*)
Log 'msg % args' with severity 'WARNING'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.warning("Houston, we have a %s", "bit of a problem", exc_info=1)

class crumpets.logging.**ProgressPrinter**(*_, **__)
Bases: tqdm.std.tqdm

class crumpets.logging.**SilentLogger**
Bases: object

Replacement logger for a logger that does not log anything. Useful when running multiple processes, but not all of them should log results.

critical(***kwargs*)

debug(***kwargs*)

error(***kwargs*)

exception(*exc_info=True*, ***kwargs*)

fatal(***kwargs*)

info(***kwargs*)

log(*level*, ***kwargs*)

warning(***kwargs*)

crumpets.logging.**get_logfilename**(*prefix=''*, *dateformat='%Y-%m-%dt%H-%M-%S'*, *pathformat='%s%s.log'*)

crumpets.logging.**make_printer**(*bar_format='{desc} {percentage:3.0f}% {elapsed}<{remaining}, {rate_fmt}{postfix}'*, *miniters=0*, *mininterval=0.5*, *smoothing=0.1*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, ***kwargs*)
Create a ProgressPrinter with appropriate parameters for training. See tqdm documentation for details on parameters. :return:

crumpets.logging.**print**(**args*)
Overwrites the builtin print function with *tqdm.tqdm.write*, so things are printed properly while tqdm is active. :param args: :return:

crumpets.presets module

crumpets.procname module

crumpets.procname.**setprocname**(*_, **__)

crumpets.rng module

class crumpets.rng.**MixtureRNG**(*prob=1*, *scale_range=None*, *shift_range=None*, *noise_range=None*, *brightness_range=None*, *color_range=None*, *contrast_range=None*, *blur_range=None*, *rotation_sigma=0*, *aspect_sigma=0*, *interpolations=None*, *hmirror=0*, *vmirror=0*, *shear_range=None*)

Bases: *crumpets.rng.RNG*

Old crumpets 2-style RNG that uses a mix of base probability, Gaussian and uniform distributions to generate parameters. See *randomize_image()* for more details on allowed values.

..note: **_range* parameters must be 2 numbers *(a,b)*. They define a uniform distribution *U[a,b]*

..note: **_sigma* parameters define the standard deviation of a Gaussian distribution.

Parameters

- **prob** – probability that pixel-based augmentations are applied; each augmentation rolls separately; does not apply to spatial transforms like scale, rotation, etc.
- **scale_range** – influences the *'scale'* value
- **shift_range** – influences the *'shift'* values
- **noise_range** – influences the *'noise'* value
- **brightness_range** – influences the *'gamma_gray'* value; brightness is converted to gamma by `convert_gamma()`
- **color_range** – influences the *'gamma_color'* values; color values are converted to gamma by `convert_gamma()`
- **contrast_range** – influences the *'contrast'* value
- **blur_range** – influences the *'blur'* value; effective sigma for Gaussian blur is *blur*image_width*
- **rotation_sigma** – influences the *'angle'* value
- **aspect_sigma** – influences the *'aspect'* value
- **interpolations** – list of possible interpolation methods to use; influences the *'interp_method'* value
- **hmirror** – probability that horizontal mirror is applied; influences the *'hmirror'* value
- **vmirror** – probability that vertical mirror is applied; influences the *'vmirror'* value
- **shear_range** – influences the *'shear'* values

class crumpets.rng.**NoRNG**

Bases: *crumpets.rng.RNG*

Decidedly non-random RNG that simply returns an empty dict. Useful for validation runs.

class crumpets.rng.**RNG**
Bases: object

Abstract base class for augmentation random number generators (RNG). For further information about the semantics of individual parameters, have a look at *randomize_image()*.

crumpets.rng.**human2gamma**(*human*)
Convert some human-understandable value to a value that is suitable for use in gamma correction.

Values outside of *[-1,1]* are clipped.

