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 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 are 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 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 I have built a fast ScatterNet implementation in PyTorch with the DT$\mathbb{C}$WT 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, available on GitHub as *PyTorch Wavelets*.

In parallel with my efforts, the original authors of the ScatterNet have improved their implementation, also building it on PyTorch. My proposed DT$\mathbb{C}$WT ScatterNet is $15-35$x 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 using a Fourier 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 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 or Pytorch.
3. Load the scattered data from disk and train on it.

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

- It was slow and needed to be run on CPUs.
- It was 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 could not easily do preprocessing techniques like random shifts and flips, as each of these would change the scattered data.
- The scattered features were often larger than the original images, and required you to store entire datasets twice (or more) times.
- The features were fixed and could only be used as a front end to any deep learning system.

To address these shortcomings, all of the above limitations became 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 could be 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 could 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 could do Scattering in PyTorch **kymatio** addressing the above design constraints. The key difference between our proposed and their improved packages is the use of the DT$\mathbb{C}$WT as the core (their software still uses the original Morlet wavelet design).

As part of our explanation we give pseudo code for the forward and backward passes of all our key layers. This will help us later when we want to build on simpler designs.

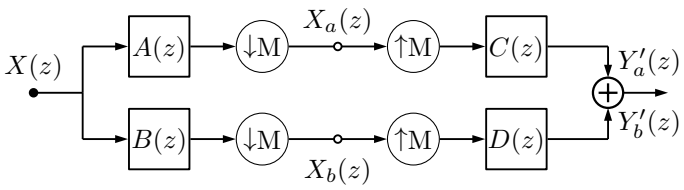


Figure 1.1: **Block Diagram of 2-D DWT.** The components of a filter bank DWT in two dimensions.

1.2 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.

Writing a DWT in PyTorch 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.

1.2.1 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 $Y(z) = H(z)X(z)$:

$$\Delta X(z) = H(z^{-1})\Delta Y(z) \tag{1.2.1}$$

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, PyTorch has 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.

1.2.2 A brief description of autograd

1.2.3 The Forward and Backward Algorithms

For clarity and repeatability, we give pseudocode for all the core operations developed in *PyTorch Wavelets*. By the end of this thesis, it should be clear how every attempted method has been implemented.

Let us start by giving generic names to the above mentioned primitives. In PyTorch, convolution followed by downsampling is done with a call to `torch.nn.functional.conv2d` with the stride parameter set to 2. There are similar such functions for other deep learning packages; let us refer to them all as `conv2d_down`. As mentioned earlier, in all such packages, this function's name is misleading as it in fact does correlation. As such we need to be careful to reverse the filters before calling it (`flip`). Convolution followed by upsampling can be done with a call to `torch.nn.functional.conv_transpose2d` with the stride parameter set to 2; let us refer to this notional function as `conv2d_up`. 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`.

1.2.3.1 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 $H \times W$.

1.2.3.2 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 steps 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.

1.2.3.3 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. 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, 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.

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

function AFB1D(x, h_0, h_1, *mode*, *axis*)
  $h_0,\ h_1 \leftarrow \text{flip}(h_0),\ \text{flip}(h_1)$ ▷ flip the filters for `conv2d_down`
  if axis == -1 **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
  $x \leftarrow \text{pad}(x,\ b,\ a,\ mode)$ ▷ 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 == -1 **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
  $lo \leftarrow \text{pad}(lo,\ p,\ p,\ \text{“}zero\text{”})$ ▷ pre pad the signal with zeros
  $hi \leftarrow \text{pad}(hi,\ p,\ p,\ \text{“}zero\text{”})$ ▷ pre pad the signal with zeros
  $x \leftarrow \texttt{conv2d\_up}(lo,\ g_0) + \texttt{conv2d\_up}(hi,\ g_1)$
  return x
end function

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

1.4 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:

$$r = |z| = \sqrt{x^2 + y^2} \tag{1.4.1}$$

This has two partial derivatives, $\frac{\partial z}{\partial x}$, $dydxzy$:

$$\frac{\partial r}{\partial x} = \frac{x}{\sqrt{x^2 + y^2}} = \frac{x}{r} \tag{1.4.2}$$

$$\frac{\partial r}{\partial y} = \frac{y}{\sqrt{x^2 + y^2}} = \frac{y}{r} \tag{1.4.3}$$

