torchimage

Release 0.0.1

Tianyi Miao

Jul 02, 2021

CONTENTS

CHAPTER

ONE

TORCHIMAGE PACKAGE

1.1 Subpackages

1.1.1 torchimage.cfa package

Subpackages

torchimage.cfa.bayer_conv package

Submodules

torchimage.cfa.bayer_conv.bayer_conv_2d module

Customized neural network layer for bayer array convolution 4 distinct filter groups for R, B, GR, and GB

`class torchimage.cfa.bayer_conv.bayer_conv_2d.`**`BayerConv2d`**(*in_channels*, *out_channels*, *kernel_size*, *bias*)

Bases: `torch.nn.modules.module.Module`

`forward`(*x: torch.Tensor*, *sensor_alignment: str*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`forward_unshuffle`(*x: torch.Tensor*, *sensor_alignment: str*)

`training: bool`

`class torchimage.cfa.bayer_conv.bayer_conv_2d.`**`BayerConv2dUnshuffle`**(*in_channels*, *out_channels*, *kernel_size*, *bias*)

Bases: *`torchimage.cfa.bayer_conv.bayer_conv_2d.BayerConv2d`*

`forward`(*x: torch.Tensor*, *sensor_alignment: str*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

> **Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

`class torchimage.cfa.bayer_conv.bayer_conv_2d.`**`PreTrainedBilinearInterpolator`**`(`*weight_dict: dict*`)`

Bases: `torch.nn.modules.module.Module`

`forward``(`*x*, *sensor_alignment*, *clamp=1.0*`)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

> **Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

`torchimage.cfa.bayer_conv.bayer_conv_2d.`**`get_padding_layer`**`(`*kernel_size*, *beg_row*, *beg_col*`)`

The padding layer accepts a batch of unsqueezed bayer arrays of shape (n_samples, c, height, width) and returns the batch where the bayer arrays are padded (n_samples, c, height + kernel_size - 2, width + kernel_size - 2)

The padding layer serves 2 purposes: 1. Align the center of the convolutional kernel to the target pixels in the 2D bayer array.

> Because the convolutional kernel always starts "sliding" at top left, we can control the offset (where the center is/where the kernel begins)

2. Deal with borders elegantly. The intuition of bilinear interpolation stems from "taking the average of nearby

   pixels" of nearest neighbor algorithms, so when the only available neighbors are on on side, taking just that neighbor's value is equivalent to taking the average of two copies.

`torchimage.cfa.bayer_conv.bayer_conv_2d.`**`get_sensor_beg_index`**`(`*sensor_alignment*`)`

Input: sensor alignment specification (str) such as "GRBG" Output: dict that maps pixel type (i.e. GR for G at R row) to its starting index (i.e. (0, 0)) at mod 2

`torchimage.cfa.bayer_conv.bayer_conv_2d.`**`last_step_demosaic`**`(`*x: torch.Tensor*, *y: torch.Tensor*, *sensor_alignment: str*`)`

torchimage.cfa.bayer_conv.gradient_corrected module

Gradient-corrected bilinear interpolation, the algorithm in MATLAB's demosaic function. Implemented as a collection of (5, 5) filters

Module contents

`class torchimage.cfa.bayer_conv.`**`BayerConv2d`**(*in_channels*, *out_channels*, *kernel_size*, *bias*)

Bases: `torch.nn.modules.module.Module`

`forward`(*x: torch.Tensor*, *sensor_alignment: str*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`forward_unshuffle`(*x: torch.Tensor*, *sensor_alignment: str*)

`training: bool`

`class torchimage.cfa.bayer_conv.`**`BayerConv2dUnshuffle`**(*in_channels*, *out_channels*, *kernel_size*, *bias*)

Bases: *`torchimage.cfa.bayer_conv.bayer_conv_2d.BayerConv2d`*

`forward`(*x: torch.Tensor*, *sensor_alignment: str*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

`torchimage.cfa.bayer_conv.`**`last_step_demosaic`**(*x: torch.Tensor*, *y: torch.Tensor*, *sensor_alignment: str*)

torchimage.cfa.utils package

Submodules

torchimage.cfa.utils.bayer_1c_4c module

Conversion between 2 alternative representations of a Bayer array.

1-channel representation: `(..., 1, 2h, 2w)`

4-channel representation: `(..., 4, h, w)` where the 4 color channels are separated

Notice that the depth dimension (number of color channels) is always on axis `-3`. Negative indexing is more compatible with different dataset shapes.

A single bayer array under 1-channel representation is a `(1, 2h, 2w)` tessellation like the one below, where `ABCD` can be any of the `RGGB`, `BGGR`, `GRBG`, `GBRG` sensor alignments:

```
ABABAB
CDCDCD
ABABAB
CDCDCD
```

When converted to 4-channel representation, the same bayer array will be as follows (stack of 4 2D arrays):

```
AAA BBB CCC DDD
AAA BBB CCC DDD
```

The process is deterministic and fully invertible (doesn't involve any loss of information).

See also:

`Demosaic`

`torchimage.cfa.utils.bayer_1c_4c.`**`bayer_1c_to_4c`**`(`*x: torch.Tensor*`)`

Convert 1-channel bayer array to 4-channel bayer array.

Input shape is automatically validated, so it's not necessary to check again outside of the scope.

Parameters **`x`** (*`torch.Tensor`*) – Input 1-channel Bayer array of shape `(..., 1, h, w)`

Returns

y – Output 4-channel Bayer array of shape `(..., 4, h//2, w//2)`

Has the same dtype and device as the input tensor.

Return type torch.Tensor

`torchimage.cfa.utils.bayer_1c_4c.`**`bayer_4c_to_1c`**`(`*x: torch.Tensor*`)`

Convert 4-channel bayer array to 1-channel bayer array.

Input shape is automatically validated, so it's not necessary to check again outside of the scope.

Parameters **`x`** (*`torch.Tensor`*) – Input 4-channel Bayer array of shape `(..., 4, h, w)`

Returns

y – Output 1-channel Bayer array of shape `(..., 1, 2*h, 2*w)`

Has the same dtype and device as the input tensor.

Return type torch.Tensor

torchimage.cfa.utils.bayer_to_rgb module

Lightweight, deterministic methods to convert bayer images to RGB images.

Currently, only downsampling is implemented. A bayer image of shape `(..., 1, 2h, 2w)` or `(..., 4, h, w)` will be converted to an RGB image of shape `(..., 3, h, w)`. We condense every 2-by-2 block of bayer detector results into a single pixel, taking R pixel's value as the new R, B pixel's value as the new B, and the average of 2 G pixel's values as the new G.

For more advanced demosaicing algorithms (such as nearest neighbor, gradient-corrected bilinear interpolation, or neural networks), please refer to other modules in `demosaic`.

`torchimage.cfa.utils.bayer_to_rgb.`**`downsample_1c_bayer_to_rgb`**`(`*x: torch.Tensor*, *sensor_alignment: str*`)`

Downsample a 1-channel bayer image to an RGB image.

Input shape and sensor alignment are automatically validated, so it's not necessary to check again outside of the scope.

