Pytorch Wavelets Documentation

Release 0.1.1

Fergal Cotter

Oct 03, 2019

Contents:

1 Introduction **1**
- 1.1 Installation . . . 4
- 1.2 Notes . . . 4
- 1.3 Provenance . . . 6

2 DWT in Pytorch Wavelets **7**
- 2.1 Differences to PyWavelets . . . 7
- 2.2 Example . . . 8
- 2.3 Other Notes . . . 8

3 DTCWT in Pytorch Wavelets **9**
- 3.1 Notes . . . 9
- 3.2 Example . . . 10
- 3.3 Advanced Options . . . 11

4 Notes on Speed **13**

5 API Guide **15**
- 5.1 Decimated WT . . . 15
- 5.2 Dual Tree Complex WT . . . 16

6 Indices and tables **19**

Bibliography **21**

Python Module Index **23**

Index **25**

CHAPTER 1

Introduction

build failing

This package provides support for computing the 2D discrete wavelet and the 2d dual-tree complex wavelet transforms, their inverses, and passing gradients through both using pytorch.

The implementation is designed to be used with batches of multichannel images. We use the standard pytorch implementation of having 'NCHW' data format.

This repo originally was only for the use of the DTCWT, but I have added some DWT support. This is still in development, and has the following known issues:

- Uses reflection padding instead of symmetric padding for the DWT
- Doesn't compute the DWT separably, instead uses the full $N \, x \, N$ kernel.