Except for the singularity at the origin, these partial derivatives are restricted to be in the range $[-1, 1]$. Also note that the complex magnitude is convex in x and y as:

Algorithm 1.2 2-D DWT and its gradient

function DWT(x, h_0, h_1, $mode$)
  save h_0, h_1, $mode$ ▷ For the backwards pass
  lo, $hi \leftarrow$ afb1d(x, h_0, h_1, $mode$, $axis = -2$) ▷ column filter
  ll, $lh \leftarrow$ afb1d(lo, h_0, h_1, $mode$, $axis = -1$) ▷ row filter
  hl, $hh \leftarrow$ afb1d(hi, h_0, h_1, $mode$, $axis = -1$) ▷ row filter
  return ll, lh, hl, hh
end function

function DWT_BACKPROP(Δll, Δlh, Δhl, Δhh)
  load h_0, h_1, $mode$
  h_0, $h_1 \leftarrow$ flip(h_0), flip(h_1) ▷ flip the filters as in (1.2.1)
  $\Delta lo \leftarrow$ sfb1d(Δll, Δlh, h_0, h_1, $mode$, $axis = -2$)
  $\Delta hi \leftarrow$ sfb1d(Δhl, Δhh, h_0, h_1, $mode$, $axis = -2$)
  $\Delta x \leftarrow$ sfb1d(Δlo, Δhi, h_0, h_1, $mode$, $axis = -1$)
  return Δx
end function

$$\nabla^2 r(x,y) = \frac{1}{r^3}\begin{bmatrix} y^2 & -xy \\ -xy & x^2 \end{bmatrix} = \frac{1}{r^3}\begin{bmatrix} y \\ -x \end{bmatrix}\begin{bmatrix} y & -x \end{bmatrix} \geq 0 \tag{1.4.4}$$

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

$$\begin{aligned} \Delta z &= \Delta r \frac{\partial r}{\partial x} + j\Delta r \frac{\partial r}{\partial y} && (1.4.5)\\ &= \Delta r \frac{x}{r} + j\Delta r \frac{y}{r} && (1.4.6)\\ &= \Delta r e^{j\theta} && (1.4.7) \end{aligned}$$

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 1.3.

These partial derivatives are variable around 0, in particular the second derivative goes to infinity at the origin **Show a plot of this**. This is not a feature commonly seen with other nonlinearities such as the tanh, sigmoid and ReLU, however it is not necessarily a bad thing. The bounded nature of the first derivative means we will not have any 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:

$$r_s = \sqrt{x^2 + y^2 + b^2} - b \tag{1.4.8}$$

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 **Plot this**. We can then increase/decrease

Algorithm 1.3 Magnitude forward and backward steps

function MAG($x,\ y$)
  $r \leftarrow \sqrt{x^2+y^2}$
  $\theta \leftarrow \arctan 2(y,\ x)$ ▷ $\arctan 2$ handles $x=0$
  save θ
  return r
end function

function MAG_BACKPROP(Δr)
  load θ
  $\Delta x \leftarrow \Delta r \cos\theta$ ▷ Reinsert phase
  $\Delta y \leftarrow \Delta r \sin\theta$ ▷ Reinsert phase
  return $\Delta x,\ \Delta y$
end function

Algorithm 1.4 DT$\mathbb{C}$WT ScatterNet Layer

function DTCWT_SCAT(x)
  $yl,\ yh \leftarrow \text{DT}\mathbb{C}\text{WT}(x)$
  $S_0 \leftarrow \text{avg\_pool}(yl, 2)$ ▷ the lowpass is double the sample size of the bandpass
  $U_1 \leftarrow \text{mag}(yh.real,\ yh.imag)$
  $Z \leftarrow \text{concatenate}(S_0,\ U_1)$ ▷ stack 1 lowpass with 6 magnitudes
  return Z
end function

the size of b as a hyperparameter in optimization. The partial derivatives now become:

$$\frac{\partial r_s}{\partial x} = \frac{x}{\sqrt{x^2+y^2+b^2}} = \frac{x}{r_s} \tag{1.4.9}$$

$$\frac{\partial r_s}{\partial y} = \frac{y}{\sqrt{x^2+y^2+b^2}} = \frac{y}{r_s} \tag{1.4.10}$$

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 DT$\mathbb{C}$WT and the magnitude operation, it is straightforward to get a DT$\mathbb{C}$WT scattering layer, shown in Algorithm 1.4. 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. 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.

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	Backprop
ScatNetLight[1]	Fourier-based Morlet wavelets	Flexible	Periodic	No	No
KyMatIO[2]	Fourier-based Morlet wavelets	Flexible	Periodic	Yes	Yes
DT$\mathbb{C}$WT Scat	Separable & spatial DT$\mathbb{C}$WT	6	Flexible	Yes	Yes

Table 1.2: **Comparison of speed for the forward and backward passes of the competing ScatterNet Implementations.** Tests were run on the reference architecture described in 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[1]	> 200.00	n/a	n/a	n/a
KyMatIO[2]	95.00	130.00	3.50	4.50
DT$\mathbb{C}$WT Scat Type A	8.00	9.30	0.23	0.29
DT$\mathbb{C}$WT Scat Type B	3.20	4.80	0.11	0.06

1.5 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 [1] and the newly developed KyMatIO [2]? Table 1.1 lists the different properties and options of the competing packages.

1.5.1 Speed and Memory Use

Table 1.2 lists the speed of the various transforms as tested on our reference architecture 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, as of version 0.2 of KyMatIO, the DT$\mathbb{C}$WT 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 1.2–section 1.4.

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 $J = 2,\ K = 8,\ m = 2$ $y \in \mathbb{R}^{243 \times 8 \times 8}$	Scat $J = 2,\ K = 6,\ m = 2$ $y \in \mathbb{R}^{147 \times 8 \times 8}$	Scat $J = 2,\ K = 6,\ m = 2$ $y \in \mathbb{R}^{147 \times 8 \times 8}$
conv1, $w \in \mathbb{R}^{C \times 243 \times 3 \times 3}$	conv1, $w \in \mathbb{R}^{C \times 147 \times 3 \times 3}$	
conv2, $w \in \mathbb{R}^{C \times C \times 3 \times 3}$		
conv3, $w \in \mathbb{R}^{2C \times C \times 3 \times 3}$		
conv4, $w \in \mathbb{R}^{2C \times 2C \times 3 \times 3}$		
avg pool, 8×8		
fc, $w \in \mathbb{R}^{2C \times N_c}$		

1.5.2 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 [3], [4]. 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.
- Tiny ImageNet[5]: 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.

1.5.2.1 Hyperparameter Choice

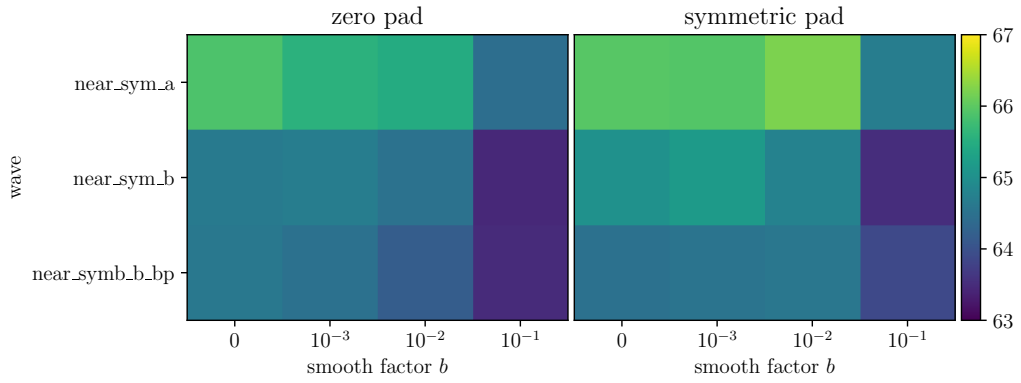
Before comparing to the Morlet based ScatterNet, we can test test different padding schemes, wavelet lengths and magnitude smoothing parameters (see (1.4.8)) for the DT$\mathbb{C}$WT ScatterNet. We test these over a grid of values described in Table 1.4. The different wavelets have different lengths 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 **??**.

