indexedconv Documentation

Release 1.1

Mikael Jacquemont, Thomas Vuillaume, Luca Antiga

Nov 04, 2022

Notes

CHAPTER 1

Indexed Convolution

The indexed operations allow the user to perform convolution and pooling on non-Euclidian grids of data given that the neighbors pixels of each pixel is known and provided.

It gives an alternative to masking or resampling the data in order to apply standard Euclidian convolution. This solution has been developed in order to apply convolutional neural networks to data from physics experiments that propose specific pixels arrangements.

It is used in the GammaLearn project for the Cherenkov Telescope Array.

Here you will find the code for the indexed operations as well as applied examples. The current implementation has been done for pytorch.

Documentation may be found online.

CHAPTER 2

Install

Install from IndexedConv folder:

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

Install with pip:

```
pip install indexedconv
```

Install with conda:

conda install -c gammalearn indexedconv

2.1 Requirements

```
"torch>=1.4",
"torchvision",
"numpy",
"matplotlib",
"h5py"
```

CHAPTER 3

Running an experiment

For example, to train the network with indexed convolution on the CIFAR10 dataset transformed to hexagonal:

```
python examples/cifar_indexed.py main_folder data_folder experiment_name --hexa --
↪batch 125 --epochs 300 --seeds 1 2 3 4 --device cpu
```

In order to train on the AID dataset, it must be downloaded and can be found here.

CHAPTER 4

Authors

The development of the indexed convolution is born from a collaboration between physicists and computer scientists.

- Luca Antiga, Orobix
- Mikael Jacquemont, LAPP (CNRS), LISTIC (USMB)
- Thomas Vuillaume, LAPP (CNRS)

CHAPTER 5

References

Please cite:

5.1 Publication

https://www.scitepress.org/Link.aspx?doi=10.5220/0007364303620371

Jacquemont, M.; Antiga, L.; Vuillaume, T.; Silvestri, G.; Benoit, A.; Lambert, P. and Maurin, G. (2019). **Indexed Operations for Non-rectangular Lattices Applied to Convolutional Neural Networks**. In Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 5: VISAPP, ISBN 978-989-758-354-4, pages 362-371. DOI: 10.5220/0007364303620371

If you want to use and refer to the code implementation of IndexedConv, please cite:

CHAPTER 6

Contributing

All contributions are welcome.

Start by contacting the authors, either directly by email or by creating a GitHub issue.

CHAPTER 7

License

MIT License

Copyright (c) 2018 IndexedConv

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 8

Examples

The indexed convolution has been tested and validated on standard dataset whose images have been modified into hexagonal lattices.

CHAPTER 9

engine

9.1 Indexed Convolution

9.1.1 IndexedConv

class indexedconv.engine.**IndexedConv**(*in_channels*, *out_channels*, *indices*, *bias=True*)

Applies a convolution over an input tensor where neighborhood relationships between elements are explicitly provided via an *indices* tensor.

The output value of the layer with input size (N, C_{in}, L) and output (N, C_{out}, L) can be described as:

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{c=0}^{C_{in}-1} \sum_{i=0}^{L-1} \sum_{k=0}^{K} weight(C_{out_j}, c, k) * input(N_i, c, indices(i, k))$$

where

indices is a L x K tensor, where *K* is the size of the convolution kernel,
providing the indices of the *K* neighbors of input element *i*.
A -1 entry means zero-padding.

Parameters

- **in_channels** (*int*) – Number of channels in the input tensor
- **out_channels** (*int*) – Number of channels produced by the convolution
- **indices** (*LongTensor*) – index tensor of shape (L x kernel_size), having on each
- **the indices of neighbors of each element of the input a -1 indicates the absence of a** (*row*) –
- **which is handled as zero-padding** (*neighbor,*) –
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`

Shape:

- Input: (N, C_{in}, L)
- Output: (N, C_{out}, L)

Variables

- **weight** (*Tensor*) – the learnable weights of the module of shape (out_channels, in_channels, kernel_size)
- **bias** (*Tensor*) – the learnable bias of the module of shape (out_channels)

