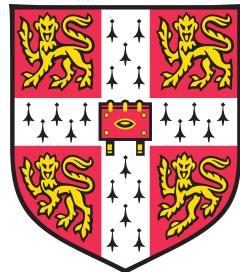


Uses of Complex Wavelets in Deep Convolutional Neural Networks



Fergal Brian Cotter

Supervisor: Prof. Nick Kingsbury
Prof. Joan Lasenby

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
PhD

For Cordelia

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. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Fergal Brian Cotter
August 2019

Acknowledgements

I would like to thank my supervisor Nick Kingsbury, who has dedicated so much of his time to support and help my research. He has not only been instructing and knowledgeable, but very kind and supportive. I also could not have asked for a better and more diligent proof-reader. I will fondly remember the time we spent together brainstorming ideas in his beautiful room in Trinity College.

I would also like to thank my advisor-cum-supervisor Joan Lasenby, who has always been very friendly and supportive, and for ‘adopting’ me when Nick officially retired.

I would like to thank the Division F computing staff, in particular Phill Richardson, Peter Grandi and Raf Czlonka. They spent many hours answering my support emails, installing packages and bringing the servers back online quickly whenever they went down. I know I made many more support requests than the average student, and I appreciate the effort they put in.

I would like to say thank you to my excellent lab friends, I could not have asked for a better group of people to do my research alongside. In particular, thank you: Adam Greig and Rich Wareham for inspiring me to be a better programmer; Jacob Vorstrup and Oliver Bonner for your support, friendship, and lectures on tea; Sam Duffield for his proof-reading, spare bed, and kindness; Hugo Hadfield for inspiring me to import numba; Mahed Abroshan and Ehsan Asadi for motivating me to be a better chess player and problem solver; Parham Bouroumand, Jiaming Liang, Kuan Hseih, James Li, Edmund Bonikowski and the many other excellent people in SigProc. I would particularly like to thank Amarjot Singh and Jos van der Westhuizen. Amarjot for his collaboration and inspiration; and Jos, for his friendship and his infectious ambition.

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

And finally I would like to thank my loving parents Bill and Mary-Rose for their ongoing emotional 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 a scene. Recent advances in supervised learning methods, particularly convolutional neural networks (CNNs), have pushed forth 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, and the networks themselves are often very difficult and time-consuming to train. It is very important that we improve our current approaches in every way possible.

A CNN is built from connecting many learned convolutional layers in series. These convolutional layers are fairly crude in terms of signal processing - they are arbitrary taps of a finite impulse response filter, learned through stochastic gradient descent from random initial conditions. We believe that if we reformulate the problem, we may achieve many insights and benefits in training CNNs. Noting that modern CNNs are mostly viewed from and analyzed in the spatial domain, this thesis aims to view the convolutional layers in the frequency domain (viewing things in the frequency domain has proved useful in the past for denoising, filter design, compression and many other tasks). In particular, we use *complex wavelets* (rather than the Fourier transform or the discrete wavelet transform) as basis functions to reformulate image understanding with deep networks.

In this thesis, we explore the most popular and well-developed form of using complex wavelets in deep learning, the ScatterNet from Stephane Mallat. We explore its current limitations by building a DeScatterNet and found that while it has many nice properties, it may not be sensitive to the most appropriate shapes for understanding natural images.

We then develop a *locally invariant* convolutional layer, a combination of a complex wavelet transform, a modulus operation, and a learned mixing. To do this, we derive backpropagation equations and allow gradients to flow back through the (previously fixed) ScatterNet front end. Connecting several such locally invariant layers allows us to build *learnable ScatterNet*, a more flexible and general form of the ScatterNet (while still maintaining its desired properties).

We show that the learnable ScatterNet can provide significant improvements over the regular ScatterNet when being used as a front end for a learning system. Additionally, we show that the locally invariant convolutional layer can directly replace convolutional layers in a deep CNN (and not just at the front-end). The locally invariant convolutional layers

naturally downsample the input (because of the complex modulus) while increasing the channel dimension (because of the multiple wavelet orientations used). This is an operation that often happens in a CNN by a combination of a pooling and convolutional layer. It was at these locations in a CNN where the learnable ScatterNet performed best, implying it may be useful as learnable pooling layer.

Finally, we develop a system to learn complex weights that act directly on the wavelet coefficients of signals, in place of a convolutional layer. We call this layer the *wavelet gain layer* and show it can be used alongside convolutional layers. The network designer may then choose to learn in the pixel *or* wavelet domains. This layer shows a lot of promise and affords more control over what regions of the frequency space we want our layer to learn from. Our experiments show that it can improve on learning in the pixel domain for early layers of a CNN.

Table of contents

List of figures	xvii
List of tables	xix
1 Introduction	1
1.1 Motivation	3
1.2 Approach	5
1.2.1 Why Complex Wavelets?	6
1.3 Method	7
1.3.1 ScatterNets	7
1.3.2 Learnable ScatterNets	7
1.3.3 Wavelet Domain Filtering	8
1.4 Thesis Layout and Contributions to Knowledge	8
1.4.1 Contributions and Publications	9
1.4.2 Related Research	10
2 Background	11
2.1 Chapter Layout	11
2.2 Supervised Machine Learning	11
2.2.1 Priors on Parameters and Regularization	13
2.2.2 Loss Functions and Minimizing the Objective	14
2.2.3 Stochastic Gradient Descent	15
2.2.4 Gradient Descent and Learning Rate	16
2.2.5 Momentum and Adam	16
2.3 Neural Networks	18
2.3.1 The Neuron and Single-Layer Neural Networks	18
2.3.2 Multilayer Perceptrons	19
2.3.3 Backpropagation	20
2.4 Convolutional Neural Networks	24
2.4.1 Convolutional Layers	24

2.4.2	Pooling	28
2.4.3	Dropout	28
2.4.4	Batch Normalization	29
2.5	Relevant Architectures and Datasets	30
2.5.1	Datasets	30
2.5.2	LeNet	31
2.5.3	AlexNet	31
2.5.4	VGGnet	32
2.5.5	The All Convolutional Network	32
2.5.6	Residual Networks	33
2.6	The Fourier and Wavelet Transforms	33
2.6.1	The Fourier Transform	34
2.6.2	The Continuous Wavelet Transform	35
2.6.3	Discretization and Frames	36
2.6.4	Discrete Wavelet Transform	38
2.6.5	Complex Wavelets	39
2.6.6	Sampled Morlet Wavelets	41
2.6.7	The DT \mathbb{C} WT	44
2.6.8	Summary of Methods	48
2.7	ScatterNets	48
2.7.1	Desirable Properties	48
2.7.2	Definition	50
2.7.3	Resulting Properties	51
3	A Faster ScatterNet	55
3.1	Chapter Layout	55
3.2	Design Constraints	56
3.2.1	Original Design	56
3.2.2	Improvements	57
3.3	A Brief Description of Autograd	57
3.4	Fast Calculation of the DWT and IDWT	57
3.4.1	The Input	58
3.4.2	Preliminary Notes	58
3.4.3	Primitives	58
3.4.4	1-D Filter Banks	60
3.4.5	2-D Transforms and Their Gradients	60
3.5	Fast Calculation of the DT \mathbb{C} WT	61
3.6	The DT \mathbb{C} WT ScatterNet	64
3.6.1	The Magnitude Operation	64

3.6.2	Putting it all Together	65
3.6.3	DTCWT ScatterNet Hyperparameter Choice	66
3.7	Comparisons	67
3.7.1	Speed	67
3.7.2	Performance	67
3.8	Conclusion	70
4	Visualizing and Improving Scattering Networks	73
4.1	Chapter Layout	74
4.2	Related Work	74
4.3	The Scattering Transform	76
4.3.1	Scattering Colour Images	77
4.4	The Inverse Scatter Network	78
4.4.1	Inverting the Low-Pass Filtering	78
4.4.2	Inverting the Magnitude Operation	79
4.4.3	Inverting the Wavelet Decomposition	79
4.4.4	The DTCWT ScatterNet	79
4.5	Visualization with Inverse Scattering	80
4.6	Channel Saliency	82
4.6.1	Experiment Setup	82
4.6.2	Results	83
4.7	Corners, Crosses and Curves	86
4.8	Conclusion	88
5	A Learnable ScatterNet: Locally Invariant Convolutional Layers	89
5.1	Chapter Layout	89
5.2	Related Work	90
5.3	Recap of Useful Terms	90
5.3.1	Convolutional Layers	90
5.3.2	Wavelet Transforms	91
5.3.3	Scattering Transforms	91
5.4	Locally Invariant Layer	92
5.4.1	Properties	93
5.5	Implementation Details	94
5.5.1	Parameter Memory Cost	95
5.5.2	Activation Memory Cost	95
5.5.3	Computational Cost	95
5.5.4	Forward and Backward Algorithm	96
5.6	Layer Introduction with MNIST	97

5.6.1	Proposed Expansions	98
5.6.2	Expanded MNIST experiments	100
5.7	Ablation Experiments with CIFAR and Tiny ImageNet	101
5.8	A New Hybrid ScatterNet	102
5.9	Conclusion	106
6	Learning in the Wavelet Domain	109
6.1	Chapter Layout	110
6.2	Background	110
6.2.1	Related Work	110
6.2.2	Notation	111
6.2.3	DTCWT Notation	112
6.2.4	Learning in Multiple Spaces	114
6.3	The DTCWT Gain Layer	114
6.3.1	The Output	116
6.3.2	Backpropagation	117
6.3.3	Examples	119
6.3.4	Implementation Details	119
6.4	Gain Layer Experiments	122
6.4.1	CNN activation regression	123
6.4.2	Ablation Studies	124
6.4.3	Network Analysis	127
6.5	Wavelet-Based Nonlinearities	129
6.5.1	ReLUs in the Wavelet Domain	131
6.5.2	Thresholding	131
6.6	Gain Layer Nonlinearity Experiments	132
6.6.1	Ablation Experiments with Nonlinearities	134
6.7	Conclusion	135
7	Conclusion and Further Work	137
7.1	Summary of Key Results	137
7.2	Future Work	139
7.2.1	Faster Transforms and More Scales	139
7.2.2	Expanding Tests on Invariant Layers	139
7.2.3	Expanding Tests on Gain Layer	140
7.2.4	ResNets and Lifting	140
7.2.5	Protecting against Attacks	141
7.2.6	Convolutional Sparse Coding	142
7.2.7	Weight Matrix Properties	142

7.3	Final Remarks	143
Appendix A	Architecture Used for Experiments	145
A.1	Run Times of some of the Proposed Layers	145
Appendix B	Extra Proofs and Algorithms	148
B.1	Gradients of Sample Rate Changes	148
B.1.1	Decimation Gradient	148
B.1.2	Interpolation Gradient	149
B.2	Gradient of Wavelet Analysis Decomposition	149
B.3	Extra Algorithms	149
Appendix C	Invertible Transforms and Optimization	152
C.1	Background	152
C.2	Proof	153
Appendix D	DT^CWT Single Subband Gains	155
D.1	Revisiting the Shift-Invariance of the DT ^C WT	155
D.2	Gains in the Subbands	157
Appendix E	Complex CNN Operations	160
E.1	Convolution	160
E.2	Regularization	161
E.3	ReLU Applied to the Real and Imaginary Parts Independently	162
E.4	Soft Shrinkage	162
E.5	Batch Normalization and ReLU Applied to the Complex Magnitude	163
Appendix F	Wavelet Gain Layer Additional Results	165
References		168

List of figures

1.1	Convolutional architecture example	3
1.2	Example first layer filters and the first three layer's outputs	4
2.1	Trajectory of gradient descent in an ellipsoidal parabola	15
2.2	Trajectories of SGD with different initial learning rates	17
2.3	A single neuron	18
2.4	Common neural network nonlinearities and their gradients	20
2.5	Multi-layer perceptron	21
2.6	General block form for autograd	24
2.7	A convolutional layer	26
2.8	Max vs Average 2×2 pooling	29
2.9	LeNet-5 architecture	32
2.10	The residual unit from ResNet	33
2.11	Importance of phase over magnitude for images	34
2.12	Typical wavelets from the 2-D separable DWT	37
2.13	Sensitivity of DWT coefficients to zero crossings and small shifts	40
2.14	Single Morlet filter with varying slants and window sizes	42
2.15	Two Morlet wavelet families and their tiling of the frequency plane	43
2.16	Analysis FBs for the 1-D DT \mathbb{C} WT	45
2.17	The DWT high-high vs the DT \mathbb{C} WT high-high frequency support	47
2.18	Wavelets from the 2-D DT \mathbb{C} WT	47
2.19	A Lipschitz continuous function	49
2.20	The Scattering Transform	52
3.1	2-D DWT filter bank layout	59
3.2	2-D DT \mathbb{C} WT filter bank layout	62
3.3	Hyperparameter results for the DT \mathbb{C} WT ScatterNet on various datasets	68
4.1	Deconvolution network block diagram	75
4.2	The inverse scattering network	78

4.3	Comparison of scattering to convolutional features	81
4.4	Tiny ImageNet changes in accuracy from channel occlusion	84
4.5	CIFAR changes in accuracy from channel occlusion	85
4.6	Channel weights for first learned layer in a hybrid ScatterNet-CNN	86
4.7	Shapes possible by filtering across the wavelet orientations with complex coefficients	87
5.1	Block Diagram of Proposed Invariant Layer for $j = J = 1$	93
5.2	Ablation results for the invariant layer	103
5.3	Ablation results for Tiny ImageNet	104
6.1	Architecture using the DWT as a frontend to a CNN	111
6.2	Proposed new forward pass in the wavelet domain	113
6.3	Diagram of the proposed method to learn in the wavelet domain	115
6.4	Forward and backward filter bank diagrams for DT ^c WT gain layer	116
6.5	DT ^c WT subbands	118
6.6	Example outputs from an impulse input for the proposed gain layers	120
6.7	Normalized MSE for conv layer and wavelet gain layer regression	123
6.8	Large kernel ablation results for CIFAR and Tiny ImageNet	126
6.9	Bandpass gain properties for network with only gain layers	128
6.10	Deconvolution reconstructions for the reference architecture and purely gain layer architecture	130
6.11	CIFAR-100 Ablation results with the gain layer	135
7.1	Residual vs lifting layers	141
7.2	Adversarial examples that can fool AlexNet	142
B.1	Gradient of DWT analysis	150
D.1	Filter bank diagram of 1-D DT ^c WT	156
D.2	Filter bank diagram of 1-D DT ^c WT with subband gains	157
F.1	Small kernel ablation results for CIFAR	166
F.2	Small kernel ablation results for Tiny ImageNet	167

List of tables

2.1	Redundancy of scattering transform	54
3.1	Hyperparameter settings for the DTCWT ScatterNet	66
3.2	Comparison of properties of different ScatterNet packages	69
3.3	Comparison of execution time for the forward and backward passes of the competing ScatterNet implementations	69
3.4	Hybrid architectures for performance comparison	71
3.5	Performance comparison for a DTCWT-based vs. a Morlet-based ScatterNet	71
5.1	Architectures for MNIST hyperparameter experiments	97
5.2	Hyperparameter settings for the MNIST experiments	97
5.3	Architecture performance comparison	98
5.4	Modified architecture performance comparison	100
5.5	CIFAR and Tiny ImageNet base architecture	101
5.6	Hybrid ScatterNet models	105
5.7	Hybrid ScatterNet models with convolutional layer first	106
5.8	Hybrid ScatterNet top-1 classification accuracies on CIFAR	107
5.9	Hybrid ScatterNet top-1 classification accuracies on Tiny ImageNet	107
6.1	Ablation base architecture	125
6.2	Different Nonlinearities in the Gain Layer	134
A.1	Run time speeds for different layers with increasing spatial size	146
A.2	Run time speeds for different layers with increasing channel size	147

Chapter 1

Introduction

It has long been the goal of computer vision researchers to be able to develop systems that can reliably recognize objects in a scene. Achieving this unlocks a huge range of applications that can benefit society as a whole: fully autonomous vehicles, automatic labelling of uploaded videos/images for searching, interpretation and screening of security video feeds, and many more, all far-reaching and extremely valuable. Many of these tasks are very tedious for humans and would be done much better by machines if the missed-detection rate can be kept low enough. The challenge does not lie in finding the right application, but in the difficulty of training a computer to see.

Some of the difficulties associated with vision are the presence of nuisance variables such as changes in lighting condition, changes in viewpoint, and background clutter. These variables do not affect the scene but can drastically change the pixel representation of it. Humans, even at early stages of their lives, have little difficulty filtering out these nuisance variables and are excellent at extracting the necessary information from a scene. To design a robust system, it makes sense to take account of how our brains see and understand scenes.

Unfortunately, biological vision is also a complex system. It has more to it than simply collecting photons in the eye. An excerpt from a recent Neurology paper [1] sums up the problem well:

It might surprise some to learn that visual information is significantly degraded as it passes from the eye to the visual cortex. Thus, of the unlimited information available from the environment, only about 10^{10} bits/sec are deposited in the retina . . . only $\sim 6 \times 10^6$ bits/sec leave the retina and only 11^4 bits/sec make it to layer IV of V1 [2], [3]. These data clearly leave the impression that visual cortex receives an impoverished representation of the world . . . it should be noted that estimates of the bandwidth of conscious awareness itself (i.e., what we ‘see’) are in the range of 100 bits/sec or less [2], [3].

Current video cameras somewhat act as a combination of the first and second stage of this system, collecting photons in photosensitive sensors and then converting this to a stream of images. Standard definition digital television typically has a bit rate between 3×10^6 and 10^7 bits/sec (slightly larger but comparable to the 10^6 bits/sec travelling through the optic nerve).

If we are to build effective vision systems, it makes sense to emulate this compression of information between the optic nerve and the later stages of the visual cortex. Hubel and Wiesel revolutionized our understanding of the (primary visual) V1 cortex in their Nobel prize-winning work (awarded in 1981 in Physiology/Medicine) by studying cats [4], [5], macaques and spider monkeys [6]. They found that neurons in the V1 cortex fired most strongly when edges of a particular (i.e., neuron-dependent) orientation were presented to the animal, so long as the edge was inside the receptive field of this neuron. Continuing on this work, Blakemore and Cooper [7] analysed the perception of kittens that had restricted visual information presented to them. In one of their experiments, the kittens were kept in darkness and then exposed for a few hours a day to only horizontal or vertical lines. After five months, they were taken into natural environments and their reactions were monitored. The two groups of cats would only play with objects when presented in an orientation that matched the orientation of their original environment. This suggests that these early layers of perception are *learned*.

The current state of the art in image understanding systems are Convolutional Neural Networks (CNNs). These are a learned model that cascades many convolutional filters serially in layers, separated by nonlinearities. They are seemingly inspired by the visual cortex in the way that they are hierarchically connected, progressively compressing the information into a richer representation.

Figure 1.1 shows an example of a CNN architecture, AlexNet [8]. Inputs are resized to a manageable size, in this case, 224×224 pixels. Multiple convolutional filters of size 11×11 are convolved over this input to give 96 output *channels* (or *activation maps*). In the figure, these are split onto two graphics cards or GPUs for memory purposes. These are then passed through a pointwise nonlinear function, or *nonlinearity*. The activations are pooled (a form of downsampling) and convolved with more filters to give 256 new channels at the second stage. This is repeated 3 more times until the 13×13 output with 256 channels is unravelled and passed through a fully connected neural network to classify the image as one of 1000 possible classes.

CNNs have garnered lots of attention since 2012 when AlexNet nearly halved the top-5 classification error rate (from 26% to 16%) in the ImageNet Large Scale Visual Recognition Competition (ILSVRC) [9]¹. In the years since the complexity of CNNs has grown significantly. AlexNet had only 5 convolutional layers, whereas the 2015 ILSVRC winner ResNet [15]

¹The previous state of the art classifiers had been built by combining keypoint extractors like SIFT[10] and HOG[11] with classifiers such as Support Vector Machines[12] and Fisher Vectors[13], for example [14].

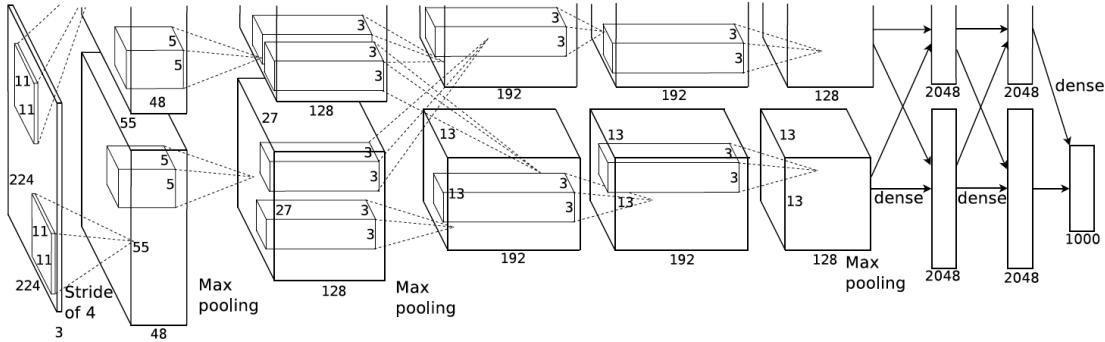


Figure 1.1: **Convolutional architecture example.** The previous layer’s activations are combined with a learned convolutional filter. Note that while the activation maps are 3-D arrays, the convolution is only a 2-D operation. This means the filters have the same number of channels as the input and produce only one output channel. Multiple channels are made by convolving with multiple filters. Not shown here are the nonlinearities that happen in between convolution operations. Image is taken from [8].

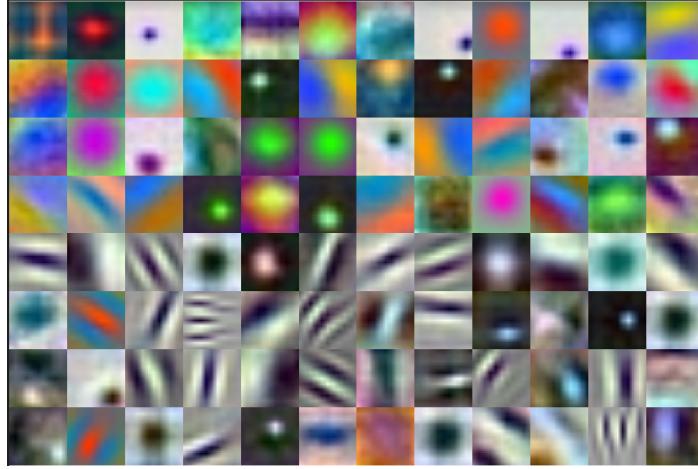
achieved 3.57% top-5 error with 151 convolutional layers (and had some experiments with 1000 layer networks).

1.1 Motivation

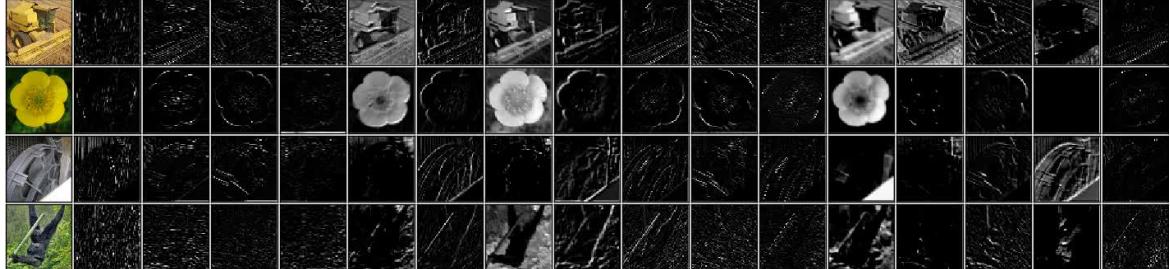
Despite their success, CNNs are often criticized for being *black-box* methods. You can view the first layer of filters quite easily (see Figure 1.2a) as they exist in RGB space, but beyond that things get trickier as the filters have a third, *channel* dimension, typically much larger than the two spatial dimensions. Additionally, it is not clear what the input channels themselves correspond to. For illustration purposes, we have also shown some example activations from the first three convolutional layers for AlexNet in Figure 1.2(b)-(d)². For the output from the first convolutional layer (conv1) in Figure 1.2b, we can accurately guess that some of the filters are responding to edges or colour information, but as we go deeper to the second (conv2) and third (conv3), it becomes less and less clear what each activation is responding to.

