Chapter 1

A Faster ScatterNet

The drive of this thesis is in exploring if wavelet theory, in particular the DT$\mathbb{C}$WT, has any place in deep learning and if it does, quantifying how beneficial it can be. The introduction of more powerful GPUs and fast and popular deep learning frameworks such as PyTorch, Tensorflow and Caffe in the past few years has helped the field of deep learning grow very rapidly. Never before has it been so possible and so accessible to test new designs and ideas for a machine learning algorithm than today. Despite this rapid growth, there has been little interest in building wavelet analysis software in modern frameworks.

This poses a challenge and an opportunity. To pave the way for more detailed investigation (both by myself in the rest of this thesis, and by other researchers who want to explore wavelets applied to deep learning), we must have the right foundation and tools to facilitate research.

A good example of this is the current implementation of the ScatterNet. While ScatterNets have been the most promising start in using wavelets in a deep learning system, they have tended to be orders of magnitude slower, and significantly more difficult to run than a standard convolutional network.

Additionally, any researchers wanting to explore the DWT in a deep learning system have had to rewrite the filter bank implementation themselves, ensuring they correctly handle boundary conditions and ensure correct filter tap alignment to achieve perfect reconstruction.

This chapter describes how we have built a fast ScatterNet implementation in PyTorch with the DT$\mathbb{C}$WT as its wavelet transform. At the core of the DT$\mathbb{C}$WT is an efficient implementation of the DWT. The result is an open-source library that provides all three (the DWT, DT$\mathbb{C}$WT and DT$\mathbb{C}$WT ScatterNet), available on GitHub as *PyTorch Wavelets* [1].

In parallel with our efforts, the original authors of the ScatterNet have improved their implementation, also building it on PyTorch. Our proposed DT$\mathbb{C}$WT ScatterNet is 7 to 15 times faster than their improved implementation (depending on the padding style and wavelet length), while using less GPU memory.

1.1 The Design Constraints

1.1.1 Original Design

The original authors implemented their ScatterNet in Matlab [2] using a Fourier-domain based Morlet wavelet transform. The standard procedure for using ScatterNets in a deep learning framework up until recently has been to:

1. Pre-scatter a dataset using conventional CPU-based hardware and software and store the features to disk. This can take several hours to several days depending on the size of the dataset and the number of CPU cores available.
2. Build a network in another framework, usually Tensorflow [3] or Pytorch [4].
3. Load the scattered data from disk and train on it.

We saw that this approach was suboptimal for a number of reasons:

- It is slow and must run on CPUs.
- It is inflexible to any changes you wanted to investigate in the Scattering design; you would have to re-scatter all the data and save elsewhere on disk.
- You can not easily do preprocessing techniques like random shifts and flips, as each of these would change the scattered data.
- The scattered features are often larger than the original images and require you to store entire datasets twice (or more) times.
- The features are fixed and can only be used as a front end to any deep learning system.

1.1.2 Improvements

To address these shortcomings, all of the above limitations become design constraints. In particular, the new software should be:

- Able to run on GPUs (ideally on multiple GPUs in parallel).
- Flexible and fast so that it can run as part of the forward pass of a neural network (allowing preprocessing techniques like random shifts and flips).
- Able to pass gradients through, so that it can be part of a larger network and have learning stages before scattering.

To achieve all of these goals, we choose to build our software on PyTorch, a popular open source deep learning framework that can do many operations on GPUs with native

support for automatic differentiation. PyTorch uses the CUDA and cuDNN libraries for its GPU-accelerated primitives. Its popularity is of key importance, as it means users can build complex networks involving ScatterNets without having to use or learn extra software.

As mentioned earlier, the original authors of the ScatterNet also noticed the shortcomings with their Scattering software, and recently released a new package that can do Scattering in PyTorch called KyMatIO[5]. The key difference between our proposed package and their improved package is the use of the DT$\mathbb{C}$WT for the choice of complex wavelet, rather than Morlet. While the focus of this chapter is in detailing how we have built a fast, GPU-ready, deep learning compatible library that can do the DWT, DT$\mathbb{C}$WT, and DT$\mathbb{C}$WT ScatterNet, we also compare the speed and classification performance of our package to KyMatIO, as it provides some interesting insights into some of the choice of complex wavelet that can be used for a ScatterNet.

1.2 A Brief Description of Autograd

As part of a modern deep learning framework, we need to define functional units like the one shown in **??**. Not only must we be able to calculate the forward evaluation of a block $y = f(x, w)$ given an input x and (possibly) some learnable weights w, we must also be able to calculate the passthrough and update gradients $\frac{\partial \mathcal{L}}{\partial x}$, $\frac{\partial \mathcal{L}}{\partial w}$ given $\frac{\partial \mathcal{L}}{\partial y}$. This typically involves saving $\frac{\partial y}{\partial x}$ and $\frac{\partial y}{\partial w}$ evaluated at the current values of x and w when we calculate the forward pass.

For example, the simple ReLU: $y = \max(0,\ x)$, is not memory-less. On the forward pass, we need to put a 1 in all the positions where $x > 0$, and a 0 elsewhere. For a convolutional layer, we need to save x and w to correlate with $\frac{\partial \mathcal{L}}{\partial y}$ on the backwards pass. It is up to the block designer to manually calculate the gradients and design the most efficient way of programming them.

For clarity and repeatability, we give pseudocode for all the core operations developed in our package *PyTorch Wavelets*. We carry this through to other chapters when we design different wavelet based blocks. By the end of this thesis, it should be clear how every attempted method has been implemented.

Note that the pseudo code can be for a simple routine or as part of an autograd block. The latter will be clear as it will involve saving of activations for the bacwards pass.

1.3 Fast Calculation of the DWT and IDWT

To have a fast implementation of the Scattering Transform, we need a fast implementation of the DT$\mathbb{C}$WT. Similarly, for a fast implementation of the DT$\mathbb{C}$WT we need a fast implementation of the DWT. Future work may also explore the DWT as a basis for learning,

so having an implementation that is fast and can pass gradients through will prove beneficial in and of itself.

1.3.1 Preliminary Notes

There has been much research into the best way to do the DWT on a GPU, in particular comparing the speed of Lifting [6], or second generation wavelets, to the direct convolutional methods. [7], [8] are two notable such publications, both of which find that the convolution based implemetations are better suited for the massively parallel architecture found in modern GPUs. For this reason, we implement a convolutional based DWT.