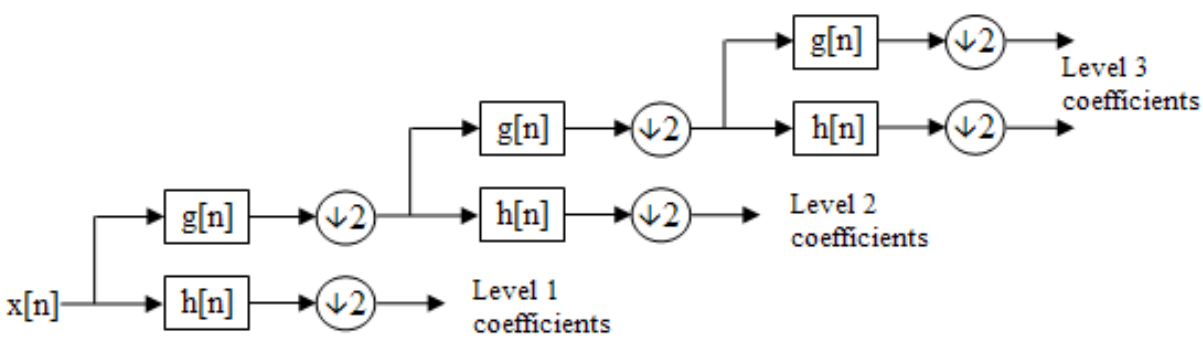


Fig. 1: The subband implementation of the discrete wavelet transform

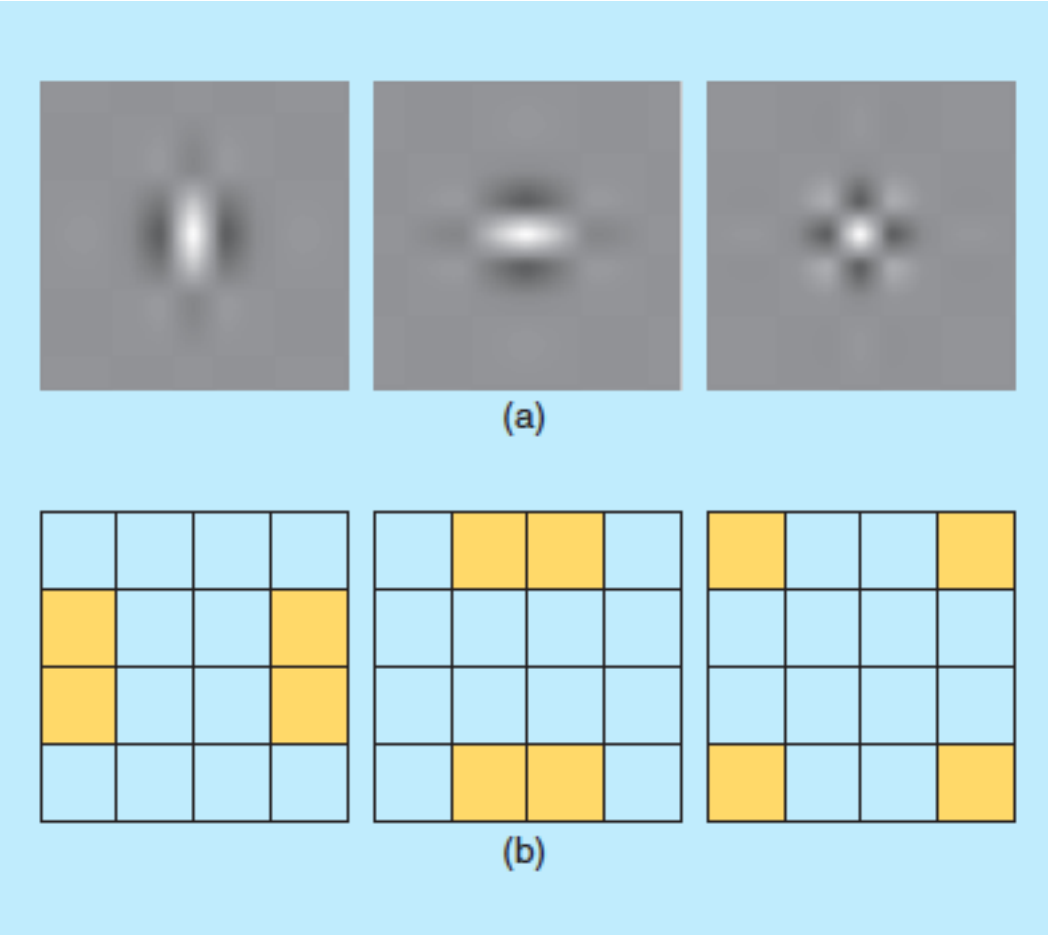


Fig. 2: The equivalent point spread functions of the dwt (a) and the areas of the frequency plane each filter selects (b). Image taken from [SBK05].

