Chapter 1

A Faster ScatterNet

The drive of this thesis is in exploring if wavelet theory, in particular the DTℂWT, 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 research (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 DTℂWT as its core. At the core of that is an efficient implementation of the DWT. The result is an open source library that provides all three (the DWT, DTℂWT and DTℂWT ScatterNet), available on GitHub as *PyTorch Wavelets* **pytorch_wavelets**.

In parallel with our efforts, the original authors of the ScatterNet have improved their implementation, also building it on PyTorch. My proposed DTℂWT ScatterNet is 15 to 35 times faster than their improved implementation, depending on the padding style and wavelet length, while using less memory.

1.1 The Design Constraints

The original authors implemented their ScatterNet in matlab **oyallon_deep_2015** 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 **abadi_tensorflow:_2015** or Pytorch **paszke_automatic_2017**.
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.

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**andreux__kymatio:__2018**, addressing the above design constraints. The key difference between our proposed package and their improved packages is the use of the DT$\mathbb{C}$WT as the core rather than Morlet wavelets. While the key focus of this chapter is in detailing how we have built a fast, GPU-ready, and deep learning compatible library that can do the DWT, DT$\mathbb{C}$WT, and DT$\mathbb{C}$WT ScatterNet, we also compare the speeds and performance of our package to KyMatIO, as it provides some interesting insights into some of the design choices that can be made with 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 **??**. In particular, 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 $\frac{\partial \mathcal{L}}{\partial x}$, $\frac{\partial \mathcal{L}}{\partial w}$. 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. Similarly 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 one of three types of functions:

1. Gradient-less code - these are lowlevel functions that can be used for both the forward and backward calculations. E.g. Algorithm 1.1. We name these `NG:<name>`, NG for no gradients.

2. Autograd blocks - the modules as shown in **??**. These always have a forward and backward pass, and are named `AG:<name>`. E.g. Algorithm 1.2.

3. Higher level modules - these make use of efficient autograd functions and are named `MOD:<name>`. E.g. Algorithm 1.3.

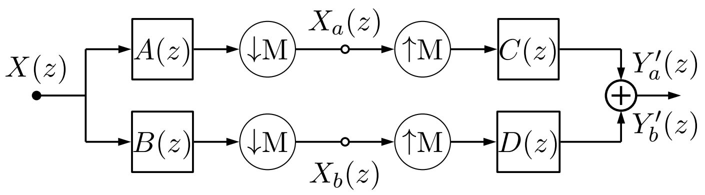


Figure 1.1: **Block Diagram of 2-D DWT.** The components of a filter bank DWT in two dimensions.

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. For a fast implementation of the DT$\mathbb{C}$WT we need a fast implementation of the DWT. Later in our work will we 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.

There has been much research into the best way to do the DWT on a GPU, in particular comparing the speed of Lifting **sweldens_lifting_1998**, or second generation wavelets, to the direct convolutional methods. **tenllado_parallel_2008**, **galiano_improving_2011** 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.

Writing a DWT in lowlevel calls is not theoretically difficult to do. There are only a few things to be wary of. Firstly, a ‘convolution’ in most deep learning packages is in fact a correlation. This does not make any difference when learning but when using preset filters, as we want to do, it means that we must take care to reverse the filters beforehand. Secondly, the automatic differentiation will naturally save activations after every step in the DWT (e.g. after row filtering, downsampling and column filtering). This is for the calculation of the backwards pass. We do not need to save these intermediate activations and we can save a lot of memory by overwriting the automatic differentiation logic and defining our own backwards pass.

1.3.1 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 loss with respect to the input.

Additionally, if we decimate by a factor of two on the forwards pass, the equivalent backwards pass is interpolating by a factor of two (this is easy to convince yourself with pen and paper).

Figure 1.1 shows the block diagram for performing the forward pass of a DWT. 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). 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 is simply 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 simply be done by calling the inverse wavelet transform on the wavelet coefficient gradients.