As the DWT and IDWT uses fixed filters, we do not need to calculate $\frac{\partial L}{\partial h}$ for the convolutional calls. This means on the forward pass we can save on memory by not storing $\frac{\partial Y}{\partial h} = X$ (see **??**). We found it easiest to explicitly rewrite forward and backward passes for the DWT, as the autograd mechanics would often unnecessarily save intermediate activations.

For example, consider an input $x \in \mathbb{R}^{128\times256\times64\times64}$. This is not an unrealistically large input for a CNN, as often minibatch sizes are 128, channels can be in the hundreds and 64×64 pixels is a relatively manageable input size. If we were to represent these numbers as floats, it would require 512MB of space on a GPU (modern GPUs have about 10GB of memory, so this single activation already requires 5%). If we perform a DWT on this input, we would require another 512MB of space for the output. Using naive autograd, PyTorch (or another framework) saves an extra 512MB of memory for the backwards pass to calculate $\frac{\partial L}{\partial h}$, even if we specify it as not requiring a gradient. This means we can save 50% of the memory cost by being explicit about what is and is not needed for backpropagation.

1.3.1.1 The Input

Recall from **??** that our input is a 3-D array with channel dimension first, followed by the two spatial dimensions. For a minibatch of images, this becomes a 4-D array, with the sample number in the first dimension. I.e., for a minibatch of N images with C channels and $H \times W$ spatial support, the input has shape: $N \times C \times H \times W$.

1.3.2 Primitives

We start with the commonly known property that for a convolutional block, the gradient with respect to the input is the gradient with respect to the output convolved with the time reverse of the filter. More formally, if $Y(z) = H(z)X(z)$:

$$\Delta X(z) = H(z^{-1})\Delta Y(z) \tag{1.3.1}$$

where $H(z^{-1})$ is the Z-transform of the time/space reverse of $H(z)$, $\Delta Y(z) \triangleq \frac{\partial L}{\partial Y}(z)$ is the gradient of the loss with respect to the output, and $\Delta X(z) \triangleq \frac{\partial L}{\partial X}(z)$ is the gradient of the

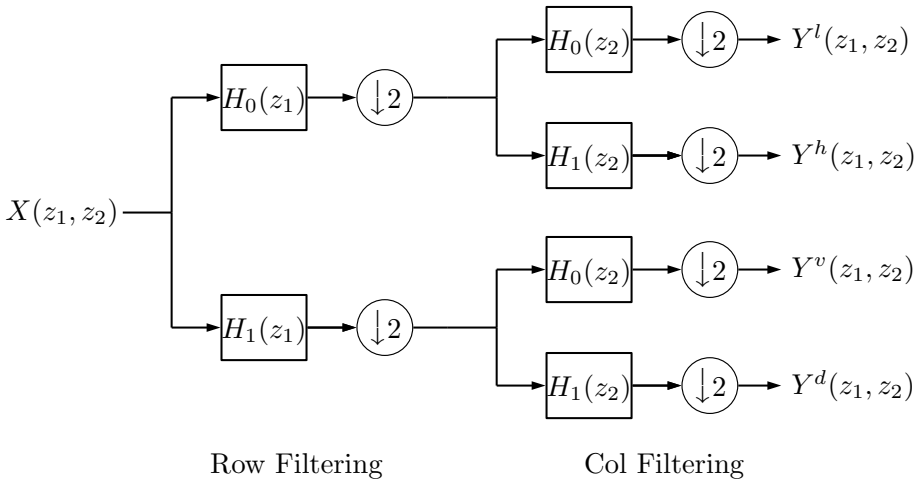


Figure 1.1: **Block Diagram of 2-D DWT.** The components of a filter bank DWT in two dimensions. Y^l, Y^h, Y^v, Y^d represent the lowpass, horizontal, vertical, and diagonal components.

loss with respect to the input. This was proved in **??**, we have just rewritten it in terms of Z-transforms here.

Additionally, if we decimate by a factor of two on the forwards pass, the equivalent backwards pass is interpolating by a factor of two (see section B.1 for a proof of this).

Figure 1.1 shows the block diagram for performing the forward pass of a DWT. We can draw a similar figure for the backwards pass by replacing all the forward operations for their passthrough gradients. Using the above two properties for the backwards pass of convolution and sample rate changes, we quickly see that the backwards pass of a wavelet transform has the same form as the inverse wavelet transform, with the time reverse of the analysis filters used as the synthesis filters. For orthogonal wavelet transforms, the synthesis filters are the time reverse of the analysis filters so backpropagation can be done by calling the inverse wavelet transform on the wavelet coefficient gradients. See section B.2 for more information on this and the equivalent figure for the backwards pass.

It is important to note that like Matlab, most deep learning frameworks have an efficient function for doing convolution followed by downsampling. Similarly, there is an efficient function to do upsampling followed by convolution (for the inverse transform). Let us call these `conv2d_down` and `conv2d_up` respectively[1]. These functions in turn call the cuDNN low-level fucntions which can only support zero padding. If another padding type is desired, it must be done beforehand with a padding function `pad`.

[1]E.g. In PyTorch, convolution followed by downsampling is done with a call to `torch.nn.functional.conv2d` with the stride parameter set to 2. Upsampling by 2 followed by convolution is done by calling `torch.nn.functional.conv2d_transpose`.

Algorithm 1.1 1-D analysis and synthesis stages of a DWT

function AFB1D(x, h_0, h_1, $mode$, $axis$)
  if axis $= 3$ **then**
    $h_0,\ h_1 \leftarrow h_0^t,\ h_1^t$ ▷ row filtering
  end if
  $p \leftarrow \lfloor (\text{len}(x) + \text{len}(h_0) - 1)/2 \rfloor$ ▷ calculate output size
  $b \leftarrow \lfloor p/2 \rfloor$ ▷ calculate pad size before
  $a \leftarrow \lceil p/2 \rceil$ ▷ calculate pad size after
  $x \leftarrow \texttt{pad}(x,\ b,\ a,\ mode)$ ▷ pre pad the signal with selected mode
  $lo \leftarrow \texttt{conv2d\_down}(x,\ h_0)$
  $hi \leftarrow \texttt{conv2d\_down}(x,\ h_1)$
  return $lo,\ hi$
end function

function SFB1D(lo, hi, g_0, g_1, $mode$, $axis$)
  if axis $= 3$ **then**
    $g_0,\ g_1 \leftarrow g_0^t,\ g_1^t$ ▷ row filtering
  end if
  $p \leftarrow \text{len}(g_0) - 2$ ▷ calculate output size
  $lo \leftarrow \texttt{pad}(lo,\ p,\ p,\ `zero')$ ▷ pre pad the signal with zeros
  $hi \leftarrow \texttt{pad}(hi,\ p,\ p,\ `zero')$ ▷ pre pad the signal with zeros
  $x \leftarrow \texttt{conv2d\_up}(lo,\ g_0) + \texttt{conv2d\_up}(hi,\ g_1)$
  return x
end function