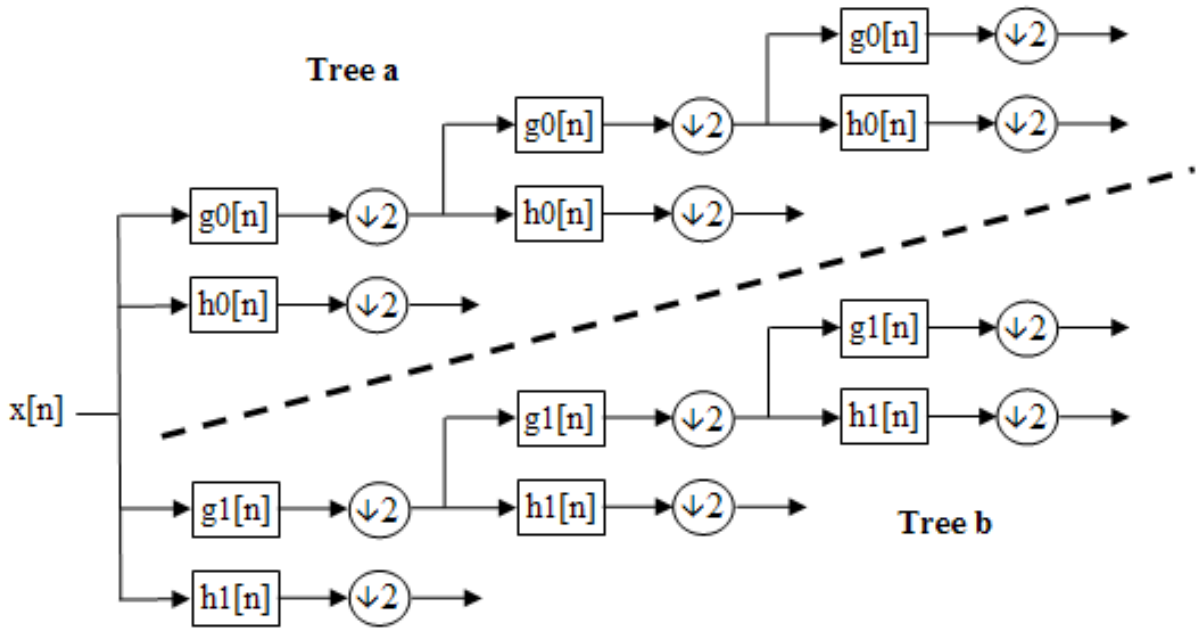


Fig. 3: The subband implementation of the dual tree complex wavelet transform

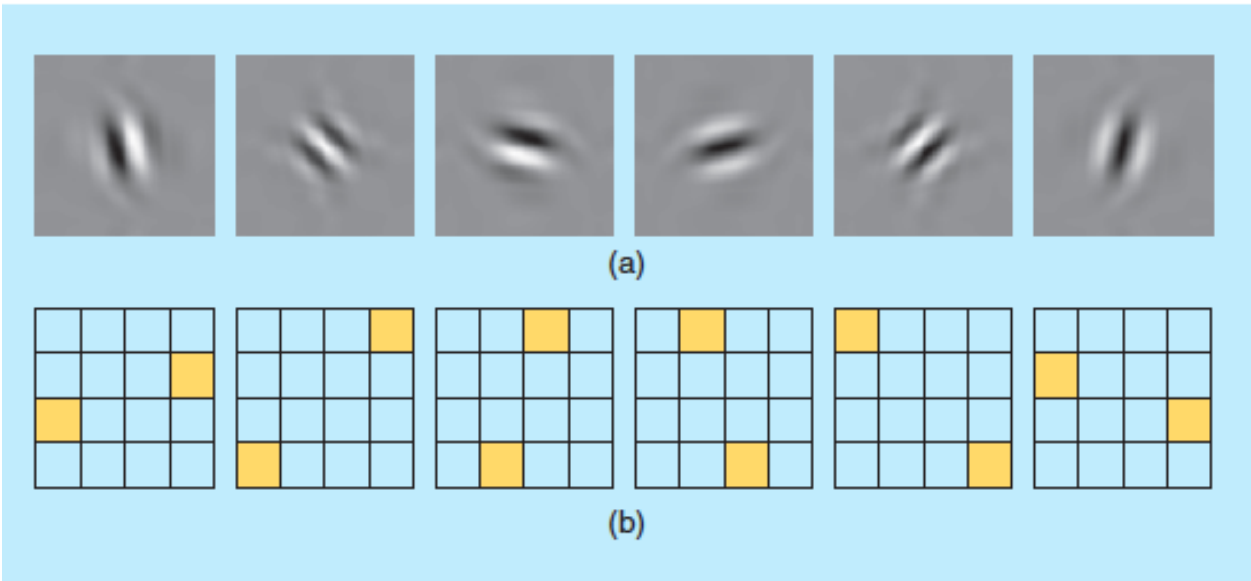


Fig. 4: The equivalent point spread functions of the dtcwt (a) and the areas of the frequency plane each filter selects (b). Image taken from [SBK05].

1.1 Installation

The easiest way to install `pytorch_wavelets` is to clone the repo and pip install it. Later versions will be released on PyPi but the docs need to updated first:

```
$ git clone https://github.com/fbcotter/pytorch_wavelets
$ cd pytorch_wavelets
$ pip install .
```

(Although the *develop* command may be more useful if you intend to perform any significant modification to the library.) A test suite is provided so that you may verify the code works on your system:

```
$ pip install -r tests/requirements.txt
$ pytest tests/
```

1.2 Notes

See the other docs

1.2.1 Floating Point Type

By default, the filters will use 32-bit precision, as is the common case with gpu operations. You can change to 64-bit by calling `torch.set_default_dtype(torch.float64)` before the transforms are constructed.

1.2.2 Running on the GPU

This should come as no surprise to pytorch users. The DWT and DTCWT transforms support cuda calling:

```
import torch
from pytorch_wavelets import DTCWTForward, DTCWTInverse
xfm = DTCWTForward(J=3, biort='near_sym_b', qshift='qshift_b').cuda()
X = torch.randn(10,5,64,64).cuda()
Yl, Yh = xfm(X)
ifm = DTCWTInverse(J=3, biort='near_sym_b', qshift='qshift_b').cuda()
Y = ifm((Yl, Yh))
```

The automated tests cannot test the gpu functionality, but do check cpu running. To test whether the repo is working on your gpu, you can download the repo, ensure you have pytorch with cuda enabled (the tests will check to see if `torch.cuda.is_available()` returns true), and run:

```
pip install -r tests/requirements.txt
pytest tests/
```

From the base of the repo.

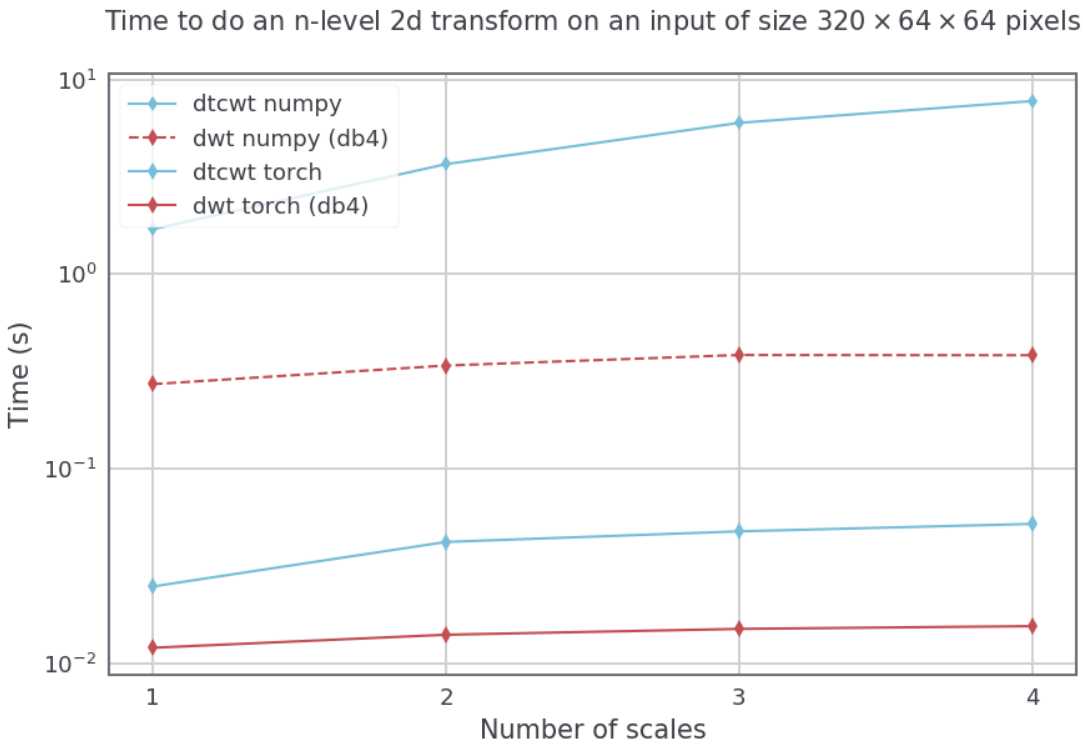
1.2.3 Backpropagation

It is possible to pass gradients through the forward and backward transforms. All you need to do is ensure that the input to each has the required_grad attribute set to true.