Parameters

- **`x`** (*`torch.Tensor`*) – Input 1-channel bayer array of shape `(..., 1, 2h, 2w)`
- **`sensor_alignment`** (*`str`*) – Sensor alignment / bayer pattern

Returns

Output 3-channel RGB array of shape `(..., 3, h, w)`.

Note that the output height and width are halved.

Return type torch.Tensor

`torchimage.cfa.utils.bayer_to_rgb.`**`downsample_4c_bayer_to_rgb`**`(`*x: torch.Tensor*, *sensor_alignment: str*`)`

Downsample a 4-channel bayer image to an RGB image.

Input shape and sensor alignment are automatically validated, so it's not necessary to check again outside of the scope.

Parameters

- **`x`** (*`torch.Tensor`*) – Input 4-channel bayer array of shape `(..., 4, h, w)`
- **`sensor_alignment`** (*`str`*) – Sensor alignment / bayer pattern

Returns Output 3-channel RGB array of shape `(..., 3, h, w)`.

Return type torch.Tensor

torchimage.cfa.utils.rgb_to_bayer module

Convert RGB images to bayer images by adding mosaic.

For every pixel, 2 out of the 3 colors will be discarded while only 1 remains. Which color remains depends on the sensor alignment specified by the parameters. Therefore, this process is:

1. Irreversible: Since converting RGB to bayer involves a loss of information, its inverse (demosaicing) is a much harder problem. 2. Deterministic

This module works with generalized nchw format.

We separate converting to 1-channel bayer and 4-channel bayer into 2 functions. This is because conversion from RGB to bayer, as well as conversion between different bayer formats, all requires creating new tensors in memory. 2-step conversion (e.g. RGB to 1c bayer, then to 4c bayer) will bear the overhead from assigning and releasing memory for intermediate tensors.

See also:

`MatLab`

`torchimage.cfa.utils.rgb_to_bayer.`**`rgb_to_1c_bayer`**`(`*x: torch.Tensor*, *sensor_alignment: str*`)`

Convert RGB image to 1-channel bayer image.

Input shape and sensor alignment are automatically validated, so it's not necessary to check again outside of the scope.

Parameters

- **`x`** (*`torch.Tensor`*) – Input RGB image of shape `(..., 3, h, w)`, where `h` and `w` must be even
- **`sensor_alignment`** (*`str`*) – Sensor alignment / bayer pattern

Returns

y – Output 1-channel bayer image of shape `(..., 1, h, w)`.

Has the same dtype and device as input.

Return type torch.Tensor

`torchimage.cfa.utils.rgb_to_bayer.`**`rgb_to_4c_bayer`**`(`*x: torch.Tensor*, *sensor_alignment: str*`)`

Convert RGB image to 4-channel bayer image.

Input shape and sensor alignment are automatically validated, so it's not necessary to check again outside of the scope.

Parameters

- **`x`** (*`torch.Tensor`*) – Input RGB image of shape `(..., 3, h, w)`, where `h` and `w` must be even
- **`sensor_alignment`** (*`str`*) – Sensor alignment / bayer pattern

Returns

y – Output 4-channel bayer image of shape `(..., 4, h//2 , w//2)`.

Has the same dtype and device as input.

Return type torch.Tensor

torchimage.cfa.utils.validation module

This module contains various validation tools for bayer and rgb images.

The validation tools deal with generalized nchw format: *(number of samples, color channels, height, width)*. The number of samples can span multiple dimensions, such as *(number of batches, number of samples in each batch, . . .)*.

`torchimage.cfa.utils.validation.`**`check_1c_bayer`**`(`*x: torch.Tensor*`)`

Check if the input consists of 1-channel bayer arrays

Parameters **`x`** (*`torch.Tensor`*) – Should be 1-channel Bayer array of shape `(..., 1, h, w)`

Raises **`AssertionError`** – Whenever the input fails a validation criterion. The assertion line itself should be helpful enough.

`torchimage.cfa.utils.validation.`**`check_4c_bayer`**`(`*x: torch.Tensor*`)`

Check if the input consists of 4-channel bayer arrays

Parameters **`x`** (*`torch.Tensor`*) – Should be 4-channel Bayer array of shape `(..., 4, h, w)`

Raises **`AssertionError`** – Whenever the input fails a validation criterion. The assertion line itself should be helpful enough.

`torchimage.cfa.utils.validation.`**`check_rgb`**`(`*x: torch.Tensor*`)`

Check if the input is a valid RGB image (or image batch) in generalized nchw format.

Parameters **`x`** (*`torch.Tensor`*) – Should be a 3-channel array of shape `(..., 3, h, w)`.

The color channels should follow the exact order of Red, Green, and Blue, but this order can't and won't be verified.

Raises **`AssertionError`** – Whenever the input fails a validation criterion. The assertion line itself should be helpful enough.

`torchimage.cfa.utils.validation.`**`check_sensor_alignment`**`(`*sensor_alignment*`)`

Check if the input is a valid bayer sensor alignment pattern.

There are only 4 possible bayer sensor alignment patterns: *GBRG*, *GRBG*, *BGGR*, *RGGB*.

Parameters **`sensor_alignment`** (*`str`*) – Should be one of *"GBRG"*, *"GRBG"*, *"BGGR"*, and *"RGGB"*. Lower case letters are allowed but discouraged.

Returns **sensor_alignment** – The sensor alignment itself if the validation is successful. Lower case letters will be automatically converted to upper case.

Return type str

Raises **`AssertionError`** – Whenever the input fails a validation criterion. The assertion line itself should be helpful enough.

Module contents

This submodule contains tools for converting between 1-channel and 4-channel bayer images, converting RGB images to bayer images, and validating input tensors or keywords for such functionalities.

`torchimage.cfa.utils.`**`bayer_1c_to_4c`**`(`*x: torch.Tensor*`)`

Convert 1-channel bayer array to 4-channel bayer array.

Input shape is automatically validated, so it's not necessary to check again outside of the scope.

Parameters **`x`** (*`torch.Tensor`*) – Input 1-channel Bayer array of shape `(..., 1, h, w)`

Returns

y – Output 4-channel Bayer array of shape `(..., 4, h//2, w//2)`

Has the same dtype and device as the input tensor.

Return type torch.Tensor

`torchimage.cfa.utils.`**`bayer_4c_to_1c`**`(`*x: torch.Tensor*`)`

Convert 4-channel bayer array to 1-channel bayer array.

Input shape is automatically validated, so it's not necessary to check again outside of the scope.

Parameters **`x`** (*`torch.Tensor`*) – Input 4-channel Bayer array of shape `(..., 4, h, w)`

Returns

y – Output 1-channel Bayer array of shape `(..., 1, 2*h, 2*w)`

Has the same dtype and device as the input tensor.

Return type torch.Tensor

`torchimage.cfa.utils.`**`check_1c_bayer`**`(`*x: torch.Tensor*`)`

Check if the input consists of 1-channel bayer arrays

Parameters **`x`** (*`torch.Tensor`*) – Should be 1-channel Bayer array of shape `(..., 1, h, w)`

Raises **`AssertionError`** – Whenever the input fails a validation criterion. The assertion line itself should be helpful enough.

`torchimage.cfa.utils.`**`check_4c_bayer`**`(`*x: torch.Tensor*`)`

Check if the input consists of 4-channel bayer arrays

Parameters **`x`** (*`torch.Tensor`*) – Should be 4-channel Bayer array of shape `(..., 4, h, w)`

Raises **`AssertionError`** – Whenever the input fails a validation criterion. The assertion line itself should be helpful enough.