With both `conv2d_down` and `conv2d_up`, we want to apply the same filter to each channel of the input and *not* sum them up (recall (**??**) sums over the channel dimension after doing convolution). There are options available to prevent the summing in most frameworks, related to doing *grouped convolution*. For a good explanation on grouped convolution, we recommend Chapter 5 of [9]. In our code below, we assume that `conv2d_down` and `conv2d_up` act independently on the C channels and do not sum across them, giving an output with the same shape as the input - $N \times C \times H \times W$.

1.3.3 1-D Filter Banks

Let us assume that the analysis ($h_0,\ h_1$) and synthesis ($g_0,\ g_1$) filters are already in the form needed to do *column* filtering. The necessary steps to do the 1-D analysis and synthesis filtering are described in Algorithm 1.1. We do not need to define backpropagation functions for the `afb1d` (analysis filter bank 1-d) and `sfb1d` (synthesis filter bank 1-D) functions as they are each others backwards pass.

Algorithm 1.2 2-D DWT and its gradient

function DWT.FORWARD(x, h_0^c, h_1^c, h_0^r, h_1^r, $mode$)
  save h_0^c, h_1^c, h_0^r, h_1^r, $mode$ ▷ For the backwards pass
  lo, $hi \leftarrow$ `afb1d`(x, h_0^r, h_1^r, $mode$, $axis = 3$) ▷ row filter
  ll, $lh \leftarrow$ `afb1d`(lo, h_0^c, h_1^c, $mode$, $axis = 2$) ▷ column filter
  hl, $hh \leftarrow$ `afb1d`(hi, h_0^c, h_1^c, $mode$, $axis = 2$) ▷ column filter
  return ll, lh, hl, hh
end function

function DWT.BACKWARD(Δll, Δlh, Δhl, Δhh)
  load h_0^c, h_1^c, h_0^r, h_1^r, $mode$
  $\Delta lo \leftarrow$ `sfb1d`(Δll, Δlh, h_0^c, h_1^c, $mode$, $axis = 2$)
  $\Delta hi \leftarrow$ `sfb1d`(Δhl, Δhh, h_0^c, h_1^c, $mode$, $axis = 2$)
  $\Delta x \leftarrow$ `sfb1d`(Δlo, Δhi, h_0^r, h_1^r, $mode$, $axis = 3$)
  return Δx
end function

1.3.4 2-D Transforms and Their Gradients

Having built the 1-D filter banks, we can easily generalize them to 2-D. Furthermore, we can define the backwards steps of both the forward 2-D DWT and the inverse 2-D DWT using these filter banks. We show how to do do this in Algorithm 1.2. The inverse transform logic is moved to the appendix Algorithm B.1.

Note that we have allowed for different row and column filters in Algorithm 1.2. Most commonly used wavelets will use the same filter for both directions (e.g. the orthogonal Daubechies family), but later when we use the DT$\mathbb{C}$WT, we will want to have different horizontal and vertical filters.

Further, note that the only things that need to be saved are the filters, as seen in Algorithm 1.2.2. These are typically only a few floats, giving us a large saving over relying on autograd.

A multiscale DWT can easily be made by calling Algorithm 1.2 multiple times on the lowpass output. Again, no intermediate activations need be saved, giving this implementation almost no memory overhead.

1.4 Fast Calculation of the DT$\mathbb{C}$WT

We have built upon previous implementations of the DT$\mathbb{C}$WT [10]–[12]. The DT$\mathbb{C}$WT gets is name from having two trees, each with their own set of filters, $a = \{h_0^a, h_1^a, g_0^a, g_1^a\}$ and $b = \{h_0^b, h_1^b, g_0^b, g_1^b\}$. In one dimension, this can be done with two DWTs, one using the a filters and the other using the b filters. In two dimensions, we must instead do four multiscale

Algorithm 1.3 2-D DT$\mathbb{C}$WT

function DTCWT(x, J, $mode$)
  load h_0^a, h_1^a, h_0^b, h_1^b
  ll^{aa}, ll^{ab}, ll^{ba}, $ll^{bb} \leftarrow x$
  for $1 \leq j \leq J$ **do**
    ll^{aa}, lh^{aa}, hl^{aa}, $hh^{aa} \leftarrow \mathrm{DWT}_{aa}(ll^{aa}, h_0^a, h_1^a, h_0^a, h_1^a, mode)$
    ll^{ab}, lh^{ab}, hl^{ab}, $hh^{ab} \leftarrow \mathrm{DWT}_{ab}(ll^{ab}, h_0^a, h_1^a, h_0^b, h_1^b, mode)$
    ll^{ba}, lh^{ba}, hl^{ba}, $hh^{ba} \leftarrow \mathrm{DWT}_{ba}(ll^{ba}, h_0^b, h_1^b, h_0^a, h_1^a, mode)$
    ll^{bb}, lh^{bb}, hl^{bb}, $hh^{bb} \leftarrow \mathrm{DWT}_{bb}(ll^{bb}, h_0^b, h_1^b, h_0^b, h_1^b, mode)$
    $yh[j] \leftarrow$ Q2C(
      lh^{aa}, hl^{aa}, hh^{aa},
      lh^{ab}, hl^{ab}, hh^{ab},
      lh^{ba}, hl^{ba}, hh^{ba},
      lh^{bb}, hl^{bb}, hh^{bb})
  end for
  $yl \leftarrow \mathrm{interleave}(ll^{aa}, ll^{ab}, ll^{ba}, ll^{bb})$
  return yl, yh
end function

DWTs. We call each DWT aa, ab, ba and bb, where the first letter indicates which filter to use for the row operations, and the second letter indicates which filter to use for the columns.

The four DWTs give twelve bandpass coefficients at each scale. These are added and subtracted from each other to get the six orientations' real and imaginary components (see Algorithm B.3 for the code for how to do this). The four lowpass coefficients from each scale are used for the next scale DWTs. At the final scale, they are interleaved to get four times the expected lowpass output area expected from a single decimated DWT. See Algorithm 1.3 for the code on how to do the full DT$\mathbb{C}$WT.