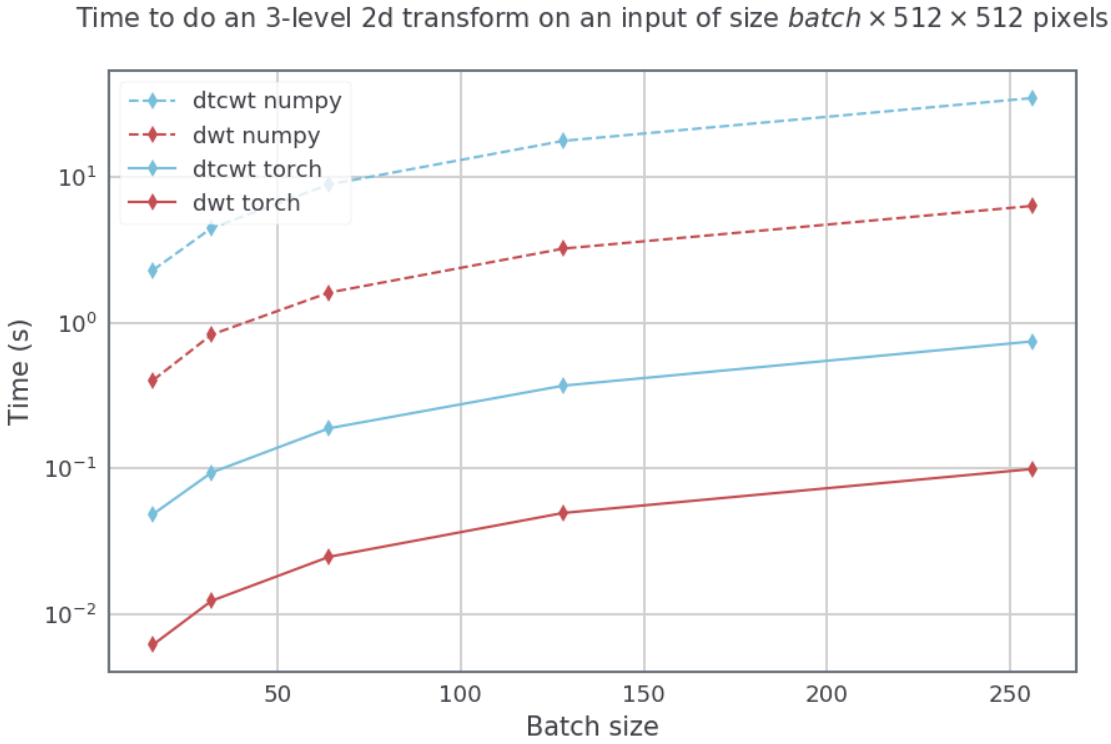
1.2.4 Speed Tests

We compare doing the dtcwt with the python package and doing the dwt with PyWavelets to doing both in pytorch_wavelets, using a GTX1080. The numpy methods were run on a 14 core Xeon Phi machine using intel's parallel python. For the dtwcwt we use the *near_sym_a* filters for the first scale and the *qshift_a* filters for subsequent scales. For the dwt we use the *db4* filters.

For a fixed input size, but varying the number of scales (from 1 to 4) we have the following speeds (averaged over 5 runs):



For an input size with height and width 512 by 512, we also vary the batch size for a 3 scale transform. The resulting speeds were:



1.3 Provenance

Based on the Dual-Tree Complex Wavelet Transform Pack for MATLAB by Nick Kingsbury, Cambridge University. The original README can be found in ORIGINAL_README.txt. This file outlines the conditions of use of the original MATLAB toolbox.

CHAPTER 2

DWT in Pytorch Wavelets

While pytorch_wavelets was initially built as a repo to do the dual tree wavelet transform efficiently in pytorch, I have also built a thin wrapper over PyWavelets, allowing the calculation of the 2D-DWT in pytorch on a GPU on a batch of images.

Older versions did the DWT non separably. As of v1.0.0 we now have code to do it separably. The old non-separable code is still there and is surprisingly sometimes faster. You can test the two out to see which is better for you by changing the *separable* flag in the DWT/IDWT constructor.

The DWT/IDWT now supports most of the padding schemes that PyWavelets uses. In particular:

- symmetric padding
- reflection padding
- zero padding
- periodization

You can see the source here. It is pretty minimal and should be clear what is going on.

In particular, the DWT and IWT classes initialize the filter banks as pytorch tensors (taking care to flip them as pytorch uses cross-correlation not convolution). It then performs non-separable 2D convolution on the input, using strided convolution to calculate the LL, LH, HL, and HH subbands. It also takes care of padding to match the PyWavelets implementation.

2.1 Differences to PyWavelets

2.1.1 Inputs

The pytorch_wavelets DWT expects the standard pytorch image format of NCHW - i.e., a batch of N images, with C channels, height H and width W. For a single RGB image, you would need to make it a torch tensor of size `(1, 3, H, W)`, or for a batch of 100 grayscale images, you would need to make it a tensor of size `(100, 1, H, W)`.