`torchimage.cfa.utils.`**`check_rgb`**`(`*x: torch.Tensor*`)`

Check if the input is a valid RGB image (or image batch) in generalized nchw format.

Parameters **`x`** (*`torch.Tensor`*) – Should be a 3-channel array of shape `(..., 3, h, w)`.

The color channels should follow the exact order of Red, Green, and Blue, but this order can't and won't be verified.

Raises **`AssertionError`** – Whenever the input fails a validation criterion. The assertion line itself should be helpful enough.

`torchimage.cfa.utils.`**`check_sensor_alignment`**`(`*sensor_alignment*`)`

Check if the input is a valid bayer sensor alignment pattern.

There are only 4 possible bayer sensor alignment patterns: *GBRG*, *GRBG*, *BGGR*, *RGGB*.

Parameters **`sensor_alignment`** (*`str`*) – Should be one of *"GBRG"*, *"GRBG"*, *"BGGR"*, and *"RGGB"*. Lower case letters are allowed but discouraged.

Returns **sensor_alignment** – The sensor alignment itself if the validation is successful. Lower case letters will be automatically converted to upper case.

Return type str

Raises **`AssertionError`** – Whenever the input fails a validation criterion. The assertion line itself should be helpful enough.

`torchimage.cfa.utils.`**`downsample_1c_bayer_to_rgb`**`(`*x: torch.Tensor*, *sensor_alignment: str*`)`

Downsample a 1-channel bayer image to an RGB image.

Input shape and sensor alignment are automatically validated, so it's not necessary to check again outside of the scope.

Parameters

- **`x`** (*`torch.Tensor`*) – Input 1-channel bayer array of shape `(..., 1, 2h, 2w)`
- **`sensor_alignment`** (*`str`*) – Sensor alignment / bayer pattern

Returns

Output 3-channel RGB array of shape `(..., 3, h, w)`.

Note that the output height and width are halved.

Return type torch.Tensor

`torchimage.cfa.utils.`**`downsample_4c_bayer_to_rgb`**`(`*x: torch.Tensor*, *sensor_alignment: str*`)`

Downsample a 4-channel bayer image to an RGB image.

Input shape and sensor alignment are automatically validated, so it's not necessary to check again outside of the scope.

Parameters

- **`x`** (*`torch.Tensor`*) – Input 4-channel bayer array of shape `(..., 4, h, w)`
- **`sensor_alignment`** (*`str`*) – Sensor alignment / bayer pattern

Returns Output 3-channel RGB array of shape `(..., 3, h, w)`.

Return type torch.Tensor

`torchimage.cfa.utils.`**`rgb_to_1c_bayer`**`(`*x: torch.Tensor*, *sensor_alignment: str*`)`

Convert RGB image to 1-channel bayer image.

Input shape and sensor alignment are automatically validated, so it's not necessary to check again outside of the scope.

Parameters

- **`x`** (*`torch.Tensor`*) – Input RGB image of shape `(..., 3, h, w)`, where `h` and `w` must be even
- **`sensor_alignment`** (*`str`*) – Sensor alignment / bayer pattern

Returns

y – Output 1-channel bayer image of shape `(..., 1, h, w)`.

Has the same dtype and device as input.

Return type torch.Tensor

`torchimage.cfa.utils.`**`rgb_to_4c_bayer`**`(`*x: torch.Tensor*, *sensor_alignment: str*`)`

Convert RGB image to 4-channel bayer image.

Input shape and sensor alignment are automatically validated, so it's not necessary to check again outside of the scope.

Parameters

- **`x`** (*`torch.Tensor`*) – Input RGB image of shape `(..., 3, h, w)`, where `h` and `w` must be even
- **`sensor_alignment`** (*`str`*) – Sensor alignment / bayer pattern

Returns

y – Output 4-channel bayer image of shape `(..., 4, h//2 , w//2)`.

Has the same dtype and device as input.

Return type torch.Tensor

Module contents

Algorithms related to color filter arrays (CFA)

1.1.2 torchimage.filtering package

Submodules

torchimage.filtering.generic module

Module contents

`class torchimage.filtering.`**`EdgeFilter`**`(`*edge_kernel*, *smooth_kernel*, ***, *normalize=False*, *same_padder=None*`)`

Bases: `torch.nn.modules.module.Module`

`all_components``(`*x*, *axes=None*`)`

`component``(`*x*, *edge_axis*, *smooth_axes*`)`

`horizontal``(`*x*`)`

`magnitude``(`*x*, *axes=None*, ***, *epsilon=0.0*, *p=2*`)`

`training: bool`

`vertical``(`*x*`)`

`class torchimage.filtering.`**`Farid`**`(`***, *normalize=False*, *same_padder='reflect'*`)`

Bases: `torchimage.filtering.edges.EdgeFilter`

`training: bool`

`class torchimage.filtering.`**`GaussianGrad`**`(`*kernel_size*, *sigma*, *edge_kernel_size=None*, *edge_sigma=None*, *normalize=True*, *edge_order=1*, *same_padder='reflect'*`)`

Bases: `torchimage.filtering.edges.EdgeFilter`

`training: bool`

`class torchimage.filtering.`**`Laplace`**`(`***, *same_padder='reflect'*`)`

Bases: `torch.nn.modules.module.Module`

Edge detection with discrete Laplace operator

Unlike SeparablePoolNd, which sequentially applies 1d convolution on previous output at each axis, Laplace simultaneously applies the kernel to each axis, generating n output tensors in parallel; these output tensors are then added to obtain a final output. Therefore, the equivalent kernel of SeparablePoolNd is the outer product of each 1d kernel; the equivalent kernel of LaplacePoolNd is the sum of 1d kernels (after they are expanded to the total number of dimensions).

For instance, 1d Laplace is [1, -2, 1] and 2d Laplace is [[0, 1, 0],

[1, -4, 1], [0, 1, 0]]

This method is less general than scipy's generic_laplace, because we cannot customize a non-separable second derivative function.

This requires every 1d pooling to return a tensor of exactly the same shape, so we recommend `same=True`.

`forward``(`*x: torch.Tensor*, *axes*`)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

`class torchimage.filtering.`**`LaplacianOfGaussian`**`(`*kernel_size*, *sigma*, ***, *same_padder='reflect'*`)`

Bases: `torch.nn.modules.module.Module`

The same as scipy.ndimage.gaussian_laplace

`forward``(`*x: torch.Tensor*, *axes=None*`)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

`class torchimage.filtering.`**`Prewitt`**`(`***, *normalize=False*, *same_padder='reflect'*`)`

Bases: *`torchimage.filtering.edges.EdgeFilter`*

`training: bool`

`class torchimage.filtering.`**`Scharr`**`(`***, *normalize=False*, *same_padder='reflect'*`)`

Bases: *`torchimage.filtering.edges.EdgeFilter`*

`training: bool`

`class torchimage.filtering.`**`Sobel`**`(`***, *normalize=False*, *same_padder='reflect'*`)`

Bases: *`torchimage.filtering.edges.EdgeFilter`*

`training: bool`

`class torchimage.filtering.`**`UnsharpMask`**`(`*blur:* torchimage.pooling.base.BasePoolNd, *amount=0.5*, *threshold=0*, ***, *padder: Optional[*torchimage.padding.generic_pad.Padder*] = None*`)`

Bases: `object`

Sharpen an image with unsharp masking.