A requirement of the DT$\mathbb{C}$WT is the need to use different filters for the first scale to all subsequent scales [13]. We have not shown this in Algorithm 1.3 for simplicity, but it would simply mean we would have to handle the $j = 1$ case separately.

1.5 The Magnitude Operation

Now that we have a forward and backward pass for the DT$\mathbb{C}$WT, the final missing piece is the magnitude operation. If $z = x + jy$, then:

$$r = |z| = \sqrt{x^2 + y^2} \tag{1.5.1}$$

Algorithm 1.4 Magnitude forward and backward steps

```
function MAG.FORWARD(x, y)
    r ← √(x² + y²)
    θ ← arctan2(y, x)                    ▷ arctan2 handles x = 0
    save θ
    return r
end function

function MAG.BACKWARD(Δr)
    load θ
    Δx ← Δr cos θ                        ▷ Reinsert phase
    Δy ← Δr sin θ                        ▷ Reinsert phase
    return Δx, Δy
end function
```

This has two partial derivatives, $\frac{\partial r}{\partial x}$, $\frac{\partial r}{\partial y}$:

$$\frac{\partial r}{\partial x} = \frac{x}{\sqrt{x^2+y^2}} = \frac{x}{r} \tag{1.5.2}$$

$$\frac{\partial r}{\partial y} = \frac{y}{\sqrt{x^2+y^2}} = \frac{y}{r} \tag{1.5.3}$$

Given an input gradient, Δr, the passthrough gradient is:

$$\begin{aligned} \Delta z &= \Delta r \frac{\partial r}{\partial x} + j\Delta r \frac{\partial r}{\partial y} && (1.5.4)\\ &= \Delta r \frac{x}{r} + j\Delta r \frac{y}{r} && (1.5.5)\\ &= \Delta r e^{j\theta} && (1.5.6) \end{aligned}$$

where $\theta = \arctan\frac{y}{x}$. This has a nice interpretation to it as well, as the backwards pass is simply reinserting the discarded phase information. The pseudo-code for this operation is shown in Algorithm 1.4. These partial derivatives are restricted to be in the range $[-1, 1]$ but have a singularity at the origin. In particular:

$$\lim_{x\to 0^-, y\to 0} \frac{\partial r}{\partial x} = -1 \tag{1.5.7}$$

$$\lim_{x\to 0^+, y\to 0} \frac{\partial r}{\partial x} = +1 \tag{1.5.8}$$

$$\lim_{x\to 0, y\to 0^-} \frac{\partial r}{\partial y} = -1 \tag{1.5.9}$$

$$\lim_{x\to 0, y\to 0^+} \frac{\partial r}{\partial y} = +1 \tag{1.5.10}$$

Algorithm 1.5 DT$\mathbb{C}$WT ScatterNet Layer

function DTCWT_SCAT($x,\ J=2,\ M=2$)
  $Z \leftarrow x$
  for $1 \le m \le M$ **do**
    $yl,\ yh \leftarrow$ DT$\mathbb{C}$WT($Z,\ J=1,\ mode=$'symmetric')
    $S \leftarrow$ avg_pool($yl,\ 2$)
    $U \leftarrow$ mag(Re(yh), Im(yh))
    $Z \leftarrow$ concatenate($S,\ U,\ axis=1$) ▷ stack 1 lowpass with 6 magnitudes
  end for
  if $J > M$ **then**
    $Z \leftarrow$ avg_pool($Z,\ 2^{J-M}$)
  end if
  return Z
end function

Rather than using the subgradient method [14], which is commonly used to handle the magnitude operation, we propose to smooth the magnitude operator:

$$r_s = \sqrt{x^2 + y^2 + b^2} - b \tag{1.5.11}$$

This keeps the magnitude near zero for small x, y but does slightly shrink larger values. The gain we get however is a new smoother gradient surface, and we no longer have to worry about dividing by zero issues when calculating gradients. We can choose the size of b as a hyperparameter in optimization. The partial derivatives now become:

$$\frac{\partial r_s}{\partial x} = \frac{x}{\sqrt{x^2 + y^2 + b^2}} = \frac{x}{r_s} \tag{1.5.12}$$

$$\frac{\partial r_s}{\partial y} = \frac{y}{\sqrt{x^2 + y^2 + b^2}} = \frac{y}{r_s} \tag{1.5.13}$$

There is a memory cost associated with this, as we will now need to save both $\frac{\partial r_s}{\partial x}$ and $\frac{\partial r_s}{\partial y}$ as opposed to saving only the phase. Algorithm B.2 has the pseudo-code for the smooth magnitude.

1.6 The DT$\mathbb{C}$WT ScatterNet

Now that we have the DT$\mathbb{C}$WT and the magnitude operation, it is straightforward to get a DT$\mathbb{C}$WT scattering layer, shown in Algorithm 1.5.

For a second order ScatterNet, the first C channels of Z from Algorithm 1.5 are the S_0 coefficients, the next JKC are the S_1 coefficients and the final $\frac{1}{2}(J-1)JK^2C$ channels are the S_2 coefficients.

Table 1.1: **Hyperparameter settings for the** DT$\mathbb{C}$WT **scatternet.**

Hyperparameter	Values
Wavelet	near_sym_a 5,7 tap filters near_sym_b 13,19 tap filters near_sym_b_bp 13,19 tap filters
Padding Scheme	symmetric zero
Magnitude Smoothing b	0 1e-3 1e-2 1e-1

Note that for ease in handling the different sample rates of the lowpass and the bandpass, we have averaged the lowpass over a 2×2 window and downsampled it by 2 in each direction. This slightly affects the higher order coefficients, as the true DT$\mathbb{C}$WT needs the doubly sampled lowpass for the second scale. We noticed little difference in performance from doing the true DT$\mathbb{C}$WT and the decimated one.

1.6.1 DT$\mathbb{C}$WT Hyperparameter Choice

Before comparing to the Morlet based ScatterNet, we can test test different padding schemes, wavelet lengths and magnitude smoothing parameters (see (1.5.11)) for the DT$\mathbb{C}$WT ScatterNet. We test these over a grid of values described in Table 1.1. The different wavelets have different lengths and hence different frequency responses. Additionally, the 'near_sym_b_bp' wavelet is a rotationally symmetric wavelet with diagonal passband brought in by a factor of $\sqrt{2}$, making it have similar $|\boldsymbol{\omega}|$ to the horizontal and vertical bandpasses.