Examples:

```
>>> indices = (10 * torch.rand(50, 3)).type(torch.LongTensor)
>>> m = nn.IndexedConv(16, 3, indices)
>>> input = torch.randn(20, 16, 50)
>>> output = m(input)
```

9.2 Indexed Pooling

9.2.1 IndexedMaxPool2d

class indexedconv.engine.**IndexedMaxPool2d**(*indices*)

Compute the Max Pooling 2d operation on a batch of features of vector images wrt a matrix of indices

Parameters **indices** (*LongTensor*) – index tensor of shape (L x kernel_size), having on each row the indices of neighbors of each element of the input a -1 indicates the absence of a neighbor, which is handled as zero-padding

Shape:

- Input: (N, C, L_{in})
- Output: (N, C, L_{out})

9.2.2 IndexedAveragePool2d

class indexedconv.engine.**IndexedAveragePool2d**(*indices*)

Compute the Average Pooling 2d operation on a batch of features of vector images wrt a matrix of indices

Parameters **indices** (*LongTensor*) – index tensor of shape (L x kernel_size), having on each row the indices of neighbors of each element of the input a -1 indicates the absence of a neighbor, which is handled as zero-padding

Shape:

- Input: (N, C, L_{in})
- Output: (N, C, L_{out})

CHAPTER 10

utils

10.1 Datasets

10.1.1 HDF5Dataset

class indexedconv.utils.**HDF5Dataset**(*path*, *transform=None*, *target_transform=None*)

Loads data in a Dataset from a HDF5 file.

Parameters

- **path** (*str*) – The path to the HDF5 file.
- **transform** (*callable, optional*) – A callable or a composition of callable to be applied to the data.
- **target_transform** (*callable, optional*) – A callable or a composition of callable to be applied to the labels.

10.1.2 NumpyDataset

class indexedconv.utils.**NumpyDataset**(*data*, *labels*, *transform=None*, *target_transform=None*)

Loads data in a Dataset from `numpy.array`.

Parameters

- **data** (*numpy.array*) – The data to load.
- **labels** (*numpy.array*) – The labels of the data.
- **transform** (*callable, optional*) – A callable or a composition of callable to be applied to the data.
- **target_transform** (*callable, optional*) – A callable or a composition of callable to be applied to the labels.

10.2 Transforms

10.2.1 NumpyToTensor

class indexedconv.utils.**NumpyToTensor**
Converts a numpy array to a tensor.

10.2.2 SquareToHexa

class indexedconv.utils.**SquareToHexa**
Converts an image with a square grid to an image with a hexagonal one.

10.3 Image processing

class indexedconv.utils.**PCA**(*D*, *n_components*)
Credit E. Hoogeboom http://github.com/ehoogeboom/hexaconv

fit(*D*, *n_components*)
The computation works as follows:

- The covariance is `C = 1/(n-1) * D * D.T`
- The eigendecomp of C is: `C = V Sigma V.T`
- Let `Y = 1/sqrt(n-1) * D`
- Let `U S V = svd(Y)`,
- Then the columns of U are the eigenvectors of `Y * Y.T = C`
- And the singular values S are the sqrts of the eigenvalues of C
- We can apply PCA by multiplying by U.T

transform(*D*, *whiten=False*, *ZCA=False*, *regularizer=1e-05*)
We want to whiten, which can be done by multiplying by $\sigma^{-1/2}U.T$

Any orthogonal transformation of this is also white, and when `ZCA=True` we choose $U\sigma^{-1/2}U.T$

indexedconv.utils.**normalize**(*images*)
Normalizes images.

Parameters **images** – image tensor of shape (c, n, m)

indexedconv.utils.**build_hexagonal_position**(*index_matrix*)
Computes the position of the pixels in the hexagonal grid from the index matrix.

Parameters **index_matrix** (*tensor*) – The index matrix representing the index of each pixel in the axial addressing system.

indexedconv.utils.**square_to_hexagonal**(*image*)
Rough sampling of square images to hexagonal grid

Parameters **image** – image tensor of shape (c, n, m)

indexedconv.utils.**square_to_hexagonal_index_matrix**(*image*)
Creates the index matrix of square images in a hexagonal grid (axial addressing system).