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. In 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,

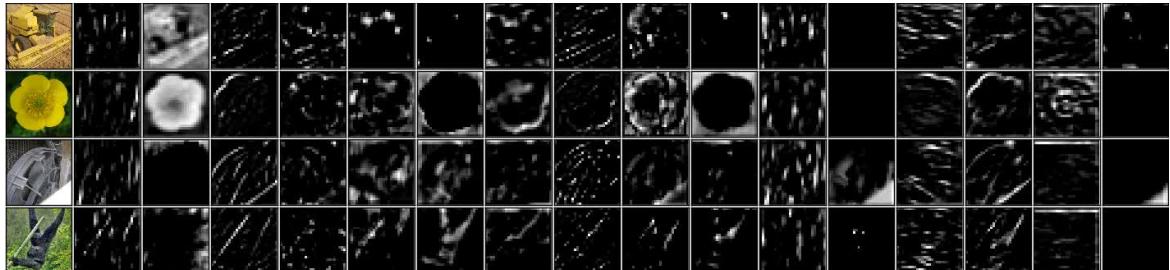
²These activations are taken after a specific nonlinearity that sets negative values to 0, hence the large black regions.



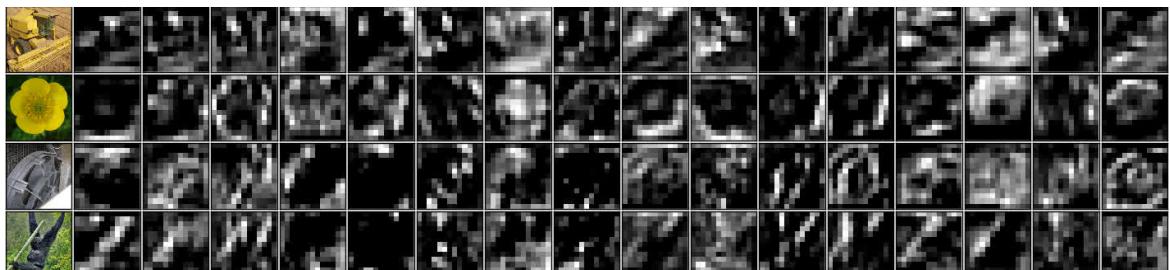
(a) conv1 filters



(b) conv1 activations



(c) conv2 activations



(d) conv3 activations

Figure 1.2: Example first layer filters and the first three layer's outputs. (a) The 11×11 filters for the first stage of AlexNet. Of the 96 filters, 48 were learned on one GPU and another 48 on another GPU. Interestingly, one GPU has learned mostly lowpass/colour filters and the other has learned oriented bandpass filters. (b) - (d) Randomly chosen activations from the output of the first, second and third convolutional layers of AlexNet (see Figure 1.1) with negative values set to 0. Filters and activation images are taken from the supplementary material of [8].

has the right to know why [16], 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.

Aside from their lack of interpretability, it often takes a long time and a lot of effort to train state-of-the-art CNNs. Typical networks that have won ILSVRC since 2012 have had roughly 100 million parameters and take up to a week to train. This is optimistic and assumes that you already know the necessary optimization or architecture hyperparameters, which you often have to find out by trial and error. In a conversation we had with Yann LeCun, the attributed father of CNNs, at a Computer Vision Summer School (ICVSS 2016), LeCun highlighted this problem himself:

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

Considering the recent success of CNNs, it is becoming more and more important to understand *how* and *what* a network learns, so we can interrogate what in the input has contributed to it making its classification or regression choice. Without this information, the use of these incredibly powerful tools could be restricted to research and proprietary applications.

1.2 Approach

The structure of convolutional layers is fairly crude in terms of signal processing - arbitrary taps of an FIR filter are learned typically via stochastic gradient descent from random starting states to minimize either a mean-squared error or cross-entropy loss.

This leads us to ask a motivating question:

Is it possible to learn convolutional filters as combinations of basis functions rather than individual filter taps?

In achieving this, it is important to find ways to have an adequate richness of filtering while reducing the number of parameters needed to specify resulting filters. We want to contract the space of learning to a subspace or manifold that is more useful. In much the same way, the convolutional layer in a CNN is a restricted version of a fully connected layer in a multi-layer perceptron, yet adding this restriction allowed us to train more powerful networks.

The intuition that we explore in this thesis is that *complex wavelets* are good basis functions for filtering in CNNs.

1.2.1 Why Complex Wavelets?

Most modern approaches to CNNs are framed entirely in the spatial domain; our choice of complex wavelets as the basis function to explore comes from the deeper intuition that it may be helpful to rethink about CNNs in the *frequency domain*. Historically, the frequency domain has been an excellent space for solving many signal processing problems such as noise removal, filter design, edge detection and data compression. We believe it may prove to have advantages for CNNs too (beyond just an efficient space to do convolution in).

The Fourier transform, which uses complex sinusoids as its basis function, is perhaps the most ubiquitous tool to use for frequency domain analysis. The problem with these complex sinusoids is that they have *infinite* support. This means that small changes in one part of an image affect every Fourier coefficient. Additionally, they are not stable to small deformations, as small changes can produce unbounded changes in the representation [17].

The common remedy to this problem is to use the localized, and more stable, short-time Fourier transform (STFT). The STFT (or the Gabor transform) is a natural extension of the Fourier transform, windowing the complex sinusoids with a Gaussian (or similar) function. The STFT has the undesirable property that all frequencies are sampled with the same resolution. A close relative of the STFT is the continuous wavelet transform (CWT). The shorter duration of the wavelet basis functions as the frequency increases means that their time resolving power improves with centre-frequency. Another commonly used wavelet transform is the discrete wavelet transform (or the DWT) often favoured over the CWT because of its speed of computation. It can use many different finite support basis functions, all with different frequency localization properties, but it is usually limited to using real filters. As such, it suffers from many problems such as shift-dependence and lack of directionality in two dimensions (2-D). These problems can be remedied by using the slower CWT with complex basis functions, but we choose instead to use the dual-tree complex wavelet transform, or DT \mathbb{C} WT [18] with q-shift filters [19].

The DT \mathbb{C} WT allows for complex basis functions that have shift-invariance and directionality, while being fast to implement like the DWT (in 2-D it can be thought of as the application of 4 DWTs in parallel). It is also more easily invertible than the CWT, forming a tight frame [20], which we believe may prove to be a very important property for visualizing what a CNN is responding to.

We revisit the properties of the Fourier transform, STFT, CWT, DWT and DT \mathbb{C} WT and expand on the properties behind our choice of basis functions in the literature review section 2.6.

On top of the intuition that the wavelet domain is a good space in which to frame CNNs, there are some experimental motivating factors too. Firstly, the wavelet transform has had much success in image and video compression, particularly for JPEG2000 [21]. Good compression performance implies an ability of the basis functions to represent the input

data sparsely (as seems to happen in the brain). Secondly, the filters from the first layer of AlexNet ([Figure 1.2](#)) look like oriented wavelets. Given that there was no prior placed on the filters to make them have this similarity to wavelets, this result is noteworthy. And finally, the aforementioned work of Hubel and Wiesel suggests that the early layers of the visual system act like a Gabor transform.

These experimental observations imply that complex wavelets would do well in replacing the first layer of a CNN, but we would also like to find out if they can be used at deeper layers. Their well-understood and well-defined behaviour would help us to answer the above *how* and *why* questions. Additionally, they allow us to enforce a certain amount of smoothness and near orthogonality; smoothness seems to be important to avoid sensitivity to adversarial or spoofing attacks [\[22\]](#) and near orthogonality allows you to cover a large space with fewer coefficients.

But first, we must find out *if* it is possible to get the same or nearly the same performance by using wavelets as the building blocks for CNNs, and this is the core goal of this thesis.

1.3 Method

1.3.1 ScatterNets

To explore the uses of complex wavelets in CNNs, we begin by looking at one of the most popular current uses of wavelets in image recognition tasks, the Scattering Transform.

The Scattering Transform, or the *ScatterNet*, was introduced in [\[17\]](#), [\[23\]](#) at the same time as AlexNet. It is a non-black-box network that can be thought of as a restricted complex-valued CNN [\[24\]](#). Unlike a CNN, it has predefined convolutional kernels, set to complex wavelet (and scaling) functions and uses the complex magnitude as its nonlinearity. Due to its well-defined structure, it can be analyzed and bounds on its stability to shifts, noise and deformations are found in [\[17\]](#).

For a simple task like identifying small handwritten digits, the variabilities in the data are simple and small and the ScatterNet can reduce the problem into a space which a Gaussian Support Vector Machine (or SVM [\[12\]](#)) can easily solve [\[23\]](#). For a more complex task like identifying real-world objects, the ScatterNet can somewhat reduce the variabilities and get good results with an SVM, but there is a significant performance gap between this and what a CNN can achieve. For example, in [\[25\]](#) a second-order ScatterNet can achieve 82.3% top-1 classification accuracy on CIFAR-10, a commonly used dataset, whereas modern CNNs such as [\[15\]](#) can achieve 93.4%.

1.3.2 Learnable ScatterNets

To start to address the performance gap between ScatterNet front ends and CNNs we first investigate the properties of current ScatterNets. Inspired by the visualization work of

Zeiler and Fergus [26] we build a DeScatterNet. The DeScatterNet leverages the perfect reconstruction properties of the DTCWT and allows us to investigate what in the input image the ScatterNet is responding to.

Interpreting the visualizations from the DeScatterNet leads us to the conclusion that the ScatterNet may be limiting itself by not combining the filtering of different wavelet orientations (it does not mix the channels as a CNN does). Inspired by the work of [27], we propose the learnable ScatterNet, which includes this mixing, while keeping the desirable properties of the ScatterNet (invariance to translation, additive noise, and deformations; see subsection 2.7.1 for a description of these properties).

The learnable ScatterNet can be thought of as using the scattering outputs as the *basis functions*³ for our convolutional layers. We show that this improves greatly on the ScatterNet design, and under certain constraints can improve on the performance of CNNs too.

1.3.3 Wavelet Domain Filtering

We find that the complex modulus of the ScatterNet design to be useful for some operations in a CNN, but it has a demodulating effect on the frequency energy (all the outputs have significantly more energy in lower frequencies). This limits repeated application of it as the demodulating effect compounds.

We develop a system that does not use the complex modulus; instead, it learns *complex* gains in the wavelet domain. Rather than mixing subbands together, we keep them independent and only learn to mix across the channel dimension. This is important, as it allows us to then use the inverse DTCWT to return to the pixel domain. The shift-invariant properties of the DTCWT mean the reconstructed outputs are (mostly) free from aliasing effects, despite much of the processing being carried out at significantly reduced sample rates in the wavelet domain.

We show that our layer can be used alongside regular convolutional layers. I.e., it becomes possible to ‘step’ into the wavelet domain to do wavelet filtering for one layer, before ‘stepping’ back into the pixel domain to do pixel filtering for the next layer.

1.4 Thesis Layout and Contributions to Knowledge

This thesis has one literature review chapter and four novel-work chapters:

- Chapter 2 explores some of the background necessary for starting to develop image understanding models. In particular, it covers the inspiration for CNNs and the workings of CNNs themselves, as well as covering the basics of wavelets and ScatterNets.

³Although they are not true basis functions as they are the combination of a complex wavelet with a modulus nonlinearity, and are thus data-dependent.

- Chapter 3 proposes a change to the core of the ScatterNet. In addition to performance issues with ScatterNets, they are slow and both memory-intensive and compute-intensive to calculate. This in itself is enough of an issue to make it unlikely that they would be used as part of deep networks. To overcome this, we change the computation to use the DTcWT [18] instead of Morlet wavelets, achieving a 20 to 30 times speed-up while achieving a small improvement in classification performance.
- Chapter 4 describes our *DeScatterNet*, a tool used to interrogate the structure of ScatterNets. We also perform tests to determine the usefulness of the different scattered outputs finding that many of them are not useful for image classification.
- Chapter 5 describes the *Learnable ScatterNet* we have developed to address some of the issues found from the interrogation in chapter 4. We find that a learnable ScatterNet layer performs better than a regular ScatterNet, and can improve on the performance of a CNN if used instead of pooling layers. We also find that scattering works well not just on RGB images, but can also be useful when used after one layer of learning.
- In chapter 6, we step away from ScatterNets and present the *Wavelet Gain Layer*. The gain layer uses the wavelet space as a latent space to learn representations. We find possible nonlinearities and describe how to learn in both the pixel and wavelet domain. This work showed that there may well be benefits to learning in the wavelet domain for earlier layers of CNNs, but we have not yet found advantages to using the wavelet gain layer for deeper layers.

1.4.1 Contributions and Publications

The key contributions of this thesis are:

- Software for wavelets and DTcWT based ScatterNet (described in chapter 3) and publicly available at [28].
- ScatterNet analysis and visualizations (described in chapter 4). This chapter expands on the paper we presented at MLSP2017 [29].
- Invariant Layer/Learnable ScatterNet (described in chapter 5)). This chapter expands on the paper accepted at ICIP2019 [30]. Software available at [31].
- Learning convolutions in the wavelet domain (described in chapter 6). We have published preliminary results on this work to arXiv [32] but have expanded on this paper in the chapter. Software available at [33].

1.4.2 Related Research

Readers may also be interested in the theses [34] and [35].

In [34] Singh looks at using the ScatterNet as a fixed front end and combining it with well-known machine learning methods such as SVMs, Autoencoders and Restricted Boltzmann Machines. By combining frameworks in a defined way he creates unsupervised feature extractors which can then be used with simple classifiers. Some relevant papers that make up this thesis are [36]–[38]. In [36] Singh shows that the DTCWT-ScatterNet outperforms a Morlet-ScatterNet when used as a front end for an SVM, which is similar to the work we do in chapter 3 where we show the DTCTWT-ScatterNet outperforms a Morlet-ScatterNet when used as a front end for CNNs. He then expands on this work by testing other backends in [37], [38].

In [35] Oyallon looks at ScatterNets as front ends to deeper learning systems, such as CNNs. Some relevant papers that make up Oyallon’s thesis are [25], [39]. [39] is particularly relevant as he uses a ScatterNet as a feature extractor for a CNN. We do similar research in chapter 5 but allow for learned weights in the ScatterNet in our design.

Chapter 2

Background

2.1 Chapter Layout

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 basics of deep learning, looking at general supervised learning in [section 2.2](#) before more specifically examining the structure of Neural Networks in [section 2.3](#) and CNNs in [section 2.4](#).

We then define the properties of Wavelet Transforms, looking at the difference between the discrete WT and the complex Morlet and DTCWT in [section 2.6](#). Finally, we introduce the Scattering Transform or ScatterNet, the original inspiration for this thesis in [section 2.7](#).

2.2 Supervised Machine Learning

While this subject is general and covered in many places, we take inspiration from [40] (chapters 1, 2, 7, 8) and [41] (chapter 5-10). Consider a sample space over inputs and targets $\mathcal{X} \times \mathcal{Y}$ and a data generating distribution p_{data} . Given a dataset of input-target pairs $\mathcal{D} = \{(x^{(n)}, y^{(n)})\}_{n=1}^N$ we would like to make predictions about $p_{data}(y|x)$ that generalize well to unseen data. A common way to do this is to build a parametric model to directly estimate this conditional probability. For example, regression asserts the data are distributed according to a function of the inputs plus a noise term ϵ :

$$y = f(x, \theta) + \epsilon \tag{2.2.1}$$

This noise is often modelled as a zero-mean Gaussian random variable, $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$, which means we can write:

$$p_{model}(y|x, \theta, \sigma^2) = \mathcal{N}(y; f(x, \theta), \sigma^2 I) \tag{2.2.2}$$

where (θ, σ^2) are the parameters of the model.

We can find point estimates of the parameters by maximizing the likelihood of $p_{model}(y|x, \theta)$ (or equivalently, minimizing $KL(p_{model}||p_{data})$, the KL-divergence between p_{model} and p_{data}). As the data are all assumed to be i.i.d., we can multiply individual likelihoods, and solve for θ :

$$\theta_{MLE} = \arg \max_{\theta} p_{model}(y|x, \theta) \quad (2.2.3)$$

$$= \arg \max_{\theta} \prod_{n=1}^N p_{model}(y^{(n)}|x^{(n)}, \theta) \quad (2.2.4)$$

$$= \arg \max_{\theta} \sum_{n=1}^N \log p_{model}(y^{(n)}|x^{(n)}, \theta) \quad (2.2.5)$$

Using the Gaussian regression model from above, this becomes:

$$\theta_{MLE} = \arg \max_{\theta} \sum_{n=1}^N \log p_{model}(y^{(n)}|x^{(n)}, \theta) \quad (2.2.6)$$

$$= \arg \max_{\theta} \left(-N \log \sigma - \frac{N}{2} \log(2\pi) - \sum_{n=1}^N \frac{(y^{(n)} - f(x^{(n)}, \theta))^2}{2\sigma^2} \right) \quad (2.2.7)$$

$$= \arg \min_{\theta} \frac{1}{N} \sum_{n=1}^N \frac{(y^{(n)} - f(x^{(n)}, \theta))^2}{2} \quad (2.2.8)$$

which gives us the well-known result that we would like to find parameters that minimize the mean squared error (MSE) between targets y and predictions $\hat{y} = f(x, \theta)$.

For binary classification $y \in \{0, 1\}$ and instead of the model in (2.2.2), we have:

$$p_{model}(y|x, \theta) = \text{Ber}(y; \sigma(f(x, \theta))) \quad (2.2.9)$$

where $\sigma(x)$ is the sigmoid function and Ber is the Bernoulli distribution. Note that we have used σ to refer to noise standard deviation thus far but now use $\sigma(x)$ to refer to the sigmoid and softmax functions, a confusing but common practice. $\sigma(x)$ and $\text{Ber}(y; p)$ are defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (2.2.10)$$

$$\text{Ber}(y; p) = p^{\mathbb{I}(y=1)}(1-p)^{\mathbb{I}(y=0)} \quad (2.2.11)$$

where $\mathbb{I}(x)$ is the indicator function. The sigmoid function is useful here as it can convert a real output $f(x, \theta)$ into a probability estimate. In particular, large positive values get mapped to 1, large negative values to 0, and values near 0 get mapped to 0.5 [41, Chapter 6].

This expands naturally to multi-class classification by making y a 1-hot vector in $\{0, 1\}^C$. We must also swap the Bernoulli distribution for the Multinoulli or Categorical distribution,

and the sigmoid function for a softmax. The softmax function for vector \mathbf{z} is defined as:

$$\sigma_i(\mathbf{z}) = \frac{e^{z_i}}{\sum_{k=1}^C e^{z_k}} \quad (2.2.12)$$

which has the nice properties that $0 \leq \sigma_i \leq 1$ and $\sum_i \sigma_i = 1$. The categorical distribution uses the softmax output:

$$\text{Cat}(y; \boldsymbol{\sigma}) = \prod_{c=1}^C \sigma_c^{\mathbb{I}(y_c=1)} \quad (2.2.13)$$

If we let $\hat{y}_c = \sigma_c(f(x, \theta))$, this makes (2.2.9):

$$p_{model}(y|x, \theta) = \text{Cat}(y; \sigma(f(x, \theta))) \quad (2.2.14)$$

$$= \prod_{c=1}^C \prod_{n=1}^N \left(\hat{y}_c^{(n)} \right)^{\mathbb{I}(y_c^{(n)}=1)} \quad (2.2.15)$$

As $y_c^{(n)}$ is either 0 or 1, we remove the indicator function. Maximizing this likelihood to find the ML estimate for θ :

$$\theta_{MLE} = \arg \max_{\theta} \prod_{c=1}^C \prod_{n=1}^N \left(\hat{y}_c^{(n)} \right)^{y_c^{(n)}} \quad (2.2.16)$$

$$= \arg \max_{\theta} \frac{1}{N} \sum_{n=1}^N \sum_{c=1}^C y_c^{(n)} \log \hat{y}_c^{(n)} \quad (2.2.17)$$

which we recognize as the cross-entropy between y and \hat{y} .

2.2.1 Priors on Parameters and Regularization

Maximum likelihood estimates for parameters, while straightforward, can often lead to overfitting. A common practice is to regularize learnt parameters θ by putting a prior over them. If we do not have any prior information about what we expect them to be, it may still be useful to put an uninformative prior over them. For example, if our parameters are in the reals, a commonly used uninformative prior is a Gaussian.

Let us extend the regression example from above by saying we would like the prior on the parameters θ to be a Gaussian, i.e. $p(\theta) = \mathcal{N}(0, \tau^2 I_D)$. The corresponding maximum a

posteriori (MAP) estimate is then obtained by finding:

$$\theta_{MAP} = \arg \max_{\theta} p(\theta | \mathcal{D}, \sigma^2) \quad (2.2.18)$$

$$= \arg \max_{\theta} \frac{p(y|x, \theta, \sigma^2)p(\theta)}{p(y|x)} \quad (2.2.19)$$

$$= \arg \max_{\theta} \log p(y|x, \theta, \sigma^2) + \log p(\theta) \quad (2.2.20)$$

$$= \arg \min_{\theta} \frac{1}{N} \sum_{n=1}^N \frac{(y^{(n)} - f(x^{(n)}, \theta))^2}{2} + \frac{\lambda}{2} \|\theta\|_2^2 \quad (2.2.21)$$

where $\lambda = \sigma^2/\tau^2$ is the ratio of the observation noise to the strength of the prior [40, Chapter 7]. This is equivalent to minimizing the MSE with an ℓ_2 penalty on the parameters, also known as ridge regression or penalized least squares. λ is often called *weight decay* in the neural network literature, which we will also use in this thesis.

2.2.2 Loss Functions and Minimizing the Objective

It may be useful to rewrite (2.2.18) as an objective function on the parameters $J(\theta)$:

$$J(\theta) = \frac{1}{N} \sum_{n=1}^N \frac{(y^{(n)} - f(x^{(n)}, \theta))^2}{2} + \frac{\lambda}{2} \|\theta\|_2^2 \quad (2.2.22)$$

$$= L_{data}(y, f(x, \theta)) + L_{reg}(\theta) \quad (2.2.23)$$

where L_{data} is the data loss such as MSE or cross-entropy and L_{reg} is the regularization, such as ℓ_2 or ℓ_1 penalized loss. Now $\theta_{MAP} = \arg \min_{\theta} J(\theta)$.

Finding the global minimum of the objective function is task-dependent and is often not straightforward. One commonly used technique is called *gradient descent* (GD). This is not difficult to do as it only involves calculating the gradient at a given point and taking a small step in the direction of steepest descent. The update equation for GD is:

$$\theta_{t+1} = \theta_t - \eta \frac{\partial J}{\partial \theta} \quad (2.2.24)$$

Unsurprisingly, such a simple technique has limitations. In particular, it is sensitive to the choice of step size and has a slow convergence rate when the condition number (ratio of largest to smallest eigenvalues) of the Hessian around the optimal point is large [42]. An example of this is shown in Figure 2.1. In this figure, the step size is chosen with exact line search, i.e.

$$\eta = \arg \min_s f \left(x + s \frac{\partial f}{\partial x} \right) \quad (2.2.25)$$

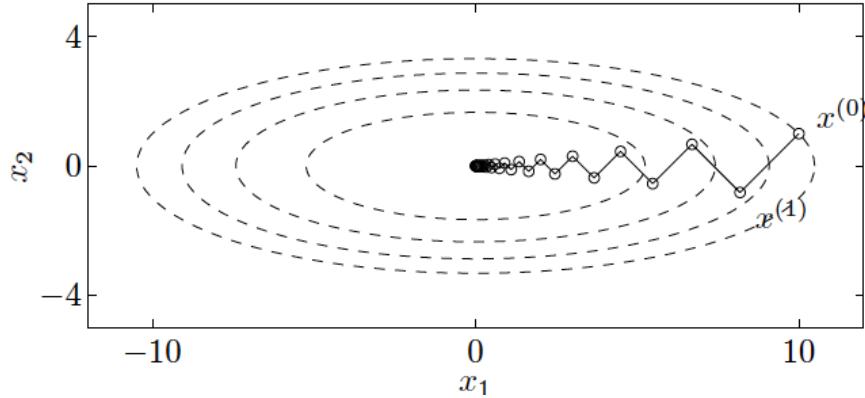


Figure 2.1: **Trajectory of gradient descent in an ellipsoidal parabola.** Some contour lines of the function $f(x) = 1/2(x_1^2 + 10x_2^2)$ and the trajectory of GD optimization using exact line search. This space has condition number 10, and shows the slow convergence of GD in spaces with largely different eigenvalues. Image is taken from [42] Figure 9.2.

To truly overcome this problem, we must know the curvature of the objective function $\frac{\partial^2 J}{\partial \theta^2}$. An example optimization technique that uses the second-order information is Newton's method [42, Chapter 9]. Such techniques sadly do not scale with size, as computing the Hessian is proportional to the number of parameters squared, and many neural networks have hundreds of thousands, if not millions of parameters. In this thesis, we only consider *first-order* optimization algorithms.

2.2.3 Stochastic Gradient Descent

Aside from the problems associated with the curvature of the function $J(\theta)$, another common issue faced with the gradient descent of (2.2.24) is the cost of computing $\frac{\partial J}{\partial \theta}$. In particular, the first term:

$$L_{data}(y, f(x, \theta)) = \mathbb{E}_{x, y \sim p_{data}} [L_{data}(y, f(x, \theta))] \quad (2.2.26)$$

$$= \frac{1}{N} \sum_{n=1}^N L_{data}\left(y^{(n)}, f(x^{(n)}, \theta)\right) \quad (2.2.27)$$

involves evaluating the entire dataset at the current values of θ . As the training set size grows into the thousands or millions of examples, this approach becomes prohibitively slow.