The basic formula of unsharp masking is: y = x + amount * (x - blur(x))

`blur`

Smoothing (blurring) filter for unsharp masking.

A filter should output a tensor whose shape is completely the same as the input tensor. The *to_filter* method from base pooling will be automatically called here, so the user only needs to specify essential parameters, such as `GaussianPoolNd(kernel_size=7, sigma=1.5)` or `AvgPoolNd(kernel_size=5)`

Type *BasePoolNd*

amount [float] The “amount” of sharpening applied to the image.

See the formula for unsharp masking for more detailed explanation.

threshold [float] Ignore differences between x and blur(x) that are below threshold. Default: `0.0`

`forward`(*x: torch.Tensor*, *axes=slice(2, None, None)*)

1.1.3 torchimage.linalg package

Submodules

torchimage.linalg.outer module

Module contents

1.1.4 torchimage.metrics package

Subpackages

torchimage.metrics.hdrvdp package

Submodules

torchimage.metrics.hdrvdp.hdrvdp3 module

`torchimage.metrics.hdrvdp.hdrvdp3.`**`hdrvdp3`**(*image_true*, *image_test*, *task*, *color_encoding*, *pixels_per_degree*, *surround=None*, *age=24*, *spectral_emission=None*, *mtf='hdrvdp'*, *rgb_display=None*, *sensitivity_correction=0.0*, *mask_p=None*, *mask_q=None*)

torchimage.metrics.hdrvdp.pixels_per_degree module

`torchimage.metrics.hdrvdp.pixels_per_degree.`**`pixels_per_degree`**(*display_diagonal_mm*, *height_pix*, *width_pix*, *viewing_distance_m*)

computer pixels per degree given display parameters and viewing distance

This is a convenience function that can be used to provide angular resolution of input images for the HDR-VDP-2.

Note that the function assumes 'square' pixels, so that the aspect ratio is resolution[0]:resolution[1].

Parameters

- **display_diagonal_mm** (*int or float*) – diagonal display size in millimeters
- **height_pix** (*int*) – display resolution in pixels as a pair of int, e.g. (1024, 768)
- **width_pix** (*int*) – display resolution in pixels as a pair of int, e.g. (1024, 768)
- **viewing_distance_m** (*float*) – viewing distance in meters, e.g. 0.5

Module contents

Submodules

torchimage.metrics.mse module

class `torchimage.metrics.mse.`**`MSE`**(*, *reduction='mean'*)

Bases: `torchimage.metrics.base.BaseMetric`

`forward_full`(*y_pred: torch.Tensor*, *y_true: torch.Tensor*)

`training: bool`

torchimage.metrics.psnr module

class `torchimage.metrics.psnr.`**`PSNR`**(*max_value=1*, *eps=0.0*, *, *reduction='mean'*)

Bases: *`torchimage.metrics.mse.MSE`*

`forward`(*y_pred: torch.Tensor*, *y_true: torch.Tensor*, *reduce_axes=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

torchimage.metrics.ssim module

class `torchimage.metrics.ssim.`**`MS_SSIM`**(*weights=None*, *use_prod=True*, *blur: torchimage.pooling.base.BasePoolNd = 'gaussian'*, *padder=None*, *K1=0.01*, *K2=0.03*, *eps=1e-08*, *use_sample_covariance=True*, *crop_border=True*)

Bases: *`torchimage.metrics.ssim.SSIM`*

`forward`(*y_pred: torch.Tensor*, *y_true: torch.Tensor*, *content_axes=slice(2, None, None)*, *reduce_axes=slice(1, None, None)*, *full=False*)

Parameters

- **`y_pred`** (*`torch.Tensor`*) – The first input tensor. Order doesn't matter because SSIM is symmetric with respect to input images.
- **`y_true`** (*`torch.Tensor`*) – The second input tensor.
- **`content_axes`** (*`None, int, slice, tuple`*) – Axes that describe the "content" of an image. This includes depth, height, and width but excludes batch or channel dimensions.
- **`reduce_axes`** (*`None, int, slice, tuple`*) – The final SSIM score will average the full SSIM map across these axes.

  If reduce_axes is None (all axes are reduce axes), the output score will be a scalar (useful as a loss function). If reduce_axes doesn't include batch axes, then it returns a 1d tensor of SSIM scores for every data point.

- **full** (*bool*) – Whether to return the full SSIM map as well. Default: False.

Returns

- **score** (*torch.Tensor*) – The SSIM score tensor where content axes are reduced.
- **full_tensor** (*torch.Tensor*) – The full output SSIM with the same shape as input images. This argument is only returned when `full=True`.

`property n_levels`

`training: bool`

`class torchimage.metrics.ssim.SSIM`(*blur: torchimage.pooling.base.BasePoolNd = 'gaussian', padder=None, K1=0.01, K2=0.03, use_sample_covariance=True, crop_border=True*)

Bases: `torch.nn.modules.module.Module`

`forward`(*y_pred: torch.Tensor, y_true: torch.Tensor, content_axes=slice(2, None, None), reduce_axes=slice(1, None, None), *, full=False*)

Parameters

- **y_pred** (*torch.Tensor*) – The first input tensor. Order doesn't matter because SSIM is symmetric with respect to input images.
- **y_true** (*torch.Tensor*) – The second input tensor.
- **content_axes** (*None, int, slice, tuple*) – Axes that describe the "content" of an image. This includes depth, height, and width but excludes batch or channel dimensions.
- **reduce_axes** (*None, int, slice, tuple*) – The final SSIM score will average the full SSIM map across these axes.

  If reduce_axes is None (all axes are reduce axes), the output score will be a scalar (useful as a loss function). If reduce_axes doesn't include batch axes, then it returns a 1d tensor of SSIM scores for every data point.
- **full** (*bool*) – Whether to return the full SSIM map as well. Default: False.

Returns

- **score** (*torch.Tensor*) – The SSIM score tensor where content axes are reduced.
- **full_tensor** (*torch.Tensor*) – The full output SSIM with the same shape as input images. This argument is only returned when `full=True`.

`training: bool`

Module contents

`class torchimage.metrics.MSE`(**, reduction='mean'*)

Bases: `torchimage.metrics.base.BaseMetric`

`forward_full`(*y_pred: torch.Tensor, y_true: torch.Tensor*)

`training: bool`

`class torchimage.metrics.MS_SSIM`(*weights=None, use_prod=True, blur: torchimage.pooling.base.BasePoolNd = 'gaussian', padder=None, K1=0.01, K2=0.03, eps=1e-08, use_sample_covariance=True, crop_border=True*)

Bases: `torchimage.metrics.ssim.SSIM`

`forward`(*y_pred: torch.Tensor*, *y_true: torch.Tensor*, *content_axes=slice(2, None, None)*, *reduce_axes=slice(1, None, None)*, *full=False*)

Parameters

- **y_pred** (*torch.Tensor*) – The first input tensor. Order doesn't matter because SSIM is symmetric with respect to input images.
- **y_true** (*torch.Tensor*) – The second input tensor.
- **content_axes** (*None, int, slice, tuple*) – Axes that describe the "content" of an image. This includes depth, height, and width but excludes batch or channel dimensions.
- **reduce_axes** (*None, int, slice, tuple*) – The final SSIM score will average the full SSIM map across these axes.