2.1.2 Returned Coefficients

We deviate slightly from PyWavelets with the format of the returned coefficients. In particular, we return a tuple of `(yl, yh)` where yl is the LL band, and *yh* is a list. The first list entry *yh[0]* are the scale 1 bandpass coefficients (finest resolution), and the last list entry *yh[-1]* are the coarsest bandpass coefficients. Note that this is the reverse of the PyWavelets format (but fits with the dtcwt standard output). Each of the bands is a single stacked tensor of the LH (horiz), HL (vertic), and HH (diag) coefficients for each scale (as opposed to PyWavelets style of returning as a tuple) with the stack along the third dimension. As the input had 4 dimensions, this output has 5 dimensions, with shape `(N, C, 3, H, W)`. This is easily transformed into the PyWavelets style by unstacking the list elements in *yh*.

2.2 Example

```
import torch
from pytorch_wavelets import DWTForward, DWTInverse # (or import DWT, IDWT)
xfm = DWTForward(J=3, mode='zero', wave='db3')  # Accepts all wave types available to␣
↪PyWavelets
ifm = DWTInverse(mode='zero', wave='db3')
X = torch.randn(10,5,64,64)
Yl, Yh = xfm(X)
print(Yl.shape)
>>> torch.Size([10, 5, 12, 12])
print(Yh[0].shape)
>>> torch.Size([10, 5, 3, 34, 34])
print(Yh[1].shape)
>>> torch.Size([10, 5, 3, 19, 19])
print(Yh[2].shape)
>>> torch.Size([10, 5, 3, 12, 12])
Y = ifm((Yl, Yh))
import numpy as np
np.testing.assert_array_almost_equal(Y.cpu().numpy(), X.cpu().numpy())
```

2.3 Other Notes

2.3.1 GPU Calculations

As you would expect, you can move the transforms to the GPU by calling `xfm.cuda()` or `ifm.cuda()`, where *xfm*, *ifm* are instances of `pytorch_wavelets.DWTForward` and `pytorch_wavelets.DWTInverse`.

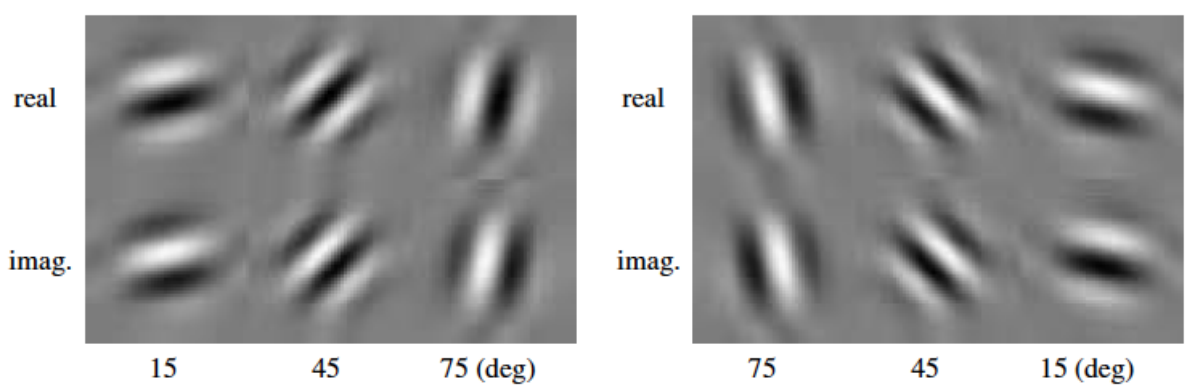
CHAPTER 3

DTCWT in Pytorch Wavelets

Pytorch wavelets is a port of dtcwt_slim, which was my first attempt at doing the DTCWT quickly on a GPU. It has since been cleaned up to run for pytorch and do the quickest forward and inverse transforms I can make, as well as being able to pass gradients through the inputs.

For those unfamiliar with the DTCWT, it is a shift invariant wavelet transform that comes with limited redundancy. Compared to the undecimated wavelet transform, which has 2^J redundancy, the DTCWT only has 2^d redundancy (where d is the number of input dimensions - i.e. 4:1 redundancy for image transforms). Instead of producing 3 output subbands like the DWT, it produces 6, which roughly represent 15, 45, 75, 105, 135 and 165 degree wavelets. On top of this, the 6 subbands have real and imaginary outputs which are in quadrature with each other (similar to windowed sine and cosine functions, or the gabor wavelet).

It is possible to calculate similar transforms (such as the morlet or gabor) using fourier transforms, but the DTCWT is faster as it uses separable convolutions.



3.1 Notes

Because of the above mentioned properties of the DTCWT, the output is slightly different to the DWT. As mentioned, it is 4:1 redundant in 2D, so we expect 4 times as many coefficients as from the decimated wavelet transform. These

extra coefficients come from:

- 6 subband outputs instead of 3 with ial and imaginary coefficients instead of just real. For an $N \times N$ image, the first level bandpass has $3N^2$ instead of $3N^2/4$ coefficients.
- The lowpass is always at double the resolution of what you'd expect it to be for the level in the wavelet tree. I.e. for a 1 level transform, the lowapss output is still $N \times N$. For a two level transform, it is $N/2 \times N/2$ and so on.