Equation (2.2.26) writes the data loss as an expectation, hinting at the fact that we can remedy this problem by using fewer samples $N_b < N$ to evaluate L_{data} . This variation is called Stochastic Gradient Descent (SGD).

Choosing the batch size is a hyperparameter choice that we must think carefully about. Setting the value very low, e.g. $N_b = 1$ can be advantageous as the noisy estimates for the gradient have a regularizing effect on the network [43]. Increasing the batch size to larger

values allows you to easily parallelize computation as well as increasing accuracy for the gradient, allowing you to take larger step sizes [44]. A good initial starting point is to set the batch size to 128 samples and increase/decrease from there [41].

2.2.4 Gradient Descent and Learning Rate

The step size parameter, η in (2.2.24) is commonly referred to as the learning rate. Choosing the right value for the learning rate is key. Unfortunately, the line search algorithm in (2.2.25) would be too expensive to compute for neural networks (as it would involve evaluating the function several times at different values), each of which takes about as long as calculating the gradients themselves. Additionally, as the gradients are typically estimated over a mini-batch and are hence noisy there may be little added benefit in optimizing the step sizes in the estimated direction.

Figure 2.2 illustrates the effect the learning rate can have over a contrived convex example. Optimizing over more complex loss surfaces only exacerbates the problem. Sadly, choosing the initial learning rate is ‘more of an art than a science’ [41], but [45], [46] have some tips on what to how to set it. We have found in our work that searching for a large learning rate that causes the network to diverge and reducing it from there can be a good search strategy. This agrees with Section 1.5 of [47] which states that for regions of the loss space which are roughly quadratic, $\eta_{max} = 2\eta_{opt}$ and any learning rate above $2\eta_{opt}$ causes divergence.

On top of the initial learning rate, the convergence of SGD methods require [45]:

$$\sum_{t=1}^{\infty} \eta_t \rightarrow \infty \quad (2.2.28)$$

$$\sum_{t=1}^{\infty} \eta_t^2 < \infty \quad (2.2.29)$$

Choosing how to do this also contains a good amount of artistry, and there is no one scheme that works best. A commonly used greedy method is to keep the learning rate constant until the training loss stabilizes and then to enter the next phase of training by setting $\eta_{k+1} = \gamma\eta_k$ where γ is a decay factor. Setting γ and the thresholds for triggering a step however must be chosen by monitoring the training loss curve and trial and error [45].

2.2.5 Momentum and Adam

One simple and very popular modification to SGD is to add *momentum*. Momentum accumulates past gradients with an exponential-decay moving average and continues to move in their direction. The name comes from the comparison of finding minima to rolling a ball over a surface – any new force (newly computed gradients) must overcome the current momentum of the ball. This has a smoothing effect on noisy gradients.

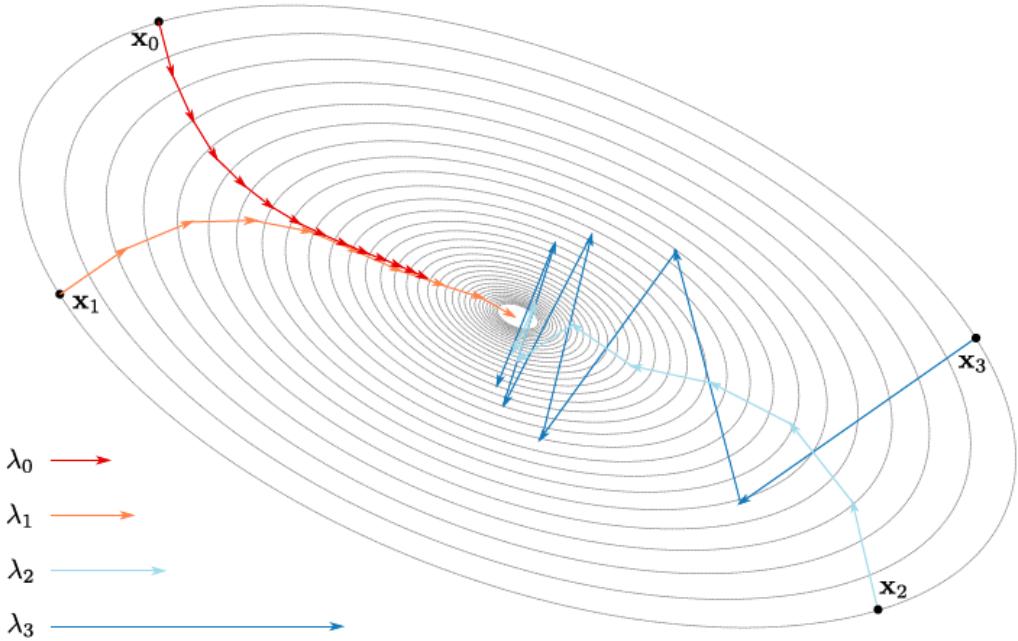


Figure 2.2: **Trajectories of SGD with different initial learning rates.** This figure illustrates the effect the step size has over the optimization process by showing the trajectory for $\eta = \lambda_i$ from equivalent starting points on a symmetric loss surface. Increasing the step size beyond λ_3 can cause the optimization procedure to diverge. Image taken from [48] Figure 2.7.

We can achieve momentum by creating a *velocity* variable v_t and modify (2.2.24) to be:

$$v_{t+1} = \alpha v_t - \eta_k \frac{\partial J}{\partial \theta} \quad (2.2.30)$$

$$\theta_{t+1} = \theta_t + v_{t+1} \quad (2.2.31)$$

where $0 \leq \alpha < 1$ is the momentum term indicating how quickly to ‘forget’ past gradients.

Another popular modification to SGD is the adaptive learning rate technique Adam [49]. There are several other adaptive schemes such as AdaGrad [50] and AdaDelta [51], but they are all quite similar, and Adam is often considered the most robust of the three [41]. The goal of all of these adaptive schemes is to take larger update steps in directions of low variance, helping to minimize the effect of large condition numbers we saw in Figure 2.1. Adam does this by keeping track of the first m_t and second v_t moments of the gradients:

$$g_{t+1} = \frac{\partial J}{\partial \theta} \quad (2.2.32)$$

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) g_{t+1} \quad (2.2.33)$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) g_{t+1}^2 \quad (2.2.34)$$

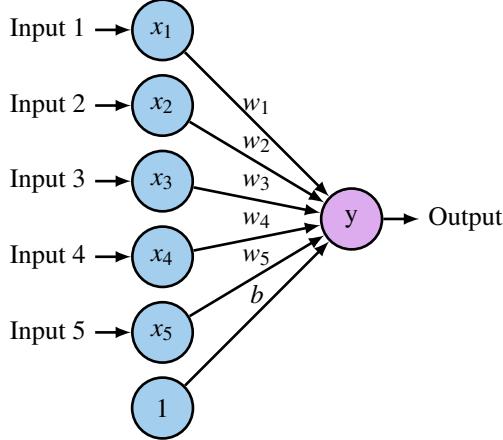


Figure 2.3: **A single neuron.** The neuron is composed of inputs x_i , weights w_i (and a bias term), as well as an activation function. Typical activation functions include the sigmoid function, tanh function and the ReLU

where $0 \leq \beta_1, \beta_2 < 1$. Note the similarity between updating the mean estimate in (2.2.33) and the velocity term in (2.2.30)¹. The parameters are then updated with:

$$\theta_{t+1} = \theta_t - \eta \frac{m_{t+1}}{\sqrt{v_{t+1}} + \epsilon} \quad (2.2.35)$$

where ϵ is a small value to avoid dividing by zero.

2.3 Neural Networks

2.3.1 The Neuron and Single-Layer Neural Networks

The neuron, shown in Figure 2.3 is the core building block of neural networks. It takes the dot product between an input vector $\mathbf{x} \in \mathbb{R}^D$ and a weight vector \mathbf{w} , before applying a chosen nonlinearity. Historically, the sigmoid nonlinearity was the most popular but today other functions have become more popular. Still, the convention has remained to name this generic nonlinearity σ . I.e.

$$y = \sigma(\langle \mathbf{x}, \mathbf{w} \rangle) = \sigma \left(\sum_{i=0}^D x_i w_i \right) \quad (2.3.1)$$

where we have used the shorthand $b = w_0$ and $x_0 = 1$. Also, note that we will use the common practice in the neural network literature to call the parameters *weights* denoted by w .

¹The m_{t+1} and v_{t+1} terms are then bias-corrected as they are biased towards zero at the beginning of training. We do not include this for conciseness.

Some of the other popular nonlinear functions σ are the tanh and ReLU:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (2.3.2)$$

$$\text{ReLU}(x) = \max(x, 0) \quad (2.3.3)$$

See [Figure 2.4](#) for plots of these. The original Rosenblatt perceptron [52] also used the Heaviside function $H(x) = \mathbb{I}(x > 0)$.

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 non-unit-norm \mathbf{w} , this can be thought of as a scaled distance). Thus, the weight vector \mathbf{w} defines a hyperplane in \mathbb{R}^D 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.

Nearly all modern neural networks use the ReLU nonlinearity and it has been credited with being a key reason for the recent surge in deep learning success [53], [54]. In particular:

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 nonlinearities 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 [55]. The ReLU, on the other hand, has a constant gradient when it is activated.
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. [56] 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.

2.3.2 Multilayer Perceptrons

As mentioned in the previous section, a single neuron can be thought of as a separating hyperplane with an activation that maps the two halves of the space to different values. Such a linear separator is limited and famously cannot solve the XOR problem [57]. Fortunately, adding a single hidden layer like the one shown in [Figure 2.5](#) can change this, and it is proveable that with an infinitely wide hidden layer, a neural network can approximate any function [58], [59]. This extension is called a multilayer perceptron, or MLP.

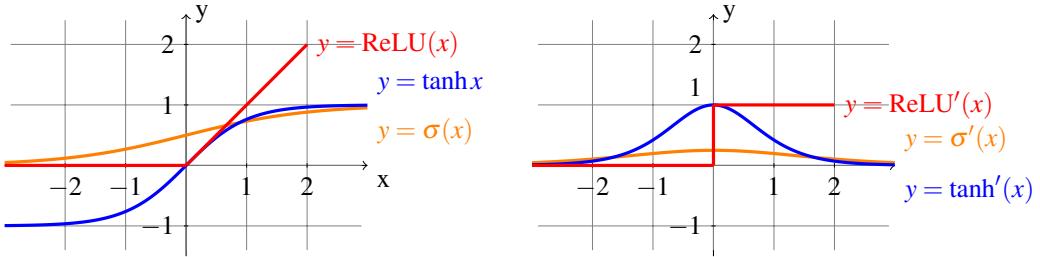


Figure 2.4: Common neural network nonlinearities and their gradients. The sigmoid, tanh and ReLU nonlinearities are commonly used activation functions for neurons. The tanh and sigmoid have the nice property of being smooth but can have saturation when the input is either largely positive or largely negative, causing little gradient to flow back through it. The ReLU does not suffer from this problem and has the additional nice property of setting values to exactly 0, making a sparser output activation.

The forward pass of such a network with one hidden layer of H units is:

$$h_i = \sigma \left(\sum_{j=0}^D x_j w_{ij}^{(1)} \right) \quad (2.3.4)$$

$$y = \sum_{k=0}^H h_k w_k^{(2)} \quad (2.3.5)$$

where $w^{(l)}$ denotes the weights for the l -th layer, of which Figure 2.5 has 2. Note that these individual layers are often called *fully connected* as each node in the previous layer affects every node in the next.

If we were to expand this network to have N_l such fully connected layers, we could rewrite the action of each layer in a recursive fashion:

$$y^{(l+1)} = W^{(l+1)} x^{(l)} \quad (2.3.6)$$

$$x^{(l+1)} = \sigma \left(y^{(l+1)} \right) \quad (2.3.7)$$

where W is now a weight matrix, acting on the vector of the previous layer's outputs $x^{(l)}$. As we are now considering every layer to be an input to the next stage, we have removed the h notation and added the superscript (l) to define the depth. $x^{(0)}$ is the network input and $y^{(N_l)}$ is the network output. Let us say that the output has C nodes, and a hidden layer $x^{(l)}$ has C_l nodes.

2.3.3 Backpropagation

It is important to truly understand backpropagation when designing neural networks, so we describe the core concepts now for a neural network with N_l layers.

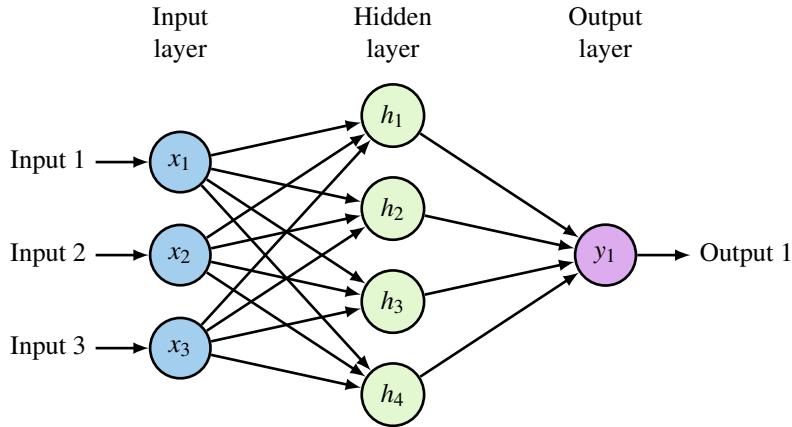


Figure 2.5: **Multi-layer perceptron.** Expanding the single neuron from Figure 2.3 to a network of neurons. The internal representation units are often referred to as the *hidden layer* as they are an intermediary between the input and output.

The delta rule, initially designed for networks with no hidden layers [60], was expanded to what we now consider *backpropagation* in [61]. While backpropagation is conceptually just the application of the chain rule, Rumelhart, Hinton, and Williams successfully updated the delta rule to networks with hidden layers, laying a key foundational step for deeper networks.

With a deep network, calculating $\frac{\partial L_{data}}{\partial w}$ may not seem easy, particularly if w is a weight in one of the earlier layers. We need to define a rule for updating the weights in all N_l layers of the network, $W^{(1)}, W^{(2)}, \dots, W^{(N_l)}$ however, only the final set $W^{(N_l)}$ are connected to the data loss function L_{data} .

2.3.3.1 Regression Loss

Let us start with writing down the derivative of L (dropping the *data* subscript for now) with respect to the network output $\hat{y} = y^{(N_l)}$ using the regression objective function (2.2.8).

$$\frac{\partial L}{\partial y^{(N_l)}} = \frac{\partial}{\partial \hat{y}} \left(\frac{1}{N} \sum_{n=1}^{N_b} \frac{1}{2} (y^{(n)} - \hat{y}^{(n)})^2 \right) \quad (2.3.8)$$

$$= \frac{1}{N} \sum_{n=1}^{N_b} (\hat{y}^{(n)} - y^{(n)}) \quad (2.3.9)$$

$$= e \in \mathbb{R} \quad (2.3.10)$$

where we have used the fact that for the regression case, $y^{(n)}, \hat{y}^{(n)} \in \mathbb{R}$.

2.3.3.2 Classification Loss

For the classification case (2.2.17), let us keep the output of the network as $y^{(N_l)} \in \mathbb{R}^C$ and define \hat{y} as the softmax applied to this vector $\hat{y}_c^{(n)} = \sigma_c(y^{(N_l, n)})$.

Note that the softmax is a vector-valued function going from $\mathbb{R}^C \rightarrow \mathbb{R}^C$ so has a Jacobian matrix S with values:

$$S_{ij} = \frac{\partial \hat{y}_i}{\partial y_j^{(N_l)}} = \begin{cases} \sigma_i(1 - \sigma_j) & \text{if } i = j \\ -\sigma_i \sigma_j & \text{if } i \neq j \end{cases} \quad (2.3.11)$$

Now, let us return to (2.2.17) and find the derivative of the objective function to this output value \hat{y} :

$$\frac{\partial L}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}} \left(\frac{1}{N} \sum_{n=1}^{N_b} \sum_{c=1}^C y_c^{(n)} \log \hat{y}_c^{(n)} \right) \quad (2.3.12)$$

$$= \frac{1}{N} \sum_{n=1}^{N_b} \sum_{c=1}^C \frac{y_c^{(n)}}{\hat{y}_c^{(n)}} \quad (2.3.13)$$

$$= d \in \mathbb{R}^C \quad (2.3.14)$$

Note that unlike (2.3.10), this derivative is vector-valued. To find $\frac{\partial L}{\partial y_j^{(N_l)}}$ we use the chain rule. It is easier to find the partial derivative with respect to one node in the output first, and then expand from here, i.e.:

$$\frac{\partial L}{\partial y_j^{(N_l)}} = \sum_{i=1}^C \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial y_j^{(N_l)}} \quad (2.3.15)$$

$$= S_j^T d \quad (2.3.16)$$

where S_j is the j th column of the Jacobian matrix S . It becomes clear now that to get the entire vector derivative for all nodes in $y^{(N_l)}$, we must multiply the transpose of the Jacobian matrix with the error term from (2.3.14):

$$\frac{\partial L}{\partial y^{(N_l)}} = S^T d \quad (2.3.17)$$

2.3.3.3 Final Layer Weight Gradient

Let the final weight layer be called $W \in \mathbb{R}^{C \times C_{N_l-1}}$ (where C_{N_l-1} is the number of outputs at the penultimate layer). We call the gradient for the final layer weights the *update* gradient. It can be computed by the chain rule again. For an individual entry in this matrix W_{ij} , the update gradient is:

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial L_{data}}{\partial y_i} \frac{\partial y_i}{\partial W_{ij}} + \lambda W_{ij} \quad (2.3.18)$$

$$= \frac{\partial L_{data}}{\partial y_i} x_j + \lambda W_{ij} \quad (2.3.19)$$

where the second term in the above two equations comes from the regularization loss that is added to the objective. The update gradient of the entire weight matrix is then:

$$\frac{\partial J}{\partial W^{(N_l)}} = \frac{\partial L_{data}}{\partial \hat{y}} x^T + 2\lambda W \quad (2.3.20)$$

$$= S^T d \left(x^{(N_l-1)} \right)^T + 2\lambda W^{(N_l)} \in \mathbb{R}^{C \times C_{N_l-1}} \quad (2.3.21)$$

2.3.3.4 Final Layer Passthrough Gradient

We also want to find the *passthrough* gradients of the final layer $\frac{\partial L}{\partial x^{(N_l-1)}}$ (these are not affected by the regularization terms, so we only need to find the gradient w.r.t. the data loss L). In a similar fashion, we first find the gradient with respect to individual elements in $x^{(N_l-1)}$ before generalizing to the entire vector:

$$\frac{\partial L}{\partial x_i} = \sum_{j=1}^C \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_i} \quad (2.3.22)$$

$$= \sum_{j=1}^C \frac{\partial L}{\partial y_j} W_{j,i} \quad (2.3.23)$$

$$= W_i^T \frac{\partial L}{\partial y} \quad (2.3.24)$$

$$(2.3.25)$$

where W_i is the i th column of W . Thus

$$\frac{\partial L}{\partial x^{(N_l-1)}} = \left(W^{(N_l)} \right)^T \frac{\partial L}{\partial y^{(N_l)}} \quad (2.3.26)$$

$$= \left(W^{(N_l)} \right)^T S^T d \quad (2.3.27)$$

This passthrough gradient then can be used to update the next layer's weights by repeating [subsubsection 2.3.3.3](#) and [subsubsection 2.3.3.4](#).

2.3.3.5 General Layer Update

The easiest way to handle this flow of gradients and the basis for most automatic differentiation packages is the block definition shown in [Figure 2.6](#). For all neural network components (even if they do not have weights), the operation must not only be able to calculate the forward pass $y = f(x, w)$ given weights w and inputs x , but also calculate the *update* and *passthrough* gradients $\frac{\partial L}{\partial w}, \frac{\partial L}{\partial x}$ given an input gradient $\frac{\partial L}{\partial y}$. The input gradient will have the same shape as y as will the update and passthrough gradients match the shape of w and x . This way, gradients for the entire network can be computed in an iterative fashion starting at the loss function and moving backwards.

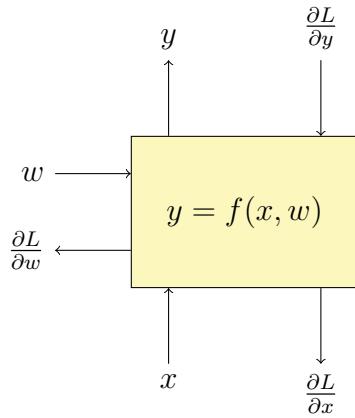


Figure 2.6: **General block form for autograd.** All neural network functions need to be able to calculate the forward pass $y = f(x, w)$ as well as the update and passthrough gradients $\frac{\partial L}{\partial w}, \frac{\partial L}{\partial x}$. Backpropagation is then easily done by allowing data to flow backwards through these blocks from the loss.

2.4 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a special type of Neural Network built mainly from *convolutional layers* rather than fully connected layers. A convolutional layer is one where the weights are shared spatially across the layer. In this way, a neuron at i is only affected by nodes from the previous layer in a given neighbourhood, rather than from every node.

First popularized in 1998 by LeCun et. al. in [62], the convolutional layer was introduced to build invariance with respect to translations, as well as reduce the parameter size of early neural networks for pattern recognition. The idea of having a locally-receptive field had already been shown to be a naturally occurring phenomenon by Hubel and Wiesel [5]. They did not become popular immediately, and another spatially based keypoint extractor, SIFT [10], was the mainstay of detection systems until the AlexNet CNN [8] won the 2012 ImageNet challenge [63] by a large margin over the next competitors, many of whom used SIFT. This CNN had 5 convolutional layers followed by 3 fully connected layers.

We now briefly describe the convolutional layer, as well as many other layers used in CNNs that have become popular in the past few years.

2.4.1 Convolutional Layers

In the analysis of neural networks so far, we have considered column vectors $x^{(l)}, y^{(l)} \in \mathbb{R}^{C_l}$. Convolutional layers for image analysis have a different format, with the spatial component of the input is preserved.

Let us first consider the definition of 2-D convolution for single-channel images:

$$y[\mathbf{n}] = (x * h)[\mathbf{n}] = \sum_{\mathbf{k}} x[\mathbf{k}]h[\mathbf{n} - \mathbf{k}] \quad (2.4.1)$$

$$= \sum_{k_1, k_2} x[k_1, k_2]h[n_1 - k_1, n_2 - k_2] \quad (2.4.2)$$

where the sum is done over the support of h . For an input $x \in \mathbb{R}^{H \times W}$ and filter $h \in \mathbb{R}^{K_H \times K_W}$ the output has spatial support $y \in \mathbb{R}^{H+K_H-1 \times W+K_W-1}$.

The filter h can also be thought of as a *matched filter* that gives the largest normalized output when the input contains the mirror of h , \tilde{h} . If the input has shapes similar to \tilde{h} in many locations, each of these locations in y will also have large outputs.

If we stack red, green and blue input channels on top of each other², we have a 3-dimensional input $x \in \mathbb{R}^{C \times H \times W}$ with $C = 3$. This third dimension is often called the *depth* dimension, to distinguish it from the two spatial dimensions. In a CNN layer, each filter h is three dimensional with depth exactly equal to C . The convolution is done only over the two spatial dimensions and the C outputs are summed at each pixel location. This makes (2.4.1):

$$y[\mathbf{n}] = \sum_{c=0}^{C-1} \sum_{\mathbf{k}} x[c, \mathbf{k}]h[c, \mathbf{n} - \mathbf{k}] \quad (2.4.3)$$

It is not enough to only have a single matched filter and often we would like to have a bank of them, each one sensitive to a different shape. For example, if one filter is sensitive to horizontal edges, we may also want to detect vertical, and diagonal edges. Let us rename the number of channels in the input layer as C_l and specify that we would like to have C_{l+1} different matched filters. We then stack each of the single-channel outputs from (2.4.3) to give the output $y \in \mathbb{R}^{C_{l+1} \times H \times W}$:

$$y[f, \mathbf{n}] = \sum_{c=0}^{C-1} \sum_{\mathbf{k}} x[c, \mathbf{k}]h[f, c, \mathbf{n} - \mathbf{k}] \quad (2.4.4)$$

After a convolutional layer, we can then apply a pointwise nonlinearity to each output location in y . Like multilayer perceptrons, this was typically the sigmoid function σ , but is now more commonly the ReLU. Revisiting (2.3.6) and (2.3.7), we can rewrite this for a

²In deep learning literature, there is not a consensus about whether to stack the outputs with the channel first ($\mathbb{R}^{C \times H \times W}$) or last ($\mathbb{R}^{H \times W \times C}$). The latter is more common in Image Processing for colour and spectral images but the former is the standard for many deep learning frameworks, including the one we use – PyTorch [64]. For this reason, we stack channels along the first dimension of our tensors.