  If reduce_axes is None (all axes are reduce axes), the output score will be a scalar (useful as a loss function). If reduce_axes doesn't include batch axes, then it returns a 1d tensor of SSIM scores for every data point.
- **full** (*bool*) – Whether to return the full SSIM map as well. Default: False.

Returns

- **score** (*torch.Tensor*) – The SSIM score tensor where content axes are reduced.
- **full_tensor** (*torch.Tensor*) – The full output SSIM with the same shape as input images. This argument is only returned when `full=True`.

property n_levels

training: bool

class `torchimage.metrics.PSNR`(*max_value=1*, *eps=0.0*, ***, *reduction='mean'*)

Bases: `torchimage.metrics.mse.MSE`

`forward`(*y_pred: torch.Tensor*, *y_true: torch.Tensor*, *reduce_axes=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class `torchimage.metrics.SSIM`(*blur: torchimage.pooling.base.BasePoolNd = 'gaussian'*, *padder=None*, *K1=0.01*, *K2=0.03*, *use_sample_covariance=True*, *crop_border=True*)

Bases: `torch.nn.modules.module.Module`

`forward`(*y_pred: torch.Tensor*, *y_true: torch.Tensor*, *content_axes=slice(2, None, None)*, *reduce_axes=slice(1, None, None)*, ***, *full=False*)

Parameters

- **y_pred** (*torch.Tensor*) – The first input tensor. Order doesn't matter because SSIM is symmetric with respect to input images.
- **y_true** (*torch.Tensor*) – The second input tensor.

- **`content_axes`** (*`None, int, slice, tuple`*) – Axes that describe the "content" of an image. This includes depth, height, and width but excludes batch or channel dimensions.
- **`reduce_axes`** (*`None, int, slice, tuple`*) – The final SSIM score will average the full SSIM map across these axes.

  If reduce_axes is None (all axes are reduce axes), the output score will be a scalar (useful as a loss function). If reduce_axes doesn't include batch axes, then it returns a 1d tensor of SSIM scores for every data point.
- **`full`** (*`bool`*) – Whether to return the full SSIM map as well. Default: False.

Returns

- **score** (*torch.Tensor*) – The SSIM score tensor where content axes are reduced.
- **full_tensor** (*torch.Tensor*) – The full output SSIM with the same shape as input images. This argument is only returned when `full=True`.

`training: bool`

1.1.5 torchimage.padding package

Submodules

torchimage.padding.pad_1d module

Pad a torch tensor along a certain dimension.

All 1-d padding utility functions share the same set of arguments.

Important note: This is an in-place function.

To avoid re-copying the input tensor, these 1d padding utility functions only accept an empty tensor that has the same shape as the final padded output and copies the values of the input tensor at its center.

The function will return its input x after modifying it.

Efficiency Warning: Currently, PyTorch doesn't support returning negative-strided views like [::-1]. torch.flip() is said to be computationally expensive due to copying the data, which might put `symmetric` and `reflect` to a disadvantage.

They are for lower-level utility only. Do *NOT* expect them to behave the same way as `F.pad` does.

param x The input tensor to be padded.

See the important note above. Let `u` be the original tensor, then `x` is an empty tensor holding `u` values at center such that `x[idx] == u`

type x torch.Tensor

param idx Indices for the ground truth tensor located at the center of the empty-padded tensor x.

Has the same length as the number of dimensions `len(idx) == x.ndim`. Each element is a `slice(beg, end, 1)` where at dimension `dim`, `x.shape[dim] - end` is the amount of padding in the end and `beg` is the amount of padding in the beginning.

Note that this has to be a tuple to properly index a high-dimensional tensor.

This tuple of index slices prevents computing padding for empty values at this dimension.

type idx tuple of slice

param dim The dimension to pad.

type dim int

param These keyword arguments are used in some padding functions only.

param negate Whether to flip signs (+, -) when flipping the signal. Default: False.

This parameter only applies to `symmetric` mode. When it is enabled, turns into half-sample anti-symmetric mode:

`antisymmetric`: `-d -c -b -a | a b c d | -d -c -b -a`

type negate bool

param before For `linear_ramp_1d`, they are the new edge values for the padded tensor. Default: 0

For `constant_1d`, they are the constants used for padding before and after ground truth.

For `stat_1d`, they are the lengths at the border to compute statistics with.

type before float

param after For `linear_ramp_1d`, they are the new edge values for the padded tensor. Default: 0

For `constant_1d`, they are the constants used for padding before and after ground truth.

For `stat_1d`, they are the lengths at the border to compute statistics with.

type after float

returns **x** – The same tensor after the padded values at `dim` are filled in.

rtype torch.Tensor

`torchimage.padding.pad_1d.`**`circular_1d`**(*x*, *idx*, *dim*)

`torchimage.padding.pad_1d.`**`constant_1d`**(*x*, *idx*, *dim*, *before*, *after*)

`torchimage.padding.pad_1d.`**`linear_ramp_1d`**(*x*, *idx*, *dim*, *before*, *after*)

`torchimage.padding.pad_1d.`**`odd_reflect_1d`**(*x*, *idx*, *dim*)

`torchimage.padding.pad_1d.`**`odd_symmetric_1d`**(*x*, *idx*, *dim*)

`torchimage.padding.pad_1d.`**`periodize_1d`**(*x*, *idx*, *dim*)

`torchimage.padding.pad_1d.`**`reflect_1d`**(*x*, *idx*, *dim*)

`torchimage.padding.pad_1d.`**`replicate_1d`**(*x*, *idx*, *dim*)

`torchimage.padding.pad_1d.`**`smooth_1d`**(*x*, *idx*, *dim*)

`torchimage.padding.pad_1d.`**`stat_1d`**(*x*, *idx*, *dim*, *before*, *after*, *mode*)

`torchimage.padding.pad_1d.`**`symmetric_1d`**(*x*, *idx*, *dim*, *negate=False*)

`torchimage.padding.pad_1d.`**`zeros_1d`**(*x*, *idx*, *dim*)

torchimage.padding.tensor_pad module

torchimage.padding.utils module

Private utility functions for padding

`torchimage.padding.utils.`**`make_idx`**`(`**args*, *dim*, *ndim*`)`

Make an index that slices exactly along a specified dimension. e.g. [:, … :, slice(*args), :, …, :]

This helper function is similar to numpy's `_slice_at_axis`.

Parameters

- **`*args`** (*`int or None`*) – constructor arguments for the slice object at target axis
- **`dim`** (*`int`*) – target axis; can be negative or positive
- **`ndim`** (*`int`*) – total number of axes

Returns **idx** – Can be used to index np.ndarray and torch.Tensor

Return type tuple of slice

`torchimage.padding.utils.`**`modify_idx`**`(`**args*, *idx*, *dim*`)`

Make an index that slices a specified dimension while keeping the slices for other dimensions the same.