Parameters **human** – value to convert to gamma

Returns value suitable for gamma correction

crumpets.shm module

class crumpets.shm.**DummyBufferManager**
Bases: object

Dummy replacement for SharedBufferManager. Supports pack and unpack, but not next methods.

next()

pack(*obj*)
Pack an object using msgpack. :param obj: object to pack :return: msgpack message bytes

unpack(*data*)
Unpack an msgpack message. :param data: msgpack message bytes :return: packed objects

class crumpets.shm.**SharedBufferManager**(*num_buffers*, *batch_size*, *buffer_specs*, *_queueclass=<bound method BaseContext.Queue of <multiprocessing.context.DefaultContext object>>*)
Bases: object

SharedBufferManager allows transparent sharing of memory between processes. On creation the specified number of shared memory buffers are created according to batch size and buffer specs.

next returns dict of numpy arrays that point to a set of shared memory buffers. *next* blocks until as set of buffers becomes available. If more than one buffer spec is given, next will always return one buffer for each spec and will only reuse a set of buffers when none of them are in use.

pack serializes an arbitrary python object to msgpack format. It detects shared buffers and replaces them with a "pointer" as extension type *EXT_SHARED*. This makes packing fast and independent of array size.

unpack detects "pointer" and replaces them with the shared buffer.

Usage:

- Sender calls next to get a set ob available buffers.
- Sender modifies buffers, calls pack and sends message to receiver.
- Receiver receives the message and calls unpack.
- Receiver uses the unpacked arrays and ensures that they are deleted at some point, either by going out of scope or explicitly deleting them. Storing shared buffers permanently may cause deadlocks.

close()
Close the queue and unblock any processes waiting on next.

next()

pack(*obj*)
Pack an object using msgpack. Any shared object are replaced by references. :param obj: object to pack :return: msgpack message bytes

unpack(*data*)
Unpack an msgpack message. Any shared object references are replaced with the object. :param data: msgpack message bytes :return: packed objects

crumpets.shm.**shared_array**(*shape*, *dtype=<class 'numpy.float32'>*)
Create a numpy array that resides in shared memory. Memory is aligned to 8 bytes.

Parameters

- **shape** – array shape
- **dtype** – numpy dtype

Returns np.ndarray

crumpets.timing module

class crumpets.timing.**ETATimer**(*goal*)
Bases: *crumpets.timing.RemainingTimer*

Simple modification of the RemainingTimer. When called, instead of returning remainin time, return ETA (estimated time of arrival).

class crumpets.timing.**RemainingTimer**(*goal*)
Bases: object

Estimates remaining time of some operation. Each time it is called, an internal counter is increased by 1. The measured operation is assumed to be done once this counter reaches goal. Out of the frequency of the calls one can compute an estimated speed and thus remaining time.

Parameters **goal** – the number of calls until the operation is assumed to be done.

crumpets.timing.**nicetime**(*seconds*)
formats given time in form of seconds to a nice string representation.

Returns string representation of time

CHAPTER

FOUR

EXAMPLES

4.1 examples package

4.1.1 Submodules

examples.dataloader_datadings module

examples.dataloader_simple module

Simple dataloader example without using datadings. The example prepares the included tinyset dataset, which is given in ./tinyset and in the form of imangenet's folder structure. Afterwards crumpets TurboDataLoader is created and run through.

examples.dataloader_simple.**main**(*show=True*, *wait_key=2000*)

examples.dataloader_simple.**prepare_dataset**(*dsdir*)

We have to prepare our example dataset tinyset s.t. we have encoded images and labels. Crumpets default worker expect msgpack packed dictionaries. Thus we have to create an iterable of packed elements, which unpacked are of form: {'image': ..., 'label': ...}.

Parameters **dsdir** – path to dataset directory

Returns iterable of msgpack packed directories

examples.pytorch_cifar10 module

Example usage of crumpets to train a custom model on Cifar10. Less complex compared to resnet example, since less parameters are considered (some are just set to their default value to make the example more intuitive). Cifar10 can either be processed to be in msgpack format or directly downloaded, using Datadings. This example is capable of using multiple gpus. If no datadir is given a default sample of 10 images is used while the loader is told that there are 2000 images to mimic a real dataset.

class examples.pytorch_cifar10.**Net**(**args: Any*, ***kwargs: Any*)

Bases: torch.nn.

forward(*sample*)

examples.pytorch_cifar10.**main**(*datadir*, *outdir*, *batch_size*, *epochs*, *lr*, *cuda=True*)

examples.pytorch_cifar10.**make_loader**(*file*, *batch_size*, *num_mini_batches*, *nworkers*, *image_rng=None*, *image_params=None*, *use_cuda=True*, *gpu_augmentation=True*)

examples.pytorch_cifar10.**make_policy**(*epochs*, *network*, *lr*, *momentum*)

examples.pytorch_resnet module

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex

CHAPTER

SIX

FURTHER INFORMATION

Tough cookie.

PYTHON MODULE INDEX

c

e

INDEX

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Y