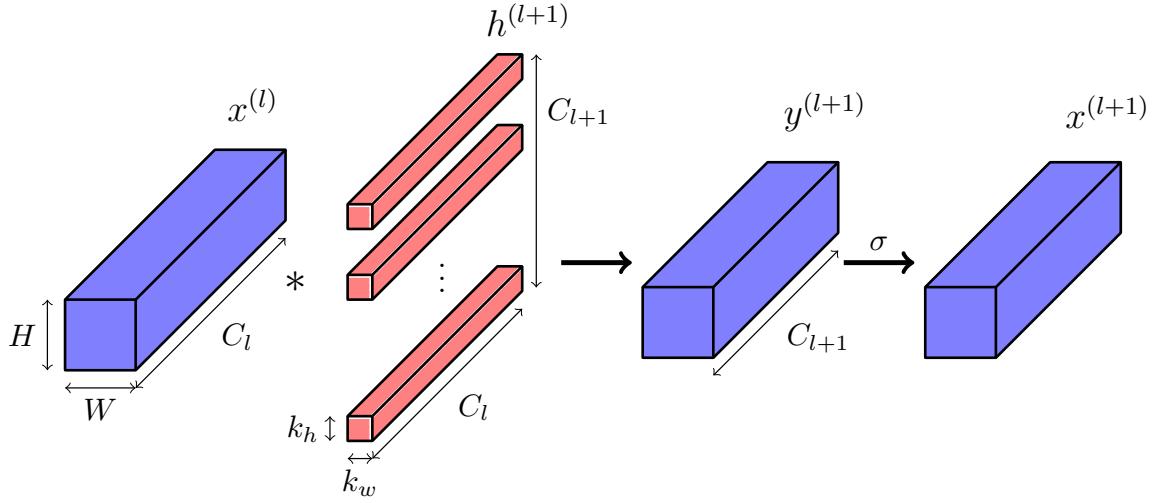


Figure 2.7: **A convolutional layer.** A convolutional layer followed by a nonlinearity σ . The previous layer’s activations are convolved with a bank of C_{l+1} filters, each of which has spatial size $k_h \times k_w$ and depth C_l . Note that there is no convolution across the channel dimension. Each filter produces one output channel in $y^{(l+1)}$.

convolutional layer at depth l with C_l input and C_{l+1} output channels:

$$y^{(l+1)}[f, \mathbf{n}] = \sum_{c=0}^{C_l-1} x^{(l)}[c, \mathbf{n}] * h^{(l)}[f, c, \mathbf{n}] \quad (2.4.5)$$

$$x^{(l+1)}[f, \mathbf{n}] = \sigma(y^{(l)}[f, \mathbf{n}]) \quad (2.4.6)$$

where $f \in \{0, 1, \dots, C_{l+1}-1\}$ indexes the filter number/output channel. A diagram representing this operation is shown in Figure 2.7.

2.4.1.1 Padding and Stride

Regular 2-D convolution expands the input from size $H \times W$ to $(H + K_H - 1) \times (W + K_W - 1)$. In neural networks, this is called *full convolution*. It is often desirable (and common) to have the same output size as input size, which can be achieved by taking the central $H \times W$ outputs of full convolution. This is often called *same-size convolution*. Another option commonly used is to only evaluate the kernels where they fully overlap the input signal, causing a reduction in the output size to $(H - K_H + 1) \times (W - K_W + 1)$. This is called *valid convolution* and was used in the original LeNet-5 [62].

Signal extension for full and same-size convolution is by default *zero padding*, and most deep learning frameworks have no ability to choose other padding schemes as part of their convolution functions. Other padding such as *symmetric padding* can be achieved by expanding the input signal before doing a valid convolution.

Stride is a commonly used term in deep learning literature. A stride of 2 means that we evaluate the filter kernel at every other input location. In signal processing, this is simply called decimation by 2.

2.4.1.2 Gradients

To get the update and passthrough gradients for the convolutional layer we will need to expand (2.4.5) (again we will drop the layer superscripts for clarity):

$$y[f, n_1, n_2] = \sum_{c=0}^{C-1} \sum_{k_1} \sum_{k_2} x[c, k_1, k_2] h[f, c, n_1 - k_1, n_2 - k_2] \quad (2.4.7)$$

The derivative for a single output $y[f, n_1, n_2]$ w.r.t. a single input $x[c, k_1, k_2]$ is then simply:

$$\frac{\partial y_{f,n_1,n_2}}{\partial x_{c,k_1,k_2}} = h[f, c, n_1 - k_1, n_2 - k_2] \quad (2.4.8)$$

It is clear from (2.4.7) that a single activation $x[c, k_1, k_2]$ affects many output values. Thus, the derivative for the loss function to this single point in x is the sum of the chain rule applied to all output positions:

$$\frac{\partial L}{\partial x_{c,k_1,k_2}} = \sum_f \sum_{n_1} \sum_{n_2} \frac{\partial L}{\partial y_{f,n_1,n_2}} \frac{\partial y_{f,n_1,n_2}}{\partial x_{c,k_1,k_2}} \quad (2.4.9)$$

$$= \sum_f \sum_{n_1} \sum_{n_2} \frac{\partial L}{\partial y_{f,n_1,n_2}} h[f, c, n_1 - k_1, n_2 - k_2] \quad (2.4.10)$$

Now we let $\Delta y[f, n_1, n_2] = \frac{\partial L}{\partial y_{f,n_1,n_2}}$ be the passthrough gradient signal from the next layer, and $\tilde{h}[\alpha, \beta, \gamma, \delta] = h[\beta, \alpha, -\gamma, -\delta]$ be a set of filters that have been mirror-imaged in the spatial domain and had their filter and channel dimensions transposed. Combining these two and substituting into (2.4.10), we get the passthrough gradient for the convolutional layer:

$$\frac{\partial L}{\partial x_{c,k_1,k_2}} = \sum_f \sum_{n_1} \sum_{n_2} \Delta y[f, n_1, n_2] \tilde{h}[c, f, k_1 - n_1, k_2 - n_2] \quad (2.4.11)$$

$$= \sum_f \Delta y[f, \mathbf{k}] * \tilde{h}[c, f, \mathbf{k}] \quad (2.4.12)$$

which is the same as (2.4.5). I.e. we can backpropagate the gradients through a convolutional layer by mirror-imaging the filters spatially, transposing them in the channel and filter dimensions, and doing a forward convolutional layer with \tilde{h} applied to Δy . Similarly, we find

the update gradients to be:

$$\frac{\partial L}{\partial h_{f,c,k_1,k_2}} = \sum_{n_1} \sum_{n_2} \frac{\partial L}{\partial y_{f,n_1,n_2}} \frac{\partial y_{f,n_1,n_2}}{\partial h_{f,c,k_1,k_2}} \quad (2.4.13)$$

$$= \sum_{n_1} \sum_{n_2} \Delta y[f, n_1, n_2] x[c, n_1 - k_1, n_2 - k_2] \quad (2.4.14)$$

$$= (\Delta y[f] \star x[c]) [k_1, k_2] \quad (2.4.15)$$

where \star is the cross-correlation operation.

2.4.2 Pooling

Pooling layers are common in CNNs where we want to build spatial invariance (and consequently reduce spatial size). As we go deeper into a CNN, it is common for the spatial size of the activation to decrease, and the channel dimension to increase. The C_l values at a given spatial location can then be thought of as a feature vector describing the presence of shapes in a given area in the input image.

Pooling is useful to add some invariance to smaller shifts when downsampling. It is often done over small spatial sizes, such as 2×2 or 3×3 . Invariance to larger shifts can be built up with multiple pooling (and convolutional) layers.

Two of the most common pooling techniques are *max pooling* and *average pooling*. Max pooling takes the largest value in its spatial area, whereas average pooling takes the mean. A visual explanation is shown in Figure 2.8. Note that pooling is typically a spatial operation, and only in rare cases is done over the channel/depth dimension.

A review of pooling methods in [65] found both max and average pooling to perform similarly well. While max pooling was the most popular in earlier state of the art networks [8], [66], there has been a recent trend towards using average pooling [67] or even to do away with pooling altogether in favour of strided convolutions (this idea was originally proposed in [68] and used notably in [15], [69], [70]).

2.4.3 Dropout

Dropout is a particularly strong regularization scheme that randomly turns off, or ‘zeros out’, neurons in a neural network [71], [72]. Each neuron has probability p of having its value set to 0 (independently of other neurons) during training time, forcing the network to be more general and preventing ‘co-adaption’ of neurons [72].

During test time, dropout is typically turned off, but can still be used to get an estimate on the uncertainty of the network by averaging over several runs [73].

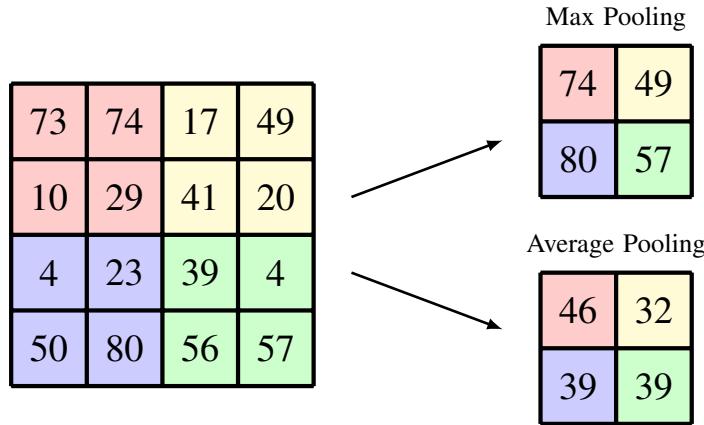


Figure 2.8: Max vs Average 2x2 pooling.

2.4.4 Batch Normalization

Batch normalization proposed in [74] is a conceptually simple technique which rescales activations of a neural network. Despite its simplicity, it has become very popular and has been found to be very useful to train deeper CNNs.

First let us define $\mu_c^{(l)}$ and $\sigma_c^{(l)}$ as the mean and standard deviations for a channel in a given activation at layer l . This mean and standard deviation is taken by averaging across the entire dataset (with N samples), and at every spatial location.

$$\mu_c^{(l)} = \frac{1}{N} \sum_{i=0}^{N-1} \sum_{\mathbf{n}} x^{(l,i)}[c, \mathbf{n}] \quad (2.4.16)$$

$$(\sigma_c^{(l)})^2 = \frac{1}{N} \sum_{i=0}^{N-1} \sum_{\mathbf{n}} \left(x^{(l,i)}[c, \mathbf{n}] \right)^2 - (\mu_c^{(l)})^2 \quad (2.4.17)$$

where $\mu, \sigma \in \mathbb{R}^C$. Batch normalization removes the mean and variance of the data, scales the data by a learnable gain γ and shifts the data to a learnable mean β , with $\gamma, \beta \in \mathbb{R}^C$. If we drop the layer superscripts, this means the action of batch normalization is defined by:

$$y[c, \mathbf{n}] = \frac{x[c, \mathbf{n}] - \mu_c}{\sigma_c + \epsilon} \gamma_c + \beta_c \quad (2.4.18)$$

where ϵ is a small value to avoid dividing by 0.

Of course, during training, we do not have access to the dataset statistics μ, σ so these values are estimated from the batch statistics. A typical practice is to keep an exponential moving average estimate of these values.

The passthrough and update gradients are:

$$\frac{\partial L}{\partial x_{c,n_1,n_2}} = \frac{\partial L}{\partial y_{c,n_1,n_2}} \frac{\gamma}{\sigma + \epsilon} \quad (2.4.19)$$

$$\frac{\partial L}{\partial \beta_c} = \sum_n \frac{\partial L}{\partial y_{c,n}} \quad (2.4.20)$$

$$\frac{\partial L}{\partial \gamma_c} = \sum_n \frac{\partial L}{\partial y_{c,n}} \frac{x_{c,n} - \mu_c}{\gamma_c + \epsilon} \quad (2.4.21)$$

Batch normalization layers are typically placed *between* convolutional layers and nonlinearities.

Consider a linear operation such as convolution with weights W acting on the previous layer's output X , defined by $Y = WX$. Batch normalization removes the sensitivity of our network to initial scaling of the weights, as $BN(aWX) = BN(WX)$. It is also particularly useful for backpropagation as scaling the weights by a constant a does not change the passthrough gradients and leads to *smaller* update gradients [74], making the network more resilient to the problems of vanishing and exploding gradients:

$$\begin{aligned} \frac{\partial BN((aW)X)}{\partial X} &= \frac{\partial BN(WX)}{\partial X} \\ \frac{\partial BN((aW)X)}{\partial (aW)} &= \frac{1}{a} \cdot \frac{\partial BN(WX)}{\partial W} \end{aligned} \quad (2.4.22)$$

2.5 Relevant Architectures and Datasets

In this section, we briefly review some relevant CNN architectures that will be helpful to refer back to in this thesis.

2.5.1 Datasets

When doing image analysis tasks it is important to know comparatively how well different networks perform on the same challenge. To achieve this, the community has developed several datasets that are commonly used to report metrics. For image classification we have chosen five such datasets, listed here in increasing order of difficulty:

1. **MNIST**: 10 classes, 6000 images per class, 28×28 pixels per image. The images contain the digits 0–9 in greyscale on a blank background. The digits have been size normalized and centred. Dataset description and files can be obtained at [75].
2. **CIFAR-10**: 10 classes, 5000 images per class, 32×32 pixels per image. The images contain classes of everyday objects like cars, dogs, planes etc. The images are colour and have little clutter or background. Dataset description can be found in [76] and files at [77].

3. **CIFAR-100:** 100 classes, 500 images per class, 32×32 pixels per image. Similar to CIFAR-10, but now with fewer images per class and ten times as many classes. Dataset description can be found in [76] and files at [77].
4. **Tiny ImageNet:** 200 classes, 500 images per class, 64×64 pixels per image. A more recently introduced dataset that bridges the gap between CIFAR and ImageNet. Images are larger than CIFAR and there are more categories. Dataset description and files can be obtained at [78].
5. **ImageNet CLS:** There are multiple types of challenges in ImageNet, but CLS is the classification challenge and is most commonly reported in papers. It has 1000 classes of objects with a varying amount of images per class. Most classes have 1300 examples in the training set, but a few have less than 1000. The images have variable size, typically a couple of hundred pixels wide and a couple of hundred pixels high. The images can have varying amounts of clutter and can be at different scales, making it a particularly difficult challenge. Dataset description is in [63] and the most reliable source of the data can be found at [79].

Several other classification datasets do exist but are not commonly used, such as PASCAL VOC [80] and Caltech-101 and Caltech-256 [81]³.

2.5.2 LeNet

LeNet-5 [62] is a good network to start with: it is simple yet contains many of the layers used in modern CNNs. Shown in Figure 2.9 it has two convolutional and three fully connected layers. The outputs of the convolutional layers are passed through a sigmoid nonlinearity and downsampled with average pooling. The first two fully-connected layers also have sigmoid nonlinearities. The loss function used is a combination of tanh functions and MSE loss.

2.5.3 AlexNet

We have already seen AlexNet [8] in chapter 1. It is arguably one of the most important architectures in the development in CNNs, as it experimentally showed that CNNs can be used for complex tasks. This required a few innovations: multiple GPUs to do fast processing on large images, the ReLU to avoid saturation, and also dropout to aid generalization. It took the original authors a week to train AlexNet on 2 GPUs.

The first layer uses convolutions with spatial support of 11×11 , followed by 5×5 and 3×3 for the final three layers.

³Tiny ImageNet is also not commonly used as it is quite new. We have included it in the main list as we have found it to be quite a useful step up from CIFAR without requiring the weeks to train experimental configurations on ImageNet.

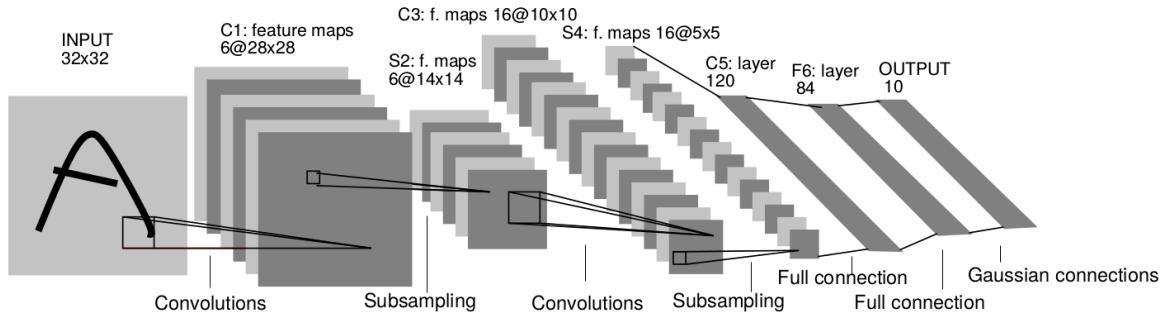


Figure 2.9: **LeNet-5 architecture.** The ‘original’ CNN architecture used for handwriting recognition. LeNet has 2 convolutional and 3 fully connected layers making 5 parameterized layers. After the second convolutional layer, the $16 \times 5 \times 5$ pixel output is unravelled to a 400 long vector. Image is taken from [62].

2.5.4 VGGnet

The Visual Geometry Group (VGG) at Oxford came second in the ILSVRC challenge in 2014 with their VGG-nets [66]. Despite this, the VGG-net remains an important network for some of the design choices it inspired. The optimal VGG-net was much deeper than AlexNet, with 19 convolutional layers on top of each other before 3 fully connected layers. These convolutional layers all used the smaller 3×3 seen only at the back of AlexNet.

This network is particularly attractive due to its simplicity, compared to the more complex Inception Network [82] which won the 2014 ILSVRC challenge. VGG-16, the 16 layer variant of VGG stacks two or three convolutional layers (and ReLUs) on top of each other before reducing spatial size with max pooling. After processing at five scales, the resulting $512 \times 14 \times 14$ activation is unravelled and passed through a fully connected layer.

These VGG networks also marked the start of a trend that has since become common, where channel depth is doubled after pooling layers. The doubling of channels and quartering the spatial size still causes a net reduction in the number of activations.

2.5.5 The All Convolutional Network

The All Convolutional Network [68] introduced two popular modifications to the VGG networks:

- They argued for the removal of max pooling layers, saying that a 3×3 convolutional layer with stride 2 works just as well.
- They removed the fully connected layers at the end of the network, replacing them with 1×1 convolutions. Note that a 1×1 convolution still has shared weights across all spatial locations. The output layer then has size $C_L \times H \times W$, where H, W are many times smaller than the input image size, and the vector of C_L coefficients at each

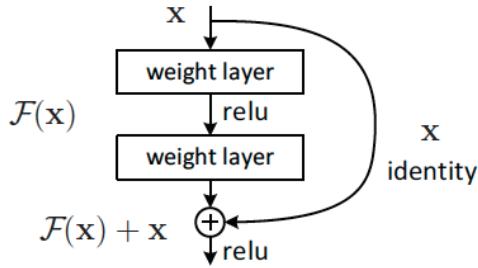


Figure 2.10: **The residual unit from ResNet.** A residual unit. The identity mapping is always present, and the network learns the difference from the identity mapping, $\mathcal{F}(x)$. Taken from [15].

spatial location can be interpreted as a vector of scores marking the presence/absence of C_L different shapes. For classification, the output can be averaged over all spatial locations, whereas for localization it may be useful to keep this spatial information.

The new network was able to achieve state of the art results on CIFAR-10 and CIFAR-100 and competitive performance on ImageNet, while only use a fraction of the parameters of other networks.

2.5.6 Residual Networks

Residual Networks or ResNets won the 2015 ILSVRC challenge, introducing the residual layer. Most state of the art models today use this residual mapping in some way [69], [70].

The inspiration for the residual layer 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 2.10. If a desired mapping is denoted $\mathcal{H}(x)$, instead of trying to learn this, they instead learn $\mathcal{F}(x) = \mathcal{H}(x) - x$. Doing this promotes a strong diagonal in the Jacobian matrix which improves conditioning for gradient descent.

Recent analyses of a ResNet without nonlinearities [83], [84] proves that SGD fails to converge for deep networks when the network mapping is far away from the identity, suggesting that a residual mapping is a good thing to do.

2.6 The Fourier and Wavelet Transforms

Many tasks in computer vision tasks are often very difficult. Pixel intensities in an image are typically not very informative in understanding what is in that image. These values are sensitive to lighting conditions and camera configurations. It would be easy to take two



Figure 2.11: 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.

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 defining an image is difficult to define, however, some things are notably more important than others. For example, the location or phase of the waves that make up an image is much more important than the magnitude of these waves, something that is not necessarily true for audio processing. A simple experiment to demonstrate this is shown in [Figure 2.11](#).

2.6.1 The Fourier Transform

For a signal $f(t) \in L_2(\mathbb{R})$ (square summable signals), the *Fourier transform* is defined as:

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t} dt \quad (2.6.1)$$

This can be extended to two dimensions for signals $f(\mathbf{u}) \in L_2(\mathbb{R}^2)$:

$$F(\boldsymbol{\omega}) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\mathbf{u})e^{-j\omega^t \mathbf{u}} d\mathbf{u} = \langle f(\mathbf{u}), e^{j\omega^t \mathbf{u}} \rangle \quad (2.6.2)$$

The Fourier transform is an invaluable signal expansion, as viewing a signal in the frequency space offers many insights, as well as affording many very useful properties (most notably the efficiency of convolution as a product of Fourier transforms). While it is a mainstay in signal processing, it can be a poor feature descriptor due to the infinite support of its basis functions - the complex sinusoids $e^{j\omega^t u}$. If a single pixel changes in the input it can 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. To achieve a more local Fourier transform we can use the

short time (or short space) Fourier Transform (STFT) or the continuous wavelet transform (CWT). The two are very similar and mainly differ in the way they handle the concept of ‘scale’. We will only discuss the CWT in this review, but for an excellent comparison of the two, we recommend [85, Chapter 1].

2.6.2 The Continuous Wavelet Transform

The *continuous wavelet transform*, like the Fourier Transform, can be used to decompose a signal into its frequency components. Unlike the Fourier transform, these frequency components can be localized in space. To achieve this, we need a bandpass filter, or *mother wavelet* ψ^4 such that:

$$\int_{-\infty}^{\infty} \psi(\mathbf{u}) d\mathbf{u} = \Psi(0) = 0 \quad (2.6.3)$$

Any function that has sufficient decay of energy with frequency and satisfies (2.6.3), is said to satisfy the *admissibility condition*.

As we are working in 2-D for image processing, consider rotations, dilations, and shifts of this function by $\theta \in [0, 2\pi]$, $a > 0$, $\mathbf{b} \in \mathbb{R}^2$ respectively, where

$$\text{Rotation: } R_\theta x(\mathbf{u}) = x(r_{-\theta}\mathbf{u}) \quad (2.6.4)$$

$$\text{Dilation: } D_a x(\mathbf{u}) = \frac{1}{a} x\left(\frac{\mathbf{u}}{a}\right) \quad (2.6.5)$$

$$\text{Translation: } T_{\mathbf{b}} x(\mathbf{u}) = x(\mathbf{u} - \mathbf{b}) \quad (2.6.6)$$

where r_θ is the 2-D rotation matrix. Now consider shifts, scales and rotations of our bandpass filter

$$\psi_{\mathbf{b}, a, \theta}(\mathbf{u}) = \frac{1}{a} \psi\left(\frac{r_{-\theta}(\mathbf{u} - \mathbf{b})}{a}\right) \quad (2.6.7)$$

which are called the *daughter wavelets*. The 2-D CWT of a signal $x(\mathbf{u})$ is defined as

$$CWT_x(\mathbf{b}, a, \theta) = \int_{-\infty}^{\infty} \psi_{\mathbf{b}, a, \theta}^*(\mathbf{u}) x(\mathbf{u}) d\mathbf{u} = \langle \psi_{\mathbf{b}, a, \theta}(\mathbf{u}), x(\mathbf{u}) \rangle \quad (2.6.8)$$

2.6.2.1 Properties

The CWT has some particularly nice properties, such as *covariance* under the three transformations (2.6.6)-(2.6.4):

$$R_{\theta_0} x \rightarrow CWT_x(r_{-\theta_0}\mathbf{b}, a, \theta + \theta_0) \quad (2.6.9)$$

$$D_{a_0} x \rightarrow CWT_x(\mathbf{b}/a_0, a/a_0, \theta) \quad (2.6.10)$$

$$T_{\mathbf{b}_0} x \rightarrow CWT_x(\mathbf{b} - \mathbf{b}_0, a, \theta) \quad (2.6.11)$$

⁴We use upright ψ, ϕ to distinguish 1-D wavelets from their 2-D counterparts ψ, ϕ

Most importantly, the CWT is now localized in space, which distinguishes it from the Fourier transform. 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 support region of the wavelets you are examining.