1.3.2 The Forward and Backward Algorithms

Let us start by giving generic names to the above mentioned primitives. We call a convolution followed by downsample `conv2d_down`[1]. As mentioned earlier, in all deep learning packages, this function's name is misleading as it in fact does correlation. As such we need to be careful to reverse the filters with `flip` before calling it. We call convolution followed by upsampling `conv2d_up`[2]. Confusingly, this function does in fact do true convolution, so we do not need to reverse any filters.

These functions in turn call the cuDNN lowlevel fucntions which can only support zero padding. If another padding type is desired, it must be done beforehand with a padding function `pad`.

1.3.2.1 The Input

In all the work in the following chapters, we would like to work on four dimensional arrays. The first dimension represents a minibatch of N images; the second is the number of channels C each image has. For a colour image, $C = 3$, but this often grows deeper in the network. Finally, the last two dimensions are the spatial dimensions, of size $H \times W$.

[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

[2]Similarly, this is done with `torch.nn.functional.conv_transpose2d` with the stride parameter set to 2 in PyTorch

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

```
1:  function NG:AFB1D($x$, $h_0$, $h_1$, $mode$, $axis$)
2:      $h_0, h_1 \leftarrow \text{flip}(h_0), \text{flip}(h_1)$        ▷ flip the filters for conv2d_down
3:      if axis == -1 then
4:          $h_0, h_1 \leftarrow h_0^t, h_1^t$        ▷ row filtering
5:      end if
6:      $p \leftarrow \lfloor(\text{len}(x)+\text{len}(h_0)-1)/2\rfloor$        ▷ calculate output size
7:      $b \leftarrow \lfloor p/2 \rfloor$        ▷ calculate pad size before
8:      $a \leftarrow \lceil p/2 \rceil$        ▷ calculate pad size after
9:      $x \leftarrow \text{pad}(x, b, a, mode)$        ▷ pre pad the signal with selected mode
10:     $lo \leftarrow$ conv2d_down($x, h_0$)
11:     $hi \leftarrow$ conv2d_down($x, h_1$)
12:     return $lo$, $hi$
13: end function

1:  function NG:SFB1D($lo$, $hi$, $g_0$, $g_1$, $mode$, $axis$)
2:      if axis == -1 then
3:          $g_0, g_1 \leftarrow g_0^t, g_1^t$        ▷ row filtering
4:      end if
5:      $p \leftarrow \text{len}(g_0) - 2$        ▷ calculate output size
6:      $lo \leftarrow \text{pad}(lo, p, p, \text{“}zero\text{”})$        ▷ pre pad the signal with zeros
7:      $hi \leftarrow \text{pad}(hi, p, p, \text{“}zero\text{”})$        ▷ pre pad the signal with zeros
8:      $x \leftarrow$ conv2d_up($lo, g_0$) + conv2d_up($hi, g_1$)
9:      return $x$
10: end function
```

1.3.2.2 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 are described in Algorithm 1.1. We do not need to define backpropagation functions for the `afb1d` and `sfb1d` functions as they are each others backwards step.

1.3.2.3 2-D Transforms and their gradients

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

The inverse transform logic is moved to the appendix Algorithm B.1. An interesting result is the similarity between the two transforms' forward and backward stages. Further,