Parameters **image** – input tensor of shape (c, n, m)

10.4 Indexed functions

indexedconv.utils.**build_kernel**(*kernel_type*, *radius=1*, *dilation=1*)

Builds the convolution kernel or mask. (Following the suggestion of Miguel Lallena).

Parameters

- **kernel_type** (*str*) – The type of kernel. Can be hexagonal ('Hex') or square ('Square').
- **radius** (*int*) – The radius of the kernel.
- **dilation** (*int*) – The dilation. A dilation of 1 means no dilation.

Returns A np.array.

indexedconv.utils.**neighbours_extraction**(*index_matrix*, *kernel_type='Hex'*, *radius=1*, *stride=1*, *dilation=1*, *retina=False*)

Builds the matrix of indices from an index matrix based on a kernel.

The matrix of indices contains for each pixel of interest its neighbours, including itself.

Parameters

- **index_matrix** (*torch.Tensor*) – Matrix of index for the images, shape(1, 1, matrix.size).
- **kernel_type** (*str*) – The kernel shape, Hex for hexagonal Square for a square and Pool for a square of size 2.
- **radius** (*int*) – The radius of the kernel.
- **stride** (*int*) – The stride.
- **dilation** (*int*) – The dilation. A dilation of 1 means no dilation.
- **retina** (*bool*) – Whether to build a retina like kernel. If True, dilation must be 1.

Returns A torch.Tensor - the matrix of the neighbours.

Example

```
>>> index_matrix = [[0, 1, -1], [2, 3, 4], [-1, 5, 6]]
[[0, 1, -1],
[2,  3, 4],
[-1, 5, 6]]
>>> kernel_type = 'Hex'
>>> radius = 1
>>> kernel
[[1, 1, 0],
[ 1, 1, 1],
[ 0, 1, 1]]
>>> stride = 1
>>> neighbours = neighbours_extraction(index_matrix, kernel_type, radius, stride)
[[-1, -1, -1,  0,  1,  2,  3],
[ -1, -1,  0,  1, -1,  3,  4],
[ -1,  0, -1,  2,  3, -1,  5],
[  0,  1,  2,  3,  4,  5,  6],
[  1, -1,  3,  4, -1,  6, -1],
[  2,  3, -1,  5,  6, -1, -1],
[  3,  4,  5,  6, -1, -1, -1]]
```

indexedconv.utils.**create_index_matrix**(*nbRow*, *nbCol*, *injTable*)
Creates the matrix of index of the pixels of the images of any shape stored as vectors.

Parameters

- **nbRow** (*int*) – The number of rows of the index matrix.
- **nbCol** (*int*) – The number of cols of the index matrix.
- **injTable** (*numpy.array*) – The injunction table, i.e. the list of the position of every pixels of the vector image in a vectorized square image.

Returns A torch.Tensor containing the index of each pixel represented in a matrix.

Example

```
>>> image = [0, 1, 2, 3, 4, 5, 6]  # hexagonal image stored as a vector
>>> # in the hexagonal space                  0 1
>>> #                                        2 3 4
>>> #                                         5 6
>>> # injunction table of the pixel position of a hexagonal image represented in␣
↪the axial addressing system
>>> injTable = [0, 1, 3, 4, 5, 7, 8]
>>> index_matrix = [[0, 1, -1], [2, 3, 4], [-1, 5, 6]]
[[0, 1, -1],
[2,  3, 4],
[-1, 5, 6]]
```

indexedconv.utils.**pool_index_matrix**(*index_matrix*, *kernel_type='Pool'*, *stride=2*)
Pools an index matrix.

Parameters

- **index_matrix** (*torch.Tensor*) – The index matrix for the images, shape(1, 1, matrix.size).
- **kernel_type** (*str*) – The kernel shape, Hex for hexagonal, Square for a square of size 3 and Pool for a square of size 2.
- **stride** (*int*) – The stride.