Parameters

- **`*args`** (*`tuple of int or None`*) – constructor arguments for the slice object at target axis
- **`idx`** (*`tuple of slice`*) – tuple of slices in the original region of interest
- **`dim`** (*`int`*) – target axis

Returns

new_idx – New tuple of slices with dimension dim substituted by slice(*args)

Can be used to index np.ndarray and torch.Tensor

Return type tuple of slice

`torchimage.padding.utils.`**`pad_width_format`**`(`*padding*, *source='numpy'*, *target='torch'*, *ndim=None*`)`

Convert between 2 padding width formats.

This function converts `pad` from `source` format to `target` format.

Padding width refers to the number of padded elements before and after the original tensor at a certain axis. Numpy and PyTorch have different formats to specify the padding widths. Because Numpy works with n-dimensional arrays while PyTorch more frequently works with (N, C, [D,]H, W) data tensors. In the latter case, starting from the last dimension seems more intuitive.

Numpy padding width format is `((before_0, after_0), (before_1, after_1), ..., (before_{n-1}, after_{n-1}))`.

PyTorch padding format is `(before_{n-1}, after_{n-1}, before_{n-2}, after_{n-2}, ..., before_{dim}, after_{dim})`, such that before after `dim` is not padded.

Parameters

- **`padding`** (*`tuple of int, or tuple of tuple of int`*) – the input padding width format to convert
- **`source`** (*`str`*) – Format specification for padding width. Either "numpy" or "torch".
- **`target`** (*`str`*) – Format specification for padding width. Either "numpy" or "torch".

- `ndim` (*int*) – Number of dimensions in the tensor of interest.

  Only used when converting from torch to numpy format.

Returns **padding** – the new padding width specification

Return type tuple of int, or tuple of tuple of int

`torchimage.padding.utils.same_padding_width`(*kernel_size*, *stride=1*, *in_size=None*)

Calculate the padding width before and after a certain axis using "same padding" method.

When stride is 1, input size at that axis doesn't matter and the output tensor will have the same shape as the input tensor, hence the name "same padding".

When stride is greater than 1, same padding can be intuitively described as "letting the kernel cover every element of the original tensor, while making padding width before and after the axis roughly the same." (unlike valid padding, which doesn't pad at all and the last pixels will be ignored if input tensor's side length doesn't match kernel size and stride)

This convention is taken from a TensorFlow documentation page which no longer exists.

Parameters

- `kernel_size` (*int*) – The convolution kernel size at that axis
- `stride` (*int*) – The convolution stride at that axis. Default: 1.
- `in_size` (*int*) – The side length of the input tensor at that axis.

  Can be None if stride is 1.

Returns **pad_before, pad_after** – The number of padded elements required by same padding before and after the axis.

Return type int

Module contents

General padding functionalities.

Padding extends the border of the original signal so the output has a certain shape.

We design this padding package with these principles:

1. **Maximal compatibility with PyTorch.** We will implicitly use `F.pad` as much as we can. (For example, we do not implement zero and constant padding) We also adjust the names of the arguments to be compatible with PyTorch
2. **Versatility** We try to incorporate as many functionalities available in other packages as possible. Specifically, we try to reproduce the behavior of `numpy.pad`, MatLab dwtmode, and PyWavelet signal extension modes.

Comparing with `torch.nn.functional.pad`, we make the following modifcations:

1. Symmetric padding is added
2. **Higher-dimension non-constant padding** To the date of this release, PyTorch has not implemented reflect (ndim >= 3+2), replicate (ndim >= 4+2), and circular (ndim >= 4+2) for high-dimensional tensors (the +2 refers to the initial batch and channel dimensions).

   We achieve n-dimensional padding by sequentially applying 1-dimensional padding from the first axis (dim 0) to the last (dim n-1). In some cases (such as constant padding with different values before and after an axis) where different orders of applying 1d padding can change the final result, we follow numpy's convention going from the first to the last dimension.

3. **Wider padding size** Padding modes reflect and circular will cause PyTorch to fail when padding size is greater than the tensor's shape at a certain dimension. i.e. Padding value causes wrapping around more than once.

Comparing with `numpy.pad`, we make the following modifcations:

1. **Bug fixes for wider padding** For modes such as symmetric and circular, numpy padding doesn't make proper repetition. For example, notice how numpy fails to repeat or flip the signal [0, 1] in on the right. >>> import numpy as np >>> np.pad([0, 1],(1, 10),mode="wrap") array([1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1]) >>> np.pad([0, 1],(1, 10),mode="symmetric") array([0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1])

   The same bug has been observed for antisymmetric as well. After reading the source code, I believe numpy's error comes from extending the padding for directions (before and after), which breaks the cycle of the original signal if before and after are relatively prime.

Padding Modes

`"empty"` - pads with undefined values `? ? ? ? | a b c d | ? ? ? ?` where the empty values are from `torch.empty`

`"constant"` - pads with a constant value `p p p p | a b c d | p p p p` where `p` is supplied by `constant_values` argument.

`p1 p1 p1 p1 | a b c d | p2 p2 p2 p2` where `p1` and `p2` are different constant values.

`"zeros"` - pads with 0; a special case of constant padding `0 0 0 0 | a b c d | 0 0 0 0`

`"symmetric"` - extends signal by *mirroring* samples. Also known as half-sample symmetric `d c b a | a b c d | d c b a`

`"antisymmetric"` - extends signal by *mirroring* and *negating* samples. Also known as half-sample antisymmetric. `-d -c -b -a | a b c d | -d -c -b -a`

`"reflect"` - signal is extended by *reflecting* samples. This mode is also known as whole-sample symmetric `d c b | a b c d | c b a`

`"replicate"` - replicates the border pixels `a a a a | a b c d | d d d d`

`"circular"` - signal is treated as a periodic one `a b c d | a b c d | a b c d`

`"periodize"` - same as circular, except the last element is replicated when signal length is odd. `a b c -> a b c c | a b c c | a b c c` Note that it first extends the signal to an even length prior to using periodic boundary conditions

`"odd_reflect"` - extend the signal by a point-reflection across the edge element `2a-d 2a-c 2a-b | a b c d | 2d-c 2d-b 2d-a | 2(2d-a)-(2d-b) 2(2d-a)-(2d-c) 2(2d-a)-d` Also known has whole-sample antisymmetric.

`"odd_symmetric"` - extend the signal by a point-reflection across a hypothetical midpoint between the edge and the symmetrically reflected edge. `2a-d 2a-c 2a-b a | a b c d | d 2d-c 2d-b 2d-a | 2d-a 2(2d-a)-(2d-b) 2(2d-a)-(2d-c) 2(2d-a)-d`

`"smooth"` - extend the signal according to the first derivatives calculated on the edges (straight line extrapolation) `a-4(b-a) a-3(b-a) a-2(b-a) a-(b-a) | a b c d | d+(d-c) d+2(d-c) d+3(d-c) d+4(d-c)`

If there's only 1 element, smooth should behave the same as replicate because we can only assume a first derivative of 0.

`"linear_ramp"` - pads with the linear ramp between end value and the array border value. `| a b c d | d+s d+2s ... d+n*s e` where `e` is the end value, `n` is the number of elements between `d` and `e`, and `s = (e-d)/(n+1)`

The end values are specified by argument `end_values`

`"maximum"`, `"mean"`, `"median"`, `"minimum"` - pads with a statistical value of all or part of the vector along each axis.