2.6.2.2 Inverse

The CWT can be inverted by using a *dual* function $\tilde{\psi}$. There are restrictions on what dual function we can use, namely the dual-wavelet pair must have an admissible constant C_ψ that satisfies the cross-admissibility constraint [86]. Assuming these constraints are satisfied, we can recover x from CWT_x .

2.6.2.3 Interpretation

As the CWT is a convolution with a zero mean function, the wavelet coefficients are only large in the regions of the parameter space (\mathbf{b}, a, θ) where $\psi_{\mathbf{b}, a, \theta}$ ‘match’ the features of the signal. As the wavelet ψ is well localized, the energy of the coefficients CWT_x will be concentrated on the significant parts of the signal.

For an excellent description of the properties of the CWT in 1-D we recommend [87] and in 2-D we recommend [85].

2.6.3 Discretization and Frames

The CWT is highly redundant. We have taken a 2-D signal and expressed it in 4 dimensions (2 offset, 1 scale and 1 rotation). In reality, we would like to sample the space of the CWT in an efficient manner. We would ideally like to fully retain all information in x (be able to reconstruct x from the samples) while sampling over (\mathbf{b}, a, θ) as little as possible to avoid redundancy. To understand how to do this we must briefly talk about frames.

A set of vectors $\phi = \{\varphi_i\}_{i \in I}$ in a Hilbert space \mathbb{H} is a *frame* if there exist two constants $0 < A \leq B < \infty$ such that for all $x \in \mathbb{H}$:

$$A\|x\|^2 \leq \sum_{i \in I} |\langle x, \varphi_i \rangle|^2 \leq B\|x\|^2 \quad (2.6.12)$$

with A, B called the *frame bounds* [20]. The frame bounds relate to the issue of stable reconstruction. In particular, no vector x with $\|x\| > 0$ should be mapped to 0, as this would violate the bound on A from below. This can be interpreted as ensuring our set ϕ covers the entire frequency space. The upper bound ensures that the transform coefficients are bounded.

Any finite set of vectors that spans the space is a frame. An orthonormal basis is a commonly known non-redundant frame where $A = B = 1$ and $|\varphi_i| = 1$ (e.g. the Discrete Wavelet Transform or the Fourier Transform). Tight frames are frames where $A = B$ and Parseval tight frames have the special case $A = B = 1$. It is possible to have frames that have

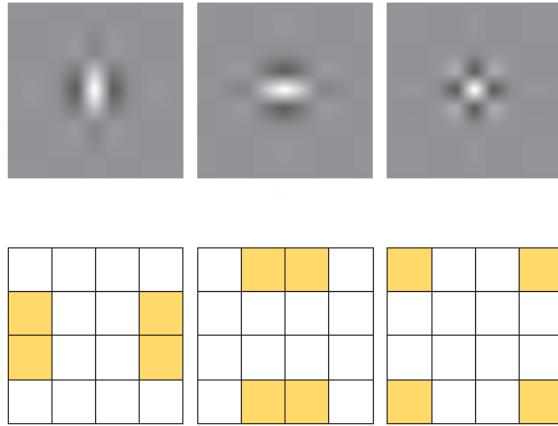


Figure 2.12: **Typical wavelets from the 2-D separable DWT.** Top: Wavelet point spread functions for ψ^v (low-high), ψ^h (high-low), and ψ^d (high-high) wavelets. High-high wavelets are in a checkerboard pattern, with no favoured orientation. Bottom: Idealized support of the spectra of each of the wavelets. Image is taken from [18].

more vectors than dimensions, and this will be the case with many expansions we explore in this thesis.

If $A = B$ and $|\varphi_i| = 1$, then A is the measure of the redundancy of the frame. Of course, for the orthogonal basis, $A = 1$ when $|\varphi_i| = 1$ so there is no redundancy. For the 2-D DT \mathbb{C} WT which we will see shortly, the redundancy is 4.

2.6.3.1 Inversion and Tightness

Equation (2.6.12) specify the constraints that make a frame representation invertible. The tighter the frame bounds, the more easy it is to invert the signal. This gives us a guide to choosing the sampling grid for the CWT.

One particular inverse operator is the *canonical dual frame*. If we define the frame operator $S = \Phi\Phi^*$ then the canonical dual of Φ is defined as $\tilde{\Phi} = \{\tilde{\varphi}_i\}_{i \in I}$ where:

$$\tilde{\varphi}_i = S^{-1}\varphi_i \quad (2.6.13)$$

then from [20] we have:

$$x = \sum_{i \in I} \langle x, \varphi_i \rangle \tilde{\varphi}_i = \sum_{i \in I} \langle x, \tilde{\varphi}_i \rangle \varphi_i \quad (2.6.14)$$

If a frame is tight, then so is its dual.

2.6.4 Discrete Wavelet Transform

(2.6.7) gave the equation for the daughter wavelets in 2-D, in 1-D at scales $a = 2^j, j \in \mathbb{Z}$, and translations $b = 2^j l, l \in \mathbb{Z}$, this is simply:

$$\psi_{j,l}(t) = \frac{1}{\sqrt{a}} \psi\left(\frac{t-b}{a}\right) = 2^{-j/2} \psi(2^{-j}t - l) \quad (2.6.15)$$

where we have chosen to redefine the dilation parameter a in terms of a scale factor j . As j increases, we move to *coarser* scales.

The 2-D DWT has one scaling function and three wavelet functions, composed of the product of 1-D wavelets in the horizontal (u_1) and vertical (u_2) directions:

$$\phi(\mathbf{u}) = \phi(u_1)\phi(u_2) \quad (2.6.16)$$

$$\psi^h(\mathbf{u}) = \phi(u_1)\psi(u_2) \quad (2.6.17)$$

$$\psi^v(\mathbf{u}) = \psi(u_1)\phi(u_2) \quad (2.6.18)$$

$$\psi^d(\mathbf{u}) = \psi(u_1)\psi(u_2) \quad (2.6.19)$$

with h, v, d indicating the sensitivity to horizontal, vertical and diagonal edges. The point spread functions for the wavelet functions are shown in Figure 2.12.

For the four equations above (2.6.16) – (2.6.19), define the daughter wavelets as:

$$\phi_{lm}^j(\mathbf{u}) = \phi_{j,l}(u_1)\phi_{j,m}(u_2) \quad (2.6.20)$$

$$\psi_{lm}^{h,j}(\mathbf{u}) = \phi_{j,l}(u_1)\psi_{j,m}(u_2) \quad (2.6.21)$$

$$\psi_{lm}^{v,j}(\mathbf{u}) = \psi_{j,l}(u_1)\phi_{j,m}(u_2) \quad (2.6.22)$$

$$\psi_{lm}^{d,j}(\mathbf{u}) = \psi_{j,l}(u_1)\psi_{j,m}(u_2) \quad (2.6.23)$$

for $l, m \in \mathbb{Z}$ where l, m define horizontal and vertical translation. We can then get an orthonormal basis with the set $\{\phi_{lm}^j, \psi_{lm}^{h,j}, \psi_{lm}^{v,j}, \psi_{lm}^{d,j}\}_{j,l,m}$. The wavelet coefficients at a chosen scale and location can then be found by taking the inner product of the signal x with the daughter wavelets.

2.6.4.1 Shortcomings

The Discrete Wavelet Transform (DWT) is an orthogonal basis. It is a natural first signal expansion to consider when frustrated with the limitations of the Fourier transform. It is also a good example of the limitations of non-redundant transforms, as it suffers from several drawbacks:

- The DWT is sensitive to the zero crossings of its wavelets. We would like singularities in the input to yield large wavelet coefficients, but this may not always be the case. See [Figure 2.13](#).
- 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 are not specific to other orientations. See [Figure 2.12](#).
- They are not shift-invariant - small shifts greatly perturb the wavelet coefficients. [Figure 2.13](#) shows this for the centre-left and centre-right images.

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. This shortcoming can be overcome with the undecimated DWT [88], [89], but it comes with a heavy computational and memory cost.

2.6.5 Complex Wavelets

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 lacked.

The Fourier transform does not suffer from a lack of directional selectivity and shift-variance because its basis functions are derived from complex sinusoids⁵:

$$e^{j\omega t} = \cos(\omega t) + j \sin(\omega t) \quad (2.6.24)$$

whereas the DWT's basis functions are purely real. 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 flips between -1 and 1, and its magnitude is a rectified sinusoid.

The nice properties of the complex sinusoids come from the fact that the cosine and sine functions of the Fourier transform form a Hilbert pair and together constitute an analytic signal.

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

$$\psi_c(t) = \psi_r(t) + j\psi_i(t) \quad (2.6.25)$$

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

⁵We have temporarily switched to 1D notation here as it is clearer and easier to use, but the results still hold for 2-D.

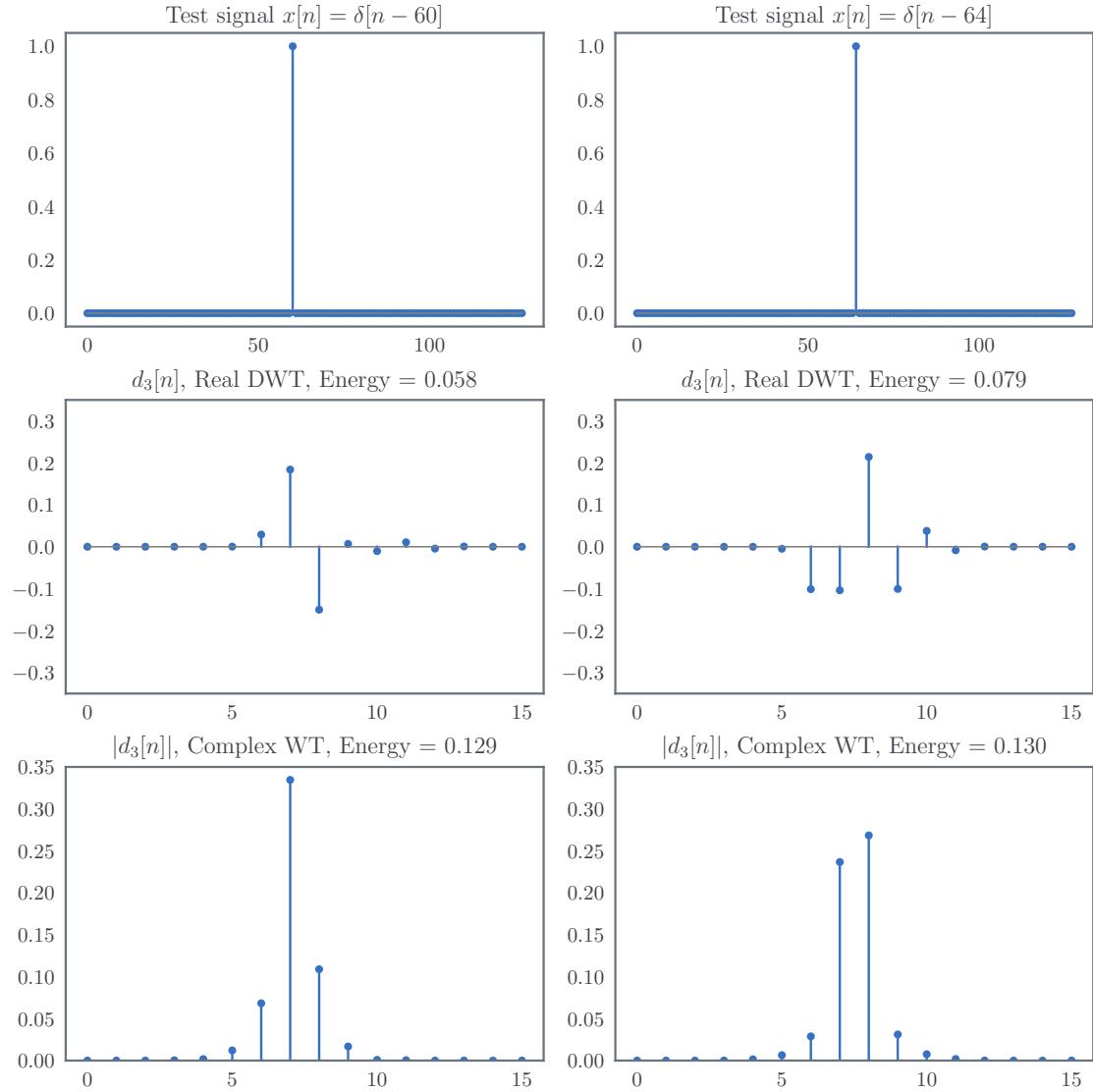


Figure 2.13: **Sensitivity of DWT coefficients to zero crossings and small shifts.** Two impulse signals $\delta(n - 60)$ and $\delta(n - 64)$ are shown (top), as well as the wavelet coefficients for scale $j = 3$ for the DWT (middle) and for the DTCWT (bottom). In the middle row, not only are the coefficients 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 DTCWT coefficients has remained far more constant, as has the shape of the envelope of the output. Image adapted from [18].

There are a number of possible ways to do a wavelet transform with complex wavelets. We examine two in particular: a Fourier-based, sampled CWT using Morlet wavelets, and the Dual-Tree Complex Wavelet Transform (DTCWT) developed by Kingsbury [18], [19], [90]–[95].

We look at the Morlet wavelet transform because it is used by Mallat et. al. in their scattering transform [23], [25], [96]–[102]. We believe the DTCWT has several advantages over the Morlet based implementation, and has been the basis for most of our work.

Let us write the wavelet transform of an input x as

$$\mathcal{W}x = \{x * \phi_J, x * \psi_\lambda\}_\lambda \quad (2.6.26)$$

where $\lambda = (j, k)$ with $j \in \{1, 2, \dots, J\}$ indexing the scale⁶ and $k \in \{0, 1, \dots, K - 1\}$ indexing the orientations of the chosen wavelet transform, whether it be the DTCWT or Morlet transform.

2.6.6 Sampled Morlet Wavelets

The wavelet transform used by Mallat et. al. in their scattering transform is an efficient implementation of the Gabor Transform. While the Gabor wavelets have the best theoretical trade-off between spatial and frequency localization, they have a (usually small) non-zero mean. This violates (2.6.3) making 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 2-D):

$$\psi(\mathbf{u}) = \frac{1}{2\pi\sigma^2} (e^{i\mathbf{u}^t \xi} - \beta) e^{-\frac{|\mathbf{u}|^2}{2\sigma^2}} \quad (2.6.27)$$

where β is this extra degree of freedom, and usually $\beta \ll 1$. σ is the size of the gaussian window and ξ is the approximate location of the peak frequency response in the Fourier plane, with $-\pi \leq \xi_1, \xi_2 \leq \pi$.

Bruna and Mallat [23] add a further additional degree of freedom in their original design by allowing for a non-circular Gaussian window over the complex sinusoid, which gives control over the angular resolution of the final wavelet. This means that (2.6.27) can be expressed as:

$$\psi(\mathbf{u}) = \frac{\gamma}{2\pi\sigma^2} (e^{i\mathbf{u}^t \xi} - \beta) e^{-\mathbf{u}^t \Sigma^{-1} \mathbf{u}} \quad (2.6.28)$$

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 2.14. A full family of Morlet wavelets at varying scales and orientations is shown in Figure 2.15.

⁶We try to number everything from zero in this thesis, but number j from 1 as is the practice in [18], [19].



Figure 2.14: **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 $\sigma = 2, \gamma = 0.5$.

2.6.6.1 Tightness and Invertibility

Recall our definition of the wavelet transform \mathcal{W} from (2.6.26). Assuming the transform is bounded, we can always scale it so that it satisfies Plancherel's equality

$$\|\mathcal{W}x\| = \|x\| \quad (2.6.29)$$

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). Scaling the transform changes the upper bound B in (2.6.12) to 1 and makes the lower bound $A = 1 - \alpha$, where α is a measure of how non-tight a frame is.

Using the capital notation to denote the Fourier transform, define the function $A(\omega)$ to be the coverage each wavelet family has over the frequency plane:

$$A(\omega) = |\Phi_J(\omega)|^2 + \sum_{\lambda} |\Psi_{\lambda}(\omega)|^2 \quad (2.6.30)$$

For a unit norm input $\|x\|^2 = 1$ and scaled wavelets, we can now change (2.6.12) to be:

$$1 - \alpha \leq A(\omega) \leq 1 \quad (2.6.31)$$

If $A(\omega)$ is ever close to 0, then there is not a good coverage of the frequency plane at that location. Figure 2.15 shows the frequency coverage of a few sample grids over the CWT parameters used by Mallat. Invertibility is possible, but not guaranteed for all configurations.

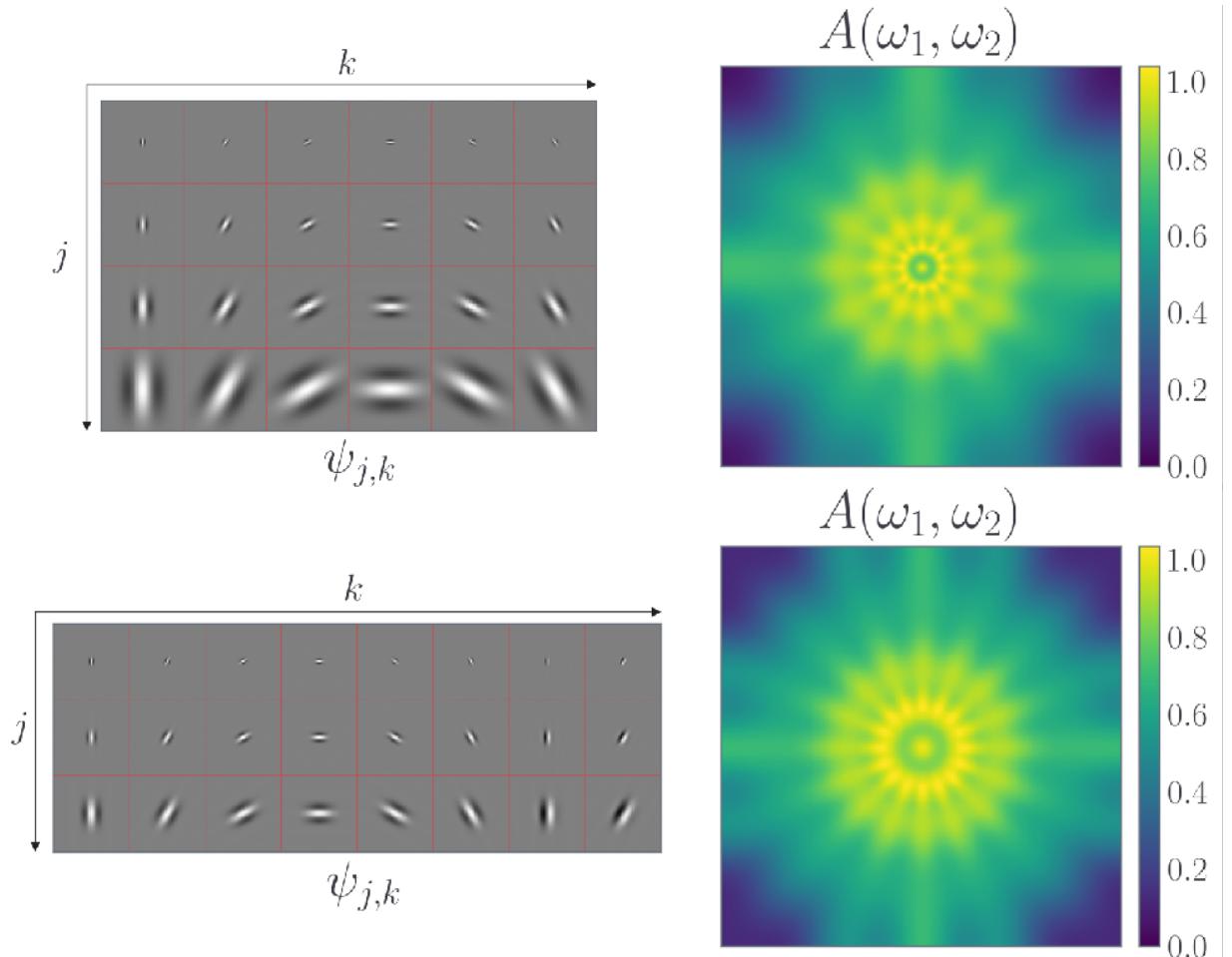


Figure 2.15: Two Morlet wavelet families and their tiling of the frequency plane. For each set of parameters, the point spread functions of the *real* wavelet bases are shown, next to their coverage of the frequency plane $A(\boldsymbol{\omega})$. Each square is 45×45 pixels. Top: $J = 3, K = 6, Q = 1$, Bottom: $J = 4, K = 8, Q = 1$. None of the configurations cover the corners of the frequency plane, but this is often mostly noise. Increasing J, K or Q gives better frequency localization but at the cost of spatial localization and added complexity. Image is adapted from [101].

The tightness of the frame is determined by the sampling grid of our wavelets parameters (\mathbf{b}, a, θ) . Common choices for sampling grids for 2-D wavelets are [85, Section 2.2]:

- For dilations, $a = 2^{j/Q}$ for $j \in \mathbb{Z}$ controlling the scale and Q , the number of scales per octave.
- For rotations, subdivide the interval $[0, \pi)$ into K sections, and choose $\theta_k = \frac{k\pi}{K}$, $k = \{0, 1, \dots, K-1\}$.
- For the translations, set the offsets $\mathbf{b} = (l2^{j/Q}, m2^{j/Q})$, $l, m \in \mathbb{Z}$.

2.6.7 The DT \mathbb{C} WT

The DT \mathbb{C} WT was first proposed by Kingsbury in [91], [92] as a way to combat many of the shortcomings of the DWT such as its poor directional selectivity and its poor shift-invariance. A thorough analysis of the properties and benefits of the DT \mathbb{C} WT is done in [18], [93]. Building on these properties, it has been used successfully for denoising and inverse problems [103]–[106], texture classification [107], [108], image registration [109], [110] and SIFT-style keypoint generation matching [111]–[115] amongst many other applications. Compared to Gabor (or Morlet) image analysis, the authors of [18] sum up the dangers as:

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

This nicely summarises the difference between this method and the Fourier based method outlined in subsection 2.6.6. 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.

2.6.7.1 Design Criteria for the DT \mathbb{C} WT

As in subsection 2.6.5, we want to have a complex mother wavelet $\psi_c = \psi_r + j\psi_i$ and complex scaling function $\phi_c = \phi_r + j\phi_i$, but now achieved with filter banks. The complex component allows for support of both the wavelet and scaling functions on only one half of the frequency plane.

The dual-tree framework shown in Figure 2.16 can achieve this by making the real and imaginary components with their own DWT. We define:

- h_0, h_1 the low and high-pass analysis filters for ϕ_r, ψ_r
- g_0, g_1 the low and high-pass analysis filters for ϕ_i, ψ_i
- \tilde{h}_0, \tilde{h}_1 the low and high pass synthesis filters for $\tilde{\phi}_r, \tilde{\psi}_r$.
- \tilde{g}_0, \tilde{g}_1 the low and high pass synthesis filters for $\tilde{\phi}_i, \tilde{\psi}_i$.

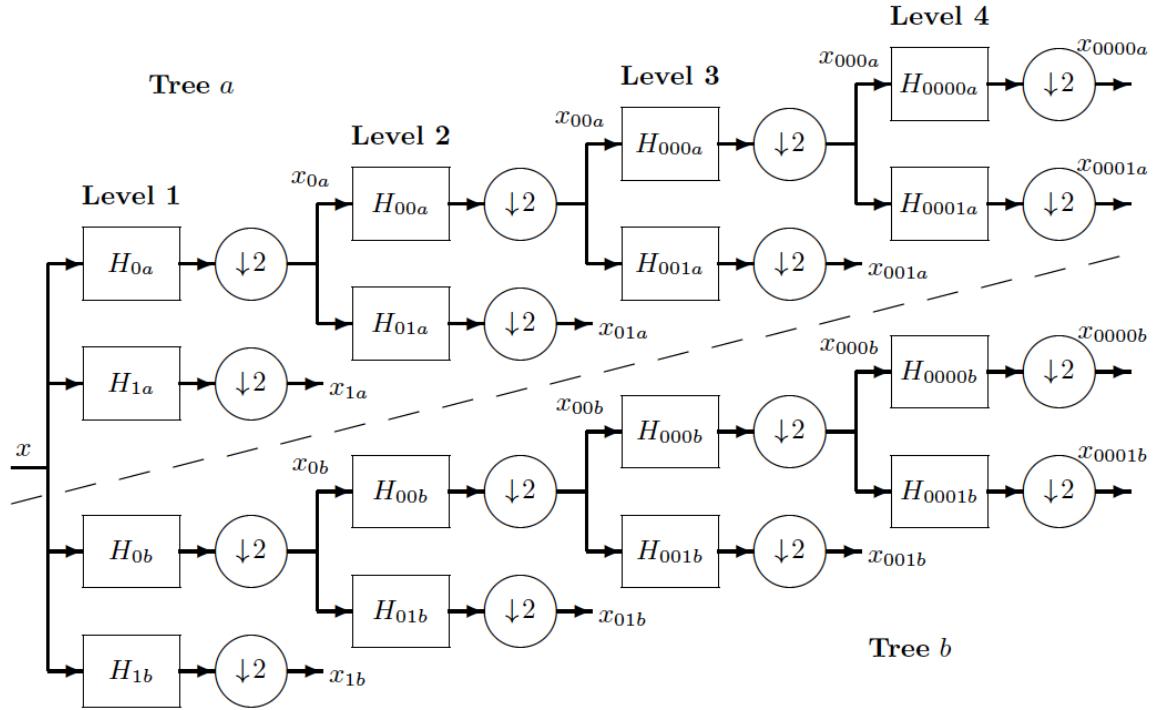


Figure 2.16: **Analysis FBs for the 1-D DTCWT.** Top ‘tree’ forms the real component of the complex wavelet ψ_r , and the bottom tree forms the imaginary (Hilbert pair) component ψ_i . Image is taken from [93].

The dilation and wavelet equations for a 1D filter bank implementation are [18]:

$$\phi_r(t) = \sqrt{2} \sum_n h_0(n) \phi_r(2t - n) \quad (2.6.32)$$

$$\psi_r(t) = \sqrt{2} \sum_n h_1(n) \phi_r(2t - n) \quad (2.6.33)$$

$$\phi_i(t) = \sqrt{2} \sum_n g_0(n) \phi_i(2t - n) \quad (2.6.34)$$

$$\psi_i(t) = \sqrt{2} \sum_n g_1(n) \phi_i(2t - n) \quad (2.6.35)$$

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 in [93] (and later proved by Selesnick in [116]) that the necessary conditions are conceptually very simple. One low-pass filter must be a *half-sample shift* of the other. I.e., if $g_0(n) = h_0(n - 1/2)$ then the corresponding wavelets are a Hilbert transform pair

$$\psi_g(t) \approx \mathcal{H}\{\psi_h(t)\} \quad (2.6.36)$$

As the DT \mathbb{C} WT is designed as an invertible filter bank implementation, this is only one of the constraints. As with conventional (real) discrete wavelets, there are also perfect reconstruction, finite support, linear phase and vanishing moment constraints to consider in the filter bank design.

The derivation of the filters that meet these conditions is covered in detail in [19], [117], and in general in [18]. The result is the option of three main 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.

2.6.7.2 2-D DT \mathbb{C} WT and its Properties

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

Figure 2.17a shows the spectrum of the wavelet when the separable product uses purely real wavelets, as is the case with the DWT. Figure 2.17b however, shows the separable product of two complex, analytic wavelets resulting in a localized and oriented 2-D wavelet.

Note that in this thesis, we name the wavelets by the direction of the edge that they are most sensitive to. For example, the 135° is the second image in Figure 2.18 and can be obtained by the separable product:

$$\psi(\mathbf{u}) = \Psi_c(u_1)\Psi_c^*(u_2) \quad (2.6.37)$$

$$= (\Psi_r(u_1) + j\Psi_i(u_1))(\Psi_r(u_2) - j\Psi_i(u_2)) \quad (2.6.38)$$

$$= (\Psi_r(u_1)\Psi_r(u_2) + \Psi_i(u_1)\Psi_i(u_2)) + j(\Psi_r(u_1)\Psi_i(u_2) - \Psi_i(u_1)\Psi_r(u_2)) \quad (2.6.39)$$

Similar equations can be obtained for the other five wavelets and the scaling function, by replacing ψ with ϕ for each direction in turn (but not both together), and not taking the complex conjugate in (2.6.37) to get the filters in the right-hand half of the frequency plane. The 2-D DT \mathbb{C} WT requires four 2-D DWTs to calculate the four possible combinations of real and imaginary components. The high and lowpass outputs from these DWTs can then be summed in different ways as in (2.6.39) to get the complex bandpass wavelets. Figure 2.18 shows the resulting wavelets both in the spatial domain and their idealized support in the frequency domain.

2.6.7.3 Tightness and Invertibility

We analysed the coverage of the frequency plane for the Morlet wavelet family 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 q-shift DT \mathbb{C} WT, the wavelet transform is also approximately unitary, i.e.,

$$\|x\|^2 \approx \|\mathcal{W}x\|^2 \quad (2.6.40)$$

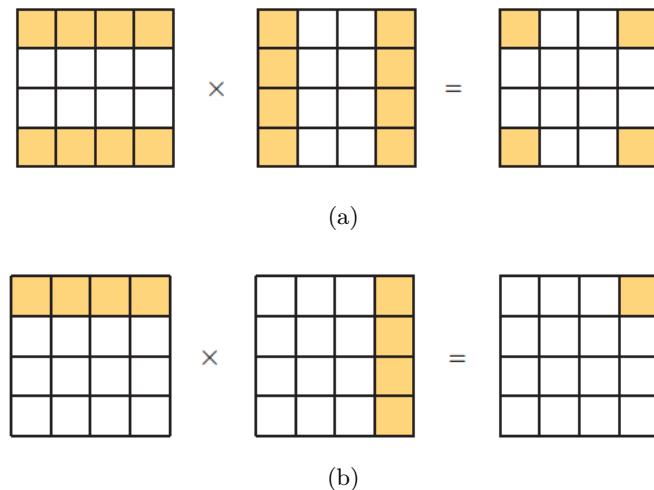


Figure 2.17: **The DWT high-high vs the DT-CWT high-high frequency support.** (a) The high-high DWT wavelet having a passband in all 4 corners of the frequency plane vs (b) the high-high DT-CWT wavelet frequency support only existing in one quadrant. Figure is taken from [18]

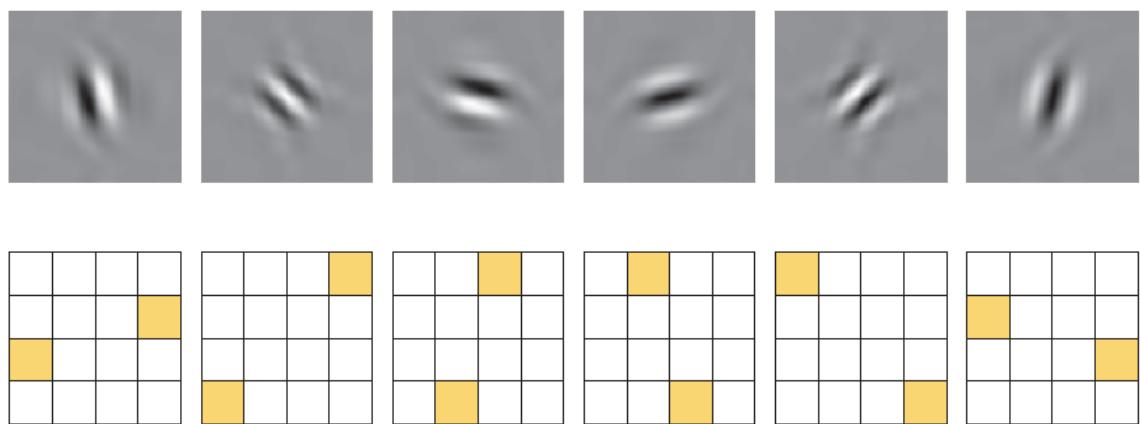


Figure 2.18: **Wavelets from the 2-D DT-CWT.** **Top:** The six oriented filters in the space domain (only the real wavelets are shown). From left to right these are the $105^\circ, 135^\circ, 165^\circ, 15^\circ, 45^\circ, 75^\circ$ wavelets. **Bottom:** Idealized support of the Fourier spectrum of each wavelet in the 2-D frequency plane. Spectra of the real wavelets are shown — the spectra of the complex wavelets ($\psi_r + j\psi_i$) only has support in the top half of the plane. Figure is taken from [18].

and the implementation is perfectly invertible as $A(\omega)$ from (2.6.30) function is unity (or very near unity) $\forall \omega \in [-\pi, \pi] \times [-\pi, pi]$. This is not a surprise, as it is a design constraint in choosing the filters, but is nonetheless important to note.

2.6.8 Summary of Methods

One final comparison to make between the DTcWT and the Morlet wavelets is their frequency coverage. The Morlet wavelets have flexibility at the cost of computational expense and can be made to have tighter angular resolution than the DTcWT. However it is not always better to keep using finer 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. We will explore this in more depth in Chapter 3.

2.7 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 [96], and then were rigorously defined by Mallat in [17]. Perhaps the clearest explanation of them, and the most relevant to our work is in [23].

While CNNs have the ability to learn invariances to nuisance variabilities, their properties and optimal configurations are not well understood. It typically takes multiple trials by an expert to find the correct hyperparameters for these networks. A scattering transform instead builds well understood and well-defined invariances.

We first review some of the desirable invariances before describing how a ScatterNet achieves them.

2.7.1 Desirable Properties

2.7.1.1 Translation-Invariance

Translation is often said to be uninformative for classification — an object appearing in the centre of the image *should* be treated the same way as a similar object appearing near the corner of an image. This can be quantified by saying a representation Φx is invariant to global translations $x_c(\mathbf{u}) = x(\mathbf{u} - \mathbf{c})$ by $\mathbf{c} = (c_1, c_2) \in \mathbb{R}^2$ if:

$$\|\Phi x_c - \Phi x\| \leq C \tag{2.7.1}$$

for some small constant $C > 0$. Note that we may instead want only local translation-invariance and restrict the distance $|\mathbf{c}|$ for which (2.7.1) is true.

Convolutional filters are naturally covariant to translations in the pixel space, so $\Phi x_c = (\Phi x)_c$, $\mathbf{c} \in \mathbb{Z}^2$. Of course, natural objects exist in continuous space and are sampled, and any

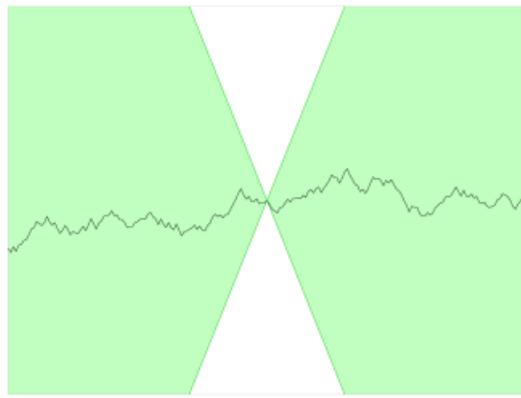


Figure 2.19: **A Lipschitz continuous function.** There is a cone for this function (shown in white) such that the graph always remains entirely outside the cone as it is shifted across. The minimum gradient needed for this to hold is called the ‘best Lipschitz constant’.

two images of the same scene taken with small camera disturbances are unlikely to be at integer pixel shifts of each other.

2.7.1.2 Stability to Noise

Stability to additive noise is another useful invariance to incorporate, as it is a common feature in sampled 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 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. Alternatively, the modulus squared has derivative everywhere but is not Lipschitz continuous as its gradient grows with x .

To be stable to additive noise, we require that for a new signal $x'(\mathbf{u}) = x(\mathbf{u}) + \epsilon(\mathbf{u})$, there must exist a bounded $C > 0$ s.t.

$$\|\Phi x' - \Phi x\| \leq C \|x' - x\| \quad (2.7.2)$$

2.7.1.3 Stability to Deformations

Small deformations are important to be invariant to but this must be limited. It is important to keep intra-class variations small but not be 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 $C_\tau > 0$ s.t.

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

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

2.7.2 Definition

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 large support (infinite in theory) 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.

Another translation-invariant and stable operator is the averaging kernel which Mallat et. al. use to make the zeroth scattering coefficient:

$$S[\emptyset]x \triangleq x * \phi_J(2^J \mathbf{u}) \quad (2.7.4)$$

which is translation-invariant to shifts less than 2^J . It, unfortunately, results in a loss of information due to the removal of high-frequency content. This is easy to see as the wavelet operator from (2.6.26) $\mathcal{W}x = \{x * \phi_J, x * \psi_\lambda\}_\lambda$ contains all the information of x , whereas the zeroth scattering coefficient is simply the lowpass portion of \mathcal{W} .

This high-frequency content can be ‘recovered’ by keeping the wavelet coefficients. The wavelet terms, like a convolutional layer in a CNN, are only covariant to shifts rather than invariant. This covariance happens in the real and imaginary parts which both vary rapidly. Fortunately, its modulus is much smoother and gives a good measure for the frequency-localized energy content at a given spatial location⁷. Unlike the Fourier modulus, the complex wavelet modulus is stable to deformations due to the grouping together of frequencies into dyadic packets [17].

We combine the wavelet transform and modulus operators into one operator $\widetilde{\mathcal{W}}$:

$$\widetilde{\mathcal{W}}x = \{x * \phi_J, |x * \psi_\lambda|\}_\lambda \quad (2.7.5)$$

$$= \{x * \phi_J, U[\lambda]x\}_\lambda \quad (2.7.6)$$

where the U terms are called the *propagated* signals. These U terms are approximately invariant for shifts of up to 2^j . Mallat et. al. choose to keep the same level of invariance as the

⁷Interestingly, the modulus operator can often still be inverted due to the redundancies of the complex wavelet transform [118], hence it does not lose any information.

zeroth-order coefficients (2^J) by further averaging. This makes the first ordering scattering coefficients:

$$S[\lambda_1]x \triangleq U[\lambda_1]x * \phi_J = |x * \psi_{\lambda_1}| * \phi_J \quad (2.7.7)$$

Again this averaging comes at a cost of discarding high-frequency information, this time about the wavelet sparsity signal $U[\lambda] = |x * \psi_\lambda|$ instead of the input signal x . We can recover this information by repeating the above process.

$$S[\lambda_1, \lambda_2]x \triangleq U[\lambda_2]U[\lambda_1]x \quad (2.7.8)$$

$$= ||x * \psi_{\lambda_1}| * \psi_{\lambda_2}| * \phi_J \quad (2.7.9)$$

In general, let $p = (\lambda_1, \lambda_2, \dots, \lambda_m)$ be a path of length m describing the order of application of wavelets, and define:

$$U[p]x = U[\lambda_m]U[\lambda_{m-1}] \cdots U[\lambda_1]x \quad (2.7.10)$$

$$= ||\cdots |x * \psi_{\lambda_1}| * \psi_{\lambda_2}| \cdots * \psi_{\lambda_m}| \quad (2.7.11)$$

and the m th order scattering coefficient along the path p is $S[p]x = U[p]x * \phi_J$. Further, let $p + \lambda = (\lambda_1, \lambda_2, \dots, \lambda_m, \lambda)$. This allows us to recursively define the next set of *propagated* and *scattering* coefficients by using $\widetilde{\mathcal{W}}$:

$$\widetilde{\mathcal{W}}U[p]x = \{S[p]x, U[p + \lambda]x\}_\lambda \quad (2.7.12)$$

which is shown in [Figure 2.20](#)

2.7.3 Resulting Properties

For ease, let us define the ‘ m th order scattering coefficients’ as S_m which is the set of all coefficients with path length m . Further, let S be the set of all scattering coefficients of any path length. The energy $\|Sx\|^2$ we then define as

$$\|Sx\|^2 = \sum_p \|S[p]x\|^2 \quad (2.7.13)$$

We can make \mathcal{W} non-expansive with appropriate scaling. Define the energy $\|\mathcal{W}x\|^2$ as

$$\|\mathcal{W}x\|^2 = \|x * \phi\|^2 + \sum_\lambda \|x * \psi_\lambda\|^2 \quad (2.7.14)$$

then by Plancherel’s formula

$$(1 - \epsilon) \|x\|^2 \leq \|\mathcal{W}x\|^2 \leq \|x\|^2 \quad (2.7.15)$$

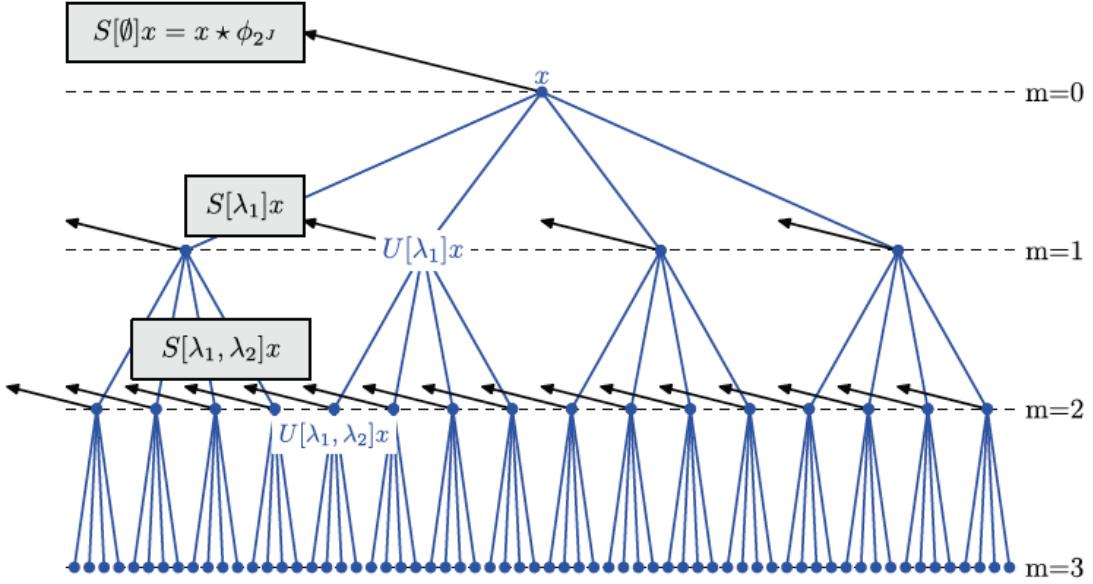


Figure 2.20: **The Scattering Transform.** Scattering outputs are the leftward pointing arrows $S[p]x$, and the intermediate coefficients $U[p]x$ are the centre nodes of the tree. Taken from [23].

For the Morlet wavelets originally used in [23], $\epsilon = 0.25$, for the DTCWT $\epsilon \approx 0$ (for the q-shift DT \mathbb{C} WT it is 0, but for the biorthogonal DT \mathbb{C} WT it is close to but not exactly 0).

2.7.3.1 Translation-Invariance

This is proven in section 2.4 of [17]. We have so far described the Scattering representation as being ‘translation-invariant for shifts up to 2^J ’. We formalize this statement here.

For a 2-D averaging filter based on a father wavelet ϕ , $\phi_J = 2^{-J}\phi(2^{-J}\mathbf{u})$ it is proven in Appendix B of [17] that shifting it by \mathbf{c} , which we denote as \mathcal{L}_c , is Lipschitz continuous:

$$\|\mathcal{L}_c\phi_J - \phi_J\| \leq 2^{-J+2}\|\nabla\phi\|_1\|\mathbf{c}\| \quad (2.7.16)$$

where $\|\nabla\phi\|_1$ is the ℓ_1 norm of the grad of ϕ .

For simplicity, let us define $A_Jx = \phi_J * x$ and $Sx = A_JUx$. Then we get:

$$\|S\mathcal{L}_c x - Sx\| = \|\mathcal{L}_c A_J Ux - A_J Ux\| \quad (2.7.17)$$

$$\leq \|\mathcal{L}_c A_J - A_J\| \|Ux\| \quad (2.7.18)$$

$$\leq 2^{-J+2}\|\nabla\phi\|_1\|\mathbf{c}\|\|x\| \quad (2.7.19)$$

2.7.3.2 Stability to Noise

As \mathcal{W} is non-expansive and the complex modulus is also non-expansive

$$\|\widetilde{\mathcal{W}}x - \widetilde{\mathcal{W}}y\| \leq \|x - y\| \quad (2.7.20)$$

We have already shown that S is the repeated application of $\widetilde{\mathcal{W}}$ in (2.7.12), we can then say

$$\|Sx - Sy\| \leq \|x - y\| \quad (2.7.21)$$

making scattering non-expansive and stable to noise. With the DTCWT, both (2.7.20) and (2.7.21) are nearly inequalities as ϵ is close to 0.

2.7.3.3 Stability to deformations

If $\mathcal{L}_\tau x = x(\mathbf{u} - \tau(\mathbf{u}))$ is an image deformed by a diffeomorphism τ with $\|\tau\|_\infty = \sup_u |\tau(\mathbf{u})|$ and $\|\nabla\tau\|_\infty = \sup_u |\nabla\tau(\mathbf{u})| < 1$ then it is proven in [17] that:

$$\|S\mathcal{L}_\tau x - Sx\| \leq CP \|x\| (2^{-J} \|\tau\|_\infty + \|\nabla\tau\|_\infty) \quad (2.7.22)$$

where $P = \text{length}(p)$ is the scattering order and C is a constant (dependent on J). For deformations with small absolute displacement relative to 2^J , the first term disappears and we have:

$$\|S\mathcal{L}_\tau x - Sx\| \leq CP \|x\| \|\nabla\tau\|_\infty \quad (2.7.23)$$

This theorem shows that S is locally Lipschitz stable to diffeomorphisms and in the case of small deformations, it linearizes them.

2.7.3.4 Energy Decay

As $m \rightarrow \infty$ the invariant coefficients of path length m , U_m , decay towards zero [17]:

$$\lim_{m \rightarrow \infty} U_m = 0 \quad (2.7.24)$$

This is an important property that suggests that we can stop scattering beyond a certain point. Experimental results [23] for image sizes on the order of a few hundred pixels by a few hundred pixels, $m = 3$ captures about 99% of the input energy. For many works using scattering transforms after [23] such as [25], [39], setting $m = 2$ was found to be sufficient.

2.7.3.5 Number of Coefficients

While we have so far talked about non-sampled signals $x(\mathbf{u})$, $\mathbf{u} \in \mathbb{R}^2$, in practice, we want to apply scattering to sampled signals $x[\mathbf{n}]$, $\mathbf{n} \in \mathbb{Z}^2$. The averaging by ϕ_J means that we

Table 2.1: **Redundancy of scattering transform.** Shows the number of output channels C_{out} and number of pixels per channel N_{out} for different scattering orders m , scales J , and orientations K for a single channel input image with N pixels.

m	J	K	C_{out}	N_{out}
1	2	6	13	$N/16$
1	2	8	17	$N/16$
2	2	6	49	$N/16$
2	2	8	81	$N/16$
2	3	6	127	$N/64$
2	3	8	217	$N/64$
3	3	6	343	$N/64$
3	3	8	729	$N/64$

can subsample Sx by 2^J in each direction. However, now we need also need to index all the paths p that can be used to create the scattering coefficients. Limiting ourselves to $m = 2$ and using a wavelet transform with J scales and K discrete orientations the number of paths for each S_m is the cardinality of the set p_m :

$$n(p_0) = 1 \quad (2.7.25)$$

$$n(p_1) = JK \quad (2.7.26)$$

$$n(p_2) = (J-1)K^2 + (J-2)K^2 + \dots + K^2 \quad (2.7.27)$$

$$= \frac{1}{2}J(J-1)K^2 \quad (2.7.28)$$

The reason $n(p_2) \neq J^2K^2$ is due to the demodulating effect of the complex modulus. As $|x * \psi_\lambda|$ is more regular than $x * \psi_\lambda$, $|x * \psi_\lambda| * \psi_{\lambda'}$ is only non-negligible if $\psi_{\lambda'}$ is located at lower frequencies than ψ_λ . This means we can discard over half of the scattering paths as their value will be near zero.

Summing up the above three equations and factoring in the reduced sample rate allowable due to averaging, for an input with N pixels, a second-order scattering representation will have $N2^{-2J} (1 + JK + \frac{1}{2}J(J-1)K^2)$ pixels. [Table 2.1](#) shows some example values of the ScatterNet redundancy for different J, K and scattering order m .

Chapter 3

A Faster ScatterNet

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

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

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

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

3.1 Chapter Layout

This chapter describes how we have built a fast ScatterNet implementation in PyTorch with the DT \mathbb{C} WT as its wavelet transform. First, we describe how to do an efficient DWT in PyTorch in [section 3.4](#) before showing how to expand this to an efficient DT \mathbb{C} WT in [section 3.5](#). We then use the DT \mathbb{C} WT to define our own ScatterNet in [section 3.6](#) (in particular, see [Algorithm 3.5](#)). All of the code is available as an open-source library at [PyTorch Wavelets \[28\]](#).

In parallel with our efforts, the original authors of the ScatterNet have improved their implementation, making a new package called KyMatIO[119]. We compare the speed and classification performance of our package to KyMatIO in section 3.7 as this provides some interesting insights into the choice of complex wavelet for a ScatterNet. This is similar to the work of [36], where Singh and Kingsbury show that a DT^CWT-ScatterNet outperforms a Morlet-ScatterNet when used as a front end to an SVM for some simpler classification tasks. We find that our proposed DT^CWT-ScatterNet is 7 to 15 times faster than KyMatIO (depending on the padding style and wavelet length), as well as giving a small improvement in performance when used as a front end to a CNN.

3.2 Design Constraints

3.2.1 Original Design

The original authors implemented their ScatterNet in Matlab [25] using a Fourier-domain based Morlet wavelet transform. The standard procedure for using ScatterNets in a deep learning framework up until recently has been to:

1. Pre-scatter a dataset using conventional CPU-based hardware and software and store the features to disk. This can take several hours to several days depending on the size of the dataset and the number of CPU cores available.
2. Build a network in another framework, usually Tensorflow [120] or Pytorch [64].
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.

3.2.2 Improvements

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

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

To achieve all of these goals, we choose to build our software on PyTorch using the DTCWT. PyTorch is 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.

3.3 A Brief Description of Autograd

As part of a modern deep learning framework, we need to define functional units like the one shown in [Figure 2.6](#). Not only must we be able to calculate the forward evaluation of a block $Y = f(X, h)$ given an input X and (possibly) some learnable weights h , we must also be able to calculate the passthrough and update gradients $\frac{\partial \mathcal{L}}{\partial X}$, $\frac{\partial \mathcal{L}}{\partial h}$ given $\frac{\partial \mathcal{L}}{\partial Y}$. This typically involves saving $\frac{\partial Y}{\partial X}$ and $\frac{\partial Y}{\partial h}$ evaluated at the current values of X and h when we calculate the forward pass.

For example, the simple ReLU: $Y = \max(0, X)$, is not memory-less. On the forward pass, we need to put a 1 in all the positions where $X > 0$, and a 0 elsewhere. For a convolutional layer, we need to save X and h to correlate with $\frac{\partial \mathcal{L}}{\partial Y}$ on the backwards pass. It is up to the block designer to manually calculate the gradients and design the most efficient way of programming them. For clarity and repeatability, we give pseudocode for all the core operations developed in our package *PyTorch Wavelets*.

3.4 Fast Calculation of the DWT and IDWT

To have a fast implementation of the Scattering Transform, we need a fast implementation of the DTCWT. Similarly, for a fast implementation of the DTCWT we need a fast implementation of the DWT. Future work may also explore the DWT as a basis for learning, so having an implementation that is fast and can pass gradients through will prove beneficial in and of itself.

3.4.1 The Input

Recall from [subsection 2.4.1](#) that our input is a 3-D array with channel dimension first, followed by the two spatial dimensions. For a minibatch of images, this becomes a 4-D array, with the sample number in the first dimension. I.e., for a minibatch of N images with C channels and $H \times W$ spatial support, the input has shape: $N \times C \times H \times W$.

3.4.2 Preliminary Notes

There has been much research into the best way to do the DWT on a GPU, in particular comparing the speed of Lifting [\[121\]](#), or second-generation wavelets, to the direct convolutional methods. [\[122\]](#), [\[123\]](#) are two notable such publications, both of which find that the convolution-based implementations are better suited for the massively parallel architecture found in modern GPUs. For this reason, we implement a convolutional-based DWT.

As the DWT and IDWT use fixed filters, we do not need to calculate $\frac{\partial L}{\partial h}$ on the backwards pass. This means on the forward pass we can save on memory by not storing $\frac{\partial Y}{\partial h} = X$ (see [subsubsection 2.4.1.2](#)). To be explicit, we derive forward rewrite forward and backward passes for the DWT, as the autograd mechanics may often unnecessarily save intermediate activations.

For example, consider an input $X \in \mathbb{R}^{128 \times 256 \times 64 \times 64}$. This is not an unrealistically large input for a CNN, as often minibatch sizes are 128, channels can be in the hundreds and 64×64 pixels is a relatively common input size. If we were to represent these numbers as floats, it would require 512MB of space on a GPU (modern GPUs have about 10GB of memory, so this single activation already requires 5%). If we perform a DWT on this input, we would require another 512MB of space for the output. Using naive autograd, PyTorch (or another framework) saves an extra 512MB of memory for the backwards pass to calculate $\frac{\partial L}{\partial h}$, even if we specify it as not requiring a gradient. This means we can save 50% of the memory cost by being explicit about what is, and what is not needed for backpropagation.

3.4.3 Primitives

We start with the commonly known property that for a 2-D convolution (with 1 input channel and 1 output channel), the passthrough gradient is the gradient w.r.t. the output convolved with the time reverse of the filter. More formally, if $Y(\mathbf{z}) = Y(z_1, z_2) = H(\mathbf{z})X(\mathbf{z})$, then:

$$\Delta X(\mathbf{z}) = H(\mathbf{z}^{-1})\Delta Y(\mathbf{z}) \quad (3.4.1)$$

where $H(\mathbf{z}^{-1})$ is the Z -transform of the time/space reverse of $H(\mathbf{z})$, $\Delta Y(\mathbf{z}) \triangleq \frac{\partial L}{\partial Y}(\mathbf{z})$ is the gradient of the loss with respect to the output, and $\Delta X(\mathbf{z}) \triangleq \frac{\partial L}{\partial X}(\mathbf{z})$ is the gradient of the loss w.r.t. the input. This was proved in [subsubsection 2.4.1.2](#), we have just rewritten it in terms of Z -transforms.

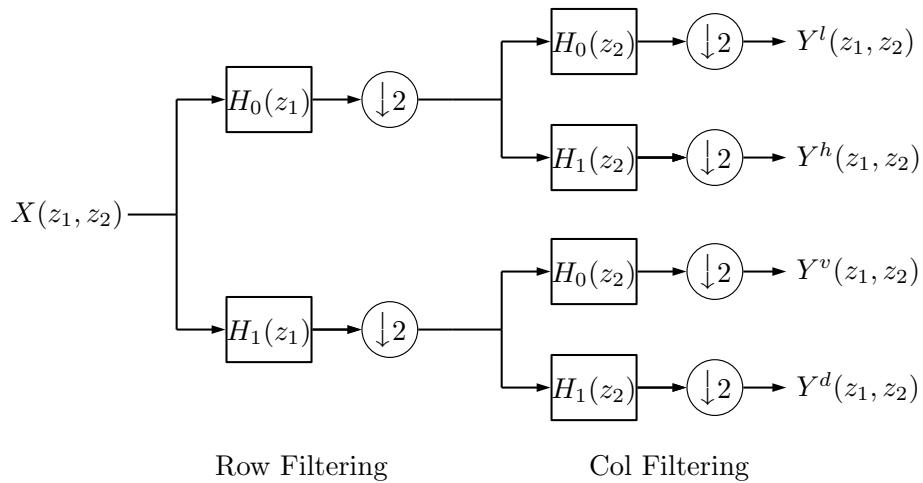


Figure 3.1: **2-D DWT filter bank layout.** The components of a filter bank DWT in two dimensions. Y^l, Y^h, Y^v, Y^d represent the lowpass, horizontal, vertical, and diagonal components.

Additionally, for a decimation block, the passthrough gradient is interpolation (see section B.1 for a proof of this).

Figure 3.1 shows the block diagram for performing the forward pass of a DWT. We can draw a similar figure for the backwards pass by replacing all the forward operations with their passthrough gradients, reversing the flow of arrows, and turning branch splits into sums. Using the above two properties, we see that the backwards pass of a DWT has the same form as the inverse wavelet transform, with the time reverse of the analysis filters used as the synthesis filters. For orthogonal wavelet transforms, the synthesis filters are the time reverse of the analysis filters so backpropagation can be done by doing the inverse wavelet transform with $\Delta Y^l(z), \Delta Y^h(z), \Delta Y^v(z), \Delta Y^d(z)$. See section B.2 for more information on this and the equivalent figure for the backwards pass.

Like Matlab, most deep learning frameworks have an efficient function for doing convolution followed by downsampling. Similarly, there is an efficient function to do upsampling followed by convolution (for the inverse transform). Let us call these `conv2d_down` and `conv2d_up` respectively¹. These functions, in turn, call the cuDNN low-level functions which can only support zero padding or no padding (we called this *valid convolution* in subsubsection 2.4.1.1). If another padding type is desired, it must be done beforehand and valid convolution used.

With both `conv2d_down` and `conv2d_up`, we want to apply the same filter to each channel of the input and *not* sum them up (recall (2.4.5) sums over the channel dimension after doing

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

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

```

1: function AFB1D( $x, h_0, h_1, mode, axis$ )
2:   if  $axis = 3$  then
3:      $h_0, h_1 \leftarrow h_0^t, h_1^t$                                  $\triangleright$  row filtering
4:   end if
5:    $p \leftarrow \lfloor (\text{len}(x) + \text{len}(h_0) - 1)/2 \rfloor$        $\triangleright$  calculate output size
6:    $b \leftarrow \lfloor p/2 \rfloor$                                       $\triangleright$  calculate pad size before
7:    $a \leftarrow \lceil p/2 \rceil$                                        $\triangleright$  calculate pad size after
8:    $x \leftarrow \text{pad}(x, b, a, mode)$                              $\triangleright$  pre pad the signal with selected mode
9:    $lo \leftarrow \text{conv2d\_down}(x, h_0)$ 
10:   $hi \leftarrow \text{conv2d\_down}(x, h_1)$ 
11:  return  $lo, hi$ 
12: end function

1: function SFB1D( $lo, hi, g_0, g_1, axis$ )
2:   if  $axis = 3$  then
3:      $g_0, g_1 \leftarrow g_0^t, g_1^t$                                  $\triangleright$  row filtering
4:   end if
5:    $p \leftarrow \text{len}(g_0) - 2$                                      $\triangleright$  calculate output size
6:    $lo \leftarrow \text{pad}(lo, p, p, 'zero')$                           $\triangleright$  pre pad the signal with zeros
7:    $hi \leftarrow \text{pad}(hi, p, p, 'zero')$                           $\triangleright$  pre pad the signal with zeros
8:    $x \leftarrow \text{conv2d\_up}(lo, g_0) + \text{conv2d\_up}(hi, g_1)$ 
9:   return  $x$ 
10: end function

```

convolution). There are options available to prevent the summing in most frameworks, related to doing *grouped convolution*. For a good explanation on grouped convolution, we recommend Chapter 5 of [48]. In our code below, we assume that `conv2d_down` and `conv2d_up` act independently on the C channels and do not sum across them, giving an output with the same shape as the input - $N \times C \times H \times W$.

3.4.4 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; to do *row* filtering, we must transpose the filters. The necessary steps to do the 1-D analysis and synthesis filtering are described in [Algorithm 3.1](#). We do not need to define backpropagation functions for the `afb1d` (analysis filter bank 1-d) and `sfb1d` (synthesis filter bank 1-D) functions as they are each other's backwards pass.

3.4.5 2-D Transforms and Their Gradients

Having built the 1-D filter banks, we can easily generalize them to 2-D. Furthermore, we can define the backwards steps of both the forward 2-D DWT and the inverse 2-D DWT using

Algorithm 3.2 2-D DWT and its gradient

```

1: function DWT.FORWARD( $x, h_0^r, h_1^r, h_0^c, h_1^c, mode$ )
2:   save  $h_0^r, h_1^r, h_0^c, h_1^c, mode$                                  $\triangleright$  For the backwards pass
3:    $lo, hi \leftarrow \text{afb1d}(x, h_0^r, h_1^r, mode, axis = 3)$            $\triangleright$  row filter
4:    $ll, lh \leftarrow \text{afb1d}(lo, h_0^c, h_1^c, mode, axis = 2)$            $\triangleright$  column filter
5:    $hl, hh \leftarrow \text{afb1d}(hi, h_0^c, h_1^c, mode, axis = 2)$            $\triangleright$  column filter
6:   return  $ll, lh, hl, hh$ 
7: end function

1: function DWT.BACKWARD( $\Delta ll, \Delta lh, \Delta hl, \Delta hh$ )
2:   load  $h_0^r, h_1^r, h_0^c, h_1^c, mode$ 
3:    $\Delta lo \leftarrow \text{sfb1d}(\Delta ll, \Delta lh, h_0^c, h_1^c, mode, axis = 2)$ 
4:    $\Delta hi \leftarrow \text{sfb1d}(\Delta hl, \Delta hh, h_0^r, h_1^r, mode, axis = 2)$ 
5:    $\Delta x \leftarrow \text{sfb1d}(\Delta lo, \Delta hi, h_0^r, h_1^r, mode, axis = 3)$ 
6:   return  $\Delta x$ 
7: end function

```

these filter banks. We show how to do this in [Algorithm 3.2](#). We first save the filters for the backward pass and then do three calls to `afb1d`; one for rows and two for columns. The inverse transform logic is moved to the appendix - in [Algorithm B.1](#).

Note that we have allowed for different row and column filters in [Algorithm 3.2](#), denoted by their r and c superscripts. Most commonly used wavelets will use the same filter for both directions (e.g. the orthogonal Daubechies family), but later when we use the DT \mathbb{C} WT, we will want to have different horizontal and vertical filters.

Further, note that the only things that need to be saved are the filters, as seen in [Algorithm 3.2](#); these are typically only a few floats.

A multiscale DWT can easily be made by calling [Algorithm 3.2](#) multiple times on the lowpass output. Again, no intermediate activations need to be saved, giving this implementation almost no memory overhead.

3.5 Fast Calculation of the DT \mathbb{C} WT

We have built upon previous implementations of the DT \mathbb{C} WT [124]–[126]. The ‘dual-tree’ part of the DT \mathbb{C} WT comes from it having two trees, each with their own set of filters: $a = \{h_0^a, h_1^a, g_0^a, g_1^a\}$ and $b = \{h_0^b, h_1^b, g_0^b, g_1^b\}$. In one dimension, this can be done with two DWTs, one using the a filters and the other using the b filters. In two dimensions, we must instead do four multiscale DWTs. We follow the notation in [93] and maintain separate imaginary operators j_1 and j_2 for the row and column processing. We call our four DWTs r, j_1, j_2, j_1j_2 and show how the filter bank system is connected in [Figure 3.2](#). The output of these four DWTs can be interpreted as a four-element ‘complex’ vector $\{a, b, c, d\} = a + bj_1 + cj_2 + dj_1j_2$. This 4-vector can be converted into a pair of complex 2-vectors by letting $j_1 = j_2 = j$ in one

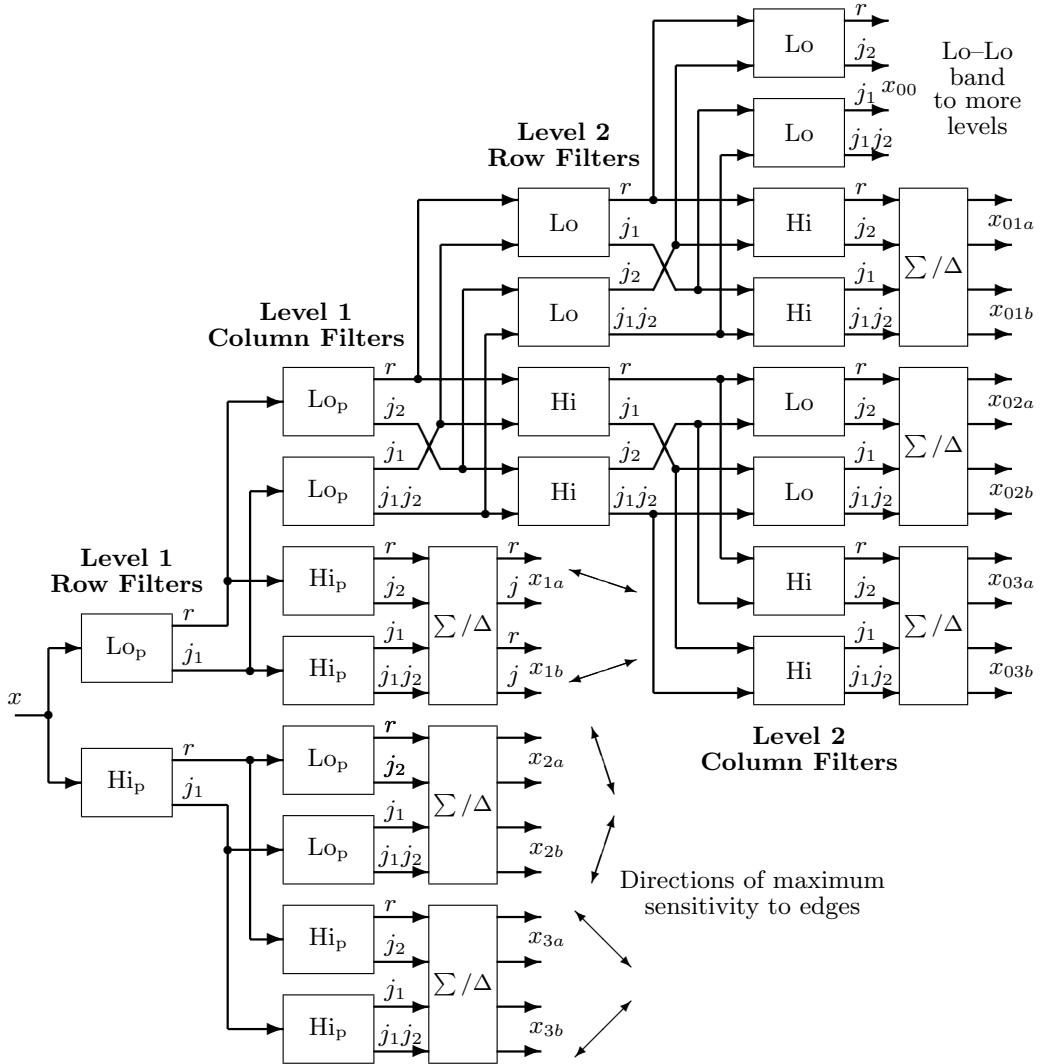


Figure 3.2: **2-D DTcCWT filter bank layout.** Two levels of the complex wavelet tree for a real 2-D input image x . The four DWTs involved in the DTcCWT are labelled with r , j_1 , j_2 and j_1j_2 . Image is taken from [93].

Algorithm 3.3 2-D DTcWT.

```

1: function DTcWT( $x, J, mode$ )
2:   load  $h_0^a, h_1^a, h_0^b, h_1^b$                                  $\triangleright$  Load from memory
3:    $ll_r, ll_{j_1}, ll_{j_2}, ll_{j_1j_2} \leftarrow x/2$ 
4:   for  $1 \leq j \leq J$  do
5:      $ll_r, lh_r, hl_r, hh_r \leftarrow \text{DWT}_r(ll_r, h_0^a, h_1^a, h_0^a, h_1^a, mode)$ 
6:      $ll_{j_1}, lh_{j_1}, hl_{j_1}, hh_{j_1} \leftarrow \text{DWT}_{j_1}(ll_{j_1}, h_0^b, h_1^b, h_0^a, h_1^a, mode)$ 
7:      $ll_{j_2}, lh_{j_2}, hl_{j_2}, hh_{j_2} \leftarrow \text{DWT}_{j_2}(ll_{j_2}, h_0^a, h_1^a, h_0^b, h_1^b, mode)$ 
8:      $ll_{j_1j_2}, lh_{j_1j_2}, hl_{j_1j_2}, hh_{j_1j_2} \leftarrow \text{DWT}_{j_1j_2}(ll_{j_1j_2}, h_0^b, h_1^b, h_0^b, h_1^b, mode)$ 
9:      $x_{1a} \leftarrow (lh_r - lh_{j_1j_2}) + j(lh_{j_1} + lh_{j_2})$ 
10:     $x_{1b} \leftarrow (lh_r + lh_{j_1j_2}) + j(-lh_{j_1} + lh_{j_2})$ 
11:     $x_{2a} \leftarrow (hl_r - hl_{j_1j_2}) + j(hl_{j_1} + hl_{j_2})$ 
12:     $x_{2b} \leftarrow (hl_r + hl_{j_1j_2}) + j(-hl_{j_1} + hl_{j_2})$ 
13:     $x_{3a} \leftarrow (hh_r - hh_{j_1j_2}) + j(hh_{j_1} + hh_{j_2})$ 
14:     $x_{3b} \leftarrow (hh_r + hh_{j_1j_2}) + j(-hh_{j_1} + hh_{j_2})$ 
15:     $yh[j] \leftarrow (x_{1b}, x_{3b}, x_{2b}, x_{2a}, x_{3a}, x_{1a})$ 
16:  end for
17:   $yl \leftarrow \text{interleave}(ll_r, ll_{j_1}, ll_{j_2}, ll_{j_1j_2})$ 
18:  return  $yl, yh$ 
19: end function

```

case and $j_1 = -j_2 = -j$ in the other case, producing the two outputs:

$$(a - d) + (b + c)j \quad (3.5.1)$$

$$(a + d) + (-b + c)j \quad (3.5.2)$$

corresponding to the first and second quadrant directional filters respectively. The sum and difference blocks Σ/Δ in [Figure 3.2](#) do this operation [93].

The four lowpass coefficients from each scale are used for the next scale DWTs. At the final scale, they are interleaved to get four times the expected lowpass output area expected from a single decimated DWT. See [Algorithm 3.3](#) for the code on how to do the full DTcWT.

A requirement of the DTcWT is the need to use different filters for the first scale to all subsequent scales [18]. We have not shown this in [Algorithm 3.3](#) for simplicity, but it would simply mean we would have to handle the $j = 1$ case separately.

We have moved the inverse DTcWT algorithm to [Algorithm B.2](#). Note that for both the DTcWT and inverse DTcWT, we rely on autograd to calculate the backwards pass by calling our defined DWT.Forward and DWT.Backward methods.

Algorithm 3.4 Magnitude forward and backward steps

```

1: function MAG.FORWARD( $x, y$ )
2:    $r \leftarrow \sqrt{x^2 + y^2}$ 
3:    $\theta \leftarrow \arctan 2(y, x)$                                  $\triangleright \arctan 2$  handles  $x = 0$ 
4:   save  $\theta$ 
5:   return  $r$ 
6: end function

1: function MAG.BACKWARD( $\Delta r$ )
2:   load  $\theta$ 
3:    $\Delta x \leftarrow \Delta r \cos \theta$                              $\triangleright$  Reinsert phase
4:    $\Delta y \leftarrow \Delta r \sin \theta$                              $\triangleright$  Reinsert phase
5:   return  $\Delta x, \Delta y$ 
6: end function

```

3.6 The DT^CWT ScatterNet

3.6.1 The Magnitude Operation

Now that we have a forward and backward pass for the DT^CWT, the final missing piece is the magnitude operation. If $z = x + jy$, then:

$$r = |z| = \sqrt{x^2 + y^2} \quad (3.6.1)$$

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

$$\frac{\partial r}{\partial x} = \frac{x}{\sqrt{x^2 + y^2}} = \frac{x}{r} \quad (3.6.2)$$

$$\frac{\partial r}{\partial y} = \frac{y}{\sqrt{x^2 + y^2}} = \frac{y}{r} \quad (3.6.3)$$

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

$$\Delta z = \Delta r \frac{\partial r}{\partial x} + j \Delta r \frac{\partial r}{\partial y} \quad (3.6.4)$$

$$= \Delta r \frac{x}{r} + j \Delta r \frac{y}{r} \quad (3.6.5)$$

$$= \Delta r e^{j\theta} \quad (3.6.6)$$

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 pseudocode for this operation is shown in [Algorithm 3.4](#). These partial derivatives are restricted to be in the range $[-1, 1]$

but have a singularity at the origin. In particular:

$$\lim_{x \rightarrow 0^-, y \rightarrow 0} \frac{\partial r}{\partial x} = -1 \quad (3.6.7)$$

$$\lim_{x \rightarrow 0^+, y \rightarrow 0} \frac{\partial r}{\partial x} = +1 \quad (3.6.8)$$

$$\lim_{x \rightarrow 0, y \rightarrow 0^-} \frac{\partial r}{\partial y} = -1 \quad (3.6.9)$$

$$\lim_{x \rightarrow 0, y \rightarrow 0^+} \frac{\partial r}{\partial y} = +1 \quad (3.6.10)$$

Rather than using the subgradient method [127] which is commonly used to handle the magnitude operation, we propose to smooth the magnitude operator:

$$r_s = \sqrt{x^2 + y^2 + b^2} - b \quad (3.6.11)$$

This keeps the magnitude near zero for small x, y but does slightly shrink larger values, however, our gradient is smoother and we no longer have to worry about dividing by zero issues when calculating gradients. We can choose the size of b as a hyperparameter in optimization, which we revisit in [subsection 3.6.3](#). The partial derivatives now become:

$$\frac{\partial r_s}{\partial x} = \frac{x}{\sqrt{x^2 + y^2 + b^2}} = \frac{x}{r_s + b} \quad (3.6.12)$$

$$\frac{\partial r_s}{\partial y} = \frac{y}{\sqrt{x^2 + y^2 + b^2}} = \frac{y}{r_s + b} \quad (3.6.13)$$

There is a memory cost associated with this, as we will now need to save both $\frac{\partial r_s}{\partial x}$ and $\frac{\partial r_s}{\partial y}$ as opposed to saving only the phase. [Algorithm B.3](#) has the pseudocode for the smooth magnitude.

3.6.2 Putting it all Together

Now that we have the DTcWT and the magnitude operation, it is straightforward to get a DTcWT scattering layer, shown in [Algorithm 3.5](#).

For a second-order ScatterNet, the first C channels of Z from [Algorithm 3.5](#) are the S_0 coefficients, the next JKC are the S_1 coefficients and the final $\frac{1}{2}(J-1)JK^2C$ channels are the S_2 coefficients.

For ease in handling the different sample rates of the lowpass and the bandpass, we have averaged the lowpass over a 2×2 window and downsampled it by 2 in each direction. This slightly affects the higher-order scattering coefficients, as the true DTcWT needs the doubly sampled lowpass for the second scale, however, we noticed little difference in performance from doing the true DTcWT and this slightly modified one.

Algorithm 3.5 DTCWT ScatterNet Layer

```

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

```

Table 3.1: Hyperparameter settings for the DTCWT ScatterNet.

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

3.6.3 DT^CWT ScatterNet Hyperparameter Choice

Before comparing to the Morlet-based ScatterNet, we test different padding schemes, wavelet lengths and magnitude smoothing parameters (see (3.6.11)) for the DTCWT ScatterNet. We test these over a grid of values described in Table 3.1.

The different wavelets have different lengths and hence different frequency responses. Additionally, the ‘near_sym_b_bp’ wavelet is a rotationally symmetric wavelet with diagonal passband brought in by a factor of $\sqrt{2}$, making it have similar $|\omega|$ to the horizontal and vertical bandpasses.

The results of these experiments are shown in Figure 3.3. The choice of options can have a significant impact on classification accuracy. Symmetric padding appears to work marginally better than zero padding. Surprisingly, the shorter filters (near_sym_a) fare better than their longer counterparts and bringing in the diagonal subbands (near_sym_b_bp) does not help.

Additionally, the smoothing bias may indeed be a good idea for the forward pass as well as aiding the backwards pass, so long as it is less than 0.1.

However, it is difficult to tell just how much ‘better’ the different filter lengths and padding schemes are, as there are many other factors to take into consideration when training a CNN. The standard deviations for the results from [Figure 3.3](#) were all in the range from 0.1 to 0.4% for the two CIFAR datasets, which is smaller than the 2% range seen from the best option to the worst. However, for the larger Tiny ImageNet dataset the standard deviations were in the range from 0.2 to 0.6%, which is comparable to the 1% difference between the ‘best’ and ‘worst’ configurations.

We proceed by using the ‘near_sym_a’ filters, with symmetric padding and a smoothing factor of 0.01, but leave all configurations in our code in [\[28\]](#) and note that more experimentation needs to be done in analysing and comparing the choice of options.

3.7 Comparisons

Now that we have the ability to do a DT^CWT based ScatterNet, how does this compare with the original Matlab implementation [\[25\]](#) and the newly developed KyMatIO [\[119\]](#)? [Table 3.2](#) lists the different properties and options of the competing packages.

3.7.1 Speed

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

See [Table 3.3](#) for the execution time results. Type A is ~ 7 times faster than the Fourier-based KyMatIO on GPUs and Type B has a ~ 14 times speedup over the Morlet backend.

Additionally, when compared with version 0.1.0 of KyMatIO, the memory footprint of the DT^CWT based implementation is only 2% of KyMatIO’s, highlighting the importance of being explicit in not saving unnecessary information for backpropagation.

3.7.2 Performance

To confirm that changing the ScatterNet core has not impeded the performance of the ScatterNet as a feature extractor, we build a simple Hybrid ScatterNet, similar to [\[39\]](#). Our net has a second-order scattering transform before four convolutional layers. See [Table 3.4](#) for

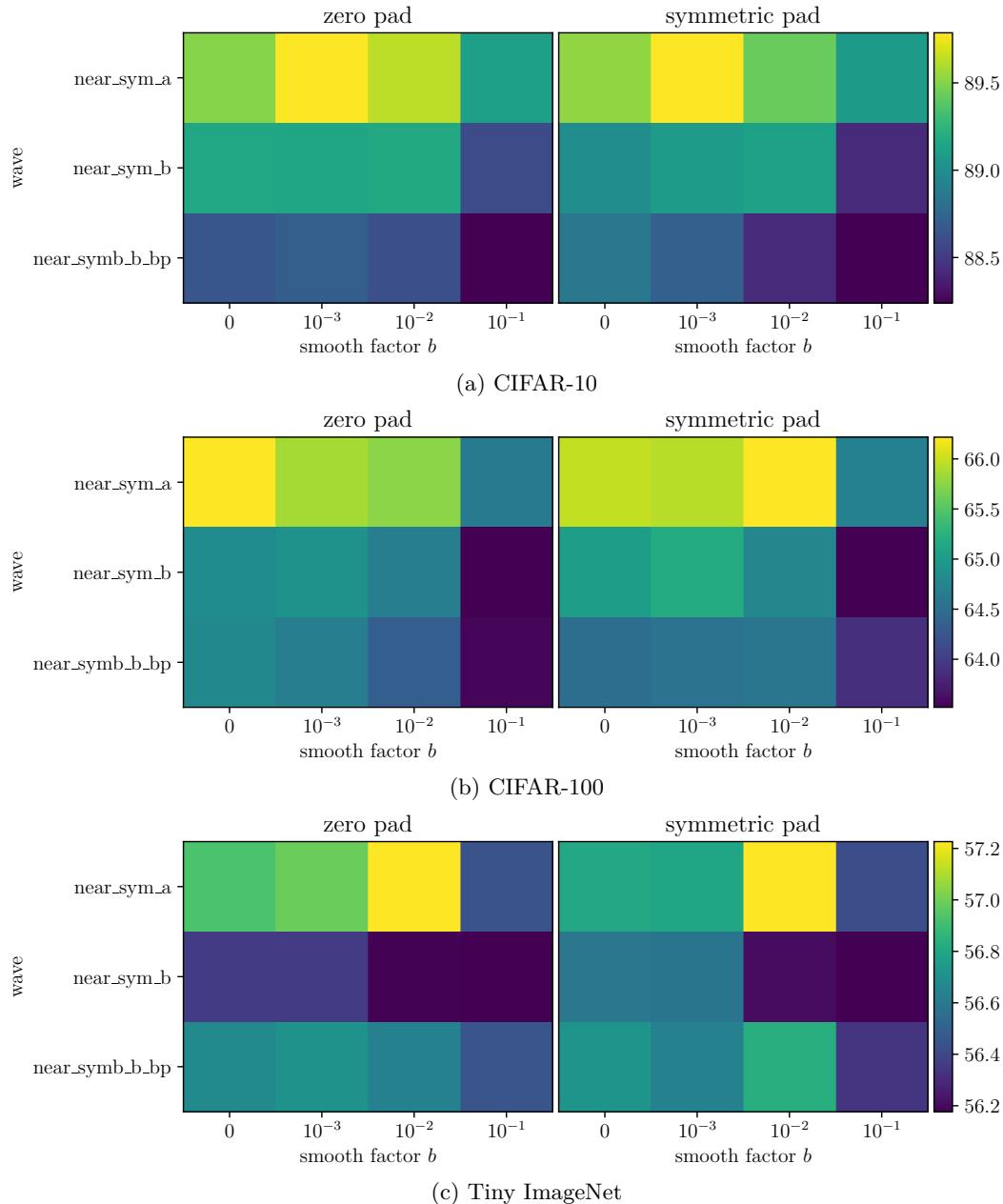


Figure 3.3: Hyperparameter results for the DTcWT ScatterNet on various datasets. Image showing relative top-1 accuracies (in %) on the given datasets using architecture described in Table 3.1. Each subfigure is a new dataset and therefore has a new colour range (shown on the right). Results are averaged over 3 runs from different starting positions. The choice of options can have a very large impact on classification accuracy. Symmetric padding is marginally better than zero padding. Surprisingly, the shorter filter (near_sym_a) fares better than its longer counterparts, and bringing in the diagonal subbands (near_sym_b_bp) does not help. Additionally, the smoothing bias may indeed be a good idea for the forward pass as well as aiding the backwards pass, so long as it is less than 0.1.

Table 3.2: **Comparison of properties of different ScatterNet packages.** In particular, the wavelet backend used, the number of orientations available, the available boundary extension methods, whether it has GPU support and whether it supports backpropagation.

Package	Backend	Orientations	Boundary Ext.	GPU	Backprop
ScatNetLight[25]	FFT-based	Flexible	Periodic	No	No
KyMatIO[119]	FFT-based	Flexible	Periodic	Yes	Yes
DTCWT Scat	Separable filter banks	6	Flexible	Yes	Yes

Table 3.3: **Comparison of execution time for the forward and backward passes of the competing ScatterNet implementations.** Tests were run on the reference architecture described in [appendix A](#). The input for these experiments is a batch of images of size $128 \times 3 \times 256 \times 256$ in 4 byte floating precision. We list two different types of options for our ScatterNet. Type A uses 16 tap filters and has symmetric padding, whereas type B uses 6 tap filters and uses zero padding at the image boundaries. Values are given to 2 significant figures, averaged over 5 runs.

Package	CPU		GPU	
	Fwd (s)	Bwd (s)	Fwd (s)	Bwd (s)
ScatNetLight[25]	> 200.00	n/a	n/a	n/a
KyMatIO[119]	95.0	130.0	1.44	2.5
DTCWT Scat Type A	3.3	3.6	0.21	0.27
DTCWT Scat Type B	2.8	3.2	0.10	0.16

the network layout. We use the optimal hyperparameter choices from the previous section, and compare these to Morlet based ScatterNet with 6 and 8 orientations.

We run tests on the following datasets:

- 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[78]: 200 classes, 500 images per class, 64×64 pixels per image.

The images in Tiny ImageNet 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 SGD 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 [31].

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 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.

3.8 Conclusion

In this chapter we have proposed changing the wavelet used in Scattering transforms from the Morlet wavelet to the spatially-separable, biorthogonal 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 program.

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

In parallel with our efforts, the original ScatterNet authors rewrote their package to do faster 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 [119].

We have experimentally verified that using a different complex wavelet slightly improves the performance of a ScatterNet front end to a CNN.

Table 3.4: **Hybrid architectures for performance comparison.** Comparison of Morlet-based ScatterNets (Morlet6 and Morlet8) to the DT^CWT-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^CWT architecture, we test different padding schemes and wavelet lengths.

Morlet8	Morlet6	DT ^C WT
Scat $J = 2, K = 8, m = 2$ $y \in \mathbb{R}^{243 \times 8 \times 8}$	Scat $J = 2, K = 6, m = 2$ $y \in \mathbb{R}^{147 \times 8 \times 8}$	Scat $J = 2, K = 6, m = 2$ $y \in \mathbb{R}^{147 \times 8 \times 8}$
conv1, $w \in \mathbb{R}^{C \times 243 \times 3 \times 3}$		conv1, $w \in \mathbb{R}^{C \times 147 \times 3 \times 3}$
	conv2, $w \in \mathbb{R}^{C \times C \times 3 \times 3}$	
	conv3, $w \in \mathbb{R}^{2C \times C \times 3 \times 3}$	
	conv4, $w \in \mathbb{R}^{2C \times 2C \times 3 \times 3}$	
	avg pool, 8×8	
	fc, $w \in \mathbb{R}^{2C \times N_c}$	

Table 3.5: **Performance comparison for a DT^CWT-based vs. a 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 ^C WT	89.8	1.1	66.2	1.1	57.3	2.7

Chapter 4

Visualizing and Improving Scattering Networks

Despite the success of CNNs, deep nets are often criticized for being ‘black box’ methods. Once you train a CNN, you can view the first layer of filters quite easily (see [Figure 1.2a](#)) as they exist in RGB space. Beyond that things get trickier, as the filters have a third, *channel* dimension, typically much larger than its two spatial dimensions. Representing these dimensions with different colours becomes tricky and uninformative, so we must do something else.

A recent paper titled ‘Why Should I Trust You?’ [\[128\]](#) explored the consequences of interpretability on human trust for 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 work has been done in this area. In particular, Zeiler and Fergus [\[26\]](#) design a DeConvNet to visualize what input patterns a filter in a given layer is mostly highly activated by. In [\[129\]](#), Mahendran and Vedaldi learn to invert representations by updating a noisy input via GD until its latent feature vector matches a desired target. Simonyan, Vedaldi, and Zisserman [\[130\]](#) develop saliency maps by projecting gradients back to the input space and measuring where they have the largest magnitude.

We introduced ScatterNets in [section 2.7](#) and looked at making them faster in [chapter 3](#). They have been one of the main successes in applying wavelets to deep learning systems, and are particularly inspiring due to their well-defined properties. They are typically used as unsupervised feature extractors [\[23\], \[25\], \[36\], \[131\]](#) and can outperform CNNs for classification tasks with reduced training set sizes, e.g. in CIFAR-10 and CIFAR-100 (Table 6 from [\[39\]](#) and Table 4 from [\[131\]](#)). They are also near state-of-the-art for Texture Discrimination tasks (Tables 1–3 from [\[99\]](#)). Despite this, there still exists a considerable gap

between Scatternets and CNNs on challenges like CIFAR-10 with the full training set (83% vs. > 90%). Even considering the benefits of ScatterNets, this gap must be addressed.

While ScatterNets have good theoretical foundations and properties [17], it is difficult to understand the second-order scattering. In particular, how useful are these coefficients for training and 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 work done for CNNs, we build a DeScatterNet to visualize what the second-order features are. We also heuristically probe a trained hybrid network (ScatterNet front end + CNN backend) and quantify the importance of the individual features.

4.1 Chapter Layout

We first redefine the operations that form a ScatterNet in [section 4.3](#) before introducing our DeScatterNet in [section 4.4](#), and show how we can use it to examine the layers of ScatterNets (using a similar technique to the CNN visualization in [26]). We use this analysis tool to highlight what patterns a ScatterNet is sensitive to ([section 4.5](#)), showing that they are very different from what their CNN counterparts are sensitive to, and possibly less useful for discriminative tasks.

We then measure the ScatterNet channel saliency by performing occlusion tests on a trained hybrid network ScatterNet-CNN, iteratively switching off individual Scattering channels and measuring the effect this has on the validation accuracy in [section 4.6](#). The results from the occlusion tests strengthen the idea that some of the ScatterNet patterns may not be well suited for deep learning systems.

We use these observations to propose an architectural change to ScatterNets, which have not changed much since their inception in [17], and show that it is possible to get visually more appealing shapes by filtering across the orientations of the ScatterNet. We present this in [section 4.7](#).

4.2 Related Work

Zeiler, Taylor, and Fergus first attempted to use ‘deconvolution’ to improve their learning [132], then later for purely visualization purposes [26]. 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. They invert a convolutional layer 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

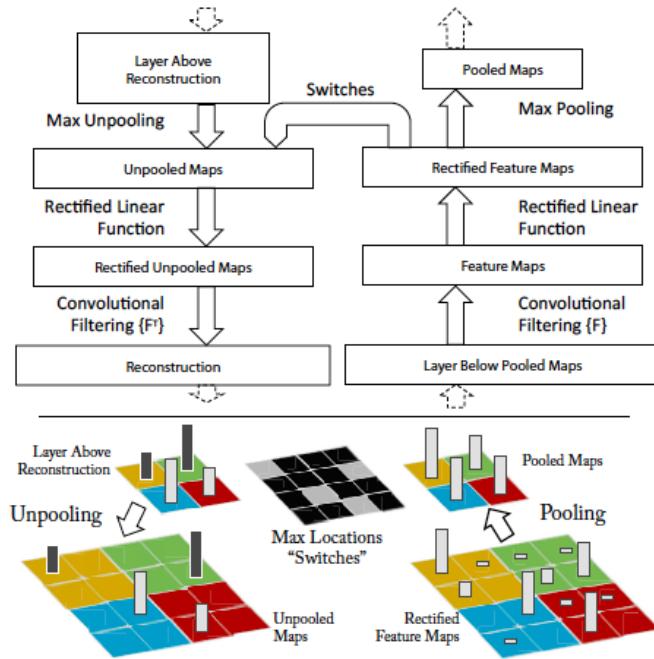


Figure 4.1: Deconvolution network block diagram. Switches are saved alongside pooled features, and the filters used for deconvolution are the transpose of the filters used for a forward pass. Taken from [26].

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. 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.

Mahendran and Vedaldi take a slightly different route on deconvolution networks [129]. They do not store this extra information but instead define a cost function to maximize. This results in visualization images that look very surreal, and can be quite different from the input.

In [68], the authors design an *all convolutional* model, where they replace the max pooling layers commonly seen in CNNs with a convolutional block with stride 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 deconvolution. They call this ‘guided backprop’.

Another interrogation tool commonly used is occlusion or perturbation. In [26], Zeiler and Fergus occlude regions in the input image and measure the impact this has on classification score. In [133], Fong and Vedaldi use gradients to find the minimal mask to apply to the input image that causes misclassification.

4.3 The Scattering Transform

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

We use the DTCWT based ScatterNet introduced in the previous chapter – [Algorithm 3.5](#) as a front end, with $K = 6$ orientations, $J = 2$ scales and $M = 2$ orders.

Consider a single-channel input signal $x(\mathbf{u})$, $\mathbf{u} \in \mathbb{R}^2$. The zeroth-order scatter coefficient is the lowpass output of a J level filter bank:

$$S_0 x(\mathbf{u}) \triangleq x(\mathbf{u}) * \phi_J(\mathbf{u}) \quad (4.3.1)$$

This is approximately invariant to translations of up to 2^J pixels¹. In exchange for gaining invariance, the S_0 coefficients have lost information (contained in the rest of the frequency space). The remaining energy of x is contained within the first-order *wavelet* coefficients:

$$W_1 x(\lambda_1, \mathbf{u}) \triangleq x * \psi_{\lambda_1} \quad (4.3.2)$$

for $\lambda_1 = (j_1, \theta_1)$, $j_1 \in \{1, 2\}$, $\theta_1 = \frac{\pi+2k\pi}{12}$ with $k \in \{0, 1, \dots, 5\}$.

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

$$U_1 x(\lambda_1, \mathbf{u}) \triangleq |x * \psi_{\lambda_1}| = \sqrt{(x * \psi_{\lambda_1}^r)^2 + (x * \psi_{\lambda_1}^i)^2} \quad (4.3.3)$$

The first-order scattering coefficient make U_1 invariant up to the coarsest scale J by averaging it:

$$S_1 x(\lambda_1, \mathbf{u}) \triangleq |x * \psi_{\lambda_1}| * \phi_J \quad (4.3.4)$$

This has $KJ = 6 \times 2 = 12$ output channels for each input channel. Later in this chapter we will want to distinguish between the first and second scale coefficients of the S_1 terms, which we will do by moving the j index to a superscript. I.e., S_1^1 and S_1^2 refer to the set of 6 S_1 terms at the first and second scales.

¹From here on, we drop the \mathbf{u} notation when indexing x , for clarity.

The second-order scattering coefficients are defined only on paths of decreasing frequency (i.e. $J_2 < J_1$) [23]

$$S_2x(\lambda_1, \lambda_2, \mathbf{u}) \triangleq |U_1x * \psi_{\lambda_2}| * \phi_J \quad (4.3.5)$$

Previous work shows that for natural images we get diminishing returns after $m = 2$. Our output is then a stack of these 3 outputs:

$$Sx = \{S_0x, S_1x, S_2x\} \quad (4.3.6)$$

with $1 + 12 + 36 = 49$ channels per input channel.

4.3.1 Scattering Colour Images

A wavelet transform like the DTcWT accepts single-channel input yet 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 colour information, and above which, we combine the three channels.

The second option uses the well known fact that the human eye is far less sensitive to higher spatial frequencies in colour 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 colour ‘blobs’, while the other half were higher frequency, greyscale, oriented wavelets.

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

We combine the three channels by modifying our magnitude operation from (3.6.11) to now be:

$$r_s = \sqrt{x_r^2 + y_r^2 + x_g^2 + y_g^2 + x_b^2 + y_b^2 + b^2 - b} \quad (4.3.7)$$

Where x_r, x_g, x_b are the real parts of the wavelet response for the red, green and blue channels, and y is the corresponding imaginary part. This only affects the S_1 coefficients and the S_2 coefficients then are calculated as per (4.3.5).

An alternative to (4.3.7) is to combine the colours *before* scattering into a luminance channel. However, we choose to use (4.3.7) instead as this has the ability to detect colour edges with constant luminance.

With $J = 2$ the resulting scattering output now has $3 + 12 + 36 = 51$ channels at $1/16$ the spatial input size.

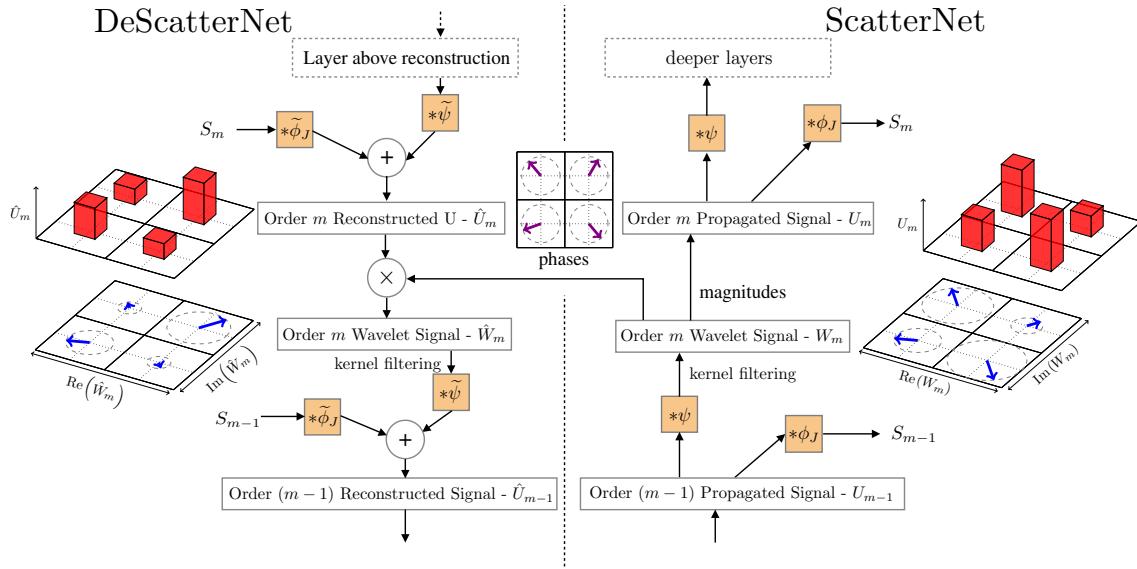


Figure 4.2: **The inverse scattering network.** Comprises a DeScattering layer (left) attached to a Scattering layer (right). We are using the same convention as [26] Figure 1 - i.e., the input signal starts in the bottom right-hand corner, passes forwards through the ScatterNet (up the right half), 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 nonlinearity). For reconstruction, we need to save the discarded phase information and reintroduce it by multiplying it with the reconstructed magnitudes.

4.4 The Inverse Scatter Network

We now introduce our inverse scattering network. This allows us to back-project scattering coefficients to the image space; it is inspired by the DeconvNet used by Zeiler and Fergus in [26] to look into the deeper layers of CNNs. Like the DeConvNet, the inverse scattering network is similar to backpropagating a single strong activation (rather than the usual gradient terms).

We emphasize that instead of thinking about perfectly reconstructing x from $S \in \mathbb{R}^{C \times H' \times W'}$, we want to see what signal/pattern in the input image caused a large activation in each channel. This can then give us a good idea of what each output channel is sensitive to, or what the ScatterNet is ‘extracting’ from the input. Note that we do not use any of the log normalization layers described in [25], [131].

4.4.1 Inverting the Low-Pass Filtering

Going from the U coefficients to the S coefficients in the forward pass involves convolving by a low pass filter ϕ_J , possibly 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 , $\tilde{\phi}_J$ (this is equivalent to the transpose convolution described in [26]).

$$\hat{S}_m x = (S_m x) * \tilde{\phi}_J \quad (4.4.1)$$

We note that interpolation usually involves lowpass smoothing of the signal, so this can all be one operation.

4.4.2 Inverting the Magnitude Operation

In the same vein as [26], we face a difficult task in inverting the nonlinearity 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 phase² to give our recovered W :

$$\hat{W}_m x = (\hat{U}_m x) e^{j\theta_m} \quad (4.4.2)$$

4.4.3 Inverting the Wavelet Decomposition

Using the DT \mathbb{C} WT makes 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 $\tilde{\psi}$:

$$\hat{U}_{m-1} x = \hat{S}_{m-1} x + \hat{W}_m x \quad (4.4.3)$$

$$= (S_{m-1} x) * \tilde{\phi}_