Returns A torch.Tensor containing the pooled matrix.

indexedconv.utils.**prepare_mask**(*indices*)
Prepares the indices and the mask for the GEMM im2col operation.

Parameters **indices** (*torch.Tensor*) – The matrix of indices containing the neighbours of each pixel of interest.

indexedconv.utils.**img2mat**(*input_images*, *index_matrix*)
Transforms a batch of features of vector images in a batch of features of matrix images.

Parameters

- **input_images** (*torch.Tensor*) – The images with shape (batch, features, image).
- **index_matrix** (*torch.Tensor*) – The index matrix containing the index of the pixels of the images. represented in a matrix

Returns A torch.Tensor

indexedconv.utils.**mat2img**(*input_matrix*, *index_matrix*)
Transforms a batch of features of matrix images in a batch of features of vector images.

Parameters

- **input_matrix** (*torch.Tensor*) – The images with shape (batch, features, matrix.size).
- **index_matrix** (*torch.Tensor*) – The index matrix for the images, shape(1, 1, matrix.size).

10.5 Utilities

indexedconv.utils.**get_gpu_usage_map**(*device_id*)

Get the current gpu usage. Inspired from gpu usage from pytorch

Parameters **device_id** (*int*) – the GPU id as GPU/Unit's 0-based index in the natural enumeration returned by the driver

Returns

dict - usage Keys are device ids as integers. Values are memory usage as integers in MB, total memory usage as integers in MB, gpu utilization in %.

indexedconv.utils.**compute_total_parameter_number**(*net*)

Computes the total number of parameters of a network.

Parameters **net** (nn.Module) – The network.

Returns A int

CHAPTER 11

networks

11.1 Networks for AID

11.1.1 WideNet

class indexedconv.nets.aid.**WideNet**(*n_out*)
ResNet like Network from HexaConv paper (Z^2).

Parameters **n_out** (*int*) – Number of features after last convolution.

11.1.2 WideNetIndexConvIndexPool

class indexedconv.nets.aid.**WideNetIndexConvIndexPool**(*index_matrix*, *camera_layout*, *n_out*)
ResNet like Network from HexaConv paper implemented with indexed convolutions and pooling.

Parameters

- **index_matrix** (*torch.Tensor*) – The index matrix corresponding to the input images.
- **camera_layout** (*str*) – The grid shape of the images.
- **n_out** (*int*) – Number of features after last convolution.

11.1.3 WideNetMasked

class indexedconv.nets.aid.**WideNetMasked**(*n_out*)
ResNet like Network from HexaConv paper implementing masked convolutions.

Parameters **n_out** (*int*) – Number of features after last convolution.

11.2 Networks for CIFAR

11.2.1 WideNet

class indexedconv.nets.cifar.**WideNet**
ResNet like Network from HexaConv paper (Z^2).

11.2.2 WideNetIndexConvIndexPool

class indexedconv.nets.cifar.**WideNetIndexConvIndexPool**(*index_matrix*, *camera_layout*)
ResNet like Network from HexaConv paper implemented with indexed convolutions and pooling.

Parameters

- **index_matrix** (*torch.Tensor*) – The index matrix corresponding to the input images.
- **camera_layout** (*str*) – The grid shape of the images.

11.2.3 WideNetIndexConvIndexPoolRetina

class indexedconv.nets.cifar.**WideNetIndexConvIndexPoolRetina**(*index_matrix*, *camera_layout*)
ResNet like Network from HexaConv paper implemented with indexed convolutions (retina like kernel) and pooling.

Parameters

- **index_matrix** (*torch.Tensor*) – The index matrix corresponding to the input images.
- **camera_layout** (*str*) – The grid shape of the images.

11.3 Network for MNIST

11.3.1 GLNet2HexaConvForMnist

class indexedconv.nets.mnist.**GLNet2HexaConvForMnist**(*index_matrix*)
Network with indexed convolutions and pooling (square kernels). 2 CL (after each conv layer, pooling is executed) 1 FC

Parameters **index_matrix** (*torch.Tensor*) – The index matrix corresponding to the input images.

Python Module Index

i

Index