The results of these experiments are shown in Figure 1.2. The choice of options can have a very large impact on the classification accuracy. Symmetric padding appears to work marginally better than zero padding. Surprisingly, the shorter filters (near_sym_a) fare better than their longer counterparts, and bringing in the diagonal subbands (near_sym_b_bp) does not help. Additionally, the smoothing bias may indeed be a good idea for the forward pass as well as aiding the backwards pass, so long as it is less than 0.1.

However, It is difficult to tell just how much 'better' the different filter lengths and padding schemes are, as there are so many other factors to take into consideration when training a CNN. The standard deviation on the results from Figure 1.2 were in the range from 0.1 to 0.4% for the two CIFAR datasets, which is smaller than the 2% range seen from the best option to the worst. However, for the larger Tiny ImageNet dataset, the standard deviations were larger, around 0.2 to 0.6%, which is comparable to the 1% range from best to worst configuration.

We proceed by using the 'near_sym_a' filters, with symmetric padding and a smoothing factor of 0.01, but leave all configurations in our code in [1] as we cannot be sure that these are the best options for all tasks.

1.7 Comparisons

Now that we have the ability to do a DT$\mathbb{C}$WT based scatternet, how does this compare with the original Matlab implementation [2] and the newly developed KyMatIO [5]? Table 1.2 lists the different properties and options of the competing packages.

1.7.1 Speed

We test the speed of the various packages on our reference architecture (see appendix A) on a moderately large input with $128 \times 3 \times 256 \times 256$ pixels. The CPU experiments used all cores available, whereas the GPU experiments ran on a single GPU. We include two permutations of our proposed scatternet with different length filters and different padding schemes. Type A uses the long 'near_sym_b' filters and has symmetric padding, and Type B uses shorter 'near_sym_a' filters and the faster zero padding scheme. We compare it to a Morlet based implementation with $K = 6$ orientations (the same number of orientations as in the DT$\mathbb{C}$WT).

See Table 1.3 for the execution time results. Type A is 7 times faster than the Fourier-based KyMatIO on GPUs and Type B has a 14 times speedup over the Morlet backend.

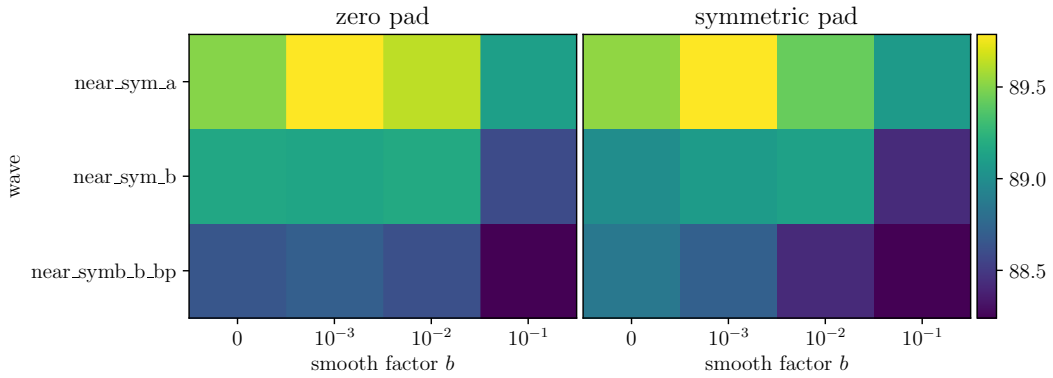
Additionally, when compared with version 0.2 of KyMatIO, the memory footprint of the DT$\mathbb{C}$WT based implementation is only 2% of KyMatIO's, highlighting the importance of being explicit in calculating the backwards pass.

1.7.2 Performance

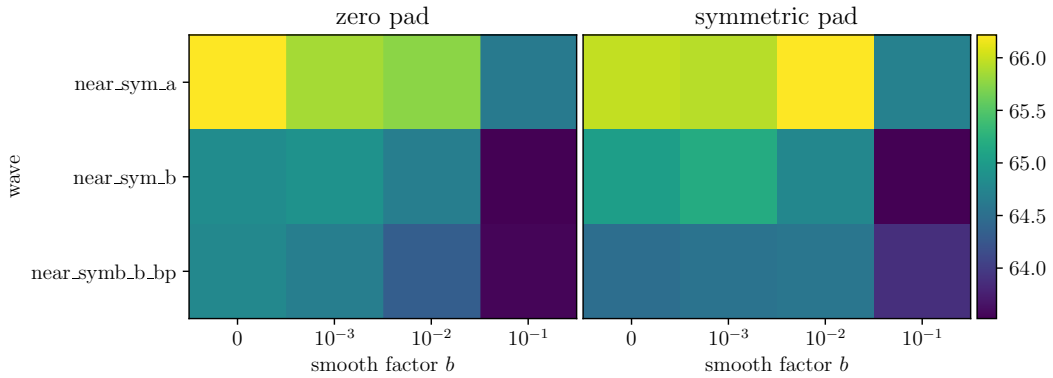
To confirm that changing the ScatterNet core has not impeded the performance of the ScatterNet as a feature extractor, we build a simple Hybrid ScatterNet, similar to [15], [16]. This puts two layers of a scattering transform at the front end of a deep learning network. We use the optimal hyperparameter choices from the previous section, and compare these to Morlet based ScatterNet with 6 and 8 orientations.

We run tests on

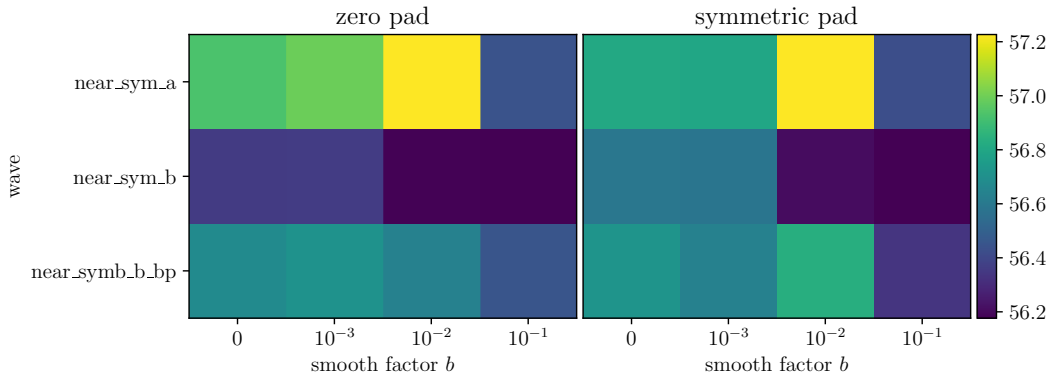
- CIFAR-10: 10 classes, 5000 images per class, 32×32 pixels per image.
- CIFAR-100: 100 classes, 500 images per class, 32×32 pixels per image.
- Tiny ImageNet[17]: 200 classes, 500 images per class, 64×64 pixels per image.