The length of the vector used for computing the statistical value is specified by argument `stat_length`.

Note that PyTorch's median behaves differently from numpy's median. When there is an even number of elements, `torch.median` returns the left element at the center, whereas `numpy.median` returns the arithmetic mean between the two elements at the center. We retain PyTorch's behavior here.

`<function>` - extend the signal with a customized function The function should have signature `pad_func(x, idx, dim)`, like other functions in pad_1d.py. You may find reading the source code from pad_1d.py and using the `_modify_index` function useful.

To pass customized keyword arguments (like end_values, constant_values, and stat_length) to a self-defined padding function, especially when they depend on `dim`, the user can define a separate function `f: dim -> kwargs` and call `f` inside `pad_func` to get the keyword arguments from dimension. I use the same method for constant (when there are multiple padding values), linear ramp and stat padding.

This is not the most elegant design but it works when it needs to, so please let me know if any improvement is urgently needed and I can fix it.

torchimage	PyWavelets	Matlab	numpy.pad	Scipy
zeros	zero	zpd	constant, cval=0	N/A
constant	N/A	N/A	constant	constant
replicate	constant	sp0	edge	nearest
smooth	smooth	spd, sp1	N/A	N/A
circular	periodic	ppd	wrap	wrap
periodize	periodization	per	N/A	N/A
symmetric	symmetric	sym, symh	symmetric	reflect
reflect	reflect	symw	reflect	mirror
antisymmetric	antisymmetric	asym, asymh	N/A	N/A
odd_reflect	antireflect	asymw	reflect, reflect_type='odd'	N/A
odd_symmetric	N/A	N/A	symmetric, reflect_type='odd'	N/A
linear_ramp	N/A	N/A	linear_ramp	N/A
maximum	N/A	N/A	maximum	N/A
mean	N/A	N/A	mean	N/A
median	N/A	N/A	median	N/A
minimum	N/A	N/A	minimum	N/A
empty	N/A	N/A	empty	N/A
<function>	N/A	N/A	<function>	N/A

class `torchimage.padding.`**`Padder`**(*pad_width=0*, *mode='constant'*, *constant_values=0*, *end_values=0.0*, *stat_length=None*)

Bases: `object`

`forward`(*x: torch.Tensor*, *axes=slice(2, None, None)*)

Pads a tensor sequentially at all specified dimensions

Parameters

- **x** (*torch.Tensor*) – The input n-dimensional tensor to be padded.
- **axes** (*sequence of int, slice, None*) – The sequence of dimensions to be padded with the exact ordering.

If axes is not provided (None), the padder will automatically right-justify the NdSpecs such that the “rightmost” axis corresponds to the “rightmost” item in an NdSpec and other entries are aligned accordingly.

If the padder has larger ndim than x (or axes), the leftmost dimensions in the padder are ignored. If x (or axes) has larger ndim than the padder, the leftmost dimensions of x are left unchanged.

If the input format is PyTorch batch-like (first 2 dimensions are batch and channel dimensions), we recommend using `axes=slice(2, None)`.

Returns **y** – Padded tensor.

Return type torch.Tensor

`pad_axis`(*x: torch.Tensor*, *i: int*, *axis: int*)

Pad a specific axis according to predefined padder parameters

This method will create a new tensor that is larger at the axis of interest (unless padding width is 0, in which case the original tensor will be returned).

It is only useful together with separable filtering and in very specific circumstances. The padder pads each axis right before the separable filtering (unfold -> matrix-vector multiplication) at that axis. This algorithm leads to the creation of many more intermediate tensors, which not only takes time copying but also consumes memory if the input tensor requires gradient.

Therefore, it is only useful when the dimension is extremely high and padding width is much larger than the size of the input tensor.

Parameters

- **x** (*torch.Tensor*) – Input tensor to be padded.
- **i** (*int*) – The index of self (padder) to be used
- **axis** (*int*) – The axis in x to be padded. Can be a positive or negative index.

Returns **y** – Padded tensor.

Return type torch.Tensor

`to_same`(*kernel_size*, *stride=1*, *in_shape=None*)

Modify this padder in-place, such that the padding width follows the convention of same padding.

Parameters

- **kernel_size** (*int or sequence of int*) – Kernel size of convolution operation
- **stride** (*int or sequence of int*) – Stride of convolution at each axis
- **in_shape** (*int or sequence of int*) – Size of input tensor at each axis of interest

Returns **new_padder** – self with pad_width re-initialized

Return type *Padder*

1.1.6 torchimage.pooling package

Submodules

torchimage.pooling.gaussian module

`class torchimage.pooling.gaussian.`**`GaussianPoolNd`**(*kernel_size*, *sigma*, *order=0*, *stride=None*, ***, *same_padder=None*)

Bases: *`torchimage.pooling.base.SeparablePoolNd`*

N-dimensional Gaussian Pooling

This module is implemented using separable pooling. Recall that Gaussian kernel is separable, i.e. An n-dimensional Gaussian kernel is the outer product of n 1D Gaussian kernels. N-dimensional Gaussian convolution is equivalent to sequentially applying n 1D Gaussian convolutions sequentially to each axis of interest. We can thus reduce the computational complexity from O(Nk^d) to O(Nkd), where N is the number of elements in the input tensor, k is kernel size, and d is the number of dimensions to convolve.

`kernel_size: torchimage.utils.ndspec.NdSpec`

`stride: torchimage.utils.ndspec.NdSpec`

`torchimage.pooling.gaussian.`**`gaussian_kernel_1d`**(*kernel_size*, *sigma*, *order*)

generate a 1-dimensional Gaussian kernel given kernel_size and sigma.

Multi-dimensional gaussian can be created by repeatedly obtaining outer products from 1-d Gaussian kernels. But when the application is Gaussian filtering (pooling) implemented through convolution, separable convolution is much more efficient.

This function uses the utility function from scipy.ndimage to generate gaussian kernels

Parameters

- **`kernel_size`** (*`int`*) – length of the 1-d Gaussian kernel

  Please be aware that while even-length gaussian kernels can be generated, they are not well-defined and will cause a shift.

- **`sigma`** (*`float`*) – standard deviation of Gaussian kernel
- **`order`** (*`int`*) – An order of 0 corresponds to convolution with a Gaussian kernel. A positive order corresponds to convolution with that derivative of a Gaussian.

Returns 1-d Gaussian kernel of length kernel_size with standard deviation sigma

Return type np.ndarray

torchimage.pooling.generic module

Module contents

We use pooling to refer to convolution-like operations that don't have learnable parameters, such as average pooling and gaussian pooling.

In torchimage, what makes pooling different from filtering is that a filter layer usually consists of its pooling layer counterpart with stride 1, after a "same" padding layer that ensures the output and the input will have the same shape. We separate filtering and pooling so that users can have greater freedom and the code becomes easier to maintain.

`class torchimage.pooling.`**`AvgPoolNd`**(*kernel_size*, *stride=None*, ***, *count_include_pad=True*, *same_padder=None*)

Bases: *`torchimage.pooling.base.SeparablePoolNd`*

`forward`(*x: torch.Tensor*, *axes=slice(2, None, None)*)
Perform separable pooling on a tensor.