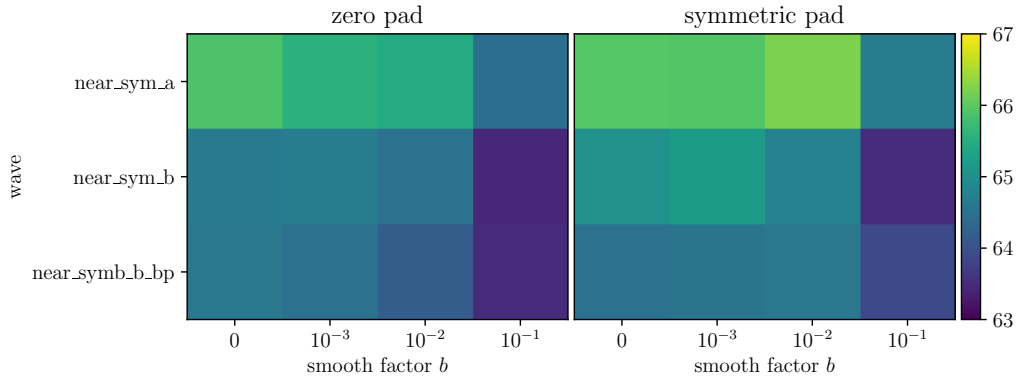
Table 1.4: **Hyperparameter settings for the** DT$\mathbb{C}$WT **scatternet.** The weight gain is the term a from **??**. Note that $\log_{10} 3.16 = 0.5$.

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

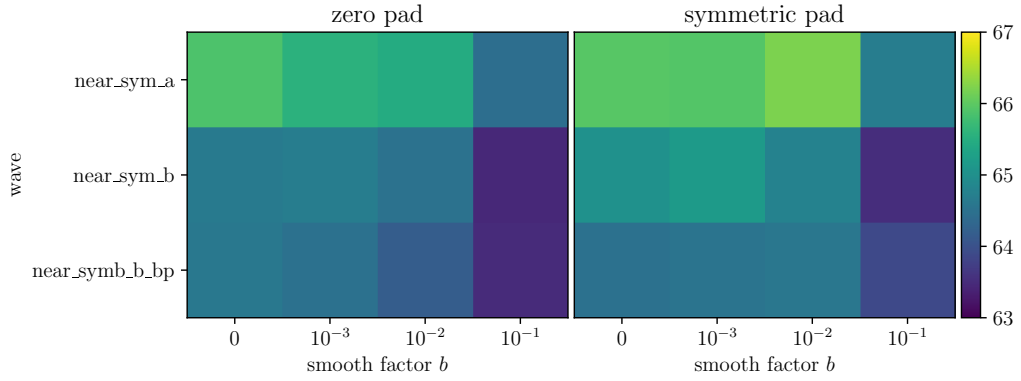
1.5.2.2



(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 1.4. Each subfloat 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. It is clear that symmetric padding is better than zero padding. Surprisingly, the shorter filter (near_sym_a) fares better than its longer counterpart, 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

Bibliography

[1] 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.

[2] 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].

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

[4] 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.

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

Appendices

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

function IDWT($ll,\ lh,\ hl,\ hl,\ g_0,\ g_1,\ mode$)
  save $g_0,\ g_1,\ mode$ ▷ For the backwards pass
  $lo \leftarrow \text{sfb1d}(ll,\ lh,\ g_0,\ g_1,\ mode,\ axis = -2)$
  $hi \leftarrow \text{sfb1d}(hl,\ hh,\ g_0,\ g_1,\ mode,\ axis = -2)$
  $x \leftarrow \text{sfb1d}(lo,\ hi,\ g_0,\ g_1,\ mode,\ axis = -1)$
  return x
end function

function IDWT_BACKPROP(δy)
  load $g_0,\ g_1,\ mode$
  $g_0,\ g_1 \leftarrow \text{flip}(g_0),\ \text{flip}(g_1)$ ▷ flip the filters as in (1.2.1)
  $\Delta lo,\ \Delta hi \leftarrow \text{afb1d}(\delta y,\ g_0,\ g_1,\ mode,\ axis = -2)$
  $\Delta ll,\ \Delta lh \leftarrow \text{afb1d}(\Delta lo,\ g_0,\ g_1,\ mode,\ axis = -1)$
  $\Delta hl,\ \Delta hh \leftarrow \text{afb1d}(\Delta hi,\ g_0,\ g_1,\ mode,\ axis = -1)$
  return $\Delta ll,\ \Delta lh,\ \Delta hl,\ \Delta hh$
end function

Algorithm B.2 Smooth Magnitude

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

function MAG_SMOOTH_BACKPROP(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