Algorithm 1.2 2-D DWT and its gradient

```
1: function AG:DWT:FWD($x$, $h_0^c$, $h_1^c$, $h_0^r$,$h_1^r$, $mode$)
2:     save $h_0^c$, $h_1^c$, $h_0^r$, $h_1^r$, $mode$                    ▷ For the backwards pass
3:     $lo$, $hi \leftarrow$ afb1d($x$, $h_0^r$, $h_1^r$, $mode$, $axis = -1$)              ▷ row filter
4:     $ll$, $lh \leftarrow$ afb1d($lo$, $h_0^c$, $h_1^c$, $mode$, $axis = -1$)          ▷ column filter
5:     $hl$, $hh \leftarrow$ afb1d($hi$, $h_0^c$, $h_1^c$, $mode$, $axis = -1$)          ▷ column filter
6:     return $ll$, $lh$, $hl$, $hh$
7: end function

1: function AG:DWT:BWD($\Delta ll$, $\Delta lh$, $\Delta hl$, $\Delta hh$)
2:     load $h_0^c$, $h_1^c$, $h_0^r$, $h_1^r$, $mode$
3:     $h_0^c$, $h_1^c \leftarrow$ flip($h_0^c$), flip($h_1^c$)                  ▷ flip the filters as in (1.3.1)
4:     $h_0^r$, $h_1^r \leftarrow$ flip($h_0^r$), flip($h_1^r$)
5:     $\Delta lo \leftarrow$ sfb1d($\Delta ll$, $\Delta lh$, $h_0^c$, $h_1^c$, $mode$, $axis = -2$)
6:     $\Delta hi \leftarrow$ sfb1d($\Delta hl$, $\Delta hh$, $h_0^c$, $h_1^c$, $mode$, $axis = -2$)
7:     $\Delta x \leftarrow$ sfb1d($\Delta lo$, $\Delta hi$, $h_0^r$, $h_1^r$, $mode$, $axis = -1$)
8:     return $\Delta x$
9: end function
```

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 (and IDWT) can easily be made by calling Algorithm 1.2 (Algorithm B.1) multiple times on the lowpass output (reconstructed image). Again, no intermediate activations need be saved, giving this implementation almost no memory overhead.

1.4 Fast Calculation of the DTℂWT

We have built upon previous implementations of the DTℂWT, in particular **kingsbury__dtcwt__2003**, **cai__2-d__2011**, **wareham__dtcwt__2014**. The DTℂWT gets is name from having two sets of filters, a and b. In two dimensions, we do four multiscale DWTs, called aa, ab, ba and bb, where the pair of letters indicates which set of wavelets is used for the row and column filtering. The twelve bandpass coefficients at each scale are added and subtracted from each other to get the six orientations' real and imaginary components Algorithm B.3. The four lowpass coefficients from each scale can be used for the next scale DWTs. At the final scale, they can be interleaved to get four times the expected decimated lowpass output area.

A requirement of the DTℂWT is the need to use different filters for the first scale to all subsequent scales **selesnick__dual-tree__2005**. We have not shown this in Algorithm 1.3 for simplicity, but it would simply mean we would have to handle the $j = 0$ case separately.

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

```
function MOD: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 $0 \leq j < J$ do
        $ll^{aa}$, $lh^{aa}$, $hl^{aa}$, $hh^{aa} \leftarrow$ AG : DWT($ll^{aa}$, $h_0^a$, $h_1^a$, $h_0^a$, $h_1^a$, $mode$)
        $ll^{ab}$, $lh^{ab}$, $hl^{ab}$, $hh^{ab} \leftarrow$ AG : DWT($ll^{ab}$, $h_0^a$, $h_1^a$, $h_0^b$, $h_1^b$, $mode$)
        $ll^{ba}$, $lh^{ba}$, $hl^{ba}$, $hh^{ba} \leftarrow$ AG : DWT($ll^{ba}$, $h_0^b$, $h_1^b$, $h_0^a$, $h_1^a$, $mode$)
        $ll^{bb}$, $lh^{bb}$, $hl^{bb}$, $hh^{bb} \leftarrow$ AG : DWT($ll^{bb}$, $h_0^b$, $h_1^b$, $h_0^b$, $h_1^b$, $mode$)
        $yh[j] \leftarrow$ 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
    $yl \leftarrow$ interleave($ll^{aa}$, $ll^{ab}$, $ll^{ba}$, $ll^{bb}$)
    return $yl$, $yh$
end function
```

1.5 Changing the ScatterNet Core

Now that we have a forward and backward pass for the DT$\mathbb{C}$WT, the final missing piece is the magnitude operation. Again, it is not difficult to calculate the gradients given the direct form, but we must be careful about their size. If $z = x + jy$, then:

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

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

$$\begin{aligned} \frac{\partial r}{\partial x} &= \frac{x}{\sqrt{x^2+y^2}} = \frac{x}{r} \qquad (1.5.2) \\ \frac{\partial r}{\partial y} &= \frac{y}{\sqrt{x^2+y^2}} = \frac{y}{r} \qquad (1.5.3) \end{aligned}$$

These partial derivatives are restricted to be in the range $[-1, 1]$ but have a singularity at the origin. In particular:

$$\lim_{x\to 0^-, y\to 0} \frac{\partial r}{\partial x} = -1 \tag{1.5.4}$$

$$\lim_{x\to 0^+, y\to 0} \frac{\partial r}{\partial x} = +1 \tag{1.5.5}$$

$$\lim_{x\to 0, y\to 0^-} \frac{\partial r}{\partial y} = -1 \tag{1.5.6}$$

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

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

$$\Delta z = \Delta r \frac{\partial r}{\partial x} + j \Delta r \frac{\partial r}{\partial y} \tag{1.5.8}$$

$$= \Delta r \frac{x}{r} + j \Delta r \frac{y}{r} \tag{1.5.9}$$

$$= \Delta r e^{j\theta} \tag{1.5.10}$$

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 very rapidly varying around 0 and the second derivatives go to infinity at the origin. This is not a feature commonly seen with other nonlinearities such as the tanh and sigmoid but it is seen with the ReLU. Small changes in the input can cause large changes in the propagated gradients. The bounded nature of the first derivative somewhat restricts the impact of possible problems so long as our optimizer does not use higher order derivatives (this is commonly the case). Nonetheless, we propose to slightly 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. We can then choose the size of b as a hyperparameter in optimization. The partial derivatives now become:

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

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

Algorithm 1.4 Magnitude forward and backward steps

```
1: function AG:MAG:FWD($x$, $y$)
2:     $r \leftarrow \sqrt{x^2 + y^2}$
3:     $\theta \leftarrow \arctan 2(y,\ x)$                    ▷ arctan2 handles $x = 0$
4:     save $\theta$
5:     return $r$
6: end function

1: function AG:MAG:BWD($\Delta r$)
2:     load $\theta$
3:     $\Delta x \leftarrow \Delta r \cos\theta$                    ▷ Reinsert phase
4:     $\Delta y \leftarrow \Delta r \sin\theta$                    ▷ Reinsert phase
5:     return $\Delta x,\ \Delta y$
6: end function
```

Algorithm 1.5 DTℂWT **ScatterNet Layer.** High level block using the above autograd functions to calculate a first order scattering

```
1: function MOD:DTCWT_SCAT($x,\ J = 2,\ M = 2$)
2:     $Z \leftarrow x$
3:     for $0 \le m < M$ do
4:         $yl,\ yh \leftarrow$ DTℂWT($Z,\ J = 1,\ mode =$ 'symmetric')
5:         $S \leftarrow$ avg_pool($yl$, 2)
6:         $U \leftarrow$ mag(Re($yh$), Im($yh$))
7:         $Z \leftarrow$ concatenate($S,\ U,\ axis = 1$)        ▷ stack 1 lowpass with 6 magnitudes
8:     end for
9:     if $J > M$ then
10:        $Z \leftarrow$ avg_pool($Z,\ 2^{J-M}$)
11:    end if
12:    return $Z$
13: end function
```

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 this.

Now that we have the DTℂWT and the magnitude operation, it is straightforward to get a DTℂWT scattering layer, shown in Algorithm 1.5. To get a multilayer scatternet, we can call the same function again on Z, which would give S_0, S_1 and U_2 and so on for higher orders.

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 DTℂWT needs the doubly sampled lowpass for the second scale. We noticed little difference in performance from doing the true DTℂWT and the decimated one.

Table 1.1: **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	B
ScatNetLight**oyallon__deep__2015**	FFT-based	Flexible	Periodic	No	N
KyMatIO**andreux__kymatio:__2018**	FFT-based	Flexible	Periodic	Yes	Y
DTℂWT Scat	Separable filter banks	6	Flexible	Yes	Y

1.6 Comparisons

Now that we have the ability to do a DTℂWT based scatternet, how does this compare with the original matlab implementation **oyallon__deep__2015** and the newly developed KyMatIO **andreux__kymatio:__2018**? Table 1.1 lists the different properties and options of the competing packages.

1.6.1 Speed and Memory Use

Table 1.2 lists the speed of the various transforms as tested on our reference architecture Appendix A. 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 long filters and uses symmetric padding, and is $15\times$ faster than the Fourier-based KyMatIO. Type B uses shorter filters and the cheaper zero padding scheme, and achieves a $35\times$ speedup over the Morlet backend. Additionally, when compared with version 0.2 of KyMatIO, the DTℂWT based implementation uses 2% of the memory for saving activations for the backwards pass, highlighting the importance of defining the custom backpropagation steps from section 1.3–section 1.5.

1.6.2 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 **oyallon__hybrid__2017**, **oyallon__scaling__2017**. This puts two layers of a scattering transform at the front end of a deep learning network. In addition to comparing our DTℂWT based scatternet to the Morlet based one, we also test using different wavelets, padding schemes and biases for the magnitude operation. 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.

Table 1.2: **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**oyallon_deep_2015**	> 200.00	n/a	n/a	n/a
KyMatIO**andreux_kymatio:_2018**	95.00	130.00	3.50	4.50
DTℂWT Scat Type A	8.00	9.30	0.23	0.29
DTℂWT Scat Type B	3.20	4.80	0.11	0.06

- Tiny ImageNet**li_tiny_nodate**: 200 classes, 500 images per class, 64×64 pixels per image.

Table 1.3 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 https://github.com/fbcotter/scatnet_learn.

1.6.2.1 DTℂWT **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 DTℂWT Scatter-Net. We test these over a grid of values described in Table 1.4. 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 **finish**.

The results of these experiments are shown in Figure 1.2. Interestingly, for all three datasets the shorter wavelet outperformed the longer wavelets.

Table 1.3: **Hybrid architectures for performance comparison.** Comparison of Morlet based scatternets (Morlet6 and Morlet8) to the DTℂWT 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 DTℂWT architecture, we test different padding schemes and wavelet lengths.

Morlet8	Morlet6	DTℂWT
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, $w \in \mathbb{R}^{2C\times N_c}$	

1.6.2.2 Results

We use the optimal hyperparameter choices from the previous section, and compare these to morlet based ScatterNet with 6 and 8 orientations. The results of this experiment are shown in Table 1.5. It is promising to see that the DTℂWT based scatternet has not only not sped up, but slightly improved upon the Morlet based ScatterNet as a frontend. Interestingly, both with Morlet and DTℂWT wavelets, 6 orientations performed better than 8, despite having fewer parameters in conv1.

1.7 Conclusion

In this chapter we have proposed changing the backend for Scattering transforms from a Morlet wavelet transform to the spatially separable DTℂWT. 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, DTℂWTs, and Scattering based on the DTℂWT, and have made this code publically available at **cotter_pytorch_2018**. We hope that this will reduce some of the barriers we initially faced in using wavelets and Scattering in deep learning.

Table 1.4: **Hyperparameter settings for the** DTℂWT **scatternet.**

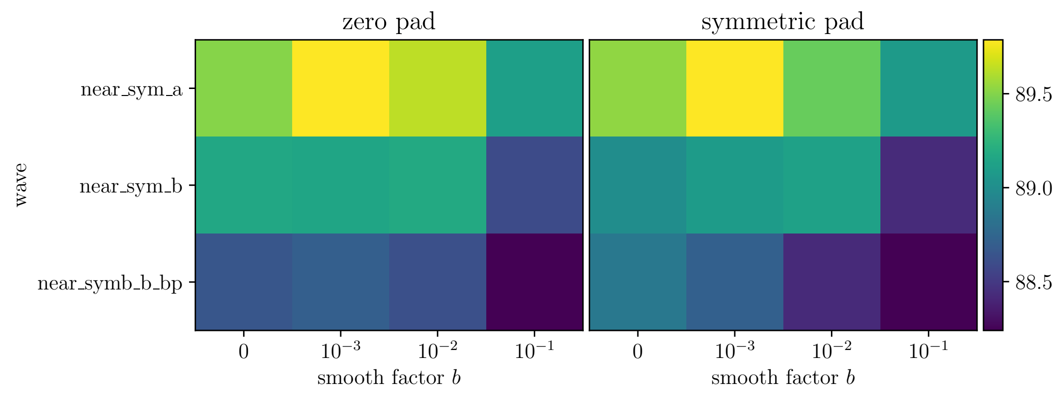
Hyperparameter	Values
Wavelet	near_sym_a 5,7 tap filters, near_sym_b 13,19 tap filters, near_sym_b_bp 13,19 tap filters
Padding Scheme	symmetric zero
Magnitude Smoothing b	0 1e-3 1e-2 1e-1

Table 1.5: **Performance comparison for a** DTℂWT **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.

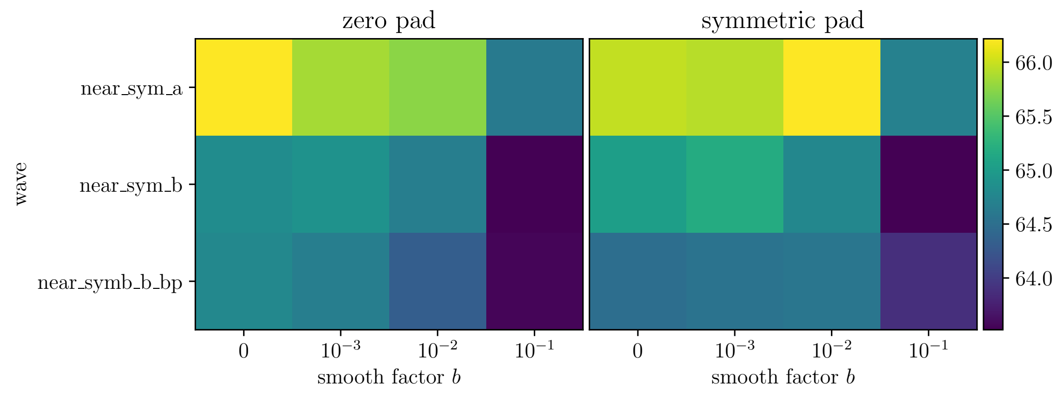
Type	CIFAR-10		CIFAR-100		Tiny ImgNet	
	Acc. (%)	Time (h)	Acc. (%)	Time (h)	Acc. (%)	Time (h)
Morlet8	88.6	3.4	65.3	3.4	57.6	5.6
Morlet6	89.1	2.4	65.7	2.4	57.5	4.4
DTℂWT	89.8	1.1	66.2	1.1	57.3	2.7

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 DTℂWT backend an order of magnitude faster than Fourier-based Morlet implementation **andreux_kymatio:_2018**.

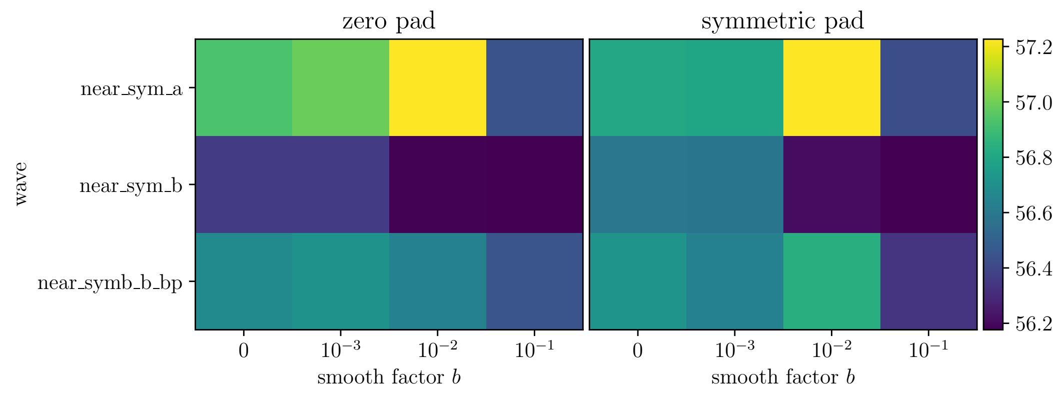
Additionally, we have experimentally verified that using a different complex wavelet core does not have a negative impact on the performance of the ScatterNet as a frontend to Hybrid networks.