(a) CIFAR-10



(b) CIFAR-100



(c) Tiny ImageNet

Figure 1.2: **Hyperparameter results for the** DT$\mathbb{C}$WT **scatternet on various datasets.** Image showing relative top-1 accuracies (in %) on the given datasets using architecture described in Table 1.1. Each subfigure is a new dataset and therefore has a new colour range (shown on the right). Results are averaged over 3 runs from different starting positions. The choice of options can have a very large impact on the classification accuracy. Symmetric padding is marginally better than zero padding. Surprisingly, the shorter filter (near_sym_a) fares better than its longer counterparts, and bringing in the diagonal subbands (near_sym_b_bp) does not help. Additionally, the smoothing bias may indeed be a good idea for the forward pass as well as aiding the backwards pass, so long as it is less than 0.1.

Table 1.2: **Comparison of properties of different ScatterNet packages.** In particular the wavelet backend used, the number of orientations available, the available boundary extension methods, whether it has GPU support and whether it supports backpropagation.

Package	Backend	Orientations	Boundary Ext.	GPU	Backprop
ScatNetLight[2]	FFT-based	Flexible	Periodic	No	No
KyMatIO[5]	FFT-based	Flexible	Periodic	Yes	Yes
DT$\mathbb{C}$WT Scat	Separable filter banks	6	Flexible	Yes	Yes

Table 1.3: **Comparison of execution time for the forward and backward passes of the competing ScatterNet Implementations.** Tests were run on the reference architecture described in appendix A. The input for these experiments is a batch of images of size $128 \times 3 \times 256 \times 256$ in 4 byte floating precision. We list two different types of options for our scatternet. Type A uses 16 tap filters and has symmetric padding, whereas type B uses 6 tap filters and uses zero padding at the image boundaries. Values are given to 2 significant figures, averaged over 5 runs.

Package	CPU		GPU	
	Fwd (s)	Bwd (s)	Fwd (s)	Bwd (s)
ScatNetLight[2]	> 200.00	n/a	n/a	n/a
KyMatIO[5]	95.0	130.0	1.44	2.5
DT$\mathbb{C}$WT Scat Type A	3.3	3.6	0.21	0.27
DT$\mathbb{C}$WT Scat Type B	2.8	3.2	0.10	0.16

Table 1.4 details the network layout for CIFAR.

For Tiny ImageNet, the images are four times the size, so the output after scattering is 16×16. We add a max pooling layer after conv4, followed by two more convolutional layers conv5 and conv6, before average pooling.

These networks are optimized with stochastic gradient descent with momentum. The initial learning rate is 0.5, momentum is 0.85, batch size $N = 128$ and weight decay is 10^{-4}. For CIFAR-10/CIFAR-100 we scale the learning rate by a factor of 0.2 after 60, 80 and 100 epochs, training for 120 epochs in total. For Tiny ImageNet, the rate change is at 18, 30 and 40 epochs (training for 45 in total). Our experiment code is available at [18].

The results of this experiment are shown in Table 1.5. It is promising to see that the DT$\mathbb{C}$WT based scatternet has not only not sped up, but slightly improved upon the Morlet based ScatterNet as a frontend. Interestingly, both with Morlet and DT$\mathbb{C}$WT wavelets, 6 orientations performed better than 8, despite having fewer parameters in conv1.

1.8 Conclusion

In this chapter we have proposed changing the backend for Scattering transforms from a Morlet wavelet transform to the spatially separable DT$\mathbb{C}$WT. This was originally inspired by the need to speed up the slow Matlab scattering package, as well as to provide GPU accelerated code that could do wavelet transforms as part of a deep learning package.

We have derived the forward and backpropagation functions necessary to do fast and memory efficient DWTs, DT$\mathbb{C}$WTs, and Scattering based on the DT$\mathbb{C}$WT, and have made this code publically available at [1]. We hope that this will reduce some of the barriers we initially faced in using wavelets and Scattering in deep learning.

In parallel with our efforts, the original ScatterNet authors rewrote their package to speed up Scattering. In theory, a spatially separable wavelet transform acting on N pixels has order $\mathcal{O}(N)$ whereas an FFT based implementation has order $\mathcal{O}(N \log N)$. We have shown experimentally that on modern GPUs, the difference is far larger than this, with the DT$\mathbb{C}$WT backend an order of magnitude faster than Fourier-based Morlet implementation [5].

Additionally, we have experimentally verified that using a different complex wavelet core does not have a negative impact on the performance of the ScatterNet as a frontend to Hybrid networks.

Table 1.4: **Hybrid architectures for performance comparison.** Comparison of Morlet based scatternets (Morlet6 and Morlet8) to the DT$\mathbb{C}$WT based scatternet on CIFAR. The output after scattering has $3(K+1)^2$ channels (243 for 8 orientations or 147 for 6 orientations) of spatial size 8×8. This is passed to 4 convolutional layers of width $C = 192$ before being average pooled and fed to a single fully connected classifier. $N_c = 10$ for CIFAR-10 and 100 for CIFAR-100. In the DT$\mathbb{C}$WT architecture, we test different padding schemes and wavelet lengths.

Morlet8	Morlet6	DT$\mathbb{C}$WT
Scat $J = 2,\ K = 8,\ m = 2$ $y \in \mathbb{R}^{243\times8\times8}$	Scat $J = 2,\ K = 6,\ m = 2$ $y \in \mathbb{R}^{147\times8\times8}$	Scat $J = 2,\ K = 6,\ m = 2$ $y \in \mathbb{R}^{147\times8\times8}$
conv1, $w \in \mathbb{R}^{C\times243\times3\times3}$	conv1, $w \in \mathbb{R}^{C\times147\times3\times3}$	
conv2, $w \in \mathbb{R}^{C\times C\times3\times3}$		
conv3, $w \in \mathbb{R}^{2C\times C\times3\times3}$		
conv4, $w \in \mathbb{R}^{2C\times2C\times3\times3}$		
avg pool, 8×8		
fc, $w \in \mathbb{R}^{2C\times N_c}$		

Table 1.5: **Performance comparison for a** DT$\mathbb{C}$WT **based ScatterNet vs Morlet based ScatterNet.** We report top-1 classification accuracy for the 3 listed datasets as well as training time for each model in hours.