Parameters

- `x` (*`torch.Tensor`*) – Input tensor.
- `axes` (*`None, int, slice, tuple of int`*) – An ordered list of axes to be processed. Default: slice(2, None)

  Axes can be repeated. If `None`, it will be all the axes from axis 0 to the last axis.

  The default `slice(2, None)` assumes that the first 2 axes are batch (N) and channel (C) dimensions.

Returns **x** – Output tensor after pooling

Return type torch.Tensor

`kernel_size: torchimage.utils.ndspec.NdSpec`

`stride: torchimage.utils.ndspec.NdSpec`

`class torchimage.pooling.BasePoolNd`(***, *same_padder=None*)
Bases: `torch.nn.modules.module.Module`

`abstract forward`(*x: torch.Tensor*, *axes*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`kernel_size: torchimage.utils.ndspec.NdSpec`

`pad`(*x: torch.Tensor*, *axes*)
Use the bound same padder to pad the input tensor before performing actual convolution

Parameters

- `x` (*`torch.Tensor`*) – Input tensor
- `axes` (*`int, slice, tuple of int`*) – Axes to convolve (processed to be nonnegative integers)

Returns **x** – padded tensor

Return type torch.Tensor

`read_stride`(*stride*)
Read an input stride after `self.kernel_size` has been initialized.

If stride is None at any axis, it will be the same as kernel size.

This is a constant function (doesn't modify self).

Parameters `stride` (*`None, int, list, NdSpec`*) – Input stride

Returns **stride** – Processed stride

Return type NdSpec

`stride: torchimage.utils.ndspec.NdSpec`

to_filter(*padder=None*)

Modify this pooling module in-place, so that the stride is 1 and a same or valid padder is supplied.

In torchimage, filtering is a special subset of pooling that has `stride=1` and (usually) same padding. (Considering that torch has not implemented a general method to perform dilated unfold on a tensor, dilation=1 is the default.)

Parameters **padder** (*None, str or* Padder) – A same padder for this pooling module in the forward stage. A not-None padder will override self.same_padder. So if self.same_padder and padder are both None, valid padding will be used.

Returns **self** – A modified self

Return type *SeparablePoolNd*

class torchimage.pooling.**GaussianPoolNd**(*kernel_size*, *sigma*, *order=0*, *stride=None*, ***, *same_padder=None*)

Bases: `torchimage.pooling.base.SeparablePoolNd`

N-dimensional Gaussian Pooling

This module is implemented using separable pooling. Recall that Gaussian kernel is separable, i.e. An n-dimensional Gaussian kernel is the outer product of n 1D Gaussian kernels. N-dimensional Gaussian convolution is equivalent to sequentially applying n 1D Gaussian convolutions sequentially to each axis of interest. We can thus reduce the computational complexity from O(Nk^d) to O(Nkd), where N is the number of elements in the input tensor, k is kernel size, and d is the number of dimensions to convolve.

kernel_size: torchimage.utils.ndspec.NdSpec

stride: torchimage.utils.ndspec.NdSpec

training: bool

class torchimage.pooling.**SeparablePoolNd**(*kernel=()*, *stride=None*, ***, *same_padder=None*)

Bases: `torchimage.pooling.base.BasePoolNd`

forward(*x: torch.Tensor*, *axes=slice(2, None, None)*)

Perform separable pooling on a tensor.

Parameters

- **x** (*torch.Tensor*) – Input tensor.
- **axes** (*None, int, slice, tuple of int*) – An ordered list of axes to be processed. Default: slice(2, None)

  Axes can be repeated. If `None`, it will be all the axes from axis 0 to the last axis.

  The default `slice(2, None)` assumes that the first 2 axes are batch (N) and channel (C) dimensions.

Returns **x** – Output tensor after pooling

Return type torch.Tensor

kernel_size: torchimage.utils.ndspec.NdSpec

stride: torchimage.utils.ndspec.NdSpec

training: bool

1.2 Submodules

1.3 torchimage.misc module

Miscellaneous utilities for PyTorch

Many of these functions have not been implemented in PyTorch

`torchimage.misc.`**`describe`**(*x: torch.Tensor*)

Describe the distribution of numbers in the input tensor.

This function mimics the behavior of DataFrame and Series describe method in pandas.

Parameters **x** (*torch.Tensor*) – Input tensor to be described

Returns **desc** – A dictionary from keywords (such as mean, std) to float values

Return type dict

`torchimage.misc.`**`outer`**(*u*, *v*)

Compute the outer product of two tensors.

For two tensors **u** of shape $(k_1, k_2, ..., k_m)$ and **v** of shape $(l_1, l_2, ..., l_n)$, their outer product $\mathbf{w} = \mathbf{u} \otimes \mathbf{v}$ is defined such that $\mathbf{w}[i_1, i_2, ..., i_m, j_1, j_2, ..., j_n] = \mathbf{u}[i_1, i_2, ..., i_m]\mathbf{v}[j_1, j_2, ..., j_n]$. **w** will have $m + n$ dimensions and the shape of $(k_1, k_2, ..., k_m, l_1, l_2, ..., l_n)$

Parameters

- **u** (*torch.Tensor or np.ndarray*) – Input tensor with arbitrary shape.
- **v** (*torch.Tensor or np.ndarray*) – Input tensor with arbitrary shape.

Returns

w – Outer product of u and v.

`w.shape` is the same as `u.shape + v.shape`.

Return type torch.Tensor

`torchimage.misc.`**`poly1d`**(*x: torch.Tensor*, *p: list*)

Calculate single-variable (one-dimensional) polynomial `y = a_n * x^n + ... + a_2 * x^2 + a_1 * x + a_0`

The extra memory required by this function is at most 2 times the size of x, one to store the output tensor and the other to keep a running ith power of x.

Parameters

- **x** (*torch.Tensor*) – Input data tensor to be substituted for the variable x.
- **p** (*sequence of float*) – List of weights in the order of descending exponents, namely [a_n, a_{n-1}, …, a_2, a_1, a_0].

Returns **y** – Output tensor with the same shape as x

Return type torch.Tensor

`torchimage.misc.`**`safe_power`**(*x*, *exponent*, ***, *epsilon=1e-06*)

Takes the power of each element in input with exponent and returns a tensor with the result.

This is a safer version of `torch.pow(out = x ** exponent)`, which avoids:

1. **NaN/imaginary output when `x < 0` and exponent has a fractional part** In this case, the function returns the signed (negative) magnitude of the complex number.

2. **NaN/infinite gradient at `x = 0` when exponent has a fractional part** In this case, the positions of 0 are added by `epsilon`, so the gradient is back-propagated as if `x = epsilon`.

However, this function doesn't deal with float overflow, such as 1e10000.

Parameters

- **`x`** (*`torch.Tensor or float`*) – The input base value.
- **`exponent`** (*`torch.Tensor or float`*) – The exponent value.

  (At least one of `x` and `exponent` must be a torch.Tensor)
- **`epsilon`** (*`float`*) – A small floating point value to avoid infinite gradient. Default: 1e-6

Returns **out** – The output tensor.

Return type torch.Tensor

1.4 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX