Uses of Complex Wavelets in Deep Convolutional Neural Networks

Fergal Cotter

Supervisor: Prof. Nick Kingsbury
Prof. Joan Lasenby

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
PhD

Trinity College April 2019

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Fergal Cotter
April 2019

Acknowledgements

I would like to thank my supervisor Nick Kingsbury who has dedicated so much of his time to help my research. He has not only been instructing and knowledgeable, but very kind and supportive. I would also like to thank my advisor, Joan Lasenby for supporting me in my first term when Nick was away, and for always being helpful. I must also acknowledge YiChen Yang and Ben Chaudhri who have done fantastic work helping me develop ideas and code for my research.

I sincerely thank Trinity College for both being my alma mater and for sponsoring me to do my research. Without their generosity I would not be here.

And finally, I would like to thank my girlfriend Cordelia, and my parents Bill and Mary-Rose for their ongoing support.

Abstract

Image understanding has long been a goal for computer vision. It has proved to be an exceptionally difficult task due to the large amounts of variability that are inherent to objects in scene. Recent advances in supervised learning methods, particularly convolutional neural networks (CNNs), have pushed the frontier of what we have been able to train computers to do.

Despite their successes, the mechanics of how these networks are able to recognize objects are little understood. Worse still is that we do not yet have methods or procedures that allow us to train these networks. The father of CNNs, Yann LeCun, summed it up as:

> There are certain recipes (for building CNNs) that work and certain recipes that don't, and we don't know why.

We believe that if we can build a well understood and well-defined network that mimicks CNN (i.e., it is able to extract the same features from an image, and able to combine these features to discriminate between classes of objects), then we will gain a huge amount of invaluable insight into what is required in these networks as well as what is learned.

In this paper we explore our attempts so far at trying to achieve this. In particular, we start by examining the previous work on Scatternets by Stephané Mallat. These are deep networks that involve successive convolutions with wavelets, a well understood and well-defined topic. We draw parallels between the wavelets that make up the Scatternet and the learned features of a CNN and clarify their differences. We then go on to build a two stage network that replaces the early layers of a CNN with a Scatternet and examine the progresses we have made with it.

Finally, we lay out our plan for improving this hybrid network, and how we believe we can take the Scatternet to deeper layers.

Table of contents

List of figures **xiii**

List of tables **xvii**

1 Introduction **1**
1.1 Gradients of Complex Activations . . . 1
1.1.1 Cauchy-Riemann Equations . . . 1
1.1.2 Complex Magntitude . . . 1

2 Background **3**
2.1 Notation . . . 3
2.2 The Fourier and Wavelet Transforms . . . 4
2.2.1 The Fourier Transform . . . 5
2.2.2 The Wavelet Transform . . . 5
2.2.3 Discrete Wavelet Transform and its Shortcomings . . . 5
2.2.4 Complex Wavelets . . . 6
2.2.5 Fourier Based Wavelet Transform . . . 9
2.2.6 The DT$\mathbb{C}$WT . . . 13
2.2.7 Summary of Methods . . . 18
2.3 Shift Invariance of the DT$\mathbb{C}$WT . . . 19
2.4 Neural Networks . . . 24
2.4.1 The Neuron and Single Layer Neural Networks . . . 24
2.4.2 Loss or Error Term . . . 24
2.4.3 Backpropagation . . . 24
2.4.4 Gradient descent vs Stochastic Gradient Descent vs Mini-Batches . . . 26
2.4.5 Learning . . . 28
2.4.6 Optimization . . . 28
2.4.7 Extending to multiple layers and different block types . . . 28
2.5 Relevant Architectures . . . 31
2.5.1 LeNet . . . 31

2.5.2 AlexNet . . . 31
2.5.3 VGG . . . 31
2.5.4 Residual Networks . . . 31
2.5.5 old . . . 31
2.5.6 Fully Connected Layers . . . 32
2.6 Scatternets . . . 33
2.7 Translation Invariant Scatternets . . . 33
2.7.1 Defining the Properties . . . 33
2.7.2 Finding the Right Operator . . . 34
2.8 Rotation and Translation Invariant Scatternets . . . 35
2.8.1 An Important note on Joint vs. Separable Invariants . . . 37
2.8.2 Defining the Properties . . . 37
2.8.3 The Operator . . . 38

3 A Faster ScatterNet **41**
3.1 The Design Constraints . . . 42
3.2 A Brief Description of Autograd . . . 43
3.3 Fast Calculation of the DWT and IDWT . . . 44
3.3.1 Primitives . . . 45
3.3.2 The Forward and Backward Algorithms . . . 45
3.4 Fast Calculation of the DT$\mathbb{C}$WT . . . 47
3.5 Changing the ScatterNet Core . . . 48
3.6 Comparisons . . . 51
3.6.1 Speed and Memory Use . . . 51
3.6.2 Performance . . . 51
3.7 Conclusion . . . 53

4 Visualizing and Improving Scattering Networks **57**
4.1 Related Work . . . 59
4.2 The Scattering Transform and Hybrid Network . . . 61
4.2.1 Scattering Color Images . . . 63
4.3 The Inverse Network . . . 63
4.3.1 Inverting the Low-Pass Filtering . . . 64
4.3.2 Inverting the Magnitude Operation . . . 65
4.3.3 Inverting the Wavelet Decomposition . . . 65
4.4 Visualization with Inverse Scattering . . . 65
4.5 Channel Saliency . . . 67
4.6 Corners, Crosses and Curves . . . 67
4.7 Discussion . . . 69

5 A Learnable ScatterNet: Locally Invariant Convolutional Layers 71
5.1 Related Work . . . 72
5.2 Recap of Useful Terms . . . 72
5.2.1 Convolutional Layers . . . 72
5.2.2 Wavelet Transforms . . . 73
5.2.3 Scattering Transforms . . . 73
5.3 Locally Invariant Layer . . . 74
5.3.1 Properties . . . 75
5.4 Implementation Details . . . 77
5.4.1 Parameter Memory Cost . . . 77
5.4.2 Activation Memory Cost . . . 77
5.4.3 Computational Cost . . . 78
5.4.4 Forward and Backward Algorithm . . . 79
5.5 Experiments . . . 79
5.5.1 Layer Introduction with MNIST . . . 79
5.5.2 Layer Comparison with CIFAR and Tiny ImageNet . . . 82
5.5.3 Network Comparison . . . 84
5.6 Conclusion . . . 84

6 Learning in the Wavelet Domain 89
6.1 A Summary of Choices . . . 89
6.2 Related Work . . . 90
6.2.1 Wavelets as a Front End . . . 90
6.2.2 Parameterizing filters in Fourier Domain . . . 91
6.3 Introduction . . . 91
6.3.1 Invertible Transforms and Optimization . . . 92
6.4 Background and Notation . . . 92
6.4.1 DWT Notation . . . 93
6.4.2 DT$\mathbb{C}$WT Notation . . . 94
6.5 The Wavelet Gain Layer . . . 94
6.5.1 The DWT Gain Layer . . . 96
6.5.2 The DT$\mathbb{C}$WT Gain Layer . . . 99
6.5.3 Examples . . . 101
6.6 Implementation Details . . . 104
6.6.1 Parameter Memory Cost . . . 104
6.6.2 Activation Memory Cost . . . 104
6.6.3 Computational Cost . . . 105
6.6.4 Parameter Initialization . . . 106
6.7 Gain Layer Experiments . . . 106

6.8 Wavelet Based Nonlinearities and Multiple Gain Layers 108
6.8.1 ReLUs in the Wavelet Domain 108
6.9 Conclusion and Future Work 108

7 Conclusion **111**
7.1 Discussion of Attempts so Far 111
7.1.1 Multiscale Scatternets + SVM 111
7.1.2 Reduced Multiscale Scatternets and CNNs 112
7.1.3 Improved Analysis Methods 113
7.2 Future Work 113
7.2.1 Closing the Gap 113
7.2.2 Moving to a new Dataset 113
7.2.3 Improved Visualizations 114
7.2.4 Going Deeper 114
7.2.5 Resiudal Scattering Layers 114
7.2.6 Exploring the Scope of Scatternets 115
7.2.7 Analysing Newer CNN Layers 115
7.2.8 Revisiting Greedy Layer-Wise Training 115
7.3 Timeline 115

References **119**

List of figures

2.1 Importance of phase over magnitude for images 4
2.2 Sensitivity of DWT coefficients to zero crossings and small shifts 7
2.3 Typical wavelets from the 2D separable DWT 8
2.4 Single Morlet filter with varying slants and window sizes 10
2.5 The full dictionary of Morlet wavelets used by Mallat 11
2.6 Fourier Implementation of the Morlet decomposition of an input image 12
2.7 Three Morlet Wavelet Families and their tiling of the frequency plane 13
2.8 Analysis FB for the DT$\mathbb{C}$WT 14
2.9 The DWT high-high vs the DT$\mathbb{C}$WT high-high frequency support 16
2.10 Wavelets from the 2D DT$\mathbb{C}$WT 16
2.11 The shift invariance ofthe DT$\mathbb{C}$WT vs. the real DWT 16
2.12 The filter bank implementation of the DT$\mathbb{C}$WT 17
2.13 DT$\mathbb{C}$WT basis functions and their frequency coverage 18
2.14 Two adversarial examples generated for AlexNet 18
2.15 Comparison of the energy spectra for DT$\mathbb{C}$WT wavelets to Morlet wavelets . 19
2.16 Full 1-D DT$\mathbb{C}$WT 20
2.17 Block Diagram of 1-D DT$\mathbb{C}$WT 20
2.18 A single neuron 25
2.19 Differences in non-linearities 29
2.20 Tight vs. overlapping pooling 29
2.21 The residual unit from ResNet 31
2.22 Standard CNN architecture 32
2.23 A Lipschitz continuous function 34
2.24 Real, Imaginary and Modulus of complex wavelet convolved with an impulse. 35
2.25 Translation Invariant Scatternet Layout 36
2.26 Three dimensional convolution with roto-scale wavelet 39

3.1 An autograd block 44
3.2 Block Diagram of 2-D DWT 44
3.3 Hyperparameter results for the DT$\mathbb{C}$WT scatternet on various datasets . . . 55

4.1 Deconvolution Network Block Diagram 60

4.2 A DeScattering layer (left) attached to a Scattering layer (right). We are using the same convention as [40] Figure 1 - the input signal starts in the bottom right hand corner, passes forwards through the ScatterNet (up the right half), and then is reconstructed in the DeScatterNet (downwards on the left half). The DeScattering layer will reconstruct an approximate version of the previous order's propagated signal. The 2×2 grids shown around the image are either Argand diagrams representing the magnitude and phase of small regions of *complex* (De)ScatterNet coefficients, or bar charts showing the magnitude of the *real* (De)ScatterNet coefficients (after applying the modulus non-linearity). For reconstruction, we need to save the discarded phase information and reintroduce it by multiplying it with the reconstructed magnitudes. 64

4.3 Visualization of a random subset of features from S_0 (all 3), S_1 (6 from the 24) and S_2 (16 from the 240) scattering outputs. We record the top 9 activations for the chosen features and project them back to the pixel space. We show them alongside the input image patches which caused the large activations. We also include reconstructions from layer conv2_2 of VGG Net [35](a popular CNN, often used for feature extraction) for reference — here we display 16 of the 128 channels. The VGG reconstructions were made with a CNN DeconvNet based on [40]. Image best viewed digitally. 66

4.4 Shapes possible by filtering across the wavelet orientations with complex coefficients 67

5.1 Block Diagram of Proposed Invariant Layer for $j = J = 1$. Activations are shaded blue, fixed parameters yellow and learned parameters red. Input $x^{(l)} \in \mathbb{R}^{C_l \times H \times W}$ is filtered by real and imaginary oriented wavelets and a lowpass filter and is downsampled. The channel dimension increases from C_l to $(2K+1)C_l$, where the number of orientations is $K = 6$. The real and imaginary parts are combined by taking their magnitude (an example of what this looks like in 1D is shown above the magnitude operator) - the components oscillating in quadrature are combined to give $z^{(l+1)}$. The resulting activations are concatenated with the lowpass filtered activations, mixed across the channel dimension, and then passed through a nonlinearity σ to give $x^{(l+1)}$. If the desired output spatial size is $H \times W$, $x^{(l+1)}$ can be bilinearly upsampled paying only a few multiplies per pixel. 76

6.1 First layer filters of the AlexNet architecture 90

6.2 Architecture using the DWT as a frontend to a CNN 91

6.3 Proposed new forward pass in the wavelet domain 95

6.4 Block Diagram of 1-D DWT Gain Layer 97
6.5 Forward and backward block diagrams for DT$\mathbb{C}$WT gain layer 98
6.6 Block Diagram of 1-D DT$\mathbb{C}$WT Gain Layer 100
6.7 DT$\mathbb{C}$WT subbands 101
6.8 Block diagrams of proposed method to learn in the wavelet domain 102
6.9 Example outputs from an impulse input for the proposed gain layers 103

7.1 Comparison of test accuracy for our network vs a two layer CNN 112
7.2 Possible future work with residual mappings 115

List of tables

3.1 Comparison of properties of different ScatterNet packages 51
3.2 Comparison of speed for the forward and backward passes of the competing ScatterNet Implementations 52
3.3 Hybrid architectures for performance comparison 53
3.4 Hyperparameter settings for the DT$\mathbb{C}$WT scatternet 54
3.5 Performance comparison for a DT$\mathbb{C}$WT based ScatterNet vs Morlet based ScatterNet 54

5.1 Architectures for MNIST hyperparameter experiments 81
5.2 Hyperparameter settings for the MNIST experiments 81
5.3 Architecture performance comparison 82
5.4 Modified architecture performance comparison 83
5.5 CIFAR and Tiny ImageNet Base Architecture 86
5.6 Results for testing VGG like architecture with convolutional and invariant layers on several datasets. An architecture with 'invX' means the equivalent convolutional layer 'convX' from **??** was swapped for our proposed layer. The top row is the reference architecture using all convolutional layers. The 'A' architecture is the originally proposed gain layer, the 'B' architecture uses the gainlayer with random shifting of the activations, and the 'C' architecture changes the mixing to be a learned 3×3 kernel acting on the invariant coefficients. Numbers are averages over 5 runs. **TODO: Rerun C models with different hypes**. 87
5.7 Hybrid ScatterNet top-1 classification accuracies on CIFAR-10 and CIFAR-100 88

6.1 Results for testing VGG like architecture with convolutional and wavelet gain layers on several datasets 107

7.1 Our plan for the remainder of the PhD 117

Chapter 1

Introduction

This work is stimulated by the intuition that wavelet decompositions, in particular complex wavelet transforms, are good building blocks for doing image recognition tasks. Their well understood and well defined behaviour as well as the similarities seen in learned networks, implies that there is potential gain for thinking about CNN layers in a new light.

To explore and test this intuition, we begin by looking at one of the most popular current uses of wavelets in image recognition tasks, in particular the Scattering Transform.

1.1 Gradients of Complex Activations

1.1.1 Cauchy-Riemann Equations

None of these will be solved. However, we can still find the partial derivatives w.r.t.the real and imaginary parts.

1.1.2 Complex Magntitude

Care must be taken when calculating gradients for the complex magnitude, as the gradient is undefined at the origin. We take the common approach of setting the gradient at the corner point to be 0.

Let us call the real and imaginary inputs to a magnitude block x and y, and we define the real output r as:

$$r = |x + jy| = \sqrt{x^2 + y^2}$$

Then the partial derivatives w.r.t.the real and imaginary inputs are:

$$\begin{aligned}\frac{\partial r}{\partial x} &= \frac{x}{\sqrt{x^2+y^2}} = \frac{x}{r} \\ \frac{\partial r}{\partial y} &= \frac{y}{\sqrt{x^2+y^2}} = \frac{y}{r}\end{aligned}$$

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

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

These partial derivatives are very variable around 0. **Show a plot of this**. We can smooth it out by adding a smoothing term:

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

This keeps the magnitude zero for small x, y but does slightly shrink larger values. The gain we get however is a new smoother gradient surface **Plot this**.

Chapter 2

Background

This thesis combines work in several fields. We provide a background for the most important and relevant fields in this chapter. We first introduce the definition and properties of the Wavelet Transforms, focussing on the DT$\mathbb{C}$WT, then we describe the fundamentals of Convolutional Neural Networks and how they learn, and finally we introduce the Scattering Transform, the original inspiration for this thesis.

2.1 Notation

We define standard notation to help the reader better understand figures and equations. Many of the terms we define here relate to concepts that have not been introduced yet, so may be unclear until later.

- **Pixel coordinates**
  When referencing spatial coordinates in an image, the preferred index is $\mathbf{u}$ for a 2D vector of coordinates, or $[u_1, u_2]$ if we wish to specify the coordinates explicitly. u_1 indexes rows from top to bottom of an image, and u_2 indexes columns from left to right. We typically use $H \times W$ for the size of the image, (but this is less strict). I.e., $u_1 \in \{0, 1, \dots H-1\}$ and $u_2 \in 0, 1, \dots W-1$.

- **Convolutional networks**
  Image convolutional neural networks often work with 4-dimensional arrays. In particular, mini-batches of images with multiple channels. When we need to, we index over the minibatch with the variable n and over the channel dimension with c. For example, we can index an activation $\mathbf{x}$ by $\mathbf{x}[n, c, u_1, u_2]$.

  To distinguish between features, filters, weights and biases of different levels in a deep network, we may add a layer subscript, or l for the general case, i.e., $\mathbf{z}_l[n, c, \mathbf{u}]$ indexes the feature map at the l-th layer of a deep network.

Figure 2.1: **Importance of phase over magnitude for images.** The phase of the Fourier transform of the first image is combined with the magnitude of the Fourier transform of the second image and reconstructed. Note that the first image has entirely won out and nothing is left visible of the cameraman.

- **Fourier transforms**
  When referring to the Fourier transform of a function, f, we typically adopt the overbar notation: i.e., $\mathcal{F}\{f\} = \bar{f}$.

- **Wavelet Filter Banks**
  Using standard notation, we define the scaling function as ϕ and the wavelet function as ψ. For a filter bank implementation of a wavelet transform, we use h for analysis and g for synthesis filters.

  In a multiscale environment, j indexes scale from $\{1, 2, \ldots, J\}$. For 2D complex wavelets, θ indexes the orientation at which the wavelet has the largest response, i.e., $\psi_{\theta,j}$ refers to the wavelet at orientation θ at the j-th scale.

2.2 The Fourier and Wavelet Transforms

Computer vision is an extremely difficult task. Greyscale intensities in an image are not very helpful in understanding what is in that image. Indeed, these values are sensitive to lighting conditions and camera configurations. It would be easy to take two photos of the same scene and get two vectors x_1 and x_2 that have a very large Euclidean distance, but to a human, would represent the same objects. What is most important in an image are the local variations of image intensity. In particular, the location or phase of the waves that make up the image. A simple experiments to demonstrate this is shown in Figure 2.1.

2.2.1 The Fourier Transform

While the Fourier transform is the core of frequency analysis, it is a poor feature descriptor due to the infinite support of its basis functions. If a single pixel changes in the input, this can theoretically change all of the Fourier coefficients. As natural images are generally non-stationary, we need to be able to isolate frequency components in local regions of an image, and not have this property of global dependence.

The Fourier transform does have one nice property, however, in that the magnitude of Fourier coefficients are invariant to global translations, a nuisance variability. We explore this theme more in our review of the Scattering Transform by Mallat et. al.in section 2.6.

2.2.2 The Wavelet Transform

The Wavelet Transform, like the Fourier Transform, can be used to decompose a signal into its frequency components. Unlike the Fourier transform though, these frequency components can be localized in space. The localization being inversely proportional to the frequency of interest which you want to measure. This means that changes in one part of the image will not affect the wavelet coefficients in another part of the image, so long as the distance between the two parts is much larger than the wavelength of the wavelets you are examining.

The corollary of this property is that wavelets can also be localised in frequency.

The Continuous Wavelet Transform (CWT) gives full flexibility on what spatial/frequency resolution is wanted, but is very redundant and computationally expensive to compute, instead the common yardstick for wavelet analysis is the Discrete Wavelet Transform (DWT) which we introduce.

2.2.3 Discrete Wavelet Transform and its Shortcomings

A particularly efficient implementation of the DWT is Mallat's maximally decimated dyadic filter tree, commonly used with an orthonormal basis set. Orthonormal basis sets are non-redundant, which makes them computationally and memory efficient, but they also have several drawbacks. In particular:

- The DWT is sensitive to the zero crossings of its wavelets. We would like singularities in the input to yield large wavelet coefficients, but if they fall at a zero crossing of a wavelet, the output can be small. See Figure 2.2.

- They have poor directional selectivity. As the wavelets are purely real, they have passbands in all four quadrants of the frequency plane. While they can pick out edges aligned with the frequency axis, they do not have admissibility for other orientations. See Figure 2.3.

- They are not shift invariant. In particular, small shifts greatly perturb the wavelet coefficients. Figure 2.2 shows this for the centre-left and centre-right images. Figure 2.11 (right) also shows this.

The lack of shift invariance and the possibility of low outputs at singularities is a price to pay for the critically sampled property of the transform. Indeed, this shortcoming can be overcome with the undecimated DWT mallat_wavelet_1998,coifman_translation-invariant_1995, but it pays a heavy price for this both computationally, and in the number of wavelet coefficients it produces, particularly if many scales J are used.

2.2.4 Complex Wavelets

So the DWT has overcome the problem of space-frequency localisation, but it has introduced new problems. Fortunately, we can improve on the DWT with complex wavelets, as they can solve these new shortcomings while maintaining the desired localization properties.

The Fourier transform does not suffer from a lack of directional selectivity and shift invariance because its basis functions are based on the complex sinusoid:

$$e^{j\omega t} = \cos(\omega t) + j\sin(\omega t) \tag{2.2.1}$$

whereas the DWT's basis functions are based on only the real sinusoid $\cos(\omega t)$[1]. As t moves along the real line, the phase of the Fourier coefficients change linearly, while their magnitude remains constant. In contrast, as t moves along the real line, the sign of the real coefficient oscillates between -1 and 1, and its magnitude changes in a non-linear way.

These nice properties come from the fact that the cosine and sine functions of the Fourier transform form a Hilbert Pair, and so together they constitute an analytic signal.

We can achieve these nice properties if the mother wavelet for our wavelet transform is analytic:

$$\psi_c(t) = \psi_r(t) + j\psi_c(t) \tag{2.2.2}$$

where $\psi_r(t)$ and $\psi_c(t)$ form a Hilbert Pair (i.e., they are 90° out of phase with each other).

There are a number of possible ways to do a wavelet transform with complex wavelets. We examine two in particular, the Fourier-based method used by Mallat et. al. in their scattering transform bruna_classification_2011, bruna_invariant_2013, bruna_scattering_2013, oyallon_generic_2013, oyallon_deep_2015, sifre_rotation_2013, sifre_rigid-motion_2014, sifre_rigid-motion_2014-1, sifre_scatnet_2013, and the separable, filter bank based DT$\mathbb{C}$WT developed by Kingsbury kingsbury_wavelet_1997, kingsbury_dual-tree_1998, kingsbury_dual-tree_1998-1, kingsbury_image_1999, kingsbury_shift_1999, kingsbury_dual-tree_2000, kingsbury_complex_2001, selesnick_dual-tree_2005.

[1] we have temporarily switched to 1D notation here as it is clearer and easier to use, but the results still hold for 2D

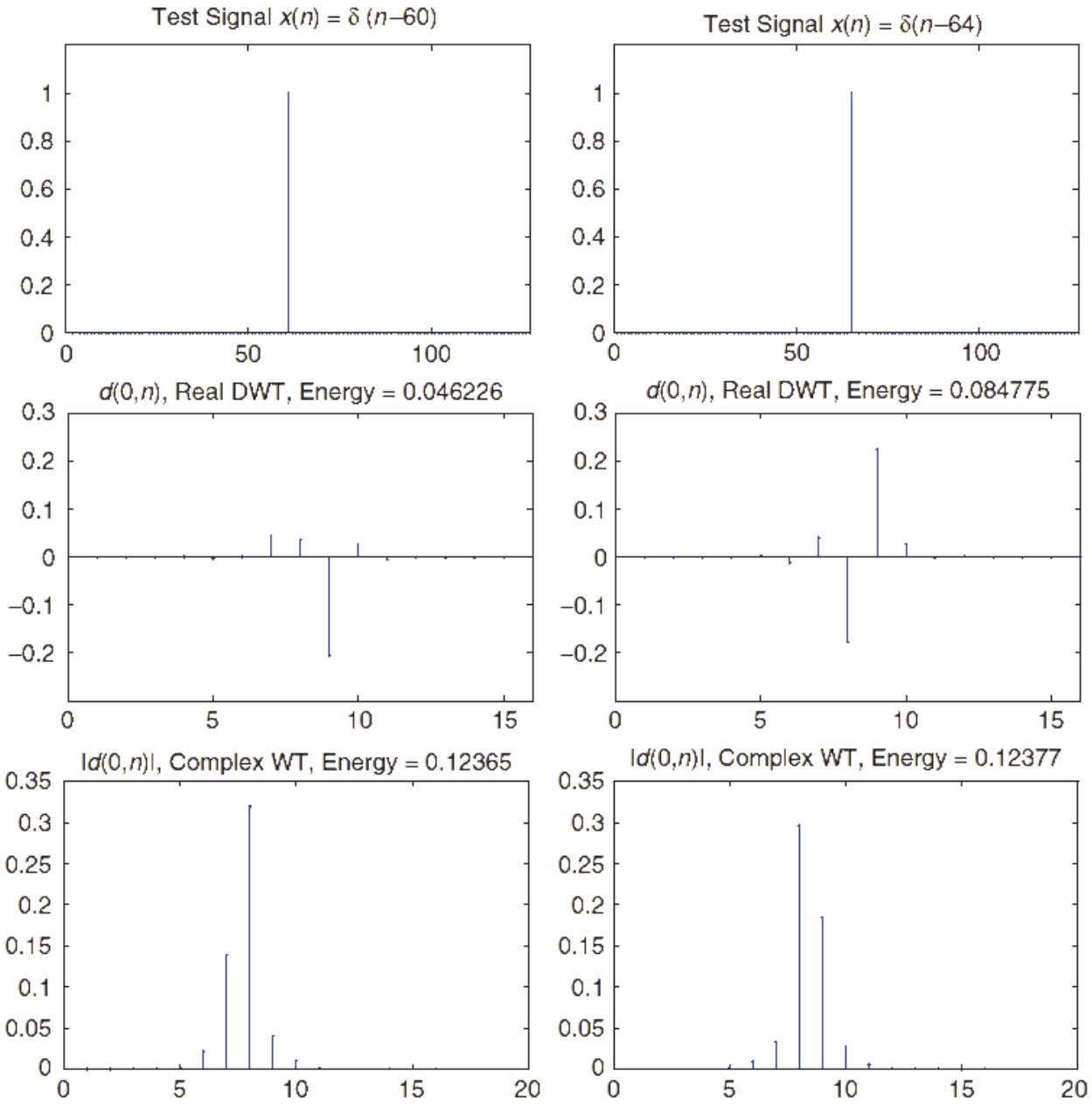


Figure 2.2: **Sensitivity of DWT coefficients to zero crossings and small shifts.** Two impulse signals $\delta(n-60)$ and $\delta(n-64)$ are shown (top), as well as the wavelet coefficients for scale $j=1$ for the DWT (middle) and for the DT$\mathbb{C}$WT (bottom). In the middle row, not only are the coefficients very different from a shifted input, but the energy has almost doubled. As the DWT is an orthonormal transform, this means that this extra energy has come from other scales. In comparison, the energy of the magnitude of the DT$\mathbb{C}$WT coefficients has remained far more constant, as has the shape of the envelope of the output. Image taken from selesnick_dual-tree_2005.

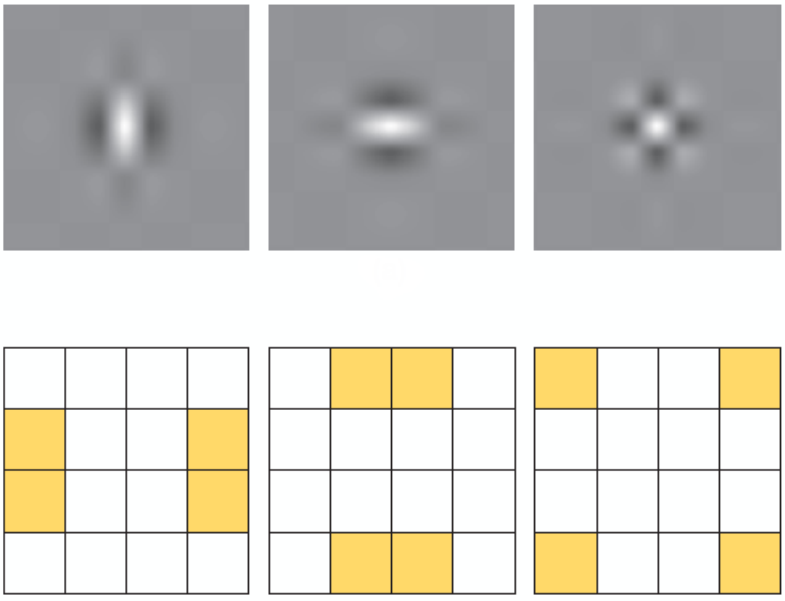


Figure 2.3: **Typical wavelets from the 2D separable DWT.** Top: Wavelet point spread functions for the low-high, high-low, and high-high wavelets. High-high wavelets are in a checkerboard pattern, with no favoured orientation. Bottom: Idealized support of the spectra of each of the wavelets. Image taken from selesnick_dual-tree_2005.

2.2.5 Fourier Based Wavelet Transform

The Fourier Based method used by Mallat et. al. is an efficient implementation of the Gabor Transform. Mallat tends to prefer to use a close relative of the Gabor wavelet — the Morlet wavelet — as the mother wavelet for his transform.

While the Gabor wavelets have the best theoretical trade-off between spatial and frequency localization, they have a non-zero mean. This makes the wavelet coefficients non-sparse, as they will all have a DC component to them, and makes them inadmissible as wavelets. Instead, the Morlet wavelet has the same shape, but with an extra degree of freedom chosen to set $\int \psi(\mathbf{u}) d\mathbf{u} = 0$. This wavelet has equation (in 2D):

$$\psi(\mathbf{u}) = \frac{1}{2\pi\sigma^2}(e^{i\mathbf{u}\xi} - \beta)e^{-\frac{|\mathbf{u}|^2}{2\sigma^2}} \tag{2.2.3}$$

where β is usually $<< 1$ and is this extra degree of freedom, σ is the size of the gaussian window, and ξ is the location of the peak frequency response — i.e., for an octave based transform, $\xi = 3\pi/4$.

Mallat et. al. add an extra degree of freedom to this by allowing for a non-circular gaussian window over the complex sinusoid, which gives control over the angular resolution of the final wavelet, so this now becomes:

$$\psi(\mathbf{u}) = \frac{\gamma}{2\pi\sigma^2}(e^{i\mathbf{u}\xi} - \beta)e^{-\mathbf{u}^t\Sigma^{-1}\mathbf{u}} \tag{2.2.4}$$

Where

$$\Sigma^{-1} = \begin{bmatrix} \frac{1}{2\sigma^2} & 0 \\ 0 & \frac{\gamma^2}{2\sigma^2} \end{bmatrix}$$

The effects of modifying the eccentricity parameter γ and the window size σ are shown in Figure 2.4. To have a full two dimensional wavelet transform, we need to rotate this mother wavelet by angle θ and scale it by j/Q, where Q is the number of scales per octave (usually 1 in image processing). This can be done by doing the following substitutions in Equation 2.2.4:

$$\begin{aligned} R_\theta &= \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \\ \mathbf{u}_\theta &= R_{-\theta}\mathbf{u} \\ \sigma_j &= 2^{\frac{j-1}{Q}}\sigma \\ \xi_j &= \frac{\xi}{2^{\frac{j-1}{Q}}} \end{aligned}$$

Mallat et. al. combine these two variables into a single coordinate

$$\lambda = (\theta, j/Q) \tag{2.2.5}$$

Figure 2.4: **Single Morlet filter with varying slants and window sizes.** Top left — 45° plane wave (real part only). Top right — plane wave with $\sigma = 3, \gamma = 1$. Bottom left — plane wave with $\sigma = 3, \gamma = 0.5$. Bottom right — plane wave with $\sigma = 2, \gamma = 0.5$.

Returning to the higher level notation, we can write the Morlet wavelet as the sum of real and imaginary parts: And scaled and rotated wavelets as:

$$\psi_\lambda(\mathbf{u}) = 2^{-j/Q}\psi(2^{-j/Q}R_\theta^{-1}\mathbf{u}) \tag{2.2.6}$$

2.2.5.1 Implementation

The Fourier Implementation of this Morlet decomposition is shown in Figure 2.6. It is based on the fact that

$$\mathcal{F}(x * \psi)(\omega) = \mathcal{F}x(\omega)\mathcal{F}\psi(\omega) \tag{2.2.7}$$

so to compute the family of outputs of $x * \psi_\lambda$, we can precompute the Fourier transform of all of the wavelets, then at run time, take the Fourier transform of the image, x, multiply with the Fourier transform of the wavelets, and then take the inverse Fourier transform of the product. The output scale can be chosen by periodizing the product of the Fourier transforms, and then compute the inverse Fourier transform at the reduced resolution.

The resulting complexity of the entire operation for an image with $N \times N$ pixels is:

- $O(N^2 \log N)$ for the forward FFT of x.
- $O(JLN^2)$ for the multiplication in the frequency domain. We can see this from Figure 2.6, there are J scales to do multiplication at, and each scale has L orientations, except for the low-low.

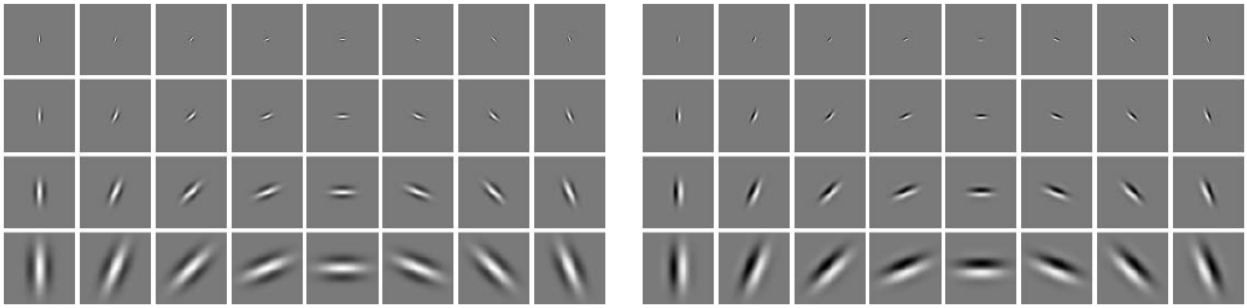


Figure 2.5: **The full dictionary of Morlet wavelets used by Mallat.** The real filters are on the left and the imaginary on the right. The first row correspond to scale $j = 1$, increasing up to $j = 4$. The first column corresponding to $\theta = 0$, rotating through $\pi/8$ up to the eighth column of $7\pi/8$, $\gamma = 1/2$.

- $O(L\sum_j (2^{-2j}N^2)\log 2^{-2j}N^2)$ for the inverse FFTs. The term inside the sum is just the $O(N^2 \log N)$ term of an inverse FFT that has been downsampled by 2^j in each direction.

And altogether:

$$T(N) = O(N^2 \log N) + O(JLN^2) + O(L\sum_j N^2 log 2^{-j} N) \tag{2.2.8}$$

2.2.5.2 Invertibility and Energy Conservation

We can write the wavelet transform of an input x as

$$\mathcal{W}x = \{x * \phi_J, x * \psi_\lambda\}_\lambda \tag{2.2.9}$$

The ℓ^2 norm of the wavelet transform is then defined by

$$\|\mathcal{W}x\|^2 = \|x * \phi_J\|^2 + \sum_\lambda \|x * \psi_\lambda\|^2 \tag{2.2.10}$$

An energy preserving transform will satisfy Plancherel's equality, so that

$$\|\mathcal{W}x\| = \|x\| \tag{2.2.11}$$

which is a nice property to have for invertibility, as well as for analysing how different signals get transformed (e.g. white noise versus standard images).

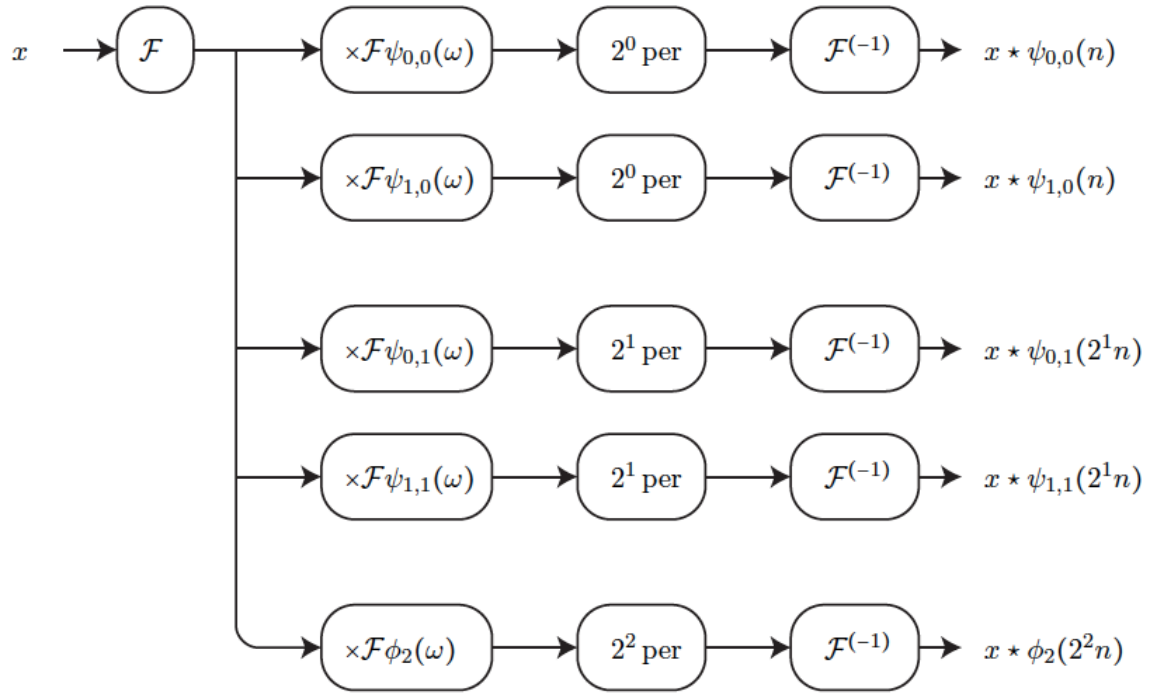


Figure 2.6: Fourier Implementation of the Morlet decomposition of an input image, with $J = 2$ scales and $L = 2$ orientations. The Fourier transform of x is calculated and multiplied with the (precomputed) Fourier transforms of a bank of Morlet filters. The results are periodized according to the target resolution, and then the inverse Fourier transform is applied. Image taken from sifre_rigid-motion_2014-1.

Figure 2.7: Three Morlet Wavelet Families and their tiling of the frequency plane. For each set of parameters, the point spread functions of the wavelet bases are shown, next to their Littlewood-Paley sum $A(\omega)$. None of the configurations cover the corners of the frequency plane, and they often exceed 1. Increasing J, L (Sifre uses C in these diagrams) or Q gives better frequency localization but at the cost of spatial localization and added complexity. Image taken from sifre_rigid-motion_2014-1.

For a transform to be invertible, we examine the measure of how tightly its basis functions tile the Fourier plane with its Littlewood-Paley function:

$$A(\omega) = |\mathcal{F}\phi_J(\omega)|^2 + \sum_{\lambda} |\mathcal{F}\psi_\lambda(\omega)|^2 \tag{2.2.12}$$

If the tiliing is α-tight, then $\forall\omega \in \mathbb{R}^2$:

$$1 - \alpha \leq A(\omega) \leq 1 \tag{2.2.13}$$

and the wavelet operator, $\mathcal{W}$ is an α frame. If $A(\omega)$ is ever close to 0, then there is not a good coverage of the frequency plane at that location. If it ever exceeds 1, then there is overlap between bases. Both of these conditions make invertibility difficult[2]. Figure 2.7 show the invertibility of a few families of wavelets used by Mallat et. al.. Invertibility is possible, but not guaranteed for all configurations. The Fourier transform of the inverse filters are defined by:

$$\mathcal{F}\phi_J^{-1}(\omega) = A(\omega)^{-1}\mathcal{F}\phi_J(\omega) \tag{2.2.14}$$

$$\mathcal{F}\psi_\lambda^{-1}(\omega) = A(\omega)^{-1}\mathcal{F}\psi_\lambda(\omega) \tag{2.2.15}$$

2.2.6 The DT$\mathbb{C}$WT

The DT$\mathbb{C}$WT was first proposed by Kingsbury in [1], [2] as a way to combat many of the shortcomings of the DWT, in particular, its poor directional selectivity, and its poor shift invariance. A thorough analsysis of the properties and benefits of the DT$\mathbb{C}$WT is done in [3], [4]. Building on these properties, it been used successfully for denoising and inverse problems [5]–[8], texture classification [9], [10], image registration [11], [12] and SIFT-style keypoint generation matching [13]–[17] amongst many other applications.

Compared to Gabor (or Morlet) image analysis, the authors of [4] sum up the dangers as:

> A typical Gabor image analysis is either expensive to compute, is noninvertible, or both.

[2] In practise, if $A(\omega)$ is only slightly greater 1 for only a few small areas of ω, approximate inversion can be achieved

Figure 2.8: Analysis FB for the DT$\mathbb{C}$WT@. Top 'tree' forms the real component of the complex wavelet ψ_r, and the bottom tree forms the imaginary (Hilbert pair) component ψ_i. Image taken from selesnick_dual-tree_2005.

This nicely summarises the difference between this method and the Fourier based method outlined in subsection 2.2.5. The DT$\mathbb{C}$WT is a filter bank (FB) based wavelet transform. It is faster to implement than the Morlet analysis, as well as being more readily invertible.

2.2.6.1 Deisgn Criteria for the DT$\mathbb{C}$WT

It was stated in subsection 2.2.4 that if the mother (and daughter) wavelets were complex, with their real and imaginary parts forming a Hilbert pair, then the wavelet transform of a signal with these $\{\psi_{j,n}\}_{j,n}$ would give a representation that had nice shift properties[3], was insensitive to zero crossings of the wavelet, and had good directional selectivity.

As in subsection 2.2.4, we want to have a complex mother wavelet ψ_c that satisfies Equation 2.2.2, but now achieved with filter banks. A slight deviation from standard filter bank notation, where h_0, h_1 are the analysis and g_0, g_1 are the synthesis filters. We define:

- h_0, h_1 the low and high-pass analysis filters for ψ_r (henceforth called ψ_h)
- g_0, g_1 the low and high-pass analysis fitlers for ψ_i (henceforth called ψ_g)
- $\tilde{h}_0, \tilde{h}_1$ the low and high-pass synthesis filters for $\tilde{\psi}_h$.
- $\tilde{g}_0, \tilde{g}_1$ the low and hig pass synthesis filters for $\tilde{\psi}_g$.

The dilation and wavelet equations for a 1D filter bank implementation are:

$$\phi_h(t) = \sqrt{2}\sum_n h_0(n)\phi_h(2t-n) \tag{2.2.16}$$

$$\psi_h(t) = \sqrt{2}\sum_n h_1(n)\phi_h(2t-n) \tag{2.2.17}$$

$$\phi_g(t) = \sqrt{2}\sum_n g_0(n)\phi_g(2t-n) \tag{2.2.18}$$

$$\psi_g(t) = \sqrt{2}\sum_n g_1(n)\phi_g(2t-n) \tag{2.2.19}$$

This implementation is shown in Figure 2.8.

Designing a filter bank implementation that results in Hilbert symmetric wavelets does not appear to be an easy task. However, it was shown by kingsbury_image_1999 (and later proved by selesnick_hilbert_2001) that the necessary conditions are conceptually very simple.

[3] in particular, that a shift in input gives the same shift in magnitude of the wavelet coefficients, and a linear phase shift

One low-pass filter must be a *half-sample shift* of the other. I.e.,

$$g_0(n) \approx h_0(n-0.5) \rightarrow \psi_g(t) \approx \mathcal{H}\{\psi_h(t)\} \tag{2.2.20}$$

As the DT$\mathbb{C}$WT is designed as an invertible filter bank implementation, this is only one of the constraints. Naturally, there are also:

- Perfect reconstruction
- Finite support
- Linear phase
- Many vanishing moments at $z = -1$ for good stopband properties

to consider when building the h's and g's. The derivation of the filters that meet these conditions is covered in detail in kingsbury_complex_2001, kingsbury_design_2003, and in general in selesnick_dual-tree_2005. The result is the option of three families of filters: biorthogonal filters ($h_0[n] = h_0[N-1-n]$ and $g_0[n] = g_0[N-n]$), q-shift filters ($g_0[n] = h_0[N-1-n]$), and common-factor filters.

2.2.6.2 The Resulting Wavelets and their Properties

While analytic wavelets in 1D are useful for their shift invariance, the real beauty of the DT$\mathbb{C}$WT is in its ability to make a separable 2D wavelet transform with oriented wavelets.

2.9a shows the spectrum of the wavelet when the separable product uses purely real wavelets, as is the case with the DWT. 2.9b however, shows the separable product of two complex, analytic wavelets resulting in a localized and oriented 2D wavelet.

I.e., for the $+45°$ wavelet[4] (which is high in both ω_1 and ω_2), the separable product is:

$$\begin{aligned} \psi(\omega_1,\omega_2) &= \psi_c(\omega_1)\overline{\psi_c(\omega_2)} && (2.2.21)\\ &= (\psi_h(\omega_1) + j\psi_g(\omega_1))\overline{(\psi_h(\omega_2) + \psi_g(\omega_2))} \\ &= \psi_h(\omega_1)\psi_h(\omega_2) + \psi_g(\omega_1)\psi_g(\omega_2) \\ &\quad + j\left(\psi_g(\omega_1)\psi_h(\omega_2) - \psi_h(\omega_1)\psi_g(\omega_2)\right) && (2.2.22) \end{aligned}$$

Similar equations can be obtained for the other five wavelets and the scaling function, by replacing ψ with ϕ for both directions, and not taking the complex conjugate in Equation 2.2.21 to get the right hand side of the frequency plane.

Figure 2.10 shows the resulting wavelets both in the spatial domain and their idealized support in the frequency domain.

Figure 2.11 shows how the DT$\mathbb{C}$WT compares with the DWT with a shifting input.

[4] note that 2.9b shows the $135°$ wavelet

(a)

(b)

Figure 2.9: (a) The high-high DWT wavelet having a passband in all 4 corners of the frequency plane vs (b) the high-high DT$\mathbb{C}$WT wavelet frequency support only existing in one quadrant. Taken from selesnick_dual-tree_2005

Figure 2.10: Wavelets from the 2d DT$\mathbb{C}$WT@. **Top:** The six oriented filters in the space domain (only the real wavelets are shown). **Bottom:** Idealized support of the Fourier spectrum of each wavelet in the 2D frequency plane. Spectra of the the real wavelets are shown — the spectra of the complex wavelets ($\psi_h + j\psi_g$) only has support in the top half of the plane. Image taken from selesnick_dual-tree_2005.

2.2.6.3 Implementation and Efficiency

Figure 2.8 showed the layout for the DT$\mathbb{C}$WT for 1D signals. We saw from Equation 2.2.22 that the 2D separable product of wavelets involved the product of ψ_g, ψ_h, ϕ_g, and ϕ_h terms, with some summing and differencing operations. Figure 2.12 shows how to efficiently implement this with FBs.

As we did for subsubsection 2.2.5.1, we calculate and compare the complexity of the DT$\mathbb{C}$WT@. To do this, we must know the length of our h and g filters. It is also important to know that we must use different filters for the first scale to the deeper scales, as this achieves better analyticity. A typical configuration will use biorthogonal filters for the first scale, then qshift for subsequent scales. These filters have the following number of taps[5]:

	h_0	h_1	g_0	g_1
biorthogonal	5	7	7	5
qshift	10	10	10	10

The resulting complexity of the entire forward wavelet transform for an image with $N \times N$ pixels is:

- First layer (Image size $= N \times N$):
  - Column filtering requires $5N^2 + 7N^2$ multiply-adds
  - Row filtering to make the LoLo term requires $5N^2$ more multiply-adds

[5]This implementation uses the shorter near_sym_a biorthogonal filters and qshift_a filters. Smoother wavelets can have slightly more taps

Figure 2.11: The shift invariance of the DT$\mathbb{C}$WT (left) vs. the real DWT (right). The DT$\mathbb{C}$WT linearizes shifts in the phase change of the complex wavelet. Image taken from kingsbury_dual-tree_1998.

Figure 2.12: The filter bank implementation of the DT℃WT@. Image taken from kingsbury_image_1999

- Row filtering to make 15° and 165° requires $5N^2 + 4N^2$ multiply-adds (the 4 here comes from the Σ/Δ function block in Figure 2.12).
- Row filtering to make the 45° and 135° requires $7N^2 + 4N^2$ multiply-adds
- Row filtering to make the $75°and105°$ requires $7N^2 + 4N^2$ multiply-adds

The total being

$$T(N) = (5+7+5+5+4+7+4+7+4)N^2 = 48N^2$$

- Second and deeper layers (Image size $= 2^{-j}N \times 2^{-j}N = M \times M$):
  - Column filtering requires $10M^2 + 10M^2$ multiply-adds
  - Row filtering to make the LoLo term requires $10M^2$ more multiply-adds
  - Row filtering to make 15° and 165° requires $10M^2 + 4M^2$ multiply-adds (the 4 here comes from the Σ/Δ function block in Figure 2.12).
  - Row filtering to make the 45° and 135° requires $10M^2 + 4M^2$ multiply-adds
  - Row filtering to make the 75° and 105° requires $10M^2 + 4M^2$ multiply-adds

The total being:

$$T(M) = (10+10+10+10+4+10+4+10+4)M^2 = 68M = 68 \times 2^{-2j}N^2$$

$$T(N) = 48N^2 + \sum_{j=1}^{J} 68 \times 2^{-2j}N^2 \approx 100N^2 \tag{2.2.23}$$

As the term $\sum_{j=1}^{J} 68 \times 2^{-j}$ is a geometric series that sums to 1/3 as $j \to \infty$. We have also rounded up the sum to be conservative.

It is difficult to compare this with the complexity of the Fourier-based implementation of the Morlet wavelet transform we have derived in subsubsection 2.2.5.1, as we cannot readily estimate the big-O order constants for the 2D FFT method. However, the central term, JLN^2, would cost $24N^2$ multiplies for four scales and six orientations. These are complex multiplies, as they are in the Fourier domain, which requires four real multiplies. On top of this, to account for the periodic repetition of the inverse FFT implementation, Mallat et. al. symmetrically extend the image by $N/2$ in each direction. This means that the central term is already on the order of $\sim 200N^2$ multiplies, without even considering the more expensive forward and inverse FFTs.

Figure 2.13: Four scales of the DT$\mathbb{C}$WT (left) and its associated frequency coverage, or $A(\omega)$ (right). Note the reduced scale compared to Figure 2.7.

Figure 2.14: Two adversarial examples generated for AlexNet. The left column shows a correctly predicted sample, the right column an incorrectly predicted example, and the centre column the difference between the two images, magnified 10 times. Image taken from szegedy_intriguing_2013.

For a simple comparison experiment, we performed the two transforms on a four core Intel i7 processor (a moderately high-end personal computer). The DT$\mathbb{C}$WT transform took roughly 0.15s on a black and white 32×32 image, vs. 0.5s for the Fourier-based method. For a larger, 512×512 image, the DT$\mathbb{C}$WT implementation took 0.35s vs. 3.5s (times were averaged over several runs).

2.2.6.4 Invertibility and Energy Conservation

We analysed the Littlewood-Paley function for the Morlet-Fourier implementation, and saw what areas of the spectrum were better covered than others. How about for the DT$\mathbb{C}$WT@?

It is important to note that in the case of the DT$\mathbb{C}$WT, the wavelet transform is also approximately unitary, i.e.,

$$\|x\|^2 \approx \|\mathcal{W}x\|^2 \tag{2.2.24}$$

and the implementation is perfectly invertible as the Littlewood-Paley function is unity (or very near unity) $\forall\omega$. See Figure 2.13. This is not a surprise, as it is a design constraint in choosing the filters, but nonetheless is important to note.

A beneficial property of energy conservation is that the noise in the input will equal the noise in the wavelet coefficients. When we introduce Scatternets, we can show that we can keep the unitary property in the scattering coefficients. This is an important property, particularly in light of the recent investigations in szegedy_intriguing_2013. This paper saw that it is easy to find cases in CNNs where a small amount of input perturbation results in a completely different class label (see Figure 2.14). Having a unitary transform limits the amount the features can change, which will make the entire network more stable to distortion and noise.

2.2.7 Summary of Methods

One final comparison to make between the DT$\mathbb{C}$WT and the Morlet wavelets is their frequency coverage. The Morlet wavelets can be made to be tighter than the DT$\mathbb{C}$WT, which gives better angular resolution — see Figure 2.15. However it is not always better to keep getting finer

(a) DT$\mathbb{C}$WT wavelets (left to right) — 15°, 45° and 75°

(c) Morlet wavelets (left to right) — 0°, 22.5°, 45°, 67.5°, and 90°

Figure 2.15: Normalized Energy spectra of the DT$\mathbb{C}$WT wavelets versus the preferred 8 orientation Morlet wavelets by Mallat for the second quadrant. Orientations listed refer to the edge orientation in the spatial domain that gives the highest response. All wavelets have been normalized to be between zero and one. The Morlet wavelets have finer angular resolution, which can give better discrimination, at the cost of decreasing stability to deformations, and requiring larger spatial support.

and finer resolutions, indeed the Fourier transform gives the ultimate in angular resolution, but as mentioned, this makes it less stable to shifts and deformations.

?? compares the advantages and disadvantages of the wavelet methods discussed in this chapter.

2.3 Shift Invariance of the DT$\mathbb{C}$WT

Firstly, let us look at what would happen if we retained only 1 of the subbands. Note that we have to keep the same band from each tree. For any pair of coefficients on the tree, this would look like Figure 2.17. E.g. if we kept x_{001a} and x_{001b} then $M = 8$ and $A(z) = H_{0a}(z)H_{00a}(z^2)H_{001a}(z^4)$ is the transfer fucntion from x to x_{001a}. The transfer functions for $B(z)$, $C(z)$ and $D(z)$ are obtained similarly. It is well known that:

$$U(z) \downarrow M \quad \rightarrow \quad U(z^M) \tag{2.3.1}$$

$$U(z) \uparrow M \quad \rightarrow \quad \frac{1}{M} \sum_{k=0}^{M-1} U(W^k z^{1/M}) \tag{2.3.2}$$

Where $W = e^{j2\pi/M}$. So downsampling followed by upsampling becomes:

$$U(z) \downarrow M \uparrow M \rightarrow \frac{1}{M} \sum_{k=0}^{M-1} U(W^k z)$$

This means that

$$Y(z) = Y_a(z) + Y_b(z) = \frac{1}{M} \sum_{k=0}^{M-1} X(W^k z)[A(W^k z)C(z) + B(W^k z)D(z)] \tag{2.3.3}$$

Figure 2.16: **Full 1-D** DT$\mathbb{C}$WT**.**

Figure 2.17: **Block Diagram of 1-D** DT$\mathbb{C}$WT**.** Note the top and bottom paths are through the wavelet or scaling functions from just level m ($M = 2^m$). Figure based on Figure 4 in [18].

litreview/images/overlaps.png

The aliasing terms for which are everywhere where $k \neq 0$ (as $X(W^k z)$ is $X(z)$ shifted by $\frac{2k\pi}{M}$). I.e. to avoid aliasing in this reduced tree, we want to make sure that $A(W^k z)C(z) + B(W^k z)D(z) = 0$ for all $k \neq 0$.

The figure below (Fig 5 from [18]) shows what $A(W^k z)$ and $C(z)$ look like for both the lowpass case (left) and the highpass case (right). Note that these responses will be similar for $B(W^k z)$ and $D(z)$. For large values of k, there is almost no overlap (i.e. $A(W^k z)C(z) \approx B(W^k z)D(z) \approx 0$, but for small values of k (particularly $k = \pm 1$), the transition bands have significant overlap with the central response. It is here that we need to use the flexibility of having 2 trees to ensure that $A(W^k z)C(z)$ and $B(W^k z)D(z)$ cancel out.

To do this, let us consider the lowpass filters first. If we let:

$$B(z) = z^{\pm M/2}A(z) \tag{2.3.4}$$
$$D(z) = z^{\mp M/2}C(z) \tag{2.3.5}$$

Then

$$A(W^k z)C(z)+B(W^k z)D(z) = A(W^k z)C(z)+(W^k z)^{\pm M/2}A(W^k z)z^{\mp M/2}C(z) \tag{2.3.6}$$
$$= A(W^k z)C(z)+e^{\frac{jk2\pi}{M}\times(\pm\frac{M}{2})}z^{\pm M/2}z^{\mp M/2}A(W^k z)C(z) \tag{2.3.7}$$
$$= A(W^k z)C(z)+(-1)^k A(W^k z)C(z) \tag{2.3.8}$$

Which cancels when k is odd

Now consider the bandpass case. For shifts of $k=1$ the right half of the left peak overlaps with the left half of the right peak. For a shift of $k=2$, the left half of the left peak overlaps with the right half of the right peak. Similarly for $k=-1$ and $k=-2$. For $|k|>2$, there is no overlap. The fact that we have overlaps at both even and odd shifts of k means that we can't use the same clever trick from before. However, what we do note is that the overlap is always caused by opposite peaks (i.e. the left with the right peak, and never the left with itself, or the right with itself). The solution then is to have B and D have upper and lower passbands of opposite polarity, whil A and C should have passbands of the same polarity.

Consider two prototype complex filters $P(z)$ and $Q(z)$ each with single passbands going from $f_s/2M \to f_s/M$ (or $\frac{\pi}{M} \to \frac{2*\pi}{M}$) - they must be complex to have support in only one half of the frequency plane. Now say $P^*(z) = \sum_r p_r^* z^{-r}$ is the z-transform of the conjugate of p_r, which has support only in the negative half of the frequency plane. Then we can get the required filters by:

$$A(z) = 2\Re[P(z)] = P(z)+P^*(z) \tag{2.3.9}$$
$$B(z) = 2\Im[P(z)] = -j[P(z)-P^*(z)] \tag{2.3.10}$$
$$C(z) = 2\Re[Q(z)] = Q(z)+Q^*(z) \tag{2.3.11}$$
$$D(z) = -2\Im[Q(z)] = j[Q(z)-Q^*(z)] \tag{2.3.12}$$

litreview/images/overlaps_complex.png

Then:

$$\begin{aligned} A(W^k z)C(z)+B(W^k z)D(z) &= [P(W^k z)+P^*(W^k z)][Q(z)+Q^*(z)]+ \\ &\quad (-j*j)[P(W^k z)-P^*(W^k z)][Q(z)-Q^*(z)] \quad (2.3.13) \\ &= P(W^k z)Q(z)[1+1]+P^*(W^k z)Q(z)[1-1]+ \\ &\quad P(W^k z)Q^*(z)[1-1]+P^*(W^k z)Q^*(z)[1+1] \quad (2.3.14) \\ &= 2P(W^k z)Q(z)+2P^*(W^k z)Q^*(z) \quad (2.3.15) \end{aligned}$$

So now we only need to ensure that $P(W^k z)$ overlaps as little as possible with $Q(z)$. This is somewhat more manageable, the diagram below shows the problem.

2.4 Neural Networks

2.4.1 The Neuron and Single Layer Neural Networks

The neuron, shown in Figure **??** is the core building block of Neural Networks. It takes the dot product between an input vector $\mathbf{x} \in \mathbb{R}^N$ and a weight vector $\mathbf{b}$, before applying a chosen nonlinearity, f. I.e.

$$y = f(\langle \mathbf{x}, \mathbf{w} \rangle) = f\left(\sum_{i=0}^{N} x_i w_i\right)$$

where we have used the shorthand $b = w_0$ and $x_0 = 1$. Note that if $\langle \mathbf{w}, \mathbf{w} \rangle = 1$ then $\langle \mathbf{x}, \mathbf{w} \rangle$ is the distance from the point $\mathbf{x}$ to the hyperplane with normal $\mathbf{w}$. With general $\mathbf{w}$ this can be thought of as a scaled distance.

Typical nonlinear functions f are the sigmoid function: (**include figure**)

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

tanh function:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

or ReLU:

$$\mathrm{ReLU}(x) = \max(x, 0)$$

The weight vector $\mathbf{w}$ defines a hyperplane in $\mathbb{R}^N$ which splits the space into two. The choice of nonlinearity then affects how points on each side of the plane are treated. For a sigmoid, points far below the plane get mapped to 0 and points far above the plane get mapped to 1 (with points near the plane having a value of 0.5). For tanh nonlinearities, these points get mapped to -1 and 1. For ReLU nonlinearities, every point below the plane ($\langle \mathbf{x}, \mathbf{w} \rangle < 0$) gets mapped to zero and every point above the plane keeps its inner product value.

Consider a single neuron acting as logistic regressor. I.e. we have $\mathcal{D} = \{\mathbf{x}_i, y_i\}$ where $y_i \in \{0, 1\}$. We can easily find the maximum likelihood estimates for $\mathbf{w}$ by minimizing the loss

2.4.2 Loss or Error Term

2.4.3 Backpropagation

With a deep network like most CNNs, calculating $\frac{\partial L}{\partial w}$ may not seem particularly obvious if w is a weight in one of the lower layers. Say we have a deep network, with L layers. We need

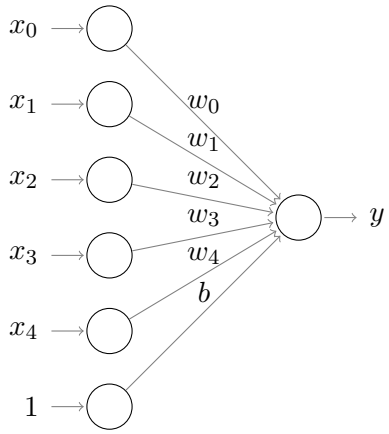


Figure 2.18: **A single neuron.** The neuron is composed of inputs x_i, weights w_i (and a bias term), as well as an activation function. Typical activation functions include the sigmoid function, tanh function and the ReLU

to define a rule for updating the weights in all L layers of the network, however, only the weights w_L are connected to the loss function, L. We assume for whatever function the last layer is that we can write down the derivative of the output with respect to the weights $\frac{\partial z_L}{\partial w_L}$. We can then write down the weight-update gradient, $\frac{\partial L}{\partial w_L}$ with application of the chain rule:

$$\frac{\partial L}{\partial w_L} = \frac{\partial L}{\partial z_L}\frac{\partial z_L}{\partial w_L} + \underbrace{\lambda w}_{\text{from the reg. loss}} \tag{2.4.1}$$

$\frac{\partial L}{\partial z_L}$ can be done simply from the equation of the loss function used. Typically this is parameterless.

Since all of the layers in a CNN are well-defined and differentiable[6] we assume that we can also write down what $\frac{\partial z_L}{\partial z_{L-1}}$ is. Repeating this process for the next layer down, we have:

$$\frac{\partial L}{\partial w_{L-1}} = \frac{\partial L}{\partial z_L}\frac{\partial z_L}{\partial z_{L-1}}\frac{\partial z_{L-1}}{\partial w_{L-1}} \tag{2.4.2}$$

We can generalize this easily like so:

$$\frac{\partial L}{\partial w_l} = \frac{\partial L}{\partial z_L} \underbrace{\prod_{i=L}^{l+1}\frac{\partial z_i}{\partial z_{i-1}}}_{\text{product to } l\text{'s output}} \frac{\partial z_l}{\partial w_l} \tag{2.4.3}$$

[6]The ReLU is not differentiable at its corner, but backpropagation can still be done easily by simply looking at the sign of the input.

So far we have defined the loss function for a given data point $(x^{(i)}, y^{(i)})$. Typically, we want our network to be able to generalize to the true real world joint distribution $P(x, y)$, minimizing the expected risk (R_E) of loss:

$$R_E(f(x,w)) = \int L(y, f(x,w)) dP(x,y) \tag{2.4.4}$$

Instead, we are limited to the training set, so we must settle for the empirical risk (R_{EMP}):

$$R_{EMP}(f(x,w)) = \frac{1}{N} \sum_{i=1}^{N} L(y^{(i)}, f(x^{(i)}, w)) \tag{2.4.5}$$

2.4.4 Gradient descent vs Stochastic Gradient Descent vs Mini-Batches

We can minimize Equation 2.4.5 with *gradient descent* rumelhart_parallel_1986. Updates can be made on a generic network parameter w with:

$$w_{t+1} = w_t - \eta \frac{\partial E_n}{\partial w} \tag{2.4.6}$$

where η is caleld the learning rate. Calculating the gradient $\frac{\partial E_n}{\partial w}$ is done by averaging the individual gradients $\frac{\partial L}{\partial w}$ over the entire training dataset. This can be very slow, particularly for large training sets.

Instead, we can learn far more quickly by using a single estimate for the weight update equation, i.e.,

$$w_{t+1} = w_t - \eta \frac{\partial L}{\partial w} \tag{2.4.7}$$

This is called *stochastic gradient descent*. Each weight update now uses a noisy estimate of the true gradient $\frac{\partial E_n}{\partial w}$. Carefully choosing the learning rate η update scheme can ensure that the network converges to a local minimum, but the process may not be smooth (the empirical risk may fluctuate, which could be interpreted as the network diverging).

An often used trade off between these two schemes is called *mini-batch gradient descent*. Here, the variance of the estimate of $\frac{\partial E_n}{\partial w}$ is reduced by averaging out the point estimate $\frac{\partial L}{\partial w}$ over a mini-batch of samples, size N_B. Typically $1 << N_B << N$, with N_B usually being around 128. This number gives a clue to another benefit that has seen the use of mini-batches become standard — they can make use of parallel processing. Instead of having to wait until the gradient from the previous data point was calculated and the network weights are updated, a network can now process N_B samples in parallel, calculating gradients for each point with the same weights, average all these gradients in one quick step, update the weights, and continue on with the next N_B samples. The update equation is now:

$$w_{t+1} = w_t - \eta \sum_{n=1}^{N_B} \frac{\partial L}{\partial w} \tag{2.4.8}$$

2.4.4.1 Loss Functions

For a given sample $q = (x, y)$, the loss function is used to measure the cost of predicting $\hat{y}$ when the true label was y. We define a loss function $L(y, \hat{y})$. A CNN is a deterministic function of its weights and inputs, $f(x, w)$ so this can be written as $L(y, f(x, w))$, or simply $L(y, x, w)$.

It is important to remember that we can choose to penalize errors however we please, although for CNNs, the vast majority of networks use the same loss function — the softmax loss. Some networks have experimented with using the hinge loss function (or 'SVM loss'), stating they could achieve improved results gu_recent_2015,tang_deep_2013.

1. *Softmax Loss:* The more common of the two, the softmax turns predictions into non-negative, unit summing values, giving the sense of outputting a probability distribution. The softmax function is applied to the C outputs of the network (one for each class):

$$p_j = \frac{e^{(f(x,w))(j)}}{\sum_{k=1}^{C} e^{(f(x,w))(k)}} \tag{2.4.9}$$

   where we have indexed the c-th element of the output vector $f(x, w)$ with $(f(x, w))(c)$. The softmax *loss* is then defined as:

$$L(y_i, x, w) = \sum_{j=1}^{C} \mathbb{1}\{y_i = j\} \log p_j \tag{2.4.10}$$

   Where $\mathbb{1}$ is the indicator function, and y_i is the true label for input i.

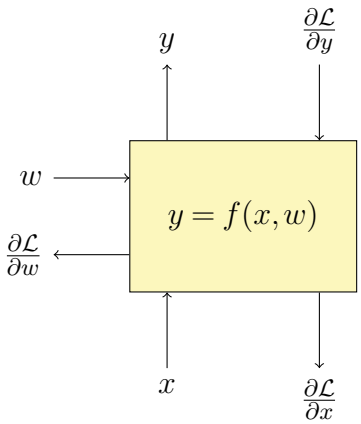
2. *Hinge Loss:* The same loss function from Support Vector Machines (SVMs) can be used to train a large margin classifier in a CNN:

$$L(y, x, w) = \sum_{l=1}^{C} \Big[\max(0, 1 - \delta(y_i, l) w^T x_i)\Big]^p \tag{2.4.11}$$

   Using a hinge loss like this introduces extra parameters, which would typically replace the final fully connected layer. The p parameter in Equation 2.4.11 can be used to choose ℓ^1 Hinge-Loss, or ℓ^2 Squared Hinge-Loss.

2.4.4.2 Regularization

Weight regularization, such as an ℓ^2 penalty is often given to the learned parameters of a system. This applies to the parameters of the fully connected, as well as the convolutional layers in a CNN. These are added to the loss function. Often the two loss components are differentiated between by their monikers - 'data loss' and 'regularization loss'. The above



equation then becomes:

$$\mathcal{L} = \mathcal{L}_{data} + \underbrace{\frac{1}{2}\lambda_{fc} \sum_{i=1}^{L_{fc}} \sum_{j} \sum_{k} (w_i[j,k])^2}_{\text{fully connected loss}} + \underbrace{\frac{1}{2}\lambda_c \sum_{i=1}^{L_c} \sum_{u_1} \sum_{u_2} \sum_{d} \sum_{n} (f_i[u_1, u_2, d, n])^2}_{\text{convolutional loss}} \tag{2.4.12}$$

Where λ_{fc} and λ_c control the regularization parameters for the network. These are often also called 'weight decay' parameters.

The choice to split the λ's between fully connected and convolutional layers was relatively arbitrary. More advanced networks can make λ a function of the layer.

2.4.5 Learning

2.4.6 Optimization

2.4.7 Extending to multiple layers and different block types

2.4.7.1 Convolutional Layers

The image/layer of features is convolved by a set of filters. The filters are typically small, ranging from 3×3 in ResNet and VGG to 11×11 in AlexNet. We have quoted only spatial size here, as the filters in a CNN are always *fully connected in depth* — i.e., they will match the number of channels their input has.

For an input $\mathbf{x} \in \mathbb{R}^{H \times W \times D}$, and filters $\mathbf{f} \in \mathbb{R}^{H' \times W' \times D \times D''}$ (D'' is the number of filters), our output $\mathbf{z} \in \mathbb{R}^{H'' \times W'' \times D''}$ will be given by:

$$z[u_1, u_2, d''] = b[d''] + \sum_{i=-\frac{H'}{2}}^{\frac{H'}{2}-1} \sum_{j=-\frac{W'}{2}}^{\frac{W'}{2}-1} \sum_{k=0}^{D-1} f[i, j, k, d''] x[u_1 - i, u_2 - j, k] \tag{2.4.13}$$

Figure 2.19: Differences in non-linearities. Green — the *sigmoid* function, Blue — the *tanh* function, and Red — the *ReLU*. The ReLU solves the problem of small gradients outside of the activation region (around $x = 0$) as well as promoting sparsity.

Figure 2.20: **??** Tight 2×2 pooling with stride 2, vs **??** overlapping 3×3 pooling with stride 2. Overlapping pooling has the possibility of having one large activation copied to two positions in the reduced size feature map, which places more emphasis on the odd columns.

2.4.7.2 ReLUs

Activation functions, neurons, or non-linearities, are the core of a neural networks expressibility. Historically, they were sigmoid or tanh functions, but these have been replaced recently by the Rectified Linear Unit (ReLU), which has equation $g(x) = \max(0, x)$. A ReLU non-linearity has two main advantages over its smoother predecessors glorot_deep_2011, nair_rectified_2010.

1. It is less sensitive to initial conditions as the gradients that backpropagate through it will be large even if x is large. A common observation of sigmoid and tanh non-linearities was that their learning would be slow for quite some time until the neurons came out of saturation, and then their accuracy would increase rapidly before levelling out again at a minimum glorot_understanding_2010. The ReLU, on the other hand, has constant gradient.

2. It promotes sparsity in outputs, by setting them to a hard 0. Studies on brain energy expenditure suggest that neurons encode information in a sparse manner. lennie_cost_2003 estimates the percentage of neurons active at the same time to be between 1 and 4%. Sigmoid and tanh functions will typically have *all* neurons firing, while the ReLU can allow neurons to fully turn off.

2.4.7.3 Pooling

Typically following a convolutional layer (but not strictly), activations are subsampled with max pooling. Pooling adds some invariance to shifts smaller than the pooling size at the cost of information loss. For this reason, small pooling is typically done often 2×2 or 3×3, and the invariance to larger shifts comes after multiple pooling (and convolutional) layers.

While initial designs of max pooling would do it in non-overlapping regions, AlexNet used 3×3 pooling with stride 2 in their breakthrough design, quoting that it gave them an increase in accuracy of roughly 0.5% and helped prevent their network from 'overfitting'. More recent networks will typically employ either this or the original 2×2 pooling with stride 2, see Figure 2.20. A review of pooling methods in mishkin_systematic_2016 found them both to perform equally well.

2.4.7.4 Batch Normalization

Batch normalization proposed only very recently in ioffe_batch_2015 is a conceptually simpler technique. Despite that, it has become quite popular and has been found to be very useful. At its core, it is doing what standard normalization is doing, but also introduces two learnable parameters — scale (γ) and offset (β). **??** becomes:

$$\tilde{z}(u_1, u_2, d) = \gamma \frac{z - E[z]}{\sqrt{Var[z]}} + \beta \tag{2.4.14}$$

These added parameters make it a *renormalization* scheme, as instead of centring the data around zero with unit variance, it can be centred around an arbitrary value with arbitrary variance. S etting $\gamma = \sqrt{Var[z]}$ and $\beta = E[z]$, we would get the identity transform. Alternatively, setting $\gamma = 1$ and $\beta = 0$ (the initial conditions for these learnable parameters), we get standard normalization.

The parameters γ and β are learned through backpropagation. As data are usually processed in batches, the gradients for γ and β are calculated per sample, and then averaged over the whole batch.

From Equation 2.4.14, let us briefly use the hat notation to represent the standard normalized input: $\hat{z} = (z - E[z])/\sqrt{Var[z]}$, then:

$$\begin{aligned} \tilde{z}^{(i)} &= \gamma \hat{z}^{(i)} + \beta \\ \frac{\partial \mathcal{L}}{\partial \gamma} &= \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}^{(i)}}{\partial \tilde{z}^{(i)}} \cdot \hat{z}^{(i)} \qquad (2.4.15) \\ \frac{\partial \mathcal{L}}{\partial \beta} &= \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}^{(i)}}{\partial \tilde{z}^{(i)}} \qquad (2.4.16) \end{aligned}$$

Batch normalization layers are typically placed *between* convolutional layers and non-linearities. I.e., if $Wu + b$ is the output of a convolutional layer, and $z = g(Wu + b)$ is the output of the non-linearity, then with the batch normalization step, we have:

$$\begin{aligned} z &= g(BN(Wu + b)) \\ &= g(BN(Wu)) \end{aligned} \tag{2.4.17}$$

Where the bias term was ignored in the convolutional layer, as it can be fully merged with the 'offset' parameter β.

This has particular benefit of removing the sensitivity of our network to our initial weight scale, as for scalar a,

$$BN(Wu) = BN((aW)u) \tag{2.4.18}$$

Figure 2.21: A residual unit. The identity mapping is always present, and the network learns the difference from the identity mapping, $\mathcal{F}(x)$. Taken from he_deep_2015.

It is also particularly useful for backpropagation, as an increase in weights leads to *smaller* gradients ioffe_batch_2015, making the network far more resilient to the problems of vanishing and exploding gradients:

$$\begin{aligned} \frac{\partial BN((aW)u)}{\partial u} &= \frac{\partial BN(Wu)}{\partial u} \\ \frac{\partial BN((aW)u)}{\partial (aW)} &= \frac{1}{a} \cdot \frac{\partial BN(Wu)}{\partial W} \end{aligned} \tag{2.4.19}$$

2.5 Relevant Architectures

2.5.1 LeNet

2.5.2 AlexNet

2.5.3 VGG

2.5.4 Residual Networks

The current state of the art design introduced a clever novel feature called a residual unithe_deep_2015,he_identity_2016. The inspiration for their design came from the difficulties experienced in training deeper networks. Often, adding an extra layer would *decrease* network performance. This is counter-intuitive as the deeper layers could simply learn the identity mapping, and achieve the same performance.

To promote the chance of learning the identity mapping, they define a residual unit, shown in Figure 2.21. If a desired mapping is denoted $\mathcal{H}(x)$, instead of trying to learn this, they instead learn $\mathcal{F}(x) = \mathcal{H}(x) - x$.

2.5.5 old

Convolutional Neural Networks (CNNs) were initially introduced by lecun_backpropagation_1989 in lecun_backpropagation_1989. Due to the difficulty of training and initializing them, they failed to be popular for more than two decades. This changed in 2012, when advancements in pre-training with unsupervised networks bengio_greedy_2007, the use of an improved non-linearity — the Rectified Linear Unit, or ReLU, new regularization methodshinton_improving_2012, and access to more powerful computers in graphics cards, or GPUs, allowed Krizhevsky, Sutskever and Hinton to develop AlexNetkrizhevsky_imagenet_2012. This network nearly halved the previous state of the art's error rate. Since then, interest in them has expanded very rapidly, and they have been successfully applied to object detec-

Figure 2.22: Standard CNN architecture. Taken from lecun_gradient-based_1998

tion ren_object_2015 and human pose estimation tompson_efficient_2015. It would take a considerable amount of effort to document the details of all the current enhancements and tricks many researches are using to squeeze extra accuracy, so for the purposes of this report we restrict ourselves to their generic design, with some time spent describing some of the more promising enhancements.

We would like to make note of some of the key architectures in the history of CNNs, which we, unfortunately, do not have space to describe:

- Yann LeCun's LeNet-5 lecun_gradient-based_1998, the state of the art design for postal digit recognition on the MNIST dataset.
- Google's GoogLeNet szegedy_going_2015 achieved 6.67% top-5 error on ILSVRC2014, introducing the new 'inception' architecture, which uses combinations of 1×1, 3×3 and 5×5 convolutions.
- Oxford's VGG simonyan_very_2014 — 6.8% and runner up in ILSVRC2014. The VGG design is very similar to AlexNet but was roughly twice as deep. More convolutional layers were used, but with smaller support — only 3×3. These were often stacked directly on top of each other without a non-linearity in between, to give the effective support of a 5×5 filter.
- Microsoft Research's ResNet he_deep_2015 achieved 4.5% top-5 error and was the winner of ILSVRC2015. This network we will talk briefly about, as it introduced a very nice novel layer — the residual layer.

Despite the many variations of CNN architectures currently being used, most follow roughly the same recipe (shown in Figure 2.22):

2.5.6 Fully Connected Layers

The convolution, pooling, and activation layers all conceptually form part of the *feature extraction* stage of a CNN. One or more fully connected layers are usually placed after these layers to form the *classifier*. One of the most elegant and indeed most powerful features of CNNs is this seamless connection between the *feature extraction* and *classifier* sub-networks, allowing the backpropagation of gradients through all layers of the entire network.

The fully connected layers in a CNN are the same as those in a classical Neural Network (NN), in that they compute a dot product between their input vector and a weight vector:

$$z_i = \sum_j W_{ij} x_j \tag{2.5.1}$$

The final output of the Fully Connected layer typically has the same number of outputs as the number of classes C in the classification problem.

2.6 Scatternets

Scatternets have been a very large influence on our work, as well as being quite distinct from the previous discussions on learned methods. They were first introduced by Bruna and Mallat in their work bruna_classification_2011, and then were rigorously defined by Mallat in mallat_group_2012. Several updates and newer models have since been released by Mallat's group, which we will review in this chapter.

It is helpful to introduce this chapter with one further note. Unlike the CNNs introduced in section 2.4, which were set up to minimize some cost function which had certain constraints to promote certain properties, the scattering operator may be thought of as an operator Φ which has some desirable properties for image understanding. These properties may ultimately help us minimize some cost function and improve our image understanding system, which we explore more in **??**.

2.7 Translation Invariant Scatternets

The translation invariant Scatternets were mostly covered in bruna_invariant_2013. This section summarises the method of this paper.

2.7.1 Defining the Properties

The first release of Scatternets aimed at building a translation invariant operator, which was also stable to additive noise and deformations. Translation is often defined as being uninformative for classification — an object appearing in the centre of the image should be treated the same way as an the same object appearing in the corner of an image, i.e., Φx is invariant to translations $x_c(\mathbf{u}) = x(\mathbf{u} - \mathbf{c})$[7] by $\mathbf{c} = (c_1, c_2) \in \mathbb{R}^2$ if

$$\Phi x_c = \Phi x \tag{2.7.1}$$

Stability to additive noise is another good choice to include in this operator, as it is a common feature in measured signals. Stability is defined in terms of Lipschitz continuity, which is a strong form of uniform continuity for functions, which we briefly introduce here.

Formally, a Lipschitz continuous function is limited in how fast it can change; there exists an upper bound on the gradient the function can take, although it doesn't necessarily need to

[7] here we adopt a slight variation on 's notation, by using boldface letters to represent vectors, as is the custom in Signal Processing

Figure 2.23: A Lipschitz continuous function is shown. There is a cone for this function (shown in white) such that the graph always remains entirely outside the cone as it's shifted across. The minimum gradient needed for this to hold is called the 'best Lipschitz constant'.

be differentiable everywhere. The modulus operator $|x|$ is a good example of a function that has a bounded derivative and so is Lipschitz continuous, but isn't differentiable everywhere.

Returning again to stability to additive noise, state that for a new signal $x'(\mathbf{u}) = x(\mathbf{u}) + \epsilon(\mathbf{u})$, there must exist a bounded $C > 0$ s.t.

$$\|\Phi x' - \Phi x\| \leq C\|x' - x\| \tag{2.7.2}$$

The final requirement is to be stable to small deformations. Enough so that we can ignore intra-class variations, but not so invariant that an object can morph into another (in the case of MNIST for example, we do not want to be so stable to deformations that 7s can map to 1s). Formally, for a new signal $x_\tau(\mathbf{u}) = x(\mathbf{u} - \tau(\mathbf{u}))$, where $\tau(\mathbf{u})$ is a non constant displacement field (i.e., not just a translation) that deforms the image, we require a $C > 0$ s.t.

$$\|\Phi x_\tau - \Phi x\| \leq C\|x\| \sup_{\mathbf{u}} |\nabla \tau(\mathbf{u})| \tag{2.7.3}$$

The term on the right $|\nabla \tau(\mathbf{u})|$ measures the deformation amplitude, so the supremum of it is a limit on the global defomation amplitude.

2.7.2 Finding the Right Operator

A Fourier modulus satisfies the first two of these requirements, in that it is both translation invariant and stable to additive noise, but it is unstable to deformations due to the infinite support of the sinusoid basis functions it uses. It also loses too much information — very different signals can all have the same Fourier modulus, e.g. a chirp, white noise and the Dirac delta function all have flat spectra.

Unlike the Fourier modulus, a wavelet transform is stable to deformations due to the grouping together frequencies into dyadic packets mallat_group_2012, however, the wavelet transform is not invariant to shifts.

We saw in **??** that the modulus of complex, analytic wavelets commuted with shifts. The real and imaginary parts are also commutative with shifts, but these vary much quicker than the modulus (Figure 2.24). Interestingly, the modulus operator, in this case, does not lose any information waldspurger_phase_2012 (due to the redundancies of the wavelet transform), which is why it may be nice to think of it as a *demodulator*.

Figure 2.24: Real, Imaginary and Modulus of complex wavelet convolved with an impulse.

The modulus can be made fully invariant by integrating, i.e.,:

$$\int Fx(\mathbf{u})d\mathbf{u} = \int |x * \psi_\lambda(\mathbf{u})|d\mathbf{u}$$

is translation invariant. Total invariance to shifts means integrating over the entire function, which may not be ideal as it loses a significant amount of information in doing this. Instead Bruna and Mallat define scales 2^J, over which their operator is invariant to shifts. Now instead of integrating, the output $\|x * \psi_\lambda\|$ is convolved with an averaging window, or conveniently, the scaling function for the chosen wavelet:

$$\phi_{2^J}(\mathbf{u}) = 2^{-2J}\phi(2^{-J}\mathbf{u})$$

Even still, this averaging means that a lot of information is lost from the first layer outputs ($\|x * \psi_\lambda\|$). Bruna and Mallat combat this by also convolving the output with wavelets that cover the rest of the frequency space, giving

$$U[p]x = U[\lambda_2]U[\lambda_1]x = \||x * \psi_{\lambda_1}| * \psi_{\lambda_2}\|$$

The choice of wavelet functions λ_1 and λ_2 is combined into a path variable, $p = (\lambda_1, \lambda_2, \ldots \lambda m)$.

Local invariants can be again computed by convolving this with another scaling function ϕ. The result is now a multiscale scattering transform, with coefficients:

$$S[p]x = U[p]x * \phi_{2^J}(\mathbf{u})$$

A graphical representation of this is shown in Figure 2.25.

2.8 Rotation and Translation Invariant Scatternets

Mallat's group refined their Scatternet architecture by expanding their list of invariants to also include rotation. They also experimented with adding scale invariance in sifre_rotation_2013, but it was limited to only averaging over scale once, and they were no longer using it in oyallon_deep_2015, so for brevity we omit it.

This work was done by two authors, each tackling different challenges. The first is texture analysis with Sifre in sifre_combined_2012, sifre_rotation_2013, sifre_rigid-motion_2014, sifre_rigid-motion_2014-1, and the second is image classification with Oyallon in oyallon_generic_2013, oyallon_deep_2015. In this section, we outline the properties and structure of this extended Scatternet.

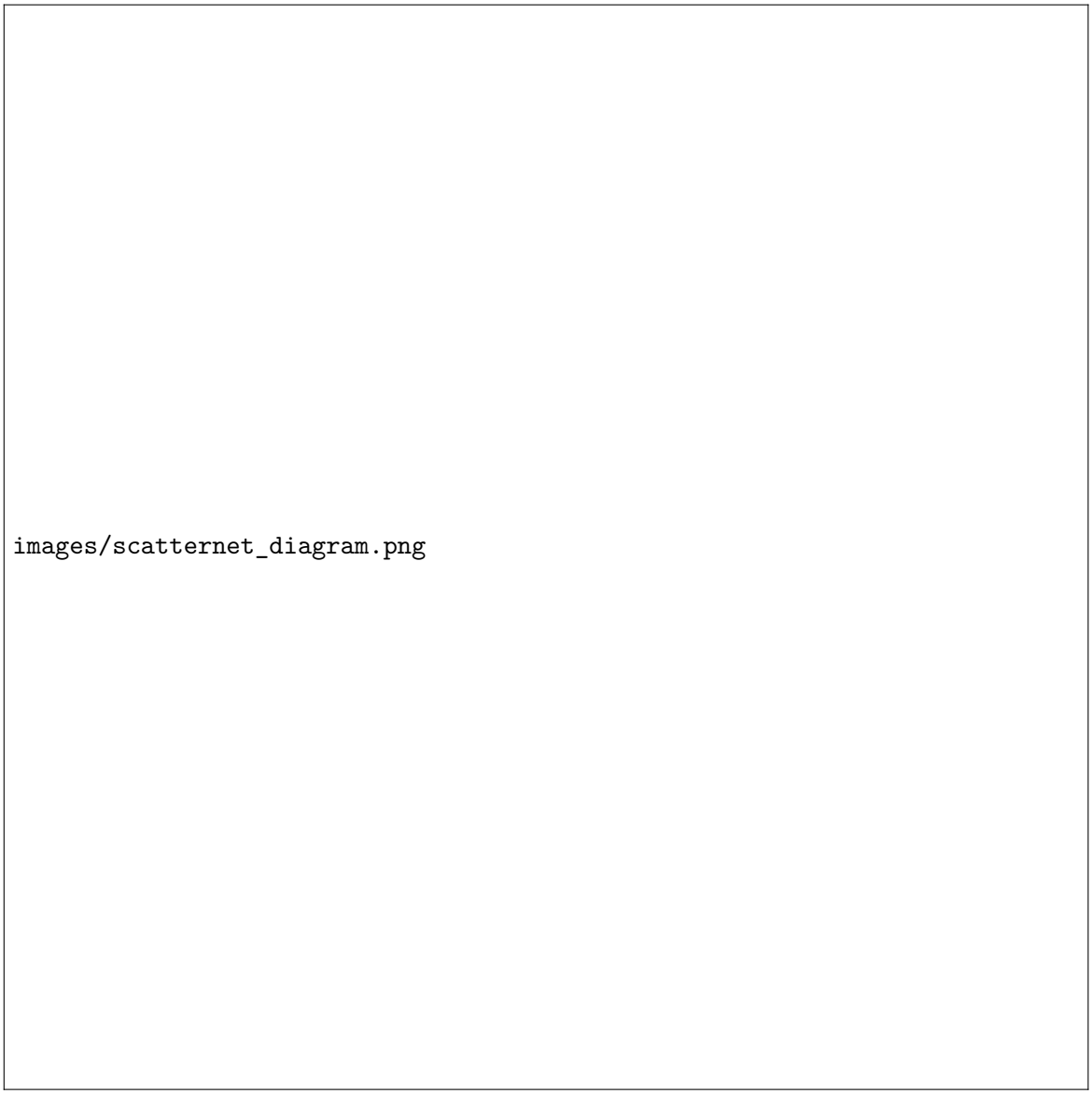


Figure 2.25: The translation invariant Scattering Transform. Scattering outputs are the leftward pointing arrows $S[p]x$, and the intermediate coefficients $U[p]x$ are the centre nodes of the tree. Taken from bruna_invariant_2013.

2.8.1 An Important note on Joint vs. Separable Invariants

When building multiple invariants, some thought must be given as to how to combine them — separably or jointly? Let us call the group of operations we want to be invariant to G, with $g \in G$ a single realization from this group — in this case, G is the group of affine transformations. We want our operator Φ to be invariant to all $g \in G$, i.e., $\Phi(gx) = \Phi(x)$. Building separable invariants would mean representing the group as $G = G_2 G_1$ (an assumption of the group, not of our model), and building $\Phi = \Phi_2 \Phi_1$, where Φ_1 is invariant to members of G_1 and covariant to members of G_2, and Φ_2 is invariant to members of G_2. I.e.,

$$\Phi_2(\Phi_1(g_1 g_2 x)) = \Phi_2(g_2 \Phi_1(x)) = \Phi_2(\Phi_1(x)) \tag{2.8.1}$$

An example of this would be in the group G of 2D translations, building horizontal invariance first, then building vertical invariance second. warn about this approach, however, as it cannot capture the action of G_2 relative to G_1. In the case of veritcal and horizontal translations, for example, it would not be able to distinguish if the patterns had moved apart as well as being shifted, whereas a joint horizontal and vertical translation invariant would be able to distinguish these two cases.

In this vein, suggest that in the case of rotation and translation invariance, a joint invariant should be used, building on the work in citti_cortical_2006, boscain_anthropomorphic_2010, sgallari_scale_2007.

2.8.2 Defining the Properties

A translation $g = (v, \theta)$ of the roto-translation group G_{rt} acting on $\mathbf{u} \in \mathbb{R}^2$ combines translation by v and rotation by R_θ as:

$$g\mathbf{u} = v + R_\theta \mathbf{u} \tag{2.8.2}$$

The product of two successive roto-translations $h = (v', \theta')$ and $g = (v, \theta)$is:

$$gh = (v + R_\theta v', \theta + \theta') \tag{2.8.3}$$

In much the similar approach to the simple translation invariant Scatternet defined above, calculate successive layers of signal coefficients $U[p]x$ that are covariant to the actions of all $g \in G_{rt}$ — i.e.,

$$U[p](gx) = gU[p]x \tag{2.8.4}$$

Creating invariants of order $m = \text{length}(p) = \text{length}([\lambda_1, \lambda_2, \ldots, \lambda_m])$ is then done by averaging $hU[p]x$ for all h in G_{rt}

$$S[p]x(g) = \sum_{h \in G_{rt}} hU[p]x\Phi_J(h^{-1}g) \tag{2.8.5}$$

This convolution averages $hU[p]x$ over all rotation angles in a spatial neighbourhood of $\mathbf{u}$ of size proportional to 2^J.

2.8.3 The Operator

2.8.3.1 Roto-Translation Invariance

Although we want to have a joint invariant for rotations and translations, this can be done with a cascade of wavelet transforms — so long as the final averaging operation is done over both rotation and translation. do just this, building a 3 layer scattering transform, the first layer of which is exactly identical to the previous translation scattering transform, i.e.,

$$\tilde{W}_1 x = (x * \phi_J, \{|x * \psi_{\theta,j}|\}) = (S_0 x, U_1 x) \tag{2.8.6}$$

The second and third layers are, however, new. The invariant part of U_1 is computed with an averaging over spatial and angle variables. *This averaging is implemented at fixed scales j* (see our note earlier about choosing separable scale invariance). For an action $g = (v, \theta)$, the averaging kernel is defined as:

$$\Phi_J(g) = \bar{\phi}(\theta) * \phi_J(u) \tag{2.8.7}$$

Where $\phi_J(u)$ is a kernel that averages each $U_1 x$ over scale 2^J, and $\bar{\phi}(\theta = (2\pi)^{-1})$ averages the result of that average over all angles.

To clarify, we look at an example architecture with $J = 2$ scales and $L = 4$ orientations. The output of the first layer $U_1 x$ would be a set of coefficients:

$$U_1 x = \{|x * \psi_{j,\theta}| \,|\, j = \{0,1\}, \theta = k\pi/4, k = \{0,1,2,3\},\} \tag{2.8.8}$$

i.e., there would be 4 high frequency coefficients, which were created with wavelets centred at $|\boldsymbol{\omega}| = 3\pi/4$, and 4 medium frequency components created with wavelets centred at $|\boldsymbol{\omega}| = 3\pi/8$. Each of these 8 will be averaged across the entire image, then each pair of 4 will be averaged across all 4 rotations, leaving 2 invariants.

To recover the information lost from averaging, also convolve $U_1 x$ with corresponding rotation and scale wavelets to pass on the high frequency information. These roto-translation wavelets, while joint, can also be computed with the cascade of separable wavelets. It may be helpful to consider the spatial variable $\mathbf{u}$ as single dimensional, and consider the rotation variable θ as a second dimension. The above equation calculated the low-low frequency component of these two variables, the remaining components are the low-high, high-low, and high-high.

Figure 2.26: Three dimensional convolution with $\Psi_{\theta_m,j_m,k_m}(u_1,u_2,\theta)$ factorised into a two dimensional convolution with $\psi_{\theta_m,j_m}(u_1,u_2)$ and a one dimensional convolution with $\psi_{k_m}(\theta)$. Colours represent the amplitude of the 3D wavelet. Image taken from sifre_rotation_2013.

We define the low frequency spatial scaling functions $\phi_J(u)$[8], the spatial wavelets $\psi_{\theta,j}(u)$, the rotation scaling function $\bar{\phi}(\theta)$ (which is just the constant $(2\pi)^{-1}$, but we write out in generic form nonetheless), and the rotation wavelet $\bar{\psi}_k(\theta)$, which is a 2π periodic wavelet.

Then, the remaining low-high, high-low, and high-high information is:

$$\Psi_{0,J,k_2}(u,\theta) = \phi_J(u) * \bar{\psi}_{k_2}(\theta) \tag{2.8.9}$$

$$\Psi_{\theta_2,j_2,}(u,\theta) = \psi_{\theta_2,j_2}(u) * \bar{\phi}(\theta) \tag{2.8.10}$$

$$\Psi_{\theta_2,j_2,k_2}(u,\theta) = \psi_{\theta_2,j_2}(u) * \bar{\psi}_{k_2}(\theta) \tag{2.8.11}$$

The k parameter is newly introduced here, and it represents the number of scales the rotation wavelet has (a typical value used by was $K = 3$). We call this combined operator Ψ_{θ_m,j_m,k_m}. See Figure 2.26 for what this looks like.

The wavelet-modulus operator then is:

$$\tilde{W}_m Y = (Y * \Phi_J(g), |Y * \Psi_{\theta_m,j_m,k_m}(g)|) \tag{2.8.12}$$

[8] we temporarily drop the boldface from the spatial parameter u to make it clearer it can be considered as single dimensional

for $m \geq 2$ and the final third order roto-translation Scatternet is:

$$Sx = (x * \phi_J(\mathbf{u}), U_1 x * \Phi_J(p_1), U_2 x * \Phi_J(p_2)) \tag{2.8.13}$$

with $p_1 = (\mathbf{u}, \theta_1, j_1)$ and $p_2 = (\mathbf{u}, \theta_1, j_1, \theta_2, j_2, k_2)$.

Chapter 3

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 we 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* **pytorch_wavelets**.

In parallel with our 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.

3.1 The Design Constraints

The original authors implemented their ScatterNet in matlab [21] 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 [22] or Pytorch [23].
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 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 can do Scattering in

PyTorch called KyMatIO[24], 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 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 this new one, as it provides some interesting insights into some of the design choices that can be made with a ScatterNet.

3.2 A Brief Description of Autograd

As part of modern deep learning framework, we need to define functional units like the one shown in Figure 3.1. In particular, not only must we be able to calculate the forward evaluation of a block given an input x and possible some learnable weights w, $y = f(x, w)$, 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 may involve 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 *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 3.1. We name these `NG:<name>`, NG for no gradients.

2. Autograd blocks - the modules as shown in Figure 3.1. These always have a forward and backward pass, and are named `AG:<name>`. E.g. Algorithm 3.2.

3. Higher level modules - these make use of efficient autograd functions and are named `MOD:<name>`. E.g. Algorithm 3.3.

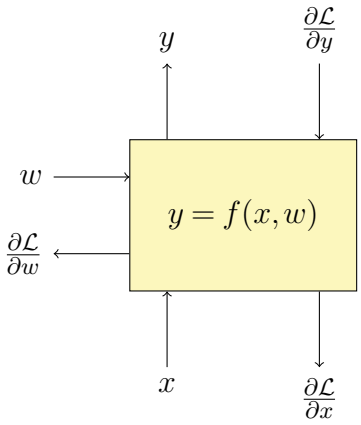


Figure 3.1: **An autograd block.** The building block of modern deep learning frameworks. Every function needs to be able to not only take an input (and possibly a weight parameter) to give an output $y = f(x, w)$, it must also be able to take a backpropagated input $\frac{\partial \mathcal{L}}{\partial y}$ and give backpropagated outputs $\frac{\partial \mathcal{L}}{\partial x}$, $\frac{\partial \mathcal{L}}{\partial w}$. The former is then passed onto the previous block's backpropagated input, and the latter, if it exists, is used to update weights. As a key step in doing this, the gradients $\frac{\partial y}{\partial x}$, $\frac{\partial y}{\partial w}$ evaluated at the current values of x and w must be saved on the forward pass. This takes up memory which can then be freed once the backpropagated outputs have been calculated.

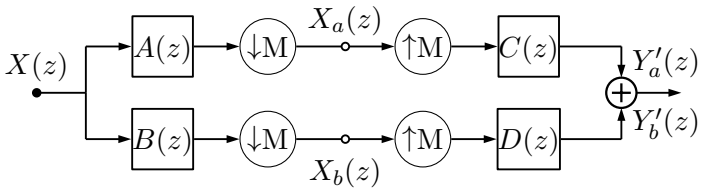


Figure 3.2: **Block Diagram of 2-D DWT.** The components of a filter bank DWT in two dimensions.

3.3 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 [25] or second generation wavelets, to the direct convolutional methods. [26], [27] are two notable such publications, both of which find that the convolution based implemetation 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.

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

$$\Delta X(z) = H(z^{-1})\Delta Y(z) \tag{3.3.1}$$

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 3.2 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.

3.3.2 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`.

3.3.2.1 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 $H \times W$.

3.3.2.2 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 steps are described in Algorithm 3.1. We do not need to define backpropagation functions for the `afb1d` and `sfb1d` functions as they are each others backwards step.

3.3.2.3 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 3.2. Note that we have allowed for different row and column filters in Algorithm 3.2. Most commonly used wavelets will use

[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 3.1 1-D analysis and synthesis stages of a DWT

function NG: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
  $lo \leftarrow$ `conv2d_down`(x, h_0)
  $hi \leftarrow$ `conv2d_down`(x, h_1)
  return $lo,\ hi$
end function

function NG: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
  $x \leftarrow$ `conv2d_up`$(lo,\ g_0) +$ `conv2d_up`$(hi,\ g_1)$
  return x
end function

the same filter for both directions (e.g. the Daubechies family), but later when we use the DTℂWT, we want to have different horizontal and vertical filtering.

The inverse transform logic is moved to the appendix **??**. 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 3.2.2. 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 3.2 (**??**) multiple times on the lowpass output (reconstructed image). Again, no intermediate activations need be saved, giving this implementation almost no memory overhead.

3.4 Fast Calculation of the DTℂWT

We have built upon previous implementations of the DTℂWT, in particular [28]–[30]. 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

Algorithm 3.2 2-D DWT and its gradient

function AG:DWT:FWD(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^c, h_1^c, $mode$, $axis = -2$) ▷ column filter
  ll, $lh \leftarrow$ afb1d(lo, h_0^r, h_1^r, $mode$, $axis = -1$) ▷ row filter
  hl, $hh \leftarrow$ afb1d(hi, h_0^r, h_1^r, $mode$, $axis = -1$) ▷ row filter
  return ll, lh, hl, hh
end function

function AG:DWT:BWD(Δll, Δlh, Δhl, Δhh)
  load h_0^c, h_1^c, h_0^r, h_1^r, $mode$
  h_0^c, $h_1^c \leftarrow$ flip(h_0^c), flip(h_1^c) ▷ flip the filters as in (3.3.1)
  h_0^r, $h_1^r \leftarrow$ flip(h_0^r), flip(h_1^r)
  $\Delta lo \leftarrow$ 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, $mode$, $axis = -1$)
  return Δx
end function

at each scale are added and subtracted from each other to get the six orientations' real and imaginary components **??**. 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 DT$\mathbb{C}$WT is the need to use different filters for the first scale to all subsequent scales [4]. We have not shown this in Algorithm 3.3 for simplicity, but it would simply mean we would have to handle the $j = 0$ case separately.

3.5 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:

$$r = |z| = \sqrt{x^2 + y^2} \tag{3.5.1}$$

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

$$\frac{\partial r}{\partial x} = \frac{x}{\sqrt{x^2 + y^2}} = \frac{x}{r} \tag{3.5.2}$$

$$\frac{\partial r}{\partial y} = \frac{y}{\sqrt{x^2 + y^2}} = \frac{y}{r} \tag{3.5.3}$$

Algorithm 3.3 2-D DTℂWT

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

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:

$$\nabla^2 r(x,y) = \frac{1}{r^3}\begin{bmatrix} y^2 & -xy \\ -xy & x^2 \end{bmatrix} = \frac{1}{r^3}\begin{bmatrix} y \\ -x \end{bmatrix}\begin{bmatrix} y & -x \end{bmatrix} \geq 0 \tag{3.5.4}$$

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

$$\begin{aligned} \Delta z &= \Delta r \frac{\partial r}{\partial x} + j\Delta r \frac{\partial r}{\partial y} && (3.5.5)\\ &= \Delta r \frac{x}{r} + j\Delta r \frac{y}{r} && (3.5.6)\\ &= \Delta r e^{j\theta} && (3.5.7) \end{aligned}$$

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 3.4.

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:

$$r_s = \sqrt{x^2 + y^2 + b^2} - b \tag{3.5.8}$$

Algorithm 3.4 Magnitude forward and backward steps

```
function AG:MAG:FWD(x, y)
    r ← sqrt(x^2 + y^2)
    θ ← arctan2(y, x)                          ▷ arctan2 handles x = 0
    save θ
    return r
end function

function AG:MAG:BWD(Δr)
    load θ
    Δx ← Δr cos θ                              ▷ Reinsert phase
    Δy ← Δr sin θ                              ▷ Reinsert phase
    return Δx, Δy
end function
```

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

$$\frac{\partial r_s}{\partial x} = \frac{x}{\sqrt{x^2+y^2+b^2}} = \frac{x}{r_s} \tag{3.5.9}$$

$$\frac{\partial r_s}{\partial y} = \frac{y}{\sqrt{x^2+y^2+b^2}} = \frac{y}{r_s} \tag{3.5.10}$$

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. **??** 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 3.5. 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. 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.

3.6 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 [21] and the newly developed KyMatIO [24]? Table 3.1 lists the different properties and options of the competing packages.

Algorithm 3.5 DT$\mathbb{C}$WT **ScatterNet Layer.** High level block using the above autograd functions to calculate a first order scattering

function MOD: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
  $U_1 \leftarrow \text{mag}(\text{Re}(yh),\ \text{Im}(yh))$
  $Z \leftarrow \text{concatenate}(S_0,\ U_1)$ ▷ stack 1 lowpass with 6 magnitudes
  return Z
end function

Table 3.1: **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[21]	FFT-based	Flexible	Periodic	No	No
KyMatIO[24]	FFT-based	Flexible	Periodic	Yes	Yes
DT$\mathbb{C}$WT Scat	Separable filter banks	6	Flexible	Yes	Yes

3.6.1 Speed and Memory Use

Table 3.2 lists the speed of the various transforms as tested on our reference architecture **??**. 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× faster than the Fourier-based KyMatIO. Type B uses shorter filters and the cheaper zero padding scheme, and achieves a 35× 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 3.3–section 3.5.

3.6.2 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 [31], [32]. This puts two layers of a scattering transform at the front end of a deep learning network. In addition to comparing our DT$\mathbb{C}$WT 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 3.2: **Comparison of speed for the forward and backward passes of the competing ScatterNet Implementations.** Tests were run on the reference architecture described in **??**. 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[21]	> 200.00	n/a	n/a	n/a
KyMatIO[24]	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

- Tiny ImageNet[33]: 200 classes, 500 images per class, 64×64 pixels per image.

Table 3.3 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.

3.6.2.1 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 (3.5.8)) for the DT$\mathbb{C}$WT ScatterNet. We test these over a grid of values described in Table 3.4. 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 3.3. Interestingly, for all three datasets the shorter wavelet outperformed the longer wavelets.

Table 3.3: **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, $w \in \mathbb{R}^{2C\times N_c}$		

3.6.2.2 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 3.5. 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.

3.7 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 [34]. We hope that this will reduce some of the barriers we faced in using wavelets and Scattering in deep learning.

Table 3.4: **Hyperparameter settings for the** DT$\mathbb{C}$WT **scatternet.**

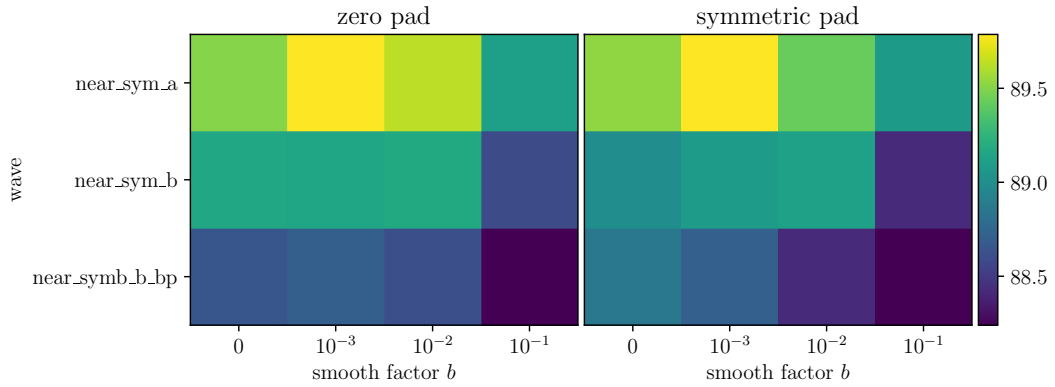
Hyperparameter	Values
Wavelet	near_sym_a 5,7 tap filters, near_sym_b 13,19 tap filters, near_sym_b_bp 13,19 tap filters
Padding Scheme	symmetric zero
Magnitude Smoothing b	0 1e-3 1e-2 1e-1

Table 3.5: **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.

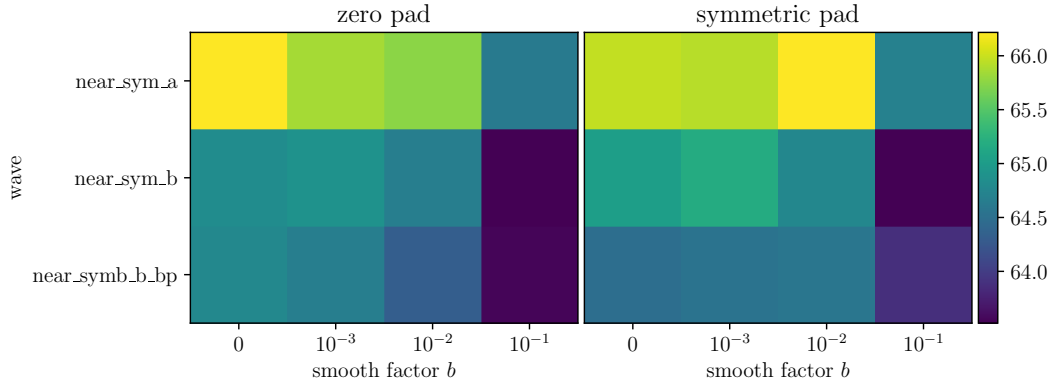
Type	CIFAR-10		CIFAR-100		Tiny ImgNet	
	Acc. (%)	Time (h)	Acc. (%)	Time (h)	Acc. (%)	Time (h)
Morlet8	88.6	3.4	65.3	3.4	57.6	5.6
Morlet6	89.1	2.4	65.7	2.4	57.5	4.4
DT$\mathbb{C}$WT	89.8	1.1	66.2	1.1	57.3	2.7

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 [24].

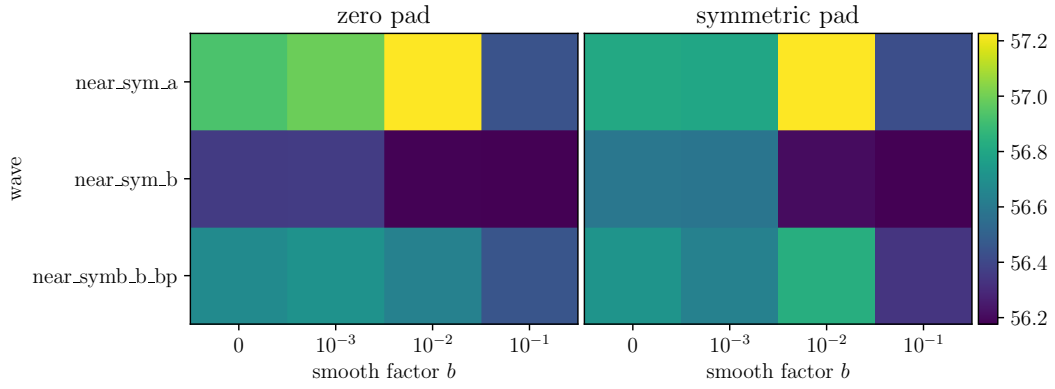
Additionally, we have experimentally verified that using a different complex wavelet backend does not have a negative impact on the performance of the ScatterNet as a frontend to Hybrid networks.



(a) CIFAR-10



(b) CIFAR-100



(c) Tiny ImageNet

Figure 3.3: **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 3.4. 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.

Chapter 4

Visualizing and Improving Scattering Networks

Deep Convolutional Neural Networks have become the *de facto* solution for image understanding tasks since the rapid development in their growth in recent years. Since AlexNet**??** in 2012, there have been many new improvements in their design, taking them 'deeper', such as the VGG network in 2014 [35], the Inception Network [36], Residual Networks in 2016 [37] and DenseNets in [38]. Each iteration has abstracted the inner workings of the model more and more.

Despite their success, they are often criticized for being 'black box' methods. Indeed, once you train a CNN, you can view the first layer of filters quite easily (see **??**) as they exist in RGB space. Beyond that things get trickier, as the filters have a third, 'depth' dimension typically much larger than its two spatial dimensions, and representing these dimensions with different colours becomes tricky and uninformative.

This has started to become a problem, and while we are happy to trust modern CNNs for isolated tasks, we are less likely to be comfortable with them driving cars through crowded cities, or making executive decisions that affect people directly. A commonly used contrived example, it is not hard to imagine a deep network that could be used to assess whether giving a bank loan to an applicant is a safe investment. Trusting a black box solution is deeply unsatisfactory in this situation. Not only from the customer's perspective, who, if declined, has the right to know why goodman_european_2016, but also from the bank's — before lending large sums of money, most banks would like to know why the network has given the all clear. 'It has worked well before' is a poor rule to live by.

A recent paper titled 'Why Should I Trust You?' [39] explored this concept in depth. They rated how highly humans trusted machine learning models. Unsurprisingly, a model with an interpretable methodology was trusted more than those which did not have one, even if it had a lower prediction accuracy on the test set. To build trust and to aid training, we need to probe these networks and visualize how and why they are making decisions.

Some good initial work has been done in this area. Zeiler and Fergus [40] design a DeConvNet to visualize what a filter in a given layer is mostly highly activated by. [41] learn to invert representations by updating a noisy input until its latent feature vector matches a desired target. [42] develop saliency maps by **Finish me**.

ScatterNets have been shown to perform very well as image classifiers. In particular, they can outperform CNNs for classification tasks with reduced training set sizes, e.g. in CIFAR-10 and CIFAR-100 (Table 6 from [32] and Table 4 from [43]). They are also near state-of-the-art for Texture Discrimination tasks (Tables 1–3 from [44]). Despite this, there still exists a considerable gap between them and CNNs on challenges like CIFAR-10 with the full training set (83% vs. 93%). Even considering the benefits of ScatterNets, this gap must be addressed.

While ScatterNets have good theoretical foundation and properties [45], it is difficult to understand the second order scattering. In particular, how useful are the second order coefficients for training? How similar are the scattered features to a modern state of the art convolutional network? To answer these questions, this chapter interrogates ScatterNet frontends. Taking inspiration from the interrogative work applied to CNNs, we build a DeScatterNet to visualize what the second order features are. We also heuristically probe a trained Hybrid Network and quantify the importance of the individual features.

Scattering transforms, or ScatterNets, have recently gained much attention and use due to their ability to extract generic and descriptive features in well defined way. They can be used as unsupervised feature extractors for image classification [20], [21], [43], [46] and texture classification [44], or in combination with supervised methods such as Convolutional Neural Networks (CNNs) to make the latter learn quicker, and in a more stable way [32].

We first revise the operations that form a ScatterNet in **??**. We then introduce our DeScatterNet (**??**), and show how we can use it to examine the layers of ScatterNets (using a similar technique to the CNN visualization in [40]). We use this analysis tool to highlight what patterns a ScatterNet is sensitive to (**??**), showing that they are very different from what their CNN counterparts are sensitive to, and possibly less useful for discriminative tasks.

We use these observations to propose an architectural change to ScatterNets, which have not changed much since their inception in [45]. Two changes of note however are the work of Sifre and Mallat in [44], and the work of Singh and Kingsbury in [43]. Sifre and Mallat introduced Rotationally Invariant ScatterNets which took ScatterNets in a new direction, as the architecture now included filtering across the wavelet orientations (albeit with heavy restrictions on the fitlers used). Singh and Kingsbury achieved improvements in performance in a Scattering system using the spatially implementable DTCWT[18] wavelets instead of the Fourier Transform (FFT) based Morlet previously used.

4.1 Related Work

Zeiler, Taylor, and Fergus first attempted to use 'deconvolution' to improve their learning [47], then later for purely visualization purposes [40]. Their method involves monitoring network nodes, seeing what input image causes the largest activity and mapping activations at different layers of the network back to the pixel space using meta-information from these images.

Figure 4.1 shows the block diagram for how deconvolution is done. Inverting a convolutional layer is done by taking the 2D transpose of each slice of the filter. Inverting a ReLU is done by simply applying a ReLU again (ensuring only positive values can flow back through the network). Inverting a max pooling step is a little trickier, as max pooling is quite a lossy operation. Zeiler, Taylor, and Fergus get around this by saving extra information on the forward pass of the model — switches that store the location of the input that caused the maximum value. This way, on the backwards pass, it is trivial to store activations to the right position in the larger feature map. Note that the positions that did not contribute to the max pooling operation remain as zero on the backwards pass. This is shown in the bottom half of Figure 4.1.

[48] design an *all convolutional* model, where they replace the max pooling layers commonly seen in CNNs with a convolutional block with sride 2, i.e. a convolution followed by decimation. They did this by first adding an extra convolutional layer before the max pooling layers, then taking away the max pooling and adding decimation after convolution, noting that removing the max pooling had little effect.

The benefit of this is that they can now reconstruct images as Zeiler and Fergus did, but without having to save switches from the max pooling operation. Additionally, they modify the handling of the ReLU in the backwards pass to combine the regular backpropagation action and the ReLU action from deconv. They call this 'guided backprop'.

Another interrogation tool commonly used is occlusion or perturbation. [40] occlude regions in the input image and measure the impact this has on classification score. [49] use gradients to find the minimal mask to apply to the input image that causes misclassification.

Mahendran and Vedaldi take a slightly different route on deconvolution networks [41]. They do not store this extra information but instead define a cost function to maximize to. This results in visualization images that look very surreal, and can be quite different from the input.

4.2 The Scattering Transform and Hybrid Network

While we have introduced the scattering trasnform before, we clarify the format we use for this chapter's analysis.

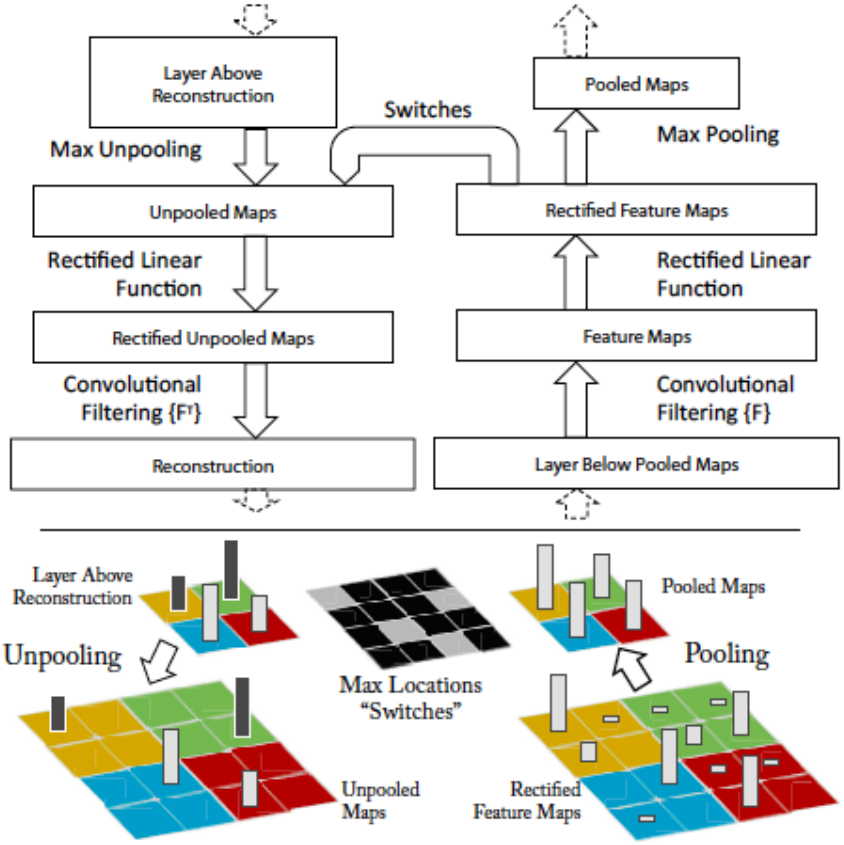


Figure 4.1: **Deconvolution Network Block Diagram.** Note the switches that are saved before the pooled features, and the filters used for deconvolution are the transpose of the filters used for a forward pass. Taken from [40].

We use the DT$\mathbb{C}$WT based scatternet introduced in the previous chapter as a front end, and Table 3.3. Consider an input signal $x(\boldsymbol{u}), \boldsymbol{u} \in \mathbb{R}^2$. The zeroth order **scatter** coefficient is the lowpass output of a J level FB:

$$S_0 x(\boldsymbol{u}) \triangleq (x * \phi_J)(\boldsymbol{u}) \tag{4.2.1}$$

This is invariant to translations of up to 2^J pixels[1]. In exchange for gaining invariance, the S_0 coefficients have lost a lot of information (contained in the rest of the frequency space). The remaining energy of x is contained within the first order **wavelet** coefficients:

$$W_1 x(\boldsymbol{u}, j_1, \theta_1) \triangleq x * \psi_{j_1,\theta_1} \tag{4.2.2}$$

for $j_1 \in \{1, 2, \ldots, J\}, \theta_1 \in \{1, 2, \ldots, L\}$. We will want to retain this information in these coefficients to build a useful classifier.

Let us call the set of available scales and orientations Λ_1 and use λ_1 to index it. For both Morlet and DT$\mathbb{C}$WT implementations, ψ is complex-valued, i.e., $\psi = \psi^r + j\psi^i$ with ψ_r and ψ_i forming a Hilbert Pair, resulting in an analytic ψ. This analyticity provides a source of invariance — small input shifts in x result in a phase rotation (but little magnitude change) of the complex wavelet coefficients[2].

Taking the magnitude of W_1 gives us the first order **propagated** signals:

$$U_1 x(\boldsymbol{u}, \lambda_1) \triangleq |x * \psi_\lambda| = \sqrt{(x * \psi^r_{\lambda_1})^2 + (x * \psi^i_{\lambda_1})^2} \tag{4.2.3}$$

The first order scattering coefficient makes U_1 invariant up to our scale J by averaging it:

$$S_1 x(\boldsymbol{u}, \lambda_1) \triangleq |x * \psi_\lambda| * \phi_J \tag{4.2.4}$$

If we define $U_0 \triangleq x$, then we can iteratively define:

$$W_m = U_{m-1} * \psi_{\lambda_m} \tag{4.2.5}$$

$$U_m = |W_m| \tag{4.2.6}$$

$$S_m = U_m * \phi_J \tag{4.2.7}$$

We repeat this for higher orders, although previous work shows that, for natural images, we get diminishing returns after $m = 2$. The output of our ScatterNet is then:

$$Sx = \{S_0 x, S_1 x, S_2 x\} \tag{4.2.8}$$

[1] From here on, we drop the $\boldsymbol{u}$ notation when indexing x, for clarity.

[2] In comparison to a system with purely real filters such as a CNN, which would have rapidly varying coefficients for small input shifts [18].

Consider an input signal $x(\boldsymbol{u}), \boldsymbol{u} \in \mathbb{R}^2$. The zeroth order **scatter** coefficient is the lowpass output of a J level FB:

$$S_0 x(\boldsymbol{u}) \triangleq (x * \phi_J)(\boldsymbol{u}) \tag{4.2.9}$$

This is invariant to translations of up to 2^J pixels[3]. In exchange for gaining invariance, the S_0 coefficients have lost a lot of information (contained in the rest of the frequency space). The remaining energy of x is contained within the first order **wavelet** coefficients:

$$W_1 x(\boldsymbol{u}, j_1, \theta_1) \triangleq x * \psi_{j_1,\theta_1} \tag{4.2.10}$$

for $j_1 \in \{1, 2, \ldots, J\}, \theta_1 \in \{1, 2, \ldots, L\}$. We will want to retain this information in these coefficients to build a useful classifier.

Let us call the set of available scales and orientations Λ_1 and use λ_1 to index it. For both Morlet and DT$\mathbb{C}$WT implementations, ψ is complex-valued, i.e., $\psi = \psi^r + j\psi^i$ with ψ_r and ψ_i forming a Hilbert Pair, resulting in an analytic ψ. This analyticity provides a source of invariance — small input shifts in x result in a phase rotation (but little magnitude change) of the complex wavelet coefficients[4].

Taking the magnitude of W_1 gives us the first order **propagated** signals:

$$U_1 x(\boldsymbol{u}, \lambda_1) \triangleq |x * \psi_\lambda| = \sqrt{(x * \psi^r_{\lambda_1})^2 + (x * \psi^i_{\lambda_1})^2} \tag{4.2.11}$$

The first order scattering coefficient makes U_1 invariant up to our scale J by averaging it:

$$S_1 x(\boldsymbol{u}, \lambda_1) \triangleq |x * \psi_\lambda| * \phi_J \tag{4.2.12}$$

If we define $U_0 \triangleq x$, then we can iteratively define:

$$W_m = U_{m-1} * \psi_{\lambda_m} \tag{4.2.13}$$
$$U_m = |W_m| \tag{4.2.14}$$
$$S_m = U_m * \phi_J \tag{4.2.15}$$

We repeat this for higher orders, although previous work shows that, for natural images, we get diminishing returns after $m = 2$. The output of our ScatterNet is then:

$$Sx = \{S_0 x, S_1 x, S_2 x\} \tag{4.2.16}$$

[3] From here on, we drop the $\boldsymbol{u}$ notation when indexing x, for clarity.

[4] In comparison to a system with purely real filters such as a CNN, which would have rapidly varying coefficients for small input shifts [18].

4.2.1 Scattering Color Images

A wavelet transform like the DT$\mathbb{C}$WT accepts single channel input, while we often work on RGB images. This leaves us with a choice. We can either:

1. Apply the wavelet transform (and the subsequent scattering operations) on each channel independently. This would triple the output size to $3C$.

2. Define a frequency threshold below which we keep color information, and above which, we combine the three channels into a single luminance channel.

The second option uses the well known fact that the human eye is far less sensitive to higher spatial frequencies in color channels than in luminance channels. This also fits in with the first layer filters seen in the well known Convolutional Neural Network, AlexNet. Roughly one half of the filters were low frequency color 'blobs', while the other half were higher frequency, grayscale, oriented wavelets.

For this reason, we choose the second option for the architecture described in this paper. We keep the 3 color channels in our S_0 coefficients, but work only on grayscale for high orders (the S_0 coefficients are the lowpass bands of a J-scale wavelet transform, so we have effectively chosen a color cut-off frequency of $2^{-J}\frac{f_s}{2}$).

For example, consider an RGB input image x of size $64 \times 64 \times 3$. The scattering transform we have described with parameters $J = 4$ and $m = 2$ would then have the following coefficients:

$$S_0 \;:\; (64 \times 2^{-J}) \times (64 \times 2^{-J}) \times 3 = 4 \times 4 \times 3 \tag{4.2.17}$$

$$S_1 \;:\; 4 \times 4 \times (LJ) = 4 \times 4 \times 24 \tag{4.2.18}$$

$$S_2 \;:\; 4 \times 4 \times \left(\frac{1}{2}L^2 J(J-1)\right) = 4 \times 4 \times 216 \tag{4.2.19}$$

$$S \;:\; 4 \times 4 \times (216 + 24 + 3) = 4 \times 4 \times 243 \tag{4.2.20}$$

4.3 The Inverse Network

We now introduce our inverse scattering network. This allows us to back project scattering coefficients to the image plane; it is inspired by the **DeconvNet** used by Zeiler and Fergus in [40] to look into the deeper layers of CNNs.

We emphasize that instead of thinking about perfectly reconstructing x from $S \in \mathbb{R}^{H' \times W' \times C}$, we want to see what signal/pattern in the input image caused a large activation in each channel. This gives us a good idea of what each output channel is sensitive to, or what it extracts from the input. Note that we do not use any of the log normalization layers described in [21], [43].

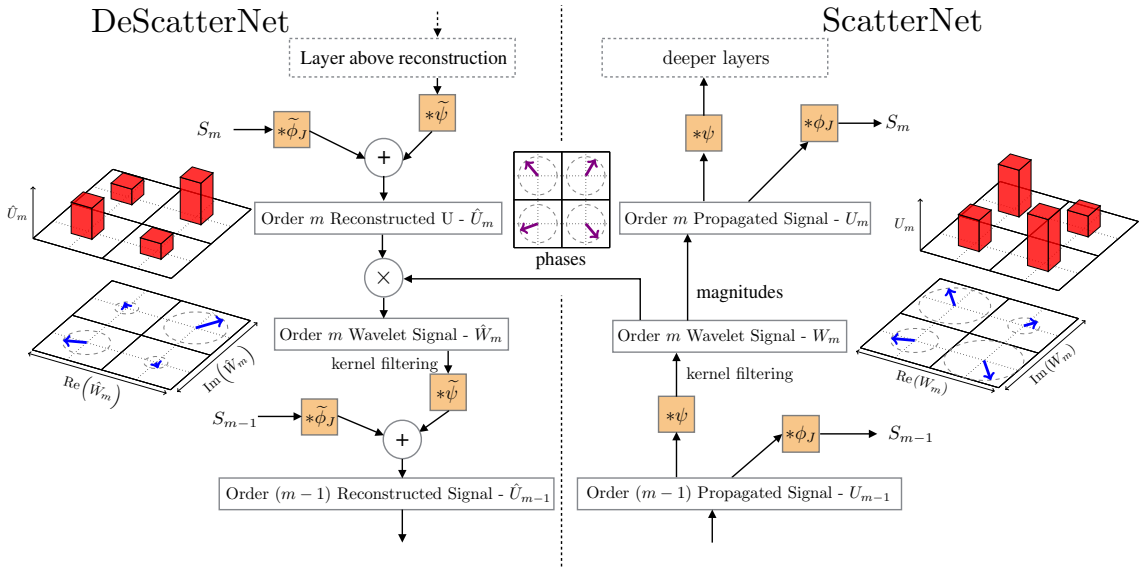


Figure 4.2: A DeScattering layer (left) attached to a Scattering layer (right). We are using the same convention as [40] Figure 1 - the input signal starts in the bottom right hand corner, passes forwards through the ScatterNet (up the right half), and then is reconstructed in the DeScatterNet (downwards on the left half). The DeScattering layer will reconstruct an approximate version of the previous order's propagated signal. The 2×2 grids shown around the image are either Argand diagrams representing the magnitude and phase of small regions of *complex* (De)ScatterNet coefficients, or bar charts showing the magnitude of the *real* (De)ScatterNet coefficients (after applying the modulus non-linearity). For reconstruction, we need to save the discarded phase information and reintroduce it by multiplying it with the reconstructed magnitudes.

4.3.1 Inverting the Low-Pass Filtering

Going from the U coefficients to the S coefficients involved convolving by a low pass filter, ϕ_J followed by decimation to make the output $(H \times 2^{-J}) \times (W \times 2^{-J})$. ϕ_J is a purely real filter, and we can 'invert' this operation by interpolating S to the same spatial size as U and convolving with the mirror image of ϕ_J, $\widetilde{\phi}_J$ (this is equivalent to the transpose convolution described in [40]).

$$\hat{S}_m = S_m * \widetilde{\phi}_J \tag{4.3.1}$$

This will not recover U as it was on the forward pass, but will recover all the information in U that caused a strong response in S.

4.3.2 Inverting the Magnitude Operation

In the same vein as [40], we face a difficult task in inverting the non-linearity in our system. We lend inspiration from the *switches* introduced in the DeconvNet; the switches in a DeconvNet

save the location of maximal activations so that on the backwards pass activation layers could be unpooled trivially. We do an equivalent operation by saving the phase of the complex activations. On the backwards pass we reinsert the phase to give our recovered W.

$$\hat{W}_m = \hat{U}_m e^{j\theta_m} \tag{4.3.2}$$

4.3.3 Inverting the Wavelet Decomposition

Using the DTCWTmakes inverting the wavelet transform simple, as we can simply feed the coefficients through the synthesis filter banks to regenerate the signal. For complex ψ, this is convolving with the conjugate transpose $\widetilde{\psi}$:

$$\begin{aligned} \hat{U}_{m-1} &= \hat{S}_{m-1} + \hat{W}_m & (4.3.3)\\ &= S_{m-1} * \widetilde{\phi}_J + \sum_{j,\theta} W_m(\boldsymbol{u}, j, \theta) * \widetilde{\psi}_{j,\theta} & (4.3.4) \end{aligned}$$

4.4 Visualization with Inverse Scattering

To examine our ScatterNet, we scatter all of the images from ImageNet's validation set and record the top 9 images which most highly activate each of the C channels in the ScatterNet. This is the *identification* phase (in which no inverse scattering is performed).

Then, in the *reconstruction* phase, we load in the $9 \times C$ images, and scatter them one by one. We take the resulting $4 \times 4 \times 243$ output vector and mask all but a single value in the channel we are currently examining.

This 1-sparse tensor is then presented to the inverse scattering network from Figure 4.2 and projected back to the image space. Some results of this are shown in Figure 4.3. This figure shows reconstructed features from the layers of a ScatterNet. For a given output channel, we show the top 9 activations projected independently to pixel space. For the first and second order coefficients, we also show the patch of pixels in the input image which cause this large output. We display activations from various scales (increasing from first row to last row), and random orientations in these scales.

The order 1 scattering (labelled with 'Order 1' in Figure 4.3) coefficients look quite similar to the first layer filters from the well known AlexNet CNN [50]. This is not too surprising, as the first order scattering coefficients are simply a wavelet transform followed by average pooling. They are responding to images with strong edges aligned with the wavelet orientation.

The second order coefficients (labelled with 'Order 2' in Figure 4.3) appear very similar to the order 1 coefficients at first glance. They too are sensitive to edge-like features, and

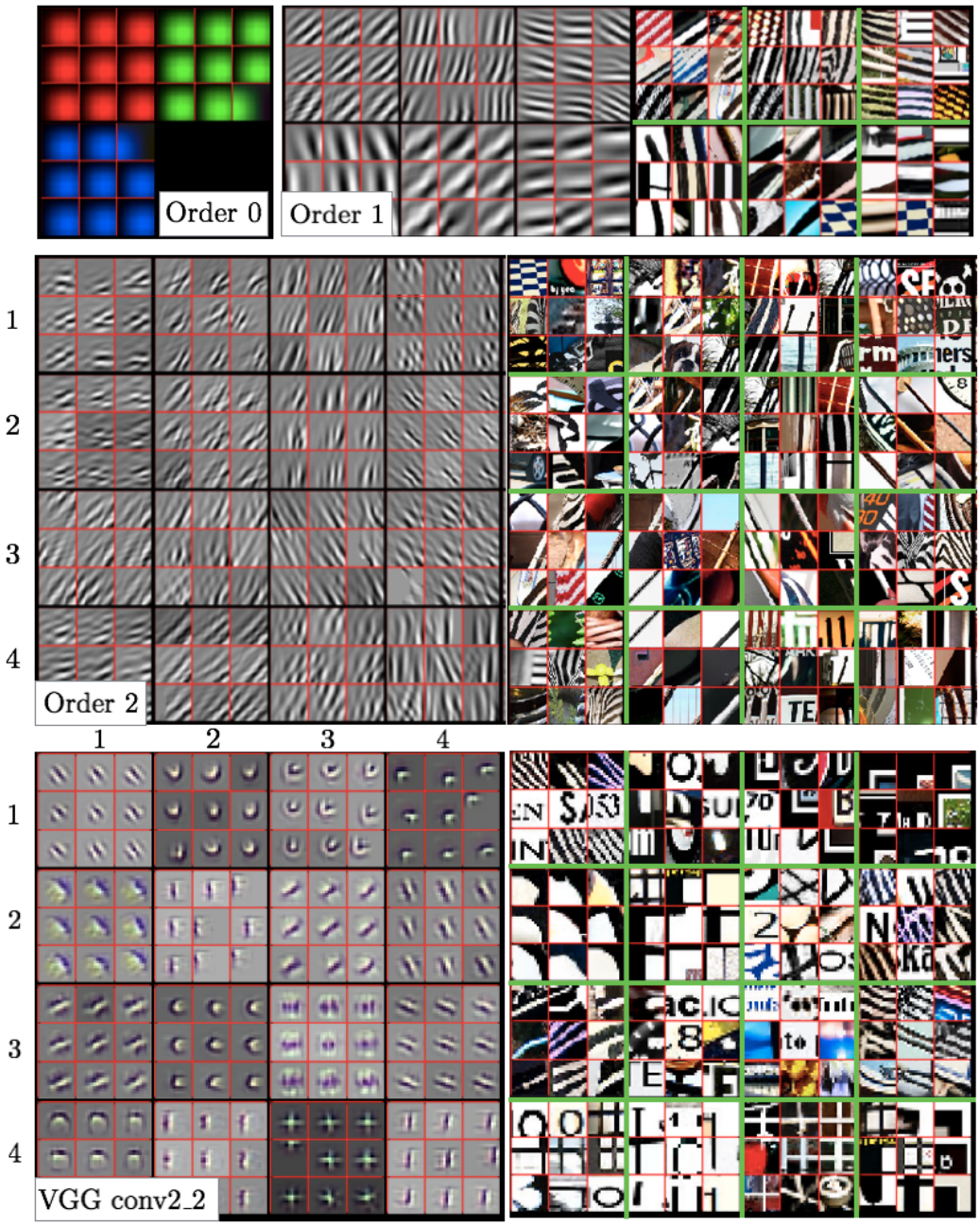


Figure 4.3: Visualization of a random subset of features from S_0 (all 3), S_1 (6 from the 24) and S_2 (16 from the 240) scattering outputs. We record the top 9 activations for the chosen features and project them back to the pixel space. We show them alongside the input image patches which caused the large activations. We also include reconstructions from layer conv2_2 of VGG Net [35](a popular CNN, often used for feature extraction) for reference — here we display 16 of the 128 channels. The VGG reconstructions were made with a CNN DeconvNet based on [40]. Image best viewed digitally.

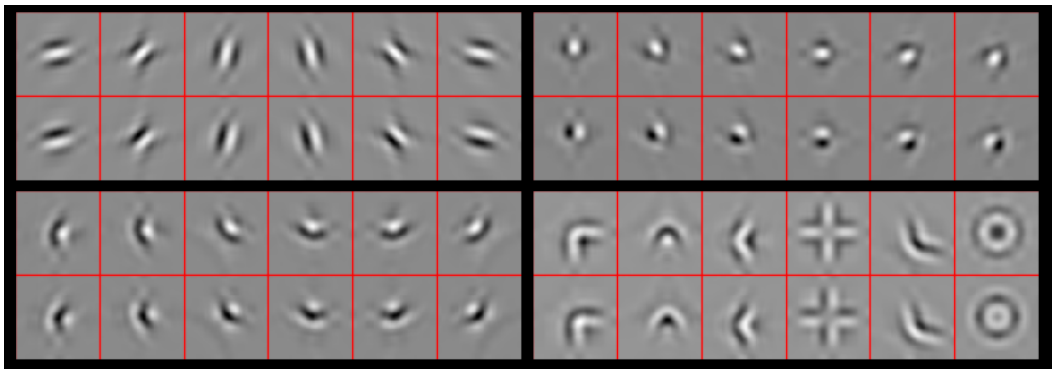


Figure 4.4: **Shapes possible by filtering across the wavelet orientations with complex coefficients.** All shapes are shown in pairs: the top image is reconstructed from a purely real output, and the bottom image from a purely imaginary output. These 'real' and 'imaginary' shapes are nearly orthogonal in the pixel space (normalized dot product < 0.01 for all but the doughnut shape in the bottom right, which has 0.15) but produce the same U', something that would not be possible without the complex filters of a ScatterNet. Top left - reconstructions from U_1 (i.e. no cross-orientation filtering). Top right- reconstructions from U_1' using a $1 \times 1 \times 12$ Morlet Wavelet, similar to what was done in the 'Roto-Translation' ScatterNet described in [21], [44]. Bottom left - reconstructions from U_1' made with a more general $1 \times 1 \times 12$ filter, described in Equation 4.6.3. Bottom right - some reconstructions possible by filtering a general $3 \times 3 \times 12$ filter.

some of them (e.g. third row, third column and fourth row, second column) are mostly just that. These are features that have the same oriented wavelet applied at both the first and second order. Others, such as the 9 in the first row, first column, and first row, fourth column are more sensitive to checker-board like patterns. Indeed, these are activations where the orientation of the wavelet for the first and second order scattering were far from each other (15° and 105° for the first row, first column and 105° and 45° for the first row, fourth column).

For comparison, we include reconstructions from the second layer of the well-known VGG CNN (labelled with 'VGG conv2_2', in Figure 4.3). These were made with a DeconvNet, following the same method as [40]. Note that while some of the features are edge-like, we also see higher order shapes like corners, crosses and curves.

4.5 Channel Saliency

4.6 Corners, Crosses and Curves

We build on these two systems, showing that with carefully designed complex filters applied across the complex spatial coefficients of a 2-D DTCWT, we can build filters that are sensitive to more recognizable shapes like those commonly seen in CNNs, such as corners and

curves (**??**). These reconstructions show that the features extracted from ScatterNets vary significantly from those learned in CNNs after the first order. In many respects, the features extracted from a CNN like VGGNet look preferable for use as part of a classification system.

[44] and [21] introduced the idea of a 'Roto-Translation' ScatterNet. Invariance to rotation could be made by applying averaging (and bandpass) filters across the L orientations from the wavelet transform *before* applying the complex modulus. Momentarily ignoring the form of the filters they apply, referring to them as $F_k \in \mathbb{C}^L$, we can think of this stage as stacking the L outputs of a complex wavelet transform on top of each other, and convolving these filters F_k over all spatial locations of the wavelet coefficients $W_m x$ (this is equivalent to how filters in a CNN are fully connected in depth):

$$V_m x(\boldsymbol{u}, j, k) = W_m x * F_k = \sum_{\theta} W_m x(\boldsymbol{u}, j, \theta) F_k(\theta) \tag{4.6.1}$$

We then take the modulus of these complex outputs to make a second propagated signal:

$$U'_m x \triangleq |V_m x| = |W_m x * F_k| = |U_{m-1} x * \psi_{\lambda_m} * F_k| \tag{4.6.2}$$

We present a variation on this idea, by filtering with a more general $F \in \mathbb{C}^{H \times W \times 12}$. We use F of length 12 rather than 6, as we use the $L = 6$ orientations and their complex conjugates; each wavelet is a 30° rotation of the previous, so with 12 rotations, we can cover the full 360°.

Figure 4.4 shows some reconstructions from these V coefficients. Each of the four quadrants show reconstructions from a different class of ScatterNet layer. All shapes are shown in real and imaginary Hilbert-like pairs; the top images in each quadrant are reconstructed from a purely real V, while the bottom inputs are reconstructed from a purely imaginary V. This shows one level of invariance of these filters, as after taking the complex magnitude, both the top and the bottom shape will activate the filter with the same strength. In comparison, for the purely real filters of a CNN, the top shape would cause a large output, and the bottom shape would cause near 0 activity (they are nearly orthogonal to each other).

In the top left, we display the 6 wavelet filters for reference (these were reconstructed from U_1, not V_1). In the top right of the figure we see some of the shapes made by using the F's from the Roto-Translation ScatterNet [21], [44]. The bottom left is where we present some of our novel kernels. These are simple corner-like shapes made by filtering with $F \in \mathbb{C}^{1 \times 1 \times 12}$

$$F = [1, j, j, 1, 0, 0, 0, 0, 0, 0, 0, 0] \tag{4.6.3}$$

The six orientations are made by rolling the coefficients in F along one sample (i.e. $[0, 1, j, j, 1, 0, \ldots]$, $[0, 0, 1, j, j, 1, 0, \ldots]$, $[0, 0, 0, 1, j, j, 1, 0, \ldots]$ $\ldots$). Coefficients roll back around (like circular convolution) when they reach the end.

Finally, in the bottom right we see shapes made by $F \in \mathbb{C}^{3\times3\times12}$. Note that with the exception of the ring-like shape which has 12 non-zero coefficients, all of these shapes were reconstructed with F's that have 4 to 8 non-zero coefficients of a possible 64. These shapes are now beginning to more closely resemble the more complex shapes seen in the middle stages of CNNs.

4.7 Discussion

This paper presents a way to investigate what the higher orders of a ScatterNet are responding to - the DeScatterNet described in section 4.3. Using this, we have shown that the second 'layer' of a ScatterNet responds strongly to patterns that are very dissimilar to those that highly activate the second layer of a CNN. As well as being dissimilar to CNNs, visual inspection of the ScatterNet's patterns reveal that they may be less useful for discriminative tasks, and we believe this may be causing the current gaps in state-of-the-art performance between the two.

We have presented an architectural change to ScatterNets that can make it sensitive to more recognizable shapes. We believe that using this new layer is how we can start to close the gap, making more generic and descriptive ScatterNets while keeping control of their desirable properties.

Chapter 5

A Learnable ScatterNet: Locally Invariant Convolutional Layers

In this chapter we explore tying together the ideas from Scattering Transforms and Convolutional Neural Networks (CNN) for Image Analysis by proposing a learnable ScatterNet. The work presented in **??** implies that while the Scattering Transform has been a promising start in using complex wavelets in image understanding tasks, there is something missing from them. To address this, we propose a learnable ScatterNet by building it with our proposed "Locally Invariant Convolutional Layers".

Previous attempts at tying ScatterNets together with CNNs in hybrid networks [31], [32], [51] have tended to keep the two parts separate, with the ScatterNet forming a fixed front end and the CNN forming a learned backend. We instead look at adding learning between scattering orders, as well as adding learned layers before the ScatterNet.

We do this by adding a second stage after a scattering order, which mixes output activations together in a learnable way. The flexibility of the mixing we introduce allows us to build a layer that acts as a Scattering Layer with no learning, or as one that acts more as a convolutional layer with a controlled number of input and output channels, or more interestingly, as a hybrid between the two.

Our experiments show that these locally invariant layers can improve accuracy when added to either a CNN or a ScatterNet. We also discover some surprising results in that the ScatterNet may be best positioned after one or more layers of learning rather than at the front of a neural network.

In **??** we briefly review convolutional layers and scattering layers before introducing our learnable scattering layers in **??**. In **??** we describe how we implement our proposed layer, and present some experiments we have run in section 5.5 and then draw conclusions about how these new ideas might improve neural networks in the future.

5.1 Related Work

There have been several similar works that look into designing new convolutional layers by separating them into two stages — a first stage that performs a non-standard filtering process, and a second stage that combines the first stage into single activations. The inception layer [52] by Szegedy *et al.* does this by filtering with different kernel sizes in the first stage, and then combining with a 1×1 convolution in the second stage. Ioannou *et al.* also do something similar by making a first stage with horizontal and vertical filters, and then combining in the second stage again with a 1×1 convolution[53]. But perhaps the most similar works are those that use a first stage with fixed filters, combining them in a learned way in the second stage. Of particular note are:

- "Local Binary Convolutional Neural Networks" [54]. This paper builds a first stage with a small 3×3 kernel filled with zeros, and randomly insert plus and minus ones into it keeping a set sparsity level. This builds a very crude spatial differentiator in random directions. The output of the first stage is then passed through a sigmoid nonlinearity before being mixed with a 1×1 convolution. The imposed structure on the first stage was found to be a good regularizer and prevented overfitting, and the combination of the mixing in the second layer allowed for a powerful and expressive layer, with performance near that of a regular CNN layer.

- "DCFNet: Deep Neural Network with Decomposed Convolutional Filters" [55]. This paper decomposes convolutional filters as linear combinations of Fourier Bessel and random bases. The first stage projects the inputs onto the chosen basis, and the second stage learns how to mix these projections with a 1×1 convolution. Unlike [54] this layer is purely linear. The supposed advantage being that the basis can be truncated to save parameters and make the input less susceptible to high frequency variations. The work found that this layer had marginal benefits over regular CNN layers in classification, but had improved stability to noisy inputs.

5.2 Recap of Useful Terms

5.2.1 Convolutional Layers

Let the output of a CNN at layer l be:

$$x^{(l)}(c, \mathbf{u}), \quad c \in \{0, \dots C_l - 1\}, \mathbf{u} \in \mathbb{R}^2$$

where c indexes the channel dimension , and $\mathbf{u}$ is a vector of coordinates for the spatial position. Of course, $\mathbf{u}$ is typically sampled on a grid, but we keep it continuous to more easily differentiate between the spatial and channel dimensions. A typical convolutional layer

in a standard CNN (ignoring the bias term) is:

$$y^{(l+1)}(f,\mathbf{u}) = \sum_{c=0}^{C_l-1} x^{(l)}(c,\mathbf{u}) * h_f^{(l)}(c,\mathbf{u}) \tag{5.2.1}$$

$$x^{(l+1)}(f,\mathbf{u}) = \sigma\left(y^{(l+1)}(f,\mathbf{u})\right) \tag{5.2.2}$$

where $h_f^{(l)}(c,\mathbf{u})$ is the fth filter of the lth layer (i.e. $f \in \{0,\ldots,C_{l+1}-1\}$) with C_l different point spread functions. σ is a non-linearity such as the ReLU, possibly combined with scaling such as batch normalization. The convolution is done independently for each c in the C_l channels and the resulting outputs are summed together to give one activation map. This is repeated C_{l+1} times to give $\left\{x^{(l+1)}(f,\mathbf{u})\right\}_{f\in\{0,\ldots,C_{l+1}-1\},\mathbf{u}\in\mathbb{R}^2}$

5.2.2 Wavelet Transforms

The 2-D wavelet transform is performed by convolving the input with a mother wavelet dilated by 2^j and rotated by θ:

$$\psi_{j,\theta}(\mathbf{u}) = 2^{-j}\psi\left(2^{-j}R_{-\theta}\mathbf{u}\right) \tag{5.2.3}$$

where R is the rotation matrix, $1 \le j \le J$ indexes the scale, and $1 \le k \le K$ indexes θ to give K angles between 0 and π. We copy notation from [20] and define $\lambda = (j,k)$ and the set of all possible λs is Λ whose size is $|\Lambda| = JK$. The wavelet transform, including lowpass, is then:

$$Wx(c,\mathbf{u}) = \{x(c,\mathbf{u}) * \phi_J(\mathbf{u}), x(c,\mathbf{u}) * \psi_\lambda(\mathbf{u})\}_{\lambda\in\Lambda} \tag{5.2.4}$$

5.2.3 Scattering Transforms

As the real and imaginary parts of complex wavelets are in quadrature with each other, taking the modulus of the resulting transformed coefficients removes the high frequency oscillations of the output signal while preserving the energy of the coefficients over the frequency band covered by ψ_λ. This is crucial to ensure that the scattering energy is concentrated towards zero-frequency as the scattering order increases, allowing sub-sampling. We define the wavelet modulus propagator to be:

$$\tilde{W}x(c,\mathbf{u}) = \{x(c,\mathbf{u}) * \phi_J(\mathbf{u}), |x(c,\mathbf{u}) * \psi_\lambda(\mathbf{u})|\}_{\lambda\in\Lambda} \tag{5.2.5}$$

Let us call these modulus terms $U[\lambda]x = |x * \psi_\lambda|$ and define a path as a sequence of λs given by $p = (\lambda_1, \lambda_2, \ldots \lambda_m)$. Further, define the modulus propagator acting on a path p by:

$$U[p]x = U[\lambda_m]\cdots U[\lambda_2]U[\lambda_1]x \tag{5.2.6}$$

$$= ||\cdots|x * \psi_{\lambda_1}| * \psi_{\lambda_2}|\cdots * \psi_{\lambda_m}| \tag{5.2.7}$$

These descriptors are then averaged over the window 2^J by a scaled lowpass filter $\phi_J = 2^{-J}\phi(2^{-J}\mathbf{u})$ giving the 'invariant' scattering coefficient

$$S[p]x(\mathbf{u}) = U[p]x * \phi_J(\mathbf{u}) \tag{5.2.8}$$

If we define $p+\lambda = (\lambda_1, \ldots, \lambda_m, \lambda)$ then we can combine (5.2.5) and (5.2.6) to give:

$$\tilde{W}U[p]x = \{S[p]x, U[p+\lambda]x\}_\lambda \tag{5.2.9}$$

Hence we iteratively apply $\tilde{W}$ to all of the propagated U terms of the previous layer to get the next order of scattering coefficients and the new U terms.

The resulting scattering coefficients have many nice properties, one of which is stability to diffeomorphisms (such as shifts and warping). From [45], if $\mathcal{L}_\tau x = x(\mathbf{u} - \tau(\mathbf{u}))$ is a diffeomorphism which is bounded with $\|\nabla\tau\|_\infty \leq 1/2$, then there exists a $K_L > 0$ such that:

$$\|S\mathcal{L}_\tau x - Sx\| \leq K_L P F(\tau) \|x\| \tag{5.2.10}$$

where $P = \text{length}(p)$ is the scattering order, and $F(\tau)$ is a function of the size of the displacement, derivative and Hessian of τ, $H(\tau)$ [45]:

$$F(\tau) = 2^{-J}\|\tau\|_\infty + \|\nabla\tau\|_\infty \max\left(\log \frac{\|\Delta\tau\|_\infty}{\|\nabla\tau\|_\infty}, 1\right) + \|H(\tau)\|_\infty \tag{5.2.11}$$

5.3 Locally Invariant Layer

We propose to mix the terms at the output of each wavelet modulus propagator $\tilde{W}$. The second term in $\tilde{W}$, the U terms are often called 'covariant' terms but in this work we will call them locally invariant, as they tend to be invariant up to a scale 2^j. We propose to mix the locally invariant terms U and the lowpass terms S with learned weights $a_{f,\lambda}$ and b_f.

For example, consider the wavelet modulus propagator from (5.2.5), and let the input to it be $x^{(l)}$. Our proposed output is then:

$$\begin{aligned} y^{(l+1)}(f,\mathbf{u}) &= \sum_\lambda \sum_{c=0}^{C-1} \left| x^{(l)}(c,\mathbf{u}) * \psi_\lambda(\mathbf{u}) \right| a_{f,\lambda}(c) \\ &+ \sum_{c=0}^{C-1} \left(x^{(l)}(c,\mathbf{u}) * \phi_J(\mathbf{u}) \right) b_f(c) \end{aligned} \tag{5.3.1}$$

Recall that λ is the tuple (j,k) for $1 \leq j \leq J$, $1 \leq k \leq K$ and is used to select the bandpass wavelet at scale j and orientation k. Note that an input to the wavelet modulus propagator $\tilde{W}$ with C channels has $(JK+1)C$ output channels – C lowpass channels and JKC modulus

bandpass channels. Let us define a new output z with index variable q such that:

$$z^{(l+1)}(q,\mathbf{u}) = \begin{cases} x^{(l)}(c,\mathbf{u}) * \phi_J(\mathbf{u}) & \text{if } 0 \le q < C \\ |x^{(l)}(c,\mathbf{u}) * \psi_\lambda(\mathbf{u})| & \text{if } C \le q < (JK+1)C \end{cases} \tag{5.3.2}$$

i.e. the lowpass channels are the first C channels of z, the modulus of the 15° ($k=1$) wavelet coefficients with $j=1$ are the next C channels, then the modulus coefficients with $k=2$ and $j=1$ are the third C channels, and so on.

We do the same for the weights a, b by defining $\tilde{a}_f = \{b_f, a_{f,\lambda}\}_\lambda$ and let:

$$\tilde{a}_f(q) = \begin{cases} b_f(c) & \text{if } 0 \le q < C \\ a_{f,\lambda}(c) & \text{if } C \le q < (JK+1)C \end{cases} \tag{5.3.3}$$

we can then use (5.3.2) and (5.3.3) to simplify (5.3.1), giving:

$$y^{(l+1)}(f,\mathbf{u}) = \sum_{q=0}^{(JK+1)C-1} z^{(l+1)}(q,\mathbf{u})\tilde{a}_f(q) \tag{5.3.4}$$

or in matrix form with $A_{f,q} = \tilde{a}_f(q)$

$$Y^{(l+1)}(\mathbf{u}) = AZ^{(l+1)}(\mathbf{u}) \tag{5.3.5}$$

This is very similar to the standard convolutional layer from (5.2.1), except we have replaced the previous layer's x with intermediate coefficients z (with $|Q| = (JK+1)C$ channels), and the convolutions of (5.2.1) have been replaced by a matrix multiply (which can also be seen as a 1×1 convolutional layer). We can then apply (5.2.2) to (5.3.4) to get the next layer's output:

$$x^{(l+1)}(f,\mathbf{u}) = \sigma\left(y^{(l+1)}(f,\mathbf{u})\right) \tag{5.3.6}$$

Figure 5.1 shows a block diagram for this process.

5.3.1 Properties

5.3.1.1 Recovering the original ScatterNet Design

The first thing to note is that with careful choice of A and σ, we can recover the original translation invariant ScatterNet [20], [32]. If $C_{l+1} = (JK+1)C_l$ and A is the identity matrix $I_{C_{l+1}}$, we remove the mixing and then $y^{(l+1)} = \tilde{W}x$.

Further, if $\sigma = \text{ReLU}$ as is commonly the case in training CNNs, it has no effect on the positive locally invariant terms U. It will affect the averaging terms if the signal is not positive, but this can be dealt with by adding a channel dependent bias term α_c to $x^{(l)}$ to

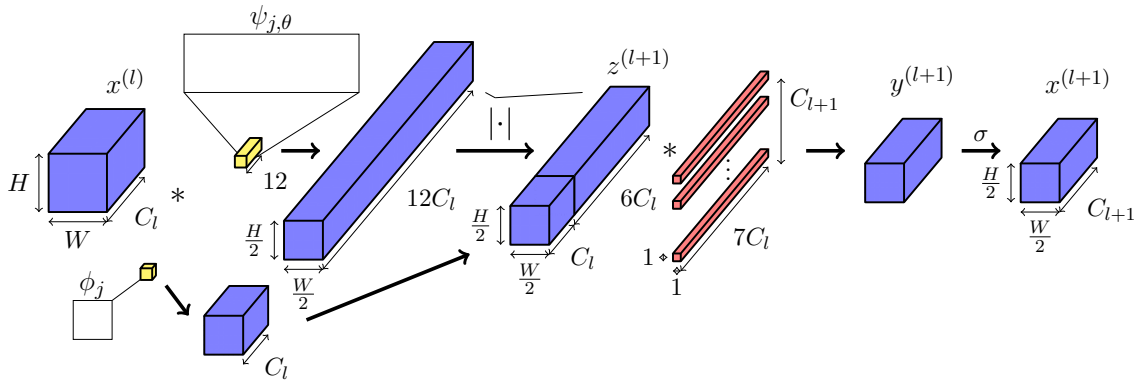


Figure 5.1: Block Diagram of Proposed Invariant Layer for $j = J = 1$. Activations are shaded blue, fixed parameters yellow and learned parameters red. Input $x^{(l)} \in \mathbb{R}^{C_l \times H \times W}$ is filtered by real and imaginary oriented wavelets and a lowpass filter and is downsampled. The channel dimension increases from C_l to $(2K+1)C_l$, where the number of orientations is $K = 6$. The real and imaginary parts are combined by taking their magnitude (an example of what this looks like in 1D is shown above the magnitude operator) - the components oscillating in quadrature are combined to give $z^{(l+1)}$. The resulting activations are concatenated with the lowpass filtered activations, mixed across the channel dimension, and then passed through a nonlinearity σ to give $x^{(l+1)}$. If the desired output spatial size is $H \times W$, $x^{(l+1)}$ can be bilinearly upsampled paying only a few multiplies per pixel.

ensure it is positive. This bias term will not affect the propagated signals as $\int \alpha_c \psi_\lambda(\mathbf{u}) d\mathbf{u} = 0$. The bias can then be corrected by subtracting $\alpha_c \|\phi_J\|_2$ from the averaging terms after taking the ReLU, then $x^{(l+1)} = \tilde{W}x$.

This makes one layer of our system equivalent to a first order scattering transform, giving S_0 and U_1 (invariant to input shifts of 2^1). Repeating the same process for the next layer again works, as we saw in (5.2.6), giving S_1 and U_2 (invariant to shifts of 2^2). If we want to build higher invariance, we can continue or simply average these outputs with an average pooling layer.

5.3.1.2 Flexibilty of the Layer

Unlike a regular ScatterNet, we are free to choose the size of C_{l+1}. This means we can set $C_{l+1} = C_l$ as is commonly the case in a CNN, and make a convolutional layer from mixing the locally invariant terms. This avoids the exponentially increasing complexity that comes with extra network layers that standard ScatterNets suffer from.

5.3.1.3 Stability to Noise and Deformations

Let us define the action of our layer on the scattering coefficients to be Vx. We would like to find a bound on $\|V\mathcal{L}_\tau x - Vx\|$. To do this, we note that the mixing is a linear operator and hence is Lipschitz continuous. The authors in [55] find constraints on the mixing weights to

make them non-expansive (i.e. Lipschitz constant 1). Further, the ReLU is non-expansive meaning the combination of the two is also non-expansive, so $\|V\mathcal{L}_\tau x - Vx\| \leq \|S\mathcal{L}_\tau x - Sx\|$, and (**??**) holds.

5.4 Implementation Details

Again, we use the DT$\mathbb{C}$WT [4] for our wavelet filters $\psi_{j,\theta}$ due to their fast implementation with separable convolutions which we will discuss more in subsection 5.4.3. There are two side effects of this choice. The first is that the number of orientations of wavelets is restricted to $K = 6$. The second is that we naturally downsample the output activations by a factor of 2 for each direction for each scale j, giving a 4^j downsampling factor overall. This represents the source of the invariance in our layer. If we do not wish to downsample the output (say to make the layer fit in a larger network), we can bilinearly interpolate the output of our layer. This is computationally cheap to do on its own, but causes the next layer's computation to be higher than necessary (there will be almost no energy for frequencies higher than $f_s/4$).

In all our experiments we set $J = 1$ for each invariant layer, meaning we can mix the lowpass and bandpass coefficients at the same resolution. Figure 5.1 shows how this is done. Note that setting $J = 1$ for a single layer does not restrict us from having $J > 1$ for the entire system, as if we have a second layer with $J = 1$ after the first, including downsampling ($\downarrow$), we would have:

$$(((x * \phi_1) \downarrow 2) * \psi_{1,\theta}) \downarrow 2 = (x * \psi_{2,\theta}) \downarrow 4 \tag{5.4.1}$$

5.4.1 Parameter Memory Cost

A standard convolutional layer with C_l input channels, C_{l+1} output channels and kernel size $L \times L$ has $L^2 C_l C_{l+1}$ parameters.

The number of learnable parameters in each of our proposed invariant layers with $J = 1$ and $K = 6$ orientations is:

$$\#\text{params} = (JK + 1)C_l C_{l+1} = 7C_l C_{l+1} \tag{5.4.2}$$

The spatial support of the wavelet filters is typically 5×5 pixels or more, and we have reduced #params to less than 3×3 per filter, while producing filters that are significantly larger than this.

5.4.2 Activation Memory Cost

A standard convolutional layer needs to save the activation $x^{(l)}$ to convolve with the back-propagated gradient $\frac{\partial L}{\partial y^{(l+1)}}$ on the backwards pass (to give $\frac{\partial L}{\partial w^{(l)}}$). For an input with C_l channels of spatial size $H \times W$, this means HWC_l floats must be saved.

Algorithm 5.1 Locally Invariant Convolutional Layer forward and backward passes

procedure LOCALINVFWD(x, A)
  $yl,\ yh \leftarrow \text{DT}\mathbb{C}\text{WT}(x^l, \text{nlevels} = 1)$ ▷ yh has 6 orientations and is complex
  $U \leftarrow \text{COMPLEXMAG}(yh)$
  $yl \leftarrow \text{AVGPOOL2x2}(yl)$ ▷ Downsample and recentre lowpass to match U size
  $Z \leftarrow \text{CONCATENATE}(yl,\ U)$ ▷ concatenated along the channel dimension
  $Y \leftarrow AZ$ ▷ Mix
  save Z ▷ For the backwards pass
  return Y
end procedure

procedure LOCALINVBWD($\frac{\partial L}{\partial Y},\ A$)
  load Z
  $\frac{\partial L}{\partial A} \leftarrow \frac{\partial L}{\partial Y} Z^T$ ▷ The weight gradient
  $\Delta Z \leftarrow A^T \frac{\partial L}{\partial Y}$
  $\Delta yl,\ \Delta U \leftarrow \text{UNSTACK}(\Delta Z)$
  $\Delta yl \leftarrow \text{AVGPOOL2x2BWD}(\Delta yl)$
  $\Delta yh \leftarrow \text{COMPLEXMAGBWD}(\Delta U)$
  $\frac{\partial L}{\partial x} \leftarrow \text{DT}\mathbb{C}\text{WTBWD}(\Delta yl,\ \Delta yh)$ ▷ The propagated gradient
  return $\frac{\partial L}{\partial x},\ \frac{\partial L}{\partial A}$
end procedure

Our layer requires us to save the activation $z^{(l+1)}$ for updating the $\tilde{a}$ terms. This has $7C_l$ channels of spatial size $\frac{HW}{4}$. This means that our proposed layer needs to save $\frac{7}{4}HWC_l$ floats, a $\frac{7}{4}$ times memory increase on the standard layer.

5.4.3 Computational Cost

A standard convolutional layer with kernel size $L \times L$ needs $L^2 C_{l+1}$ multiplies per input pixel (of which there are $C_l \times H \times W$).

There is an overhead in doing the wavelet decomposition for each input channel. A separable 2-D discrete wavelet transform (DWT) with 1-D filters of length L will have $2L\left(1 - 2^{-2J}\right)$ multiplies per input pixel for a J scale decomposition. A DT$\mathbb{C}$WT has 4 DWTs for a 2-D input, so its cost is $8L\left(1 - 2^{-2J}\right)$, with $L = 6$ a common size for the filters. It is important to note that unlike the filtering operation, this does not scale with C_{l+1}, the end result being that as C_{l+1} grows, the cost of C_l forward transforms is outweighed by that of the mixing process whose cost is proportional to $C_l C_{l+1}$.

Because we are using a decimated wavelet decomposition, the sample rate decreases after each wavelet layer. The benefit of this is that the mixing process then only works on one quarter the spatial size after the first scale and one sixteenth the spatial after the second

scale. Restricting ourselves to $J = 1$ as we mentioned in section 5.4, the computational cost is then:

$$\underbrace{\frac{7}{4}C_{l+1}}_{\text{mixing}} + \underbrace{36}_{\text{DTCWT}} \quad \text{multiplies per input pixel} \tag{5.4.3}$$

In most CNNs, C_{l+1} is several dozen if not several hundred, which makes (5.4.3) significantly smaller than $L^2 C_{l+1} = 9C_{l+1}$ multiplies for 3×3 convolutions.

5.4.4 Forward and Backward Algorithm

There are two layer hyperparameters to choose in our layer:

- The number of output channels C_{l+1}. This may be restricted by the architecture.
- The variance of the weight initialization for the mixing matrix A.

Assuming we have already chosen these values, then the forward and backward algorithms can be computed with Algorithm Algorithm 5.1.

5.5 Experiments

In this section we examine the effectiveness of our invariant layer by testing its performance on the well known datasets (listed in the order of increasing difficulty):

- MNIST: 10 classes, 6000 images per class, 28×28 pixels per image.
- 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[33]: 200 classes, 500 images per class, 64×64 pixels per image.

Our experiment code is available at https://github.com/fbcotter/invariant_convolution.

5.5.1 Layer Introduction with MNIST

To begin experiments on the proposed locally invariant layer, we look at how well a simple system works on MNIST, and compare it to an equivalent system with convolutions. Because of the small size of the MNIST challenge, we can quickly get results, allowing a large number of trials and a broad search over hyperparameters to be done. In this way, we can use the findings from these experiments to guide our work on more difficult tasks like CIFAR and Tiny ImageNet.

To minimize the effects of learning from other layers, we build a custom small network, as described in Table 5.1. The first two layers are learned convolutional/invariant layers,

followed by a fully connected layer with fixed weights that we can use to project down to the number of output classes. Finally, we add a small learned layer that linearly combines the 10 outputs from the random projection, to give 10 new outputs. This is to facilitate reordering of the outputs to the correct class. This simple network is meant to test the limits of our layer, rather than achieve state of the art performance on MNIST.

Given that our layer is quite different to a standard convolutional layer, we must do a full hyperparameter search over optimizer parameters such as the learning rate, momentum, and weight decay, as well as layer hyperparameters such as the variance of the random initialization for the mixing matrix A.

To simplify the weight variance search, we use Glorot Uniform Initialization [56] and only vary the gain value a:

$$A_{ij} \sim U\left[-a\sqrt{\frac{6}{(C_l + C_{l+1})HW}},\ a\sqrt{\frac{6}{(C_l + C_{l+1})HW}}\right] \tag{5.5.1}$$

where C_l, C_{l+1} are the number of input and output channels as before, and the kernel size is $H = W = 1$ for an invariant layer and $H = W = 3$ for a convolutional layer.

We do a grid search over these hyperparameters and use Hyperband [57] to schedule early stopping of poorly performing runs. Each run has a grace period of 5 epochs and can train for a maximum of 20 epochs. We do not do any learning rate decay. We found the package Tune [58] was very helpful in organising parallel distributed training runs. The hyperparameter options are described in Table 5.2, note that we test $4^4 = 256$ different options.

Once we find the optimal hyperparameters for each network, we then run the two architectures 10 times with different random seeds and report the mean and variance of the accuracy. The results of these runs are listed in Table 5.3.

5.5.1.1 Proposed Expansions

The results from the previous section seem to indicate that our proposed invariant layer is a slightly worse substitute for a convolutional layer. However we believe that this is due to the centred nature of the wavelet bases that were used to generate the z and later the y coefficients. A similar effect was seen in the previous chapter in **??** where the space of shapes attainable by mixing wavelet coefficients in a 3×3 area was much richer than those attainable by only mixing in a 1×1 area.

To test this hypothesis, we change Equation 5.3.4 to be:

$$y^{(l+1)}(f, \mathbf{u}) = \sum_{q \in Q} z^{(l+1)}(q, \mathbf{u}) * (\tilde{a}_f(q)\alpha_f(q, \mathbf{u})) \tag{5.5.2}$$

Where $\alpha_f(q, \mathbf{u})$ is an introduced kernel designed to allow mixing of wavelets from neighbouring spatial locations. We test a range of possible α's each with varying complexity/overhead:

Table 5.1: **Architectures for MNIST hyperparameter experiments.** The activation size rows are offset from the layer description rows to convey the input and output shapes. The project layers in both architectures are unlearned, so all of the learning has to be done by the first two layers and the reshuffle layer.

(a) Invariant Architecture

Activation Size	Layer Name + Info
$1 \times 28 \times 28$	
	inv1, $A \in \mathbb{R}^{7\times 7}$
$7 \times 14 \times 14$	
	inv2, $A \in \mathbb{R}^{49\times 49}$
$49 \times 7 \times 7$	
	unravel
2401×1	
	project, $w \in \mathbb{R}^{2401\times 10}$
10×1	
	reshuffle, $w \in \mathbb{R}^{10\times 10}$
10×1	

(b) Reference Arch with 3×3 convolutions

Activation Size	Layer Name + Info
$1 \times 28 \times 28$	
	conv1, $w \in \mathbb{R}^{7\times 1\times 3\times 3}$
$7 \times 28 \times 28$	
	maxpool1, 2×2
$7 \times 14 \times 14$	
	conv2 $w \in \mathbb{R}^{49\times 7\times 3\times 3}$
$49 \times 14 \times 14$	
	maxpool2, 2×2
$49 \times 7 \times 7$	
	unravel
2401×1	
	project, $w \in \mathbb{R}^{2401\times 10}$
10×1	
	reshuffle, $w \in \mathbb{R}^{10\times 10}$
10×1	

Table 5.2: **Hyperparameter settings for the MNIST experiments.** The weight gain is the term a from Equation 5.5.1. Note that $\log_{10} 3.16 = 0.5$.

Hyperparameter	Values
Learning Rate (lr)	$\{0.0316,\ 0.1,\ 0.316,\ 1\}$
Momentum (mom)	$\{0,\ 0.25,\ 0.5,\ 0.9\}$
Weight Decay (wd)	$\{10^{-5},\ 3.16\times 10^{-5},\ 10^{-4},\ 3.16\times 10^{-4}\}$
Weight Gain (a)	$\{0.5,\ 1.0,\ 1.5,\ 2.0\}$

(a) We randomly shift each of the $7C$ subbands horizontally by $\{-1, 0, 1\}$ pixels, and vertically by $\{-1, 0, 1\}$ pixels. This is determined at the beginning of a training session and is consistent between batches. This theoretically is free to do, but practically it involves convolving with a 3×3 kernel with a single 1 and eight 0's.

(b) Instead of shifting impulses as in the previous option, we can shift a gaussian kernel by one pixel left/right and up/down, making a smoother filter.

(c) Instead of imposing a lowpass/impulse structure, we can set α to be a random 3×3 kernel. This is chosen once at the beginning of training and then is kept fixed between batches.

Table 5.3: **Architecture performance comparison.** Numbers reported are the mean and standard deviation of accuracy over 10 runs with the optimal hyperparameters, θ. Note that for both architectures we found that lr was the most important hyperparameter to choose correctly, and had the largest impact on the performance.

		accuracy	
Architecture	$\theta = \{\text{lr}, \text{mom}, \text{wd}, \text{a}\}$	mean	std
Convolutional	$\{0.1,\ 0.5,\ 10^{-5},\ 1.5\}$	97.3	0.29
Invariant	$\{0.032,\ 0.9,\ 3.2 \times 10^{-5},\ 1.0\}$	96.6	0.26

(d) We can set the 3×3 kernel to be fully learned. This still makes for a novel layer, but now the parameter cost is 9 times higher than the 1×1 conv layer, and 7 times higher than a vanilla 3×3 convolution.

(e) We can take the top three 3×3 DCT coefficients of the $7C$ subbands, allowing us to do something like the previous option but with only a threefold parameter increase. The top three coefficients are the constant, the horizontal and the vertical filters.

Again, we search over the hyperparameter space to find the optimal hyperparameters and then run 10 runs at the best set of hyperparameters, and report the results in Table 5.4. As we expected, adding in random shifts significantly helps the invariant layer. Two systems of note are the shifted impulse (a) system and the learned 3×3 kernel (d) system. The first improves the mean accuracy by 1.3% without any extra learning. The second improves the performance by 2.4% but with a large parameter cost. To explore an equivalent system, we also list in Table 5.4 a modification to the convolutional architecture that uses 5×5 convolutions and $C_1 = 10,\ C_2 = 100$ channels, resulting in a system with comparable parameter cost to (d). We include these results at the bottom of Table 5.4 under 'Wide Convolutional'.

5.5.2 Layer Comparison with CIFAR and Tiny ImageNet

Now we look at expanding our layer to harder datasets, focusing more on the final classification accuracy. We do this again by comparing to a reference architecture. For this task, we choose a VGG-like network as our reference. It has six convolutional layers for CIFAR and eight layers for Tiny ImageNet as shown in 5.5a. The initial number of channels C we use is 64. Despite this simple design, this reference architecture achieves competitive performance for the three datasets.

We perform an ablation study where we progressively swap out convolutional layers for invariant layers keeping the input and output activation sizes the same. As there are 6 layers (or 8 for Tiny ImageNet), there are too many permutations to list the results for swapping out all layers for our locally invariant layer, so we restrict our results to swapping 1 or 2 layers. We also report the accuracy when we swap out all of the layers. **??** reports the

Table 5.4: **Modified architecture performance comparison.** Numbers reported are the mean and standard deviation of accuracy over 10 runs with the optimal hyperparameters, θ. We also list parameter cost and number of multiplies for each layer option, relative to the standard 3×3 convolutional layer to highlight the benefits/drawbacks of each option.

		cost		accuracy	
Architecture	$\theta = \{\text{lr}, \text{mom}, \text{wd}, \text{a}\}$	param	mults	mean	std
Convolutional	$\{0.1,\ 0.5,\ 10^{-5},\ 1.5\}$	1	1	97.3	0.29
Invariant	$\{0.032,\ 0.9,\ 3.2 \times 10^{-5},\ 1.0\}$	$\frac{7}{9}$	$\frac{7}{36}$	96.6	0.26
Shifted impulses (a)	$\{0.32,\ 0.5,\ 10^{-4},\ 1.0\}$	$\frac{7}{9}$	$\frac{7}{36}$	97.9	0.25
Shifted gaussians (b)	$\{1.0,\ 0.0,\ 10^{-5},\ 1.0\}$	$\frac{7}{9}$	$\frac{7}{4}$	97.7	0.56
Random 3×3 kernel (c)	$\{1.0,\ 0.9,\ 10^{-5},\ 1.0\}$	$\frac{7}{9}$	$\frac{7}{4}$	95.8	1.01
Learned 3×3 kernel (d)	$\{0.1,\ 0.5,\ 10^{-4},\ 1.0\}$	**7**	$\frac{7}{4}$	**99.0**	0.12
Learned 3 DCT coeffs (e)	$\{1.0,\ 0.0,\ 10^{-5},\ 1.0\}$	$\frac{7}{3}$	$\frac{7}{4}$	98.1	0.37
Wide Convolutional	$\{0.32,\ 0.5,\ 10^{-5},\ 1.5\}$	7	7	98.7	0.25

top-1 classification accuracies for CIFAR-10, CIFAR-100 and Tiny ImageNet. In the table, 'invX' means that the 'convX' layer from 5.5a was replaced with an invariant layer. If the convolutional layer is before a pooling layer, then we do not interpolate the output of the invariant layer and we remove the pooling layer.

In addition to testing the original 1×1 gain, we also report results for using the 'shifted impulse' and 'learned 3×3' modified architectures from subsubsection 5.5.1.1.

This network is 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).

Interestingly, we see improvements when one or two invariant layers are used near the start of a system, but not for the first layer. In particular, the best position for the invariant layer seems to be just before a sample rate change. Recalling that the magnitude operation in the ScatterNet effectively demodulates the energy from higher spatial frequencies to a lower band, it intuitively makes sense that good places for scattering layers are at positions where you want to downsample.

5.5.3 Network Comparison

In the previous section, we examined how the locally invariant layer performs when directly swapped in for a convolutional layer in an architecture designed for convolutional layers. In this section, we look at how it performs in a Hybrid ScatterNet-like [31], [32], network.

To build the second order ScatterNet, we stack two invariant layers on top of each other. For 3 input channels, the output of these layers has $3(1+6+6+36) = 147$ channels at $1/16$ the spatial input size. We then use 4 convolutional layers, similar to convC to convF in **??** with $C = 96$. In addition, we use dropout after these later convolutional layers with drop probability $p = 0.3$.

We compare a ScatterNet with no learning in between scattering orders (ScatNet A) to one with our proposal for a learned mixing matrix A (ScatNet B). Finally, we also test the hypothesis seen from subsection 5.5.2 about putting conv layers before an inv layer, and test a version with a small convolutional layer before ScatNets A and B, taking the input from 3 to 16 channels, and call these ScatNet architectures ScatNet C and D respectively.

See Table 5.7 for results from these experiments. It is clear from the improvements that the mixing layer helps the Scattering front end. Interestingly, ScatNet C and D further improve on the A and B versions (albeit with a larger parameter and multiply cost than the mixing operation). This reaffirms that there may be benefit to add learning before as well as inside the ScatterNet.

For comparison, we have also listed the performance of other architectures as reported by their authors in order of increasing complexity. Our proposed ScatNet D achieves comparable performance with the the All Conv, VGG16 and FitNet architectures. The Deep[59] and Wide[60] ResNets perform best, but with very many more multiplies, parameters and layers.

ScatNets A to D with 6 layers like convC to convG from **??** after the scattering, achieve $58.1, 59.6, 60.8$ and 62.1% top-1 accuracy on Tiny ImageNet. As these have more parameters and multiplies from the extra layers we exclude them from Table 5.7.

5.6 Conclusion

In this work we have proposed a new learnable scattering layer, dubbed the locally invariant convolutional layer, tying together ScatterNets and CNNs. We do this by adding a mixing between the layers of ScatterNet allowing the learning of more complex shapes than the ripples seen in [63]. This invariant layer can easily be shaped to allow it to drop in the place of a convolutional layer, theoretically saving on parameters and computation. However, care must be taken when doing this, as our ablation study showed that the layer only improves upon regular convolution at certain depths. Typically, it seems wise to use the invariant layer right before a sample rate change.

We have developed a system that allows us to pass gradients through the Scattering Transform, something that previous work has not yet researched. Because of this, we were able to train end-to-end a system that has a ScatterNet surrounded by convolutional layers and with our proposed mixing. We were surprised to see that even a small convolutional layer before Scattering helps the network, and a very shallow and simple Hybrid-like ScatterNet was able to achieve good performance on CIFAR-10 and CIFAR-100.

Table 5.5: **CIFAR and Tiny ImageNet Base Architecture.** Reference architecture used for experiments on CIFAR-10, CIFAR-100 and Tiny ImageNet. This architecture is based off the VGG[35] architecture. C is a hyperparameter that controls the network width, we use $C = 64$ for our initial tests. The activation size rows are offset from the layer description rows to convey the input and output shapes.

(a) CIFAR Architecture

Activation Size	Layer Name + Info
$3 \times 32 \times 32$	
	convA, $w \in \mathbb{R}^{C \times 3 \times 3 \times 3}$
$C \times 32 \times 32$	
	convB, $w \in \mathbb{R}^{C \times C \times 3 \times 3}$
$C \times 32 \times 32$	
	pool1, max pooling 2×2
$C \times 16 \times 16$	
	convC, $w \in \mathbb{R}^{2C \times C \times 3 \times 3}$
$2C \times 16 \times 16$	
	convD, $w \in \mathbb{R}^{2C \times 2C \times 3 \times 3}$
$2C \times 16 \times 16$	
	pool2, max pooling 2×2
$2C \times 8 \times 8$	
	convE, $w \in \mathbb{R}^{4C \times 2C \times 3 \times 3}$
$4C \times 8 \times 8$	
	convF, $w \in \mathbb{R}^{4C \times 4C \times 3 \times 3}$
$4C \times 8 \times 8$	
	avg, 8×8 average pooling
$4C \times 1 \times 1$	
	fc1, fully connected layer
10×1, 100×1	

(b) Tiny ImageNet Architecture

Activation Size	Layer Name + Info
$3 \times 64 \times 64$	
	convA, $w \in \mathbb{R}^{C \times 3 \times 3 \times 3}$
$C \times 64 \times 64$	
	convB, $w \in \mathbb{R}^{C \times C \times 3 \times 3}$
$C \times 64 \times 64$	
	pool1, max pooling 2×2
$C \times 32 \times 32$	
	convC, $w \in \mathbb{R}^{2C \times C \times 3 \times 3}$
$2C \times 32 \times 32$	
	convD, $w \in \mathbb{R}^{2C \times 2C \times 3 \times 3}$
$2C \times 32 \times 32$	
	pool2, max pooling 2×2
$2C \times 16 \times 16$	
	convE, $w \in \mathbb{R}^{4C \times 2C \times 3 \times 3}$
$4C \times 16 \times 16$	
	convF, $w \in \mathbb{R}^{4C \times 4C \times 3 \times 3}$
$4C \times 16 \times 16$	
	pool3, max pooling 2×2
$4C \times 8 \times 8$	
	convG, $w \in \mathbb{R}^{8C \times 4C \times 3 \times 3}$
$8C \times 8 \times 8$	
	convH, $w \in \mathbb{R}^{8C \times 8C \times 3 \times 3}$
$8C \times 8 \times 8$	
	avg, 8×8 average pooling
$8C \times 1$	
	fc1, fully connected layer
200×1	

Table 5.6: Results for testing VGG like architecture with convolutional and invariant layers on several datasets. An architecture with 'invX' means the equivalent convolutional layer 'convX' from **??** was swapped for our proposed layer. The top row is the reference architecture using all convolutional layers. The 'A' architecture is the originally proposed gain layer, the 'B' architecture uses the gainlayer with random shifting of the activations, and the 'C' architecture changes the mixing to be a learned 3×3 kernel acting on the invariant coefficients. Numbers are averages over 5 runs. **TODO: Rerun C models with different hypes**.

	CIFAR-10			CIFAR-100			Tiny ImgNet		
	A	B	C	A	B	C	A	B	C
reference	92.6	-	-	72.0	-	-	59.3	-	-
invA	92.3	92.4	92.3	71.4	71.4	70.7	60.0	60.5	59.2
invB	93.4	93.2	93.4	73.4	72.9	72.3	61.3	60.7	61.0
invC	93.7	93.2	93.4	73.7	73.5	73.0	61.7	62.0	61.0
invD	92.7	92.6	92.8	72.5	72.4	72.1	61.5	61.2	59.7
invE	91.8	92.2	92.5	71.1	71.7	71.8	60.7		
invF	92.4	92.9	92.7	70.3	71.1	71.7	59.8	59.9	58.6
invA, invB	92.6	91.8	92.4	71.1	71.2	70.2	58.9	60.4	59.3
invB, invC	92.2	91.8	92.3	71.3	70.9	70.5	59.6	59.2	59.3
invC, invD	92.5	92.0	92.9	72.1	72.2	72.2	60.7	61.4	60.0
invD, invE	90.4	91.2	91.9	68.6	69.7	70.5	58.9	59.1	58.1
invA, invC	92.2	92.0	92.5	71.7	71.0	70.7	59.4	59.6	60.1
invB, invD	92.8	92.3	82.8	73.0	72.3	71.8	61.4	60.8	59.5
invC, invE	91.5	92.0	92.3	70.9	71.7	72.4	61.2		

Table 5.7: **Hybrid ScatterNet top-1 classification accuracies on CIFAR-10 and CIFAR-100.** N_l is the number of learned convolutional layers, #param is the number of parameters, and #mults is the number of multiplies per $32 \times 32 \times 3$ image. An asterisk indicates that the value was estimated from the architecture description.

	Arch. Properties			Top 1 Accuracies	
Arch. Name	N_l	#Mparam	#Mmults	CIFAR-10	CIFAR-100
ScatNet A	4	2.6	165	89.4	67.0
ScatNet B	6	2.7	167	91.1	70.7
ScatNet C	5	3.7	251	91.6	70.8
ScatNet D	7	4.3	294	93.0	73.5
All Conv[48]	8	1.4	281*	92.8	66.3
VGG16[61]	16	138*	313*	91.6	-
FitNet[62]	19	2.5	382	91.6	65.0
ResNet-1001[59]	1000	10.2	4453*	95.1	77.3
WRN-28-10[60]	28	36.5	5900*	96.1	81.2

Chapter 6

Learning in the Wavelet Domain

In this chapter we move away from the ScatterNet ideas from the previous chapters and instead look at using the wavelet domain as a new space in which to learn. In particular the ScatterNet, and even the learnable ScatterNet proposed in the previous chapter, are built around taking complex magnitudes of the highpass wavelets. This inherently builds invariance to shifts but at the cost of making things smoother. In many ways this was beneficial, as it allowed us to subsample the output and we saw that the scattering layers worked well just before downsampling stages of a CNN. However, we would now like to explore if it is possible and at all beneficial to learn with wavelets without taking the complex magnitude. This means that the frequency support of our activations will remain in the same space in the Fourier domain.

The inspiration to this chapter is the hope that learning in the frequency/wavelet domain may afford simpler filters than learning in the pixel domain. A classic example of this is the first layer filters in AlexNet shown in Figure 6.1. These could be parameterized with only a few nonzero wavelet coefficients, or alternatively, we could take a decomposition of each input channel and keep individual subbands (or equivalently, attenuate other bands), then take the inverse wavelet transform.

Our experiments show that ... **FINISH ME**

6.1 A Summary of Choices

As mentioned in the inspiration for this chapter, many filters that have complex support in the pixel domain would have simple support in the wavelet domain, but as the previous section showed, naively reparameterizing things in a different domain may not afford us any benefit in the optimization procedure.

There are two possible ways we can try to leverage the wavelet domain for learning:

1. We can reparameterize filters in the wavelet domain if we use nonlinear optimizers like ADAM, or ℓ_1 regularization to impose sparsity. This is presented in **??**.

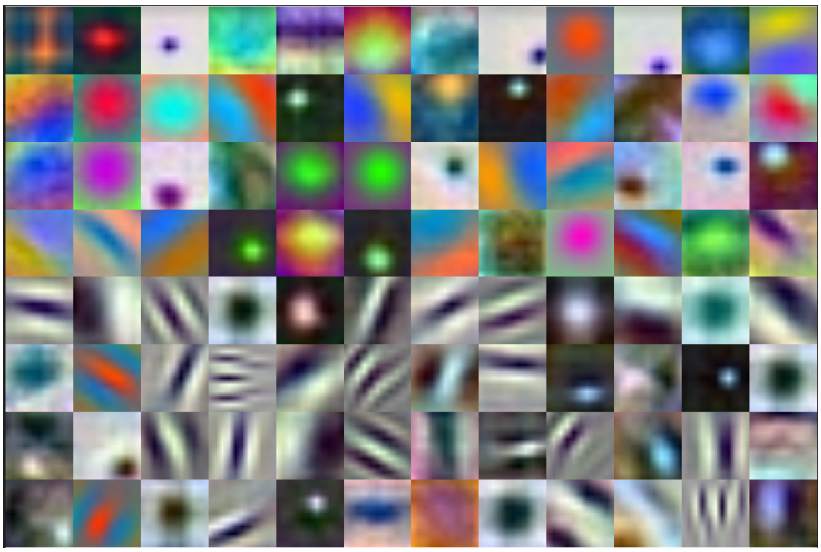


Figure 6.1: **First layer filters of the AlexNet architecture.** The first layer filters of the seminal AlexNet [50] are an inspiration for considering learning filters in the wavelet domain. Each of these 11×11 filters would only require a handful of non zero coefficients in the wavelet domain. The weights shown here were taken from a pretrained network from torchvision [64].

2. We can take wavelet transforms of the inputs and learn filters on the wavelet coefficients. We can also apply nonlinearities to the wavelet coefficients such as wavelet shrinkage. On the output of this, we have the choice of either staying in the wavelet domain or returning to the pixel domain with an inverse wavelet transform. This is presented in **??**

This chapter explores both possible methods and the merits and drawbacks of each.

6.2 Related Work

6.2.1 Wavelets as a Front End

Fujieda et. al. use a DWT in combination with a CNN to do texture classification and image annotation [65], [66]. In particular, they take a multiscale wavelet transform of the input image, combine the actviations at each scale independently with learned weights, and feed these back into the network where the activation resolution size matches the subband resolution. The architecture block diagram is shown in Figure 6.2, taken from the original paper. This work found that their dubbed 'Wavelet-CNN' could outperform competetive non wavelet based CNNs on both texture classification and image annotation.

Several works also use wavelets in deep neural networks for super-resolution [67] and for adding detail back into dense pixel-wise segmentation tasks [68]. These typically save wavelet

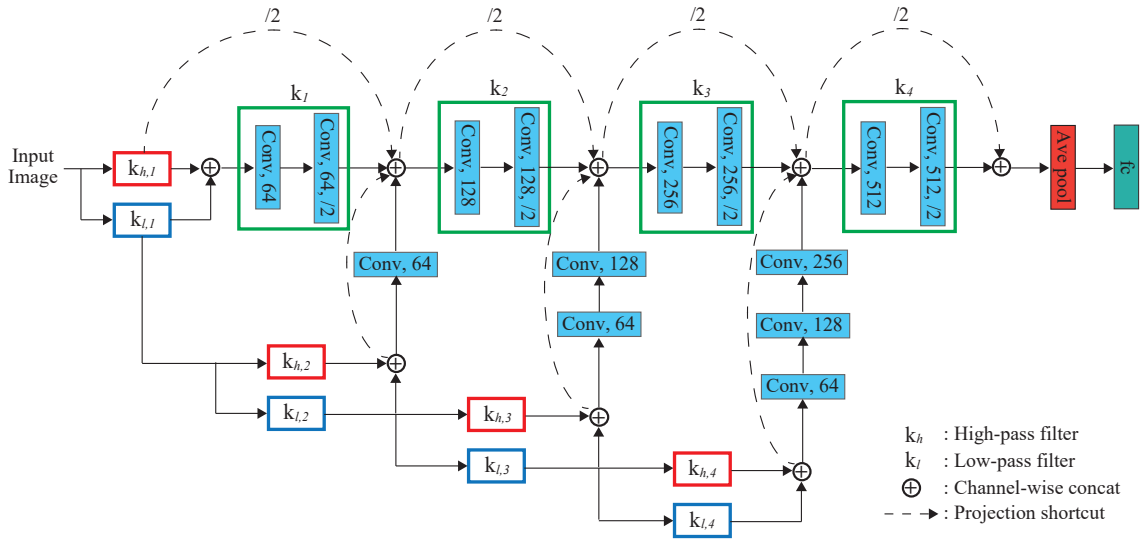


Figure 6.2: **Architecture using the DWT as a frontend to a CNN.** Figure 1 from [66]. Fujieda et. al. take a multiscale wavelet decomposition of the input before passing the input through a standard CNN. They learn convolutional layers independently on each subband and feed these back into the network at different depths, where the resolution of the subband and the network activations match.

coefficients and use them for the reconstruction phase, so are a little less applicable than the first work.

6.2.2 Parameterizing filters in Fourier Domain

In "Spectral Representations for Convolutional Neural Networks" [69], Rippel et. al. explore parameterization of filters in the DFT domain. Note that they do not necessarily do the convolution in the Frequency domain, they simply parameterize a filter $\mathbf{w} \in \mathbb{R}^{F \times C \times K \times K}$ as a set of fourier coefficients $\hat{\mathbf{w}} \in \mathbb{C}^{F \times C \times K \times \lceil K/2 \rceil}$ (the reduced spatial size is a result of enforcing that the inverse DFT of their filter to be real, so the parameterization is symmetric). On the forward pass of the neural network, they take the inverse DFT of $\hat{\mathbf{w}}$ to obtain $\mathbf{w}$ and then convolve this with the input $\mathbf{x}$ as a normal CNN would do.[1].

6.3 Introduction

We would like to explore the possibility of using the wavelet domain as a space to learn. Unlike the work of [69] which only parameterized filters in the wavelet domain and transformed the

[1]The convolution may be done by taking both the image and filter back into the fourier space but this is typically decided by the framework, which selects the optimal convolution strategy for the filter and input size. Note that there is not necessarily a saving to be gained by enforcing it to do convolution by product of FFTs, as the FFT size needed for the filter will likely be larger than $K \times K$, which would require resampling the coefficients

filters back to the pixel domain to do convolution, this chapter explores learning wholly in the wavelet domain. I.e., we want to take a wavelet decomposition of the input and learn gains to apply to these coefficients, and optionally return to the pixel domain.

As neural network training involves presenting thousands of training samples on memory limited GPUs, we want our layer to be fast and as memory efficient as possible. To achieve this we would ideally choose to use a critically sampled filter bank implementation. The fast 2-D Discrete Wavelet Transform (DWT) is a possible option, but it has two drawbacks: it has poor directional selectivity and any alteration of wavelet coefficients will cause the aliasing cancelling properties of the reconstructed signal to disappear. Another option is to use the DT$\mathbb{C}$WT [4]. This comes with a memory overhead which we discuss more in subsection 6.6.1, but it enables us to have have better directional selectivity and allows for the possibility of returning to the pixel domain with minimal aliasing [18].

We will look at the merits and drawbacks of both options, discuss how to implement them, and finally run some experiments exploring how well these layers work.

6.3.1 Invertible Transforms and Optimization

Note that an important point should be laboured about reparameterizing filters in either the wavelet or Fourier domains. That is that any invertible linear transform of the parameter space will not change the updates if a linear optimization scheme (like standard gradient descent, or SGD with momentum) is used. This is proved in **??**.

6.4 Background and Notation

As we now want to consider the DWT and the DT$\mathbb{C}$WT which are both implemented as filter bank systems, we deviate slightly from the notation in the previous chapter (which was inspired by sampling a continuous wavelet transform).

Firstly, instead of talking about the continuous spatial variable $\mathbf{u}$, we now consider the discrete spatial variable $\mathbf{n} = [n_1, n_2]$. We switch to square brackets to make this clearer. With the new discrete notation, the output of a CNN at layer l is:

$$x^{(l)}[c, \mathbf{n}], \quad c \in \{0, \dots C_l - 1\}, \mathbf{n} \in \mathbb{Z}^2 \tag{6.4.1}$$

where c indexes the channel dimension. We also make use of the 2-D Z-transform to simplify our analysis:

$$X(\mathbf{z}) = \sum_{n_1} \sum_{n_2} x[n_1, n_2] z_1^{-n_1} z_2^{-n_2} = \sum_{\mathbf{n}} x[c, \mathbf{n}] \mathbf{z}^{-\mathbf{n}} \tag{6.4.2}$$

As we are working with three dimensional arrays (two spatial and one channel) but are only doing convolution in two, we introduce a slightly modified 2-D Z-transform which includes

the channel index:

$$X(c,\mathbf{z}) = \sum_{n_1}\sum_{n_2} x[c,n_1,n_2] z_1^{-n_1} z_2^{-n_2} = \sum_{\mathbf{n}} x[c,\mathbf{n}]\mathbf{z}^{-\mathbf{n}} \tag{6.4.3}$$

Recall that a typical convolutional layer in a standard CNN gets the next layer's output in a two-step process:

$$y^{(l+1)}[f,\mathbf{n}] = \sum_{c=0}^{C_l-1} x^{(l)}[c,\mathbf{n}] * h_f^{(l)}[c,\mathbf{n}] \tag{6.4.4}$$

$$x^{(l+1)}[f,\mathbf{u}] = \sigma\left(y^{(l+1)}[f,\mathbf{u}]\right) \tag{6.4.5}$$

In shorthand, we can reduce the action of the convolutional layer in (6.4.4) to $\mathcal{H}$, saying:

$$y^{(l+1)} = \mathcal{H}x^{(l)} \tag{6.4.6}$$

With the new Z-transform notation introduced in (6.4.3), we can rewrite (6.4.4) as:

$$Y^{(l+1)}(f,\mathbf{z}) = \sum_{c=0}^{C_l-1} X^{(l)}(c,\mathbf{z})\, H_f^{(l)}(c,\mathbf{z}) \tag{6.4.7}$$

Note that we cannot rewrite (6.4.5) with Z-transforms as it is a nonlinear operation.

Also recall that with multirate systems, upsampling by M takes $X(z)$ to $X(z^M)$ and downsampling by M takes $X(z)$ to $\frac{1}{M}\sum_{k=0}^{M-1} X(W_M^k z^{1/k})$ where $W_M^k = e^{\frac{j2\pi k}{M}}$. We will drop the M subscript below unless it is unclear of the sample rate change, simply using W^k.

6.4.1 DWT Notation

In 2-D, a J scale DWT gives $3J+1$ coefficients, 3 sets of bandpass coefficients for each scale, representing the horizontal, vertical and diagonal regions of the frequency plane, and 1 set of lowpass coefficients at the final scale. Let us refer to the DWT wavelet coefficients with the typewriter font $\mathtt{u}$ and write the J scale DWT as:

$$\mathrm{DWT}_J(x) = \mathtt{u}_{lp}, \{\mathtt{u}_{j,k}\}_{1\le j\le J, 1\le k\le 3} \tag{6.4.8}$$

If x is a batch of 2-D images with multiple channels, the DWT is done indpendently on each channel in each minibatch sample. I.e. if the input is a minibatch of N samples with C channels of images of spatial size $H\times W$, then $x\in\mathbb{R}^{N\times C\times H\times W}$ and:

$$\mathtt{u}_{lp} \in \mathbb{R}^{N\times C\times \frac{H}{2^J}\times\frac{W}{2^J}} \tag{6.4.9}$$

$$\mathtt{u}_{j,k} \in \mathbb{R}^{N\times C\times \frac{H}{2^J}\times\frac{W}{2^J}} \tag{6.4.10}$$

Sometimes we may want to refer to all of the subband coefficients at a single scale with one variable, in which case we will simply drop the k subscript and call them $\mathtt{u}_j$. Additionally if we want to refer to all the coefficients (lowpass and bandpass) we will call them $\mathtt{u}$.

6.4.2 DT$\mathbb{C}$WT Notation

Unlike the DWT, a J scale DT$\mathbb{C}$WT gives $6J+1$ coefficients, 6 sets of complex bandpass coefficients for each scale (representing the oriented bands from 15 to 165 degrees) and 1 set of real lowpass coefficients.

Although we will only ever use either the DWT or the DT$\mathbb{C}$WT, to avoid confusion we use a regular font u to refer to the DT$\mathbb{C}$WT coefficients of x:

$$\text{DT}\mathbb{C}\text{WT}_J(x) = u_{lp}, \{u_{j,k}\}_{1\leq j\leq J, 1\leq k\leq 6} \tag{6.4.11}$$

Each of these coefficients then has size:

$$u_{lp} \in \mathbb{R}^{N\times C\times \frac{H}{2^{J-1}}\times \frac{W}{2^{J-1}}} \tag{6.4.12}$$

$$u_{j,k} \in \mathbb{C}^{N\times C\times \frac{H}{2^{J}}\times \frac{W}{2^{J}}} \tag{6.4.13}$$

Note that the lowpass coefficients are twice as large as in a fully decimated transform, a feature of the redundancy of the DT$\mathbb{C}$WT.

Again if we ever want to refer to all the subbands at a given scale, we will drop the k subscript and call them u_j. Likewise, u refers to the whole set of DT$\mathbb{C}$WT coefficients.

When we want to be agnostic of the chosen transform type, we use $\mathcal{W}$ and $\mathcal{W}^{-1}$ to denote the forward and inverse transform.

6.5 The Wavelet Gain Layer

At the beginning of each stage of a neural network we have the activations $x^{(l)}$. Naturally, all of these activations have their equivalent DWT coefficients $\mathtt{u}^{(l)}$ and DT$\mathbb{C}$WT coefficients $u^{(l)}$.

From (6.4.4), convolutional layers also have intermediate activations $y^{(l)}$. Let us differentiate these from the x coefficients and modify (6.4.8) and (6.4.11) to say the DWT of $y^{(l)}$ gives $\mathtt{v}$ and the DT$\mathbb{C}$WT of $y^{(l)}$ gives v.

We now propose the wavelet gain layers $\mathtt{G}$ and G for the DWT and DT$\mathbb{C}$WT respectively, or $\mathcal{G}$ for an implementation agnostic layer. The name 'gain layer' comes from the inspiration for this chapter's work, in that the first layer of CNN could be nearly done in the wavelet domain by setting subband gains to 0 and 1.

The gain layer $\mathcal{G}$ can be used instead of a convolutional layer. It is designed to work on the wavelet coefficients of an activation, $\mathtt{u}$ and u to give outputs $\mathtt{v}$ and v.

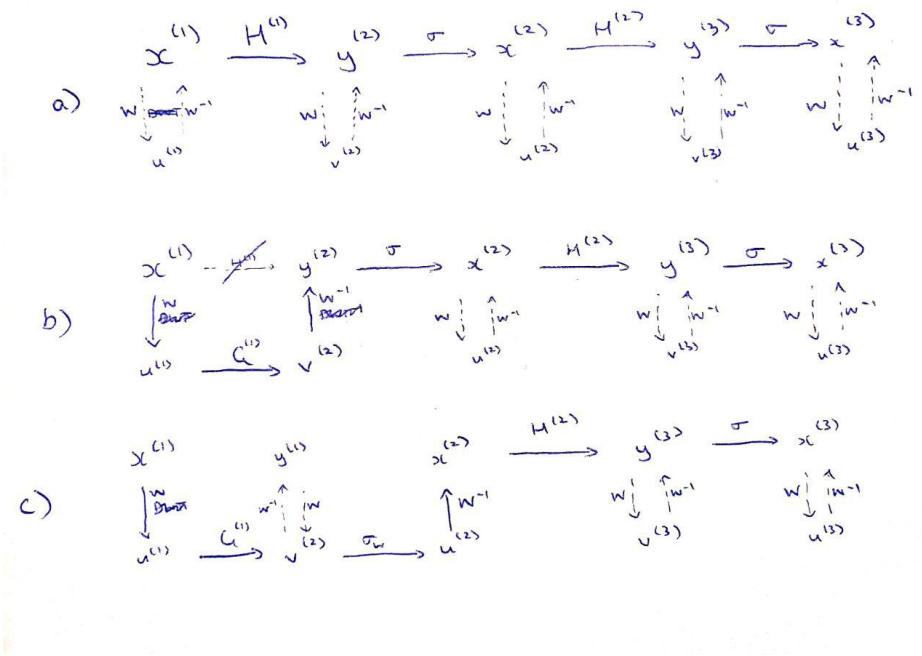


Figure 6.3: **Proposed new forward pass in the wavelet domain.** Two network layers with some possible options for processing. Solid lines denote the evaluation path and dashed lines indicate relationships. In (a) we see a regular convolutional neural network. We have included the dashed lines to make clear what we are denoting as u and v with respect to their equivalents x and y. In (b) we get to $y^{(2)}$ through a different path. First we take the wavelet transform of $x^{(1)}$ to give $u^{(1)}$, apply a wavelet gain layer $\mathcal{G}^{(1)}$, and take the inverse wavelet transform to give $y^{(2)}$. The cross through $\mathcal{H}^{(1)}$ indicates that this path is no longer present. Note that there may not be any possible $\mathcal{G}^{(1)}$ to make $y^{(2)}$ from (b) equal $y^{(2)}$ from (a). In (c) we have stayed in the wavelet domain longer, and applied a wavelet nonlinearity σ_w to give $u^{(2)}$. We then return to the pixel domain to give $x^{(2)}$ and continue on from there in the pixel domain.

This can be seen as breaking the convolutional path in Figure 6.3 and taking a new route to get to the next layer's coefficients. From here, we can return to the pixel domain by taking the corresponding inverse wavelet transform $\mathcal{W}^{-1}$. Alternatively, we can stay in the wavelet domain and apply a wavelet based nonlinearity σ_w to give the next layer's u coefficients. Ultimately we would like to explore architecture design with arbitrary sections in the wavelet and pixel domain, but to do this we must first explore:

- How effective $\mathcal{G}$ is at replacing $\mathcal{H}$.

- How effective σ_w is at replcaing σ.

6.5.1 The DWT Gain Layer

As mentioned previously, modifying the wavelet coefficients of a critically sampled DWT will necessarily result in a loss of the aliasing cancelling properties. However, in a deep neural network, this is not as obviously a bad thing as it is for denoising or deconvolution. For this reason, we note that there may be a problem in using a DWT, but proceed nonetheless.

For each subband in our J scale system, we introduce a gain term $\mathtt{g}$. Let us specify our input $\mathtt{u}$ has C_l channels, and we would like our output $\mathtt{v}$ to have C_{l+1} channels, then $\mathtt{g}$ is made up of:

$$\begin{aligned}
\mathtt{g}_{lp} &\in \mathbb{R}^{C_{l+1}\times C_l \times k_{lp} \times k_{lp}} && (6.5.1)\\
\mathtt{g}_{1,1} &\in \mathbb{R}^{C_{l+1}\times C_l \times k_1 \times k_1} && (6.5.2)\\
\mathtt{g}_{1,2} &\in \mathbb{R}^{C_{l+1}\times C_l \times k_1 \times k_1} && (6.5.3)\\
&\vdots && (6.5.4)\\
\mathtt{g}_{J,3} &\in \mathbb{R}^{C_{l+1}\times C_l \times k_J \times k_J} && (6.5.5)
\end{aligned}$$

Note that we have allowed for different kernel spatial sizes for the lowpass and for each bandpass scale $\mathtt{g}_j$. This is to allow for the flexibility of putting more emphasis on certain frequency areas if desired.

With these gains, we define $\mathtt{v} = \mathtt{G}\mathtt{u}$ to be:

$$\begin{aligned}
\mathtt{v}_{lp}[f,\mathbf{n}] &= \sum_{c=0}^{C_l-1} \mathtt{u}_{lp}[c,\mathbf{n}] * \mathtt{g}_{lp}[f,c,\mathbf{n}] && (6.5.6)\\
\mathtt{v}_{1,1}[f,\mathbf{n}] &= \sum_{c=0}^{C_l-1} \mathtt{u}_{1,1}[c,\mathbf{n}] * \mathtt{g}_{1,1}[f,c,\mathbf{n}] && (6.5.7)\\
\mathtt{v}_{1,2}[f,\mathbf{n}] &= \sum_{c=0}^{C_l-1} \mathtt{u}_{1,2}[c,\mathbf{n}] * \mathtt{g}_{1,2}[f,c,\mathbf{n}] && (6.5.8)\\
&\vdots && (6.5.9)\\
\mathtt{v}_{J,3}[f,\mathbf{n}] &= \sum_{c=0}^{C_l-1} \mathtt{u}_{J,3}[c,\mathbf{n}] * \mathtt{g}_{J,3}[f,c,\mathbf{n}] && (6.5.10)
\end{aligned}$$

I.e., we do independent mixing at each of the different subbands. For 1×1 kernels, this is simply a matrix multiply of the wavelet coefficients. This is shown visually in 6.8a.

6.5.1.1 The Output

To explore the action of the gain layer $\mathtt{G}$ on DWT coefficients, let use examine a 1-D, single scale, and single channel system. We want to find the action of the layer without any

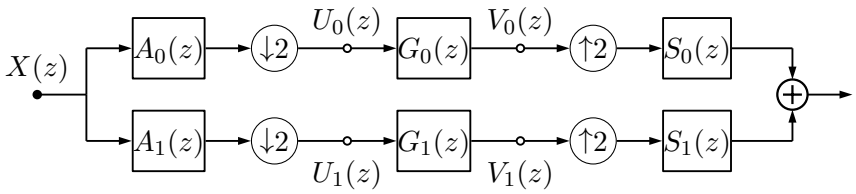


Figure 6.4: **Block Diagram of 1-D DWT Gain Layer.** Here we show the low and highpass for a single scale 1-D DWT, Gain Layer and inverse DWT. The gain layer has gains $\mathtt{g}_0$ and $\mathtt{g}_1$.

nonlinearities, i.e.,

$$y = \mathcal{W}^{-1}\mathtt{G}\mathcal{W}x \tag{6.5.11}$$

or the path taken in Figure 6.3b.

Let us call the low and highpass analysis filters of the DWT A_0 and A_1, and the synthesis filters S_0 and S_1 (these are normally called H and G, but we keep those letters reserved for the CNN and gain layer filters). From the perfect reconstruction property of the DWT, we know

$$A_0(z)S_0(z) + A_1(z)S_1(z) = 2 \tag{6.5.12}$$

and from the aliasing cancellation property of the DWT, we know:

$$A_0(-z)S_0(z) + A_1(-z)S_1(z) = 0 \tag{6.5.13}$$

If we add in gains G_0 and G_1 to the low and highpass coefficients, as shown in Figure 6.4, then the output is:

$$\begin{aligned} Y(z) = &\frac{1}{2}X(z)\left[A_0(z)S_0(z)G_0(z^2) + A_1(z)S_1(z)B(z^2)\right] + \\ &\frac{1}{2}X(-z)\left[A_0(-z)S_0(z)G_0(z^2) + A_1(-z)S_1(z)G_1(z^2)\right] \end{aligned} \tag{6.5.14}$$

If we let $D(z) = G_1(z^2) - G_0(z^2)$ then the above becomes:

$$Y(z) = \frac{1}{2}X(z)\left[A_0(z)S_0(z)G_0(z^2) + A_1(z)S_1(z)G_1(z^2)\right] + \frac{1}{2}X(-z)D(z)A_1(-z)S_1(z) \tag{6.5.15}$$

6.5.1.2 Backpropagation

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

$$\Delta X(z) = H(z^{-1})\Delta Y(z) \tag{6.5.16}$$

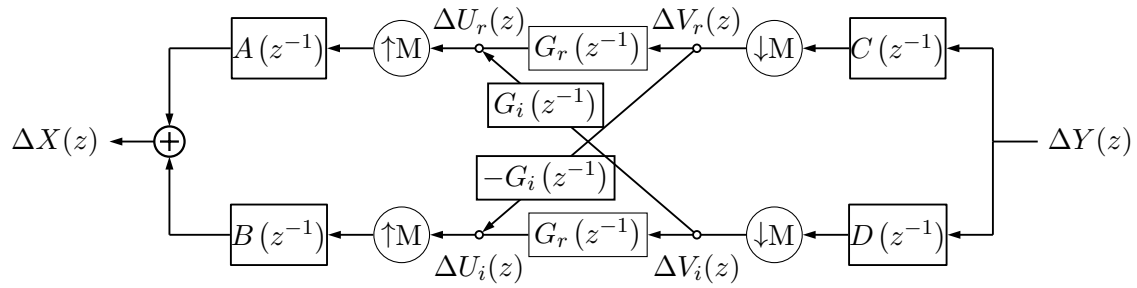


Figure 6.5: **Forward and backward block diagrams for** DT$\mathbb{C}$WT **gain layer.** Based on Figure 4 in [18]. Ignoring the G gains, the top and bottom paths (through A, C and B, D respectively) make up the the real and imaginary parts for *one subband* of the dual tree system. Combined, $A + jB$ and $C - jD$ make the complex filters necessary to have support on one side of the Fourier domain (see Figure 6.7). Adding in the complex gain $G_r + jG_i$, we can now attenuate/shape the impulse response in each of the subbands. To allow for learning, we need backpropagation. The bottom diagram indicates how to pass gradients $\Delta Y(z)$ through the layer. Note that upsampling has become downsampling, and convolution has become convolution with the time reverse of the filter (represented by z^{-1} terms).

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.

Assume we already have access to the quantity $\Delta Y(z)$ (this is the input to the backwards pass). **??** illustrates the backpropagation procedure. An interesting result is that for orthogonal wavelet transforms, $S(z^{-1}) = A(z)$, so the backwards pass of an inverse wavelet transform is equivalent to doing a forward wavelet transform. Similarly, the backwards pass of the forward transform is equivalent to doing the inverse transform. The weight update gradients are then calculated by finding $\Delta V_i(z) = S_i(z^{-1})Y(z)$ for $i = 0,\ 1$, and then convolving with the time reverse of the saved wavelet coefficients from the forward pass - $U_i(z^{-1})$.

$$\Delta G_i(z) = \Delta V_i(z) U_i(z^{-1}) \tag{6.5.17}$$

Unsurprisingly, the passthrough gradients have similar form to (6.5.15)

$$\Delta X(z) = \frac{1}{2}\Delta Y(z)\left[A_0(z)S_0(z)G_0(z^{-2}) + A_1(z)S_1(z)G_1(z^{-2})\right] + \frac{1}{2}\Delta Y(-z)D(z)A_1(-z)S_1(z) \tag{6.5.18}$$

Note that we only need to evaluate (6.5.17), (6.5.18) over the support of $G(z)$ i.e., if it is a single number we only need to calculate $\Delta G(z)|_{z=0}$.

6.5.2 The DT$\mathbb{C}$WT Gain Layer

The DT$\mathbb{C}$WT gain layer is the same in principle to the action of the DWT gain layer, but due to the different properties of the two transforms, the implementation is slightly different. Now that we have the framework for applying a complex gain at one subband, we can extend this to all of the subbands in the DT$\mathbb{C}$WT. We also reintroduce the channel dimension.

To do the mixing across the C_l channels at each subband, giving C_{l+1} output channels, we introduce the learnable filters:

$$\begin{aligned}
g_{lp} &\in \mathbb{R}^{C_{l+1}\times C_l\times k_{lp}\times k_{lp}} && (6.5.19)\\
g_{1,1} &\in \mathbb{C}^{C_{l+1}\times C_l\times k_1\times k_1} && (6.5.20)\\
g_{1,2} &\in \mathbb{C}^{C_{l+1}\times C_l\times k_1\times k_1} && (6.5.21)\\
&\vdots && (6.5.22)\\
g_{J,6} &\in \mathbb{C}^{C_{l+1}\times C_l\times k_J\times k_J} && (6.5.23)
\end{aligned}$$

where k, k are the sizes of the mixing kernels. These could be 1×1 for simple gain control, or could be larger, say 3×3, to do more complex filtering on the subbands.

With these gains we define $v = Gu$ to be:

$$\begin{aligned}
v_{lp}[f,\mathbf{n}] &= \sum_{c=0}^{C_l-1} u_{lp}[c,\mathbf{n}] * g_{lp}[f,c,\mathbf{n}] && (6.5.24)\\
v_{1,1}[f,\mathbf{n}] &= \sum_{c=0}^{C_l-1} u_{1,1}[c,\mathbf{n}] * g_{1,1}[f,c,\mathbf{n}] && (6.5.25)\\
v_{1,2}[f,\mathbf{n}] &= \sum_{c=0}^{C_l-1} u_{1,2}[c,\mathbf{n}] * g_{1,2}[f,c,\mathbf{n}] && (6.5.26)\\
&\vdots && (6.5.27)\\
v_{J,6}[f,\mathbf{n}] &= \sum_{c=0}^{C_l-1} u_{J,6}[c,\mathbf{n}] * g_{J,6}[f,c,\mathbf{n}] && (6.5.28)
\end{aligned}$$

Note that for complex signals a, b the convolution $a * b$ is defined as $(a_r * b_r - a_i * b_i) + j(a_r * b_i + a_i * b_r)$. This is shown in 6.8b.

6.5.2.1 The Output

Unlike the DWT gain layer, the DT$\mathbb{C}$WT gain layer can achieve aliasing cancelling and therefore has a transfer function. The proof of this is done in **??**.

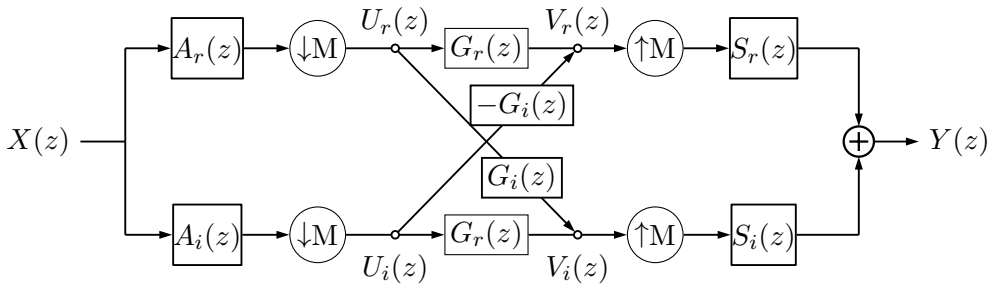


Figure 6.6: **Block Diagram of 1-D** DT$\mathbb{C}$WT **Gain Layer.** Here we show the real and imaginary trees for a single subband. Note that while it may look similar to Figure 6.4, this diagram represents the two trees for one subband rather than a single tree with a pair of subbands. The gain layer does a complex multiply, using both the real and imaginary parts of the decomposed signal. This preserves the shift invariance of the DT$\mathbb{C}$WT for the reconstructed signal Y.

For a single subband in the DT$\mathbb{C}$WT, the gain layer uses a complex learned weight g, but otherwise produces the output v in much the same way as the DWT gain layer. Figure 6.6 shows a single subband DT$\mathbb{C}$WT based gain layer. The output of this layer is:

$$Y(z) = \frac{2}{M} X(z) \left[G_r(z^M) \left(A_r(z) S_r(z) + A_i(z) S_i(z) \right) + G_i(z^M) \left(A_r(z) S_i(z) - A_i(z) S_r(z) \right) \right] \tag{6.5.29}$$

See **??** for the derivation. The G_r term modifies the subband gain $A_r S_r + A_i S_i$ and the G_i term modifies its Hilbert Pair $A_r S_i - A_i S_r$. Figure 6.7 show the contour plots for the frequency support of each of these subbands. The complex gain g can be used to reshape the frequency response for each subband independently.

6.5.2.2 Backpropagation

If H were complex, the first term in Equation 6.5.16 would be $\bar{H}(1/\bar{z})$, but as each individual block in the DT$\mathbb{C}$WT is purely real, we can use the simpler form $H(z^{-1})$.

Again, let us calculate $\Delta V_r(z)$ and $\Delta V_i(z)$ by backpropagating $\Delta Y(z)$ through the inverse DT$\mathbb{C}$WT. Again this is the same as doing the forward DT$\mathbb{C}$WT on $\Delta Y(z)$. Then the weight update equations are:

$$\Delta G_r(z) = \Delta V_r(z) U_r(z^{-1}) + \Delta V_i(z) U_i(z^{-1}) \tag{6.5.30}$$

$$\Delta G_i(z) = -\Delta V_r(z) U_i(z^{-1}) + \Delta V_i(z) U_r(z^{-1}) \tag{6.5.31}$$