(a) CIFAR-10



(b) CIFAR-100



(c) Tiny ImageNet

Figure 1.2: **Hyperparameter results for the** DTℂWT **scatternet on various datasets.** Image showing relative top-1 accuracies (in %) on the given datasets using architecture described in Table 1.4. 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. Surprisingly, 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.

Appendix A

Architecture Used for Experiments

Something

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.

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

```
1: function AG:IDWT:FWD($ll$, $lh$, $hl$, $hl$, $g_0$, $g_1$, $mode$)
2:     save $g_0$, $g_1$, $mode$                                  ▷ For the backwards pass
3:     $lo \leftarrow \text{sfb1d}(ll,\ lh,\ g_0,\ g_1,\ mode,\ axis = -2)$
4:     $hi \leftarrow \text{sfb1d}(hl,\ hh,\ g_0,\ g_1,\ mode,\ axis = -2)$
5:     $x \leftarrow \text{sfb1d}(lo,\ hi,\ g_0,\ g_1,\ mode,\ axis = -1)$
6:     return $x$
7: end function

1: function AG:IDWT:BWD($\delta y$)
2:     load $g_0$, $g_1$, $mode$
3:     $g_0,\ g_1 \leftarrow \text{flip}(g_0),\ \text{flip}(g_1)$              ▷ flip the filters as in (1.3.1)
4:     $\Delta lo,\ \Delta hi \leftarrow \text{afb1d}(\delta y,\ g_0,\ g_1,\ mode,\ axis = -2)$
5:     $\Delta ll,\ \Delta lh \leftarrow \text{afb1d}(\Delta lo,\ g_0,\ g_1,\ mode,\ axis = -1)$
6:     $\Delta hl,\ \Delta hh \leftarrow \text{afb1d}(\Delta hi,\ g_0,\ g_1,\ mode,\ axis = -1)$
7:     return $\Delta ll,\ \Delta lh,\ \Delta hl,\ \Delta hh$
8: end function
```

Algorithm B.2 Smooth Magnitude

```
1: function AG:MAG_SMOOTH:FWD($x$, $y$, $b$)
2:     $b \leftarrow \max(b,\ 0)$
3:     $r \leftarrow \sqrt{x^2 + y^2 + b^2}$
4:     $\frac{\partial r}{\partial x} \leftarrow \frac{x}{r}$
5:     $\frac{\partial r}{\partial y} \leftarrow \frac{y}{r}$
6:     save $\frac{\partial r}{\partial x},\ \frac{\partial r}{\partial x}$
7:     return $r - b$
8: end function

1: function AG:MAG_SMOOTH:BWD($\Delta r$)
2:     load $\frac{\partial r}{\partial x},\ \frac{\partial r}{\partial y}$
3:     $\Delta x \leftarrow \Delta r \frac{\partial r}{\partial x}$
4:     $\Delta y \leftarrow \Delta r \frac{\partial r}{\partial y}$
5:     return $\Delta x,\ \Delta y$
6: end function
```

Algorithm B.3 Q2C

```
1: function MOD:Q2C($x$, $y$, $b$)
2:     $b \leftarrow \max(b,\ 0)$
3:     $r \leftarrow \sqrt{x^2 + y^2 + b^2}$
4:     $\frac{\partial r}{\partial x} \leftarrow \frac{x}{r}$
5:     $\frac{\partial r}{\partial y} \leftarrow \frac{y}{r}$
6:     save $\frac{\partial r}{\partial x},\ \frac{\partial r}{\partial x}$
7:     return $r - b$
8: end function
```