Type	CIFAR-10		CIFAR-100		Tiny ImgNet	
	Acc. (%)	Time (h)	Acc. (%)	Time (h)	Acc. (%)	Time (h)
Morlet8	88.6	3.4	65.3	3.4	57.6	5.6
Morlet6	89.1	2.4	65.7	2.4	57.5	4.4
DT$\mathbb{C}$WT	89.8	1.1	66.2	1.1	57.3	2.7

References

[1] F. Cotter, *Pytorch Wavelets*, GitHub fbcotter/pytorch_wavelets, 2018.

[2] E. Oyallon and S. Mallat, "Deep Roto-Translation Scattering for Object Classification", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 2865–2873.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015.

[4] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch", Oct. 2017.

[5] M. Andreux, T. Angles, G. Exarchakis, R. Leonarduzzi, G. Rochette, L. Thiry, J. Zarka, S. Mallat, J. Andén, E. Belilovsky, J. Bruna, V. Lostanlen, M. J. Hirn, E. Oyallon, S. Zhang, C. Cella, and M. Eickenberg, "Kymatio: Scattering Transforms in Python", *arXiv:1812.11214 [cs, eess, stat]*, Dec. 2018. arXiv: 1812.11214 `[cs, eess, stat]`.

[6] W. Sweldens, "The Lifting Scheme: A Construction of Second Generation Wavelets", *SIAM J. Math. Anal.*, vol. 29, no. 2, pp. 511–546, Mar. 1998.

[7] C. Tenllado, J. Setoain, M. Prieto, L. Piñuel, and F. Tirado, "Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting", *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 3, pp. 299–310, Mar. 2008.

[8] V. Galiano, O. Lopez, M. Malumbres, and H. Migallon, "Improving the discrete wavelet transform computation from multicore to GPU-based algorithms", in *Proceedings of the 11th International Conference on Computational and Mathematical Methods in Science and Engineering*, Jun. 2011, pp. 544–555.

[9] Y. A. Ioannou, "Structural Priors in Deep Neural Networks", PhD thesis, Department of Engineering, University of Cambridge, Trumpington Street, Cambridge CB2 1PZ, United Kingdom, Sep. 2017.

[10] N. Kingsbury, *DTCWT*, 2003.

[11] S. Cai, K. Li, and I. Selesnick, *2-D Dual-Tree Wavelet Transform*, Nov. 2011.

[12] R. Wareham, C. Shaffrey, and N. Kingsbury, *Dtcwt*, 2014.

[13] I. W. Selesnick, R. G. Baraniuk, and N. G. Kingsbury, "The dual-tree complex wavelet transform", *Signal Processing Magazine, IEEE*, vol. 22, no. 6, pp. 123–151, 2005.

[14] S. Boyd, L. Xiao, and A. Mutapcic, *The Subgradient Method*, en, 2003.

[15] E. Oyallon, "A Hybrid Network: Scattering and Convnet", 2017.

[16] E. Oyallon, E. Belilovsky, and S. Zagoruyko, "Scaling the Scattering Transform: Deep Hybrid Networks", in *2017 IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017, pp. 5619–5628. arXiv: 1703.08961.

[17] F.-F. Li, "Tiny ImageNet Visual Recognition Challenge", Stanford cs231n: https://tiny-imagenet.herokuapp.com/, 2017.

[18] F. Cotter, *Learnable ScatterNet*, GitHub fbcotter/scatnet_learn, 2019.

[19] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A Research Platform for Distributed Model Selection and Training", *arXiv preprint arXiv:1807.05118*, 2018.

[20] M. Vetterli and J. Kovacevic, *Wavelets and Subband Coding*, 2nd ed., ser. Prentice Hall Signal Processing Series. Prentice Hall PTR, 2007.

Appendix A

Architecture Used for Experiments

The experiments for this thesis were run on a single server with 8 GPUs and 14 CPUs. The GPUs were each NVIDIA GeForce GTX 1080 cards released in May 2016. They each have 8GiB of RAM, 2560 CUDA cores and 320 GB/s memory bandwidth. The CPUs were Intel(R) Xeon(R) E5-2660 models.

At the completion of the project, we were running CUDA 10.0 with cuDNN 7.6 and PyTorch version 1.1.

To do hyperparameter search we used the Tune package [19] which we highly recommend, as it makes running trials in parallel very easy.

A.1 Run Times of some of the Propsed Layers

.

Appendix B

Forward and Backward Algorithms

We have listed some of the forward and backward algorithms here that are not included in the main text for the interested reader. We also have included proofs for the gradients of decimation, interpolation and an entire analysis decomposition.

B.1 Gradients of Sample Rate Changes

Consider 1D decimation and interpolation of a signal X. The results we prove here easily extrapolate to 2D, but for ease we have kept to the 1D case.

Decimation of a signal X by M is defined as:

$$Y[n] = X[Mn] \tag{B.1.1}$$

and interpolation by M as:

$$Y[n] = \begin{cases} X[\frac{n}{M}] & n = Mk,\ k \in \mathbb{Z} \\ 0 & \text{otherwise} \end{cases} \tag{B.1.2}$$

B.1.1 Decimation Gradient

From (B.1.1) the gradient $\frac{\partial Y_n}{\partial X_k}$ is:

$$\frac{\partial Y_n}{\partial X_k} = \begin{cases} 1 & k = Mn \\ 0 & \text{otherwise} \end{cases} \tag{B.1.3}$$

By the chain rule, $\frac{\partial J}{\partial X_k}$ is:

$$\frac{\partial J}{\partial X_k} = \frac{\partial J}{\partial Y_n}\frac{\partial Y_n}{\partial X_k} \tag{B.1.4}$$

$$= \begin{cases} \Delta Y[\frac{k}{M}] & k = Mn \\ 0 & \text{otherwise} \end{cases} \tag{B.1.5}$$

which is interpolating ΔY by M from (B.1.2).