3.2 Example

```
import torch
from pytorch_wavelets import DTCWTForward, DTCWTInverse
xfm = DTCWTForward(J=3, biort='near_sym_b', qshift='qshift_b')
X = torch.randn(10,5,64,64)
Yl, Yh = xfm(X)
print(Yl.shape)
>>> torch.Size([10, 5, 16, 16])
print(Yh[0].shape)
>>> torch.Size([10, 5, 6, 32, 32, 2])
print(Yh[1].shape)
>>> torch.Size([10, 5, 6, 16, 16, 2])
print(Yh[2].shape)
>>> torch.Size([10, 5, 6, 8, 8, 2])
ifm = DTCWTInverse(biort='near_sym_b', qshift='qshift_b')
Y = ifm((Yl, Yh))
```

Like with the DWT, Yh returned is a tuple. There are 2 extra dimensions - the first comes between the channel dimension of the input and the row dimension. This is the 6 orientations of the DTCWT. The second is the final dimension, which is the real an imaginary parts (complex numbers are not native to pytorch). I.e. to access the real part of the 45 degree wavelet for the first subband, you would use `Yh[0][:,:,1,:,:,0]`, and the imaginary part of the 165 degree wavelet would be `Yh[0][:,:,5,:,:,1]`.

The above images were created by doing a forward transform with an input of zeros (creates a pyramid with the correct size bands), and then setting the centre spatial value to 1 for each of the orientations at the third scale. I.e.:

```
import numpy as np
import torch
from pytorch_wavelets import DTCWTForward, DTCWTInverse
xfm = DTCWTForward(J=3)
ifm = DTCWTInverse()
x = torch.zeros(1,1,64,64)
# Create 12 outputs, one for the real and imaginary point spread functions
# for each of the 6 orientations
out = np.zeros((12,64,64)
yl, yh = xfm(x)
for b in range(6):
  for ri in range(2):
    yh[2][0,0,b,4,4,ri] = 1
    out[b*2 + ri] = ifm((yl, yh))
    yh[2][0,0,b,4,4,ri] = 0
# Can now plot the output
```

3.3 Advanced Options

There is a whole host of advanced options for calculating the DTCWT. The above example shows the use case that will work most of the time. However, here are some more ways the DTCWT can be done:

3.3.1 Custom Biorthogonal and Qshift Filters

Rather than specifying the type of filter for the layer 1 (biort), and layer 2+ (qshift) transforms, you can provide the filters directly. They should be given as a tuple of array-like objects. For the biorthogonal filters, this is a 2-tuple of low and highpass filters. For the qshift filters, this will be a 4-tuple of low for tree a, low for tree b, high for tree a and high for tree b filters.

E.g.:

```
from pytorch_wavelets import DTCWTForward
from pytorch_wavelets.dtcwt.coeffs import biort
# The standard style
xfm1 = DTCWTForward(biort='near_sym_a', J=1)
# Get our own filters, here we reverse the standard filters so they
# still have the right properties, only changing the phase
h0o, _, h1o, _ = biort('near_sym_a')
xfm2 = DTCWTForward(biort=(h0o[::-1], h1o[::-1]), J=1)
```

Note that you must be careful when doing this, as the filters are designed to have the correct phase properties, so any changes will likely result in a loss of the quarter shift and hence the shift invariant properties of the transform.

3.3.2 Skipping Highpasses

There is the option to not calculate the bandpass outputs at given scales. This can speed up the transform if you know that there is very little useful content in some areas of the frequency space. To do this, you can give a list of booleans to the *skip_hps* parameter (if it is a single boolean, that is then used for all the scales). The first value corresponds to the first scale highpass outputs, and a value of true means do not calculate them.

E.g.:

```
from pytorch_wavelets import DTCWTForward
xfm = DTCWTForward(J=3, skip_hps=[True, False, False])
yl, yh = xfm(torch.randn(1, 1, 64, 64))
print(yh[0].shape)
>>> torch.Size([0])
print(yh[1].shape)
>>> torch.Size([1, 1, 6, 16, 16, 2])
```

Naturally, the inverse transform happily accepts tensors with 0 shape (or even *None*'s) and sets that level to be all zeros.

3.3.3 Changing the output shape

By default the highpass outputs have an extra 2 dimensions, one at the end for complex values, and one after the channel dimension, for the 6 orientations. E.g. an input of shape of $(N, C_{in}, H_{in}, W_{in})$ will have bandpass coefficients with shapes $list(N, C_{in}, 6, H''_{in}, W''_{in}, 2)$, (we've put dashes next to the height and width as they will change with scale).

You can choose where the orientations and real and imaginary dimensions go with the options *o_dim* and *ri_dim*, which are by default 2 and -1.

3.3.4 Including all the lowpasses

In case you want to get all the intermediate lowpasses, you can with the *include_scale* parameter. This works a bit like the *skip_hps* where you can provide a single boolean to apply it to all the scales, or a list of booleans to fine tune which lowpasses you want.

If any of the value in *include_scale* is true, then the transform output will change, and the lowpass will be a tuple.