B.1.2 Interpolation Gradient

From (B.1.2) the gradient $\frac{\partial Y_n}{\partial X_k}$ is:

$$\frac{\partial Y_n}{\partial X_k} = \begin{cases} 1 & n = Mk \\ 0 & \text{otherwise} \end{cases} \tag{B.1.6}$$

and then the gradient $\frac{\partial J}{\partial X_k}$ is:

$$\frac{\partial J}{\partial X_k} = \frac{\partial J}{\partial Y_n}\frac{\partial Y_n}{\partial X_k} \tag{B.1.7}$$

$$= \begin{cases} \Delta Y[n] & n = Mk \\ 0 & \text{otherwise} \end{cases} \tag{B.1.8}$$

$$= \Delta Y[Mk] \tag{B.1.9}$$

which is decimation of ΔY by M.

B.2 Gradient of Wavelet Analysis Decomposition

We mention in subsection 1.3.2 that the gradient of a forward DWT with orthogonal wavelets is just the inverse DWT. This easily follows from applying the chain rule and using the gradients of each of the stages of the DWT (convolution becomes correlation, downsampling becomes upsampling). The equivalent backwards pass of Figure 1.1 is shown in Figure B.1.

For an orthogonal wavelet transform, the synthesis filters are the time reverse of the analysis filters [20, Chapter 3]. This means that the blocks $H_0(z^{-1}), H_1(z^{-1})$ can be replaced with $G_0(z), G_1(z)$ respectively, giving the inverse DWT.

B.3 Extra Algorithms

In the text we refer to the 2-D inverse DWT which we have listed in Algorithm B.1, as well as the smooth magnitude operation (Algorithm B.2) and the quadratue to complex operation (Algorithm B.3).

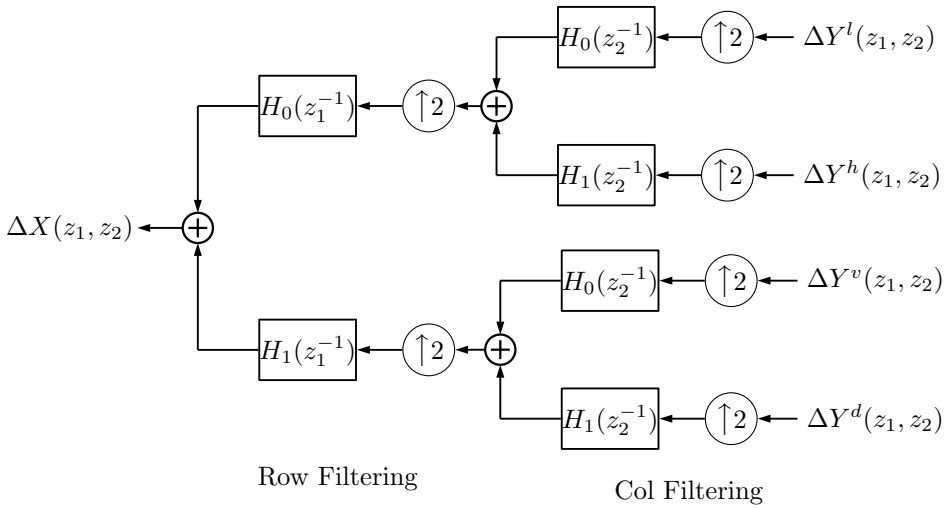


Figure B.1: **Gradient of DWT analysis.** Each block in the forward pass of Figure 1.1 has been swapped for its gradient. The resulting gradient has the same form as the inverse DWT.

Algorithm B.1 2-D Inverse DWT and its gradient

function IDWT.FORWARD($ll,\ lh,\ hl,\ hl,\ g_0^c,\ g_1^c,\ g_0^r,\ g_1^r,\ mode$)
    save $g_0^c,\ g_1^c,\ g_0^r,\ g_1^r,\ mode$ ▷ For the backwards pass
    $lo \leftarrow \text{sfb1d}(ll,\ lh,\ g_0^c,\ g_1^c,\ mode,\ axis = 2)$
    $hi \leftarrow \text{sfb1d}(hl,\ hh,\ g_0^c,\ g_1^c,\ mode,\ axis = 2)$
    $x \leftarrow \text{sfb1d}(lo,\ hi,\ g_0^r,\ g_1^r,\ mode,\ axis = 3)$
    return x
end function

function IDWT.BACKWARD(δy)
    load $g_0^c,\ g_1^c,\ g_0^r,\ g_1^r,\ mode$
    $\Delta lo,\ \Delta hi \leftarrow \text{afb1d}(\delta y,\ g_0^r,\ g_1^r,\ mode,\ axis = 3)$
    $\Delta ll,\ \Delta lh \leftarrow \text{afb1d}(\Delta lo,\ g_0^c,\ g_1^c,\ mode,\ axis = 2)$
    $\Delta hl,\ \Delta hh \leftarrow \text{afb1d}(\Delta hi,\ g_0^c,\ g_1^c,\ mode,\ axis = 2)$
    return $\Delta ll,\ \Delta lh,\ \Delta hl,\ \Delta hh$
end function

Algorithm B.2 Smooth Magnitude

function MAG_SMOOTH.FORWARD($x,\ y,\ b$)
    $b \leftarrow \max(b,\ 0)$
    $r \leftarrow \sqrt{x^2 + y^2 + b^2}$
    $\frac{\partial r}{\partial x} \leftarrow \frac{x}{r}$
    $\frac{\partial r}{\partial y} \leftarrow \frac{y}{r}$
    save $\frac{\partial r}{\partial x},\ \frac{\partial r}{\partial x}$
    return $r - b$
end function

function MAG_SMOOTH.BACKWARD(Δr)
    load $\frac{\partial r}{\partial x},\ \frac{\partial r}{\partial y}$
    $\Delta x \leftarrow \Delta r \frac{\partial r}{\partial x}$
    $\Delta y \leftarrow \Delta r \frac{\partial r}{\partial y}$
    return $\Delta x,\ \Delta y$
end function

Algorithm B.3 Q2C

function Q2C($x,\ y,\ b$)
    $b \leftarrow \max(b,\ 0)$
    $r \leftarrow \sqrt{x^2 + y^2 + b^2}$
    $\frac{\partial r}{\partial x} \leftarrow \frac{x}{r}$
    $\frac{\partial r}{\partial y} \leftarrow \frac{y}{r}$
    save $\frac{\partial r}{\partial x},\ \frac{\partial r}{\partial x}$
    return $r - b$
end function