E.g.

```
from pytorch_wavelets import DTCWTForward
xfm1 = DTCWTForward(J=3)
xfm2 = DTCWTForward(J=3, include_scale=True)
xfm3 = DTCWTForward(J=3, include_scale=[False, True, True])
x = torch.randn(1, 1, 64, 64)
yl, yh = xfm1(x)
print(yl.shape)
>>> torch.Size([1, 1, 16, 16])
# Now do xfm2 which will give back all scales
yl, yh = xfm2(x)
for l in yl:
  print(yl.shape)
>>> torch.Size([1, 1, 64, 64])
>>> torch.Size([1, 1, 32, 32])
>>> torch.Size([1, 1, 16, 16])
# Now do xfm3 which will give back the last two scales
yl, yh = xfm3(x)
for l in yl:
  print(yl.shape)
>>> torch.Size([0])
>>> torch.Size([1, 1, 32, 32])
>>> torch.Size([1, 1, 16, 16])
```

Note that to do the inverse transform, you have to give the final lowpass output. You can provide None to indicate it's all zeros, but you cannot provide all the intermediate lowpasses.

CHAPTER 4

Notes on Speed

Under tests/, the *profile_xfms* script tests the speed of several layers of the DTCWT for working on a moderately sized input $X \in \mathbb{R}^{10\times10\times128\times128}$. As a reference, an 11 by 11 convolution takes 2.53ms for a tensor of this size.

A single layer DTCWT using the 'near_sym_a' filters (lengths 5 and 7) has 6 convolutional calls. I timed them at 238us each for a total of 1.43ms. Unfortunately, there is also a bit of overhead in calculating the DTCWT, and not all non convolutional operations are free. In addition to the 6 convolutions, there were:

- 6 move ops @ 119us = 714us
- 10 pointwise add ops @ 122us = 465us
- 12 copy ops @ 35us = 381us
- 6 different add ops @ 38us = 232us
- 6 subtraction ops @ 37us = 220us
- 3 constant division ops @ 57us = 173us
- 6 more move ops @ 28us = 171us

Making the overheads 2.3ms, and 3.7ms total time.

For a two layer DTCWT, there are now 12 convolutional ops. The second layer kernels are slightly larger (10 taps each) so although they act over 1/4 the sample size, they take up an extra 1.1ms (2.5ms total for the 12 convs). The overhead for non convolution operations is 4.4ms, making 6.9ms. Roughly 3 times a long as an 11 by 11 convolution.

There is an option to not calculate the highpass coefficients for the first scale, as these often have limited useful information (see the *skip_hps* option). For a two scale transform, this takes the convolution run time down to 1.13ms and the overhead down to 2.49ms, totaling 3.6ms, or roughly the same time as the 1 layer transform.

A single layer inverse transform takes: 1.43ms (conv) + 2.7ms (overhead) totaling 4.1ms, slightly longer than the 3.7ms for the forward transform.

A two layer inverse transform takes: 2.24 (conv) + 5.9 (overhead) totaling 8.1ms, again slightly longer than the 6.9ms for the forward transform.

A single layer end to end transform takes 2.86ms (conv) + 5.8ms (overhead) = 8.6ms $\approx$ 3.7 (forward) + 4.1 (inverse).

Similarly, a two layer end to end transform takes 4.4ms (conv) + 10.4ms (overhead) = 14.8ms $\approx$ 6.9 (forward) + 8.1 (inverse).

If we use the *near_sym_b* filters for layer 1 (13 and 19 taps), the overhead doesn't increase, but the time taken to do each convolution unsurprisingly triples to 600us each (up from 200us for *near_sym_a*).

CHAPTER 5

API Guide

5.1 Decimated WT

class pytorch_wavelets.**DWTForward**(*J=1*, *wave='db1'*, *mode='zero'*)

Bases: torch.nn.modules.module.Module

Performs a 2d DWT Forward decomposition of an image

Parameters

- **J** (*int*) – Number of levels of decomposition
- **wave** (*str or pywt.Wavelet*) – Which wavelet to use. Can be a string to pass to pywt.Wavelet constructor, can also be a pywt.Wavelet class, or can be a two tuple of array-like objects for the analysis low and high pass filters.
- **mode** (*str*) – 'zero', 'symmetric', 'reflect' or 'periodization'. The padding scheme
- **separable** (*bool*) – whether to do the filtering separably or not (the naive implementation can be faster on a gpu).

forward(*x*)

Forward pass of the DWT.

Parameters **x** (*tensor*) – Input of shape $(N, C_{in}, H_{in}, W_{in})$

Returns

(yl, yh) tuple of lowpass (yl) and bandpass (yh) coefficients. yh is a list of length J with the first entry being the finest scale coefficients. yl has shape $(N, C_{in}, H'_{in}, W'_{in})$ and yh has shape $list(N, C_{in}, 3, H''_{in}, W''_{in})$. The new dimension in yh iterates over the LH, HL and HH coefficients.

Note: $H'_{in}, W'_{in}, H''_{in}, W''_{in}$ denote the correctly downsampled shapes of the DWT pyramid.

class pytorch_wavelets.**DWTInverse**(*wave='db1'*, *mode='zero'*)

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

Performs a 2d DWT Inverse reconstruction of an image

Parameters

- **wave** (*str or pywt.Wavelet*) – Which wavelet to use
- **C** – deprecated, will be removed in future

forward(*coeffs*)

Parameters **coeffs** (*yl, yh*) – tuple of lowpass and bandpass coefficients, where: yl is a lowpass tensor of shape $(N, C_{in}, H'_{in}, W'_{in})$ and yh is a list of bandpass tensors of shape $list(N, C_{in}, 3, H''_{in}, W''_{in})$. I.e. should match the format returned by DWTForward

Returns Reconstructed input of shape $(N, C_{in}, H_{in}, W_{in})$

Note: $H'_{in}, W'_{in}, H''_{in}, W''_{in}$ denote the correctly downsampled shapes of the DWT pyramid.

Note: Can have None for any of the highpass scales and will treat the values as zeros (not in an efficient way though).

5.2 Dual Tree Complex WT

class pytorch_wavelets.**DTCWTForward**(*biort='near_sym_a'*, *qshift='qshift_a'*, *J=3*, *skip_hps=False*, *include_scale=False*, *o_dim=2*, *ri_dim=-1*, *mode='symmetric'*)

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

Performs a 2d DTCWT Forward decomposition of an image

Parameters

- **biort** (*str*) – One of 'antonini', 'legall', 'near_sym_a', 'near_sym_b'. Specifies the first level biorthogonal wavelet filters. Can also give a two tuple for the low and highpass filters directly.
- **qshift** (*str*) – One of 'qshift_06', 'qshift_a', 'qshift_b', 'qshift_c', 'qshift_d'. Specifies the second level quarter shift filters. Can also give a 4-tuple for the low tree a, low tree b, high tree a and high tree b filters directly.
- **J** (*int*) – Number of levels of decomposition
- **skip_hps** (*bools*) – List of bools of length J which specify whether or not to calculate the bandpass outputs at the given scale. skip_hps[0] is for the first scale. Can be a single bool in which case that is applied to all scales.
- **include_scale** (*bool*) – If true, return the bandpass outputs. Can also be a list of length J specifying which lowpasses to return. I.e. if [False, True, True], the forward call will return the second and third lowpass outputs, but discard the lowpass from the first level transform.
- **o_dim** (*int*) – Which dimension to put the orientations in
- **ri_dim** (*int*) – which dimension to put the real and imaginary parts

forward(*x*)

Forward Dual Tree Complex Wavelet Transform

Parameters **x** (*tensor*) – Input to transform. Should be of shape $(N, C_{in}, H_{in}, W_{in})$.

Returns

(yl, yh) tuple of lowpass (yl) and bandpass (yh) coefficients. If include_scale was true, yl will be a list of lowpass coefficients, otherwise will be just the final lowpass coefficient of shape $(N, C_{in}, H'_{in}, W'_{in})$. Yh will be a list of the complex bandpass coefficients of shape $list(N, C_{in}, 6, H''_{in}, W''_{in}, 2)$, or similar shape depending on o_dim and ri_dim

Note: $H'_{in}, W'_{in}, H''_{in}, W''_{in}$ are the shapes of a DTCWT pyramid.

class pytorch_wavelets.**DTCWTInverse**(*biort='near_sym_a', qshift='qshift_a', o_dim=2, ri_dim=-1, mode='symmetric'*)

Bases: torch.nn.modules.module.Module

2d DTCWT Inverse

Parameters

- **biort** (*str*) – One of 'antonini', 'legall', 'near_sym_a', 'near_sym_b'. Specifies the first level biorthogonal wavelet filters. Can also give a two tuple for the low and highpass filters directly.
- **qshift** (*str*) – One of 'qshift_06', 'qshift_a', 'qshift_b', 'qshift_c', 'qshift_d'. Specifies the second level quarter shift filters. Can also give a 4-tuple for the low tree a, low tree b, high tree a and high tree b filters directly.
- **J** (*int*) – Number of levels of decomposition.
- **o_dim** (*int*) – which dimension the orientations are in
- **ri_dim** (*int*) – which dimension to put th real and imaginary parts in

forward(*coeffs*)

Parameters **coeffs** (*yl, yh*) – tuple of lowpass and bandpass coefficients, where: yl is a tensor of shape $(N, C_{in}, H'_{in}, W'_{in})$ and yh is a list of the complex bandpass coefficients of shape $list(N, C_{in}, 6, H''_{in}, W''_{in}, 2)$, or similar depending on o_dim and ri_dim

Returns Reconstructed output

Note: Can accept Nones or an empty tensor (torch.tensor([])) for the lowpass or bandpass inputs. In this cases, an array of zeros replaces that input.

Note: $H'_{in}, W'_{in}, H''_{in}, W''_{in}$ are the shapes of a DTCWT pyramid.

Note: If include_scale was true for the forward pass, you should provide only the final lowpass output here, as normal for an inverse wavelet transform.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Bibliography

[SBK05] Ivan W. Selesnick, Richard G. Baraniuk, and Nick G. Kingsbury. The dual-tree complex wavelet transform. *Signal Processing Magazine, IEEE*, 22(6):123–151, 2005. 01602.

Python Module Index

p

pytorch_wavelets, 16

Index

D

`DTCWTForward` (*class in pytorch_wavelets*), 16
`DTCWTInverse` (*class in pytorch_wavelets*), 17
`DWTForward` (*class in pytorch_wavelets*), 15
`DWTInverse` (*class in pytorch_wavelets*), 15

F

`forward()` (*pytorch_wavelets.DTCWTForward method*), 16
`forward()` (*pytorch_wavelets.DTCWTInverse method*), 17
`forward()` (*pytorch_wavelets.DWTForward method*), 15
`forward()` (*pytorch_wavelets.DWTInverse method*), 16

P

`pytorch_wavelets` (*module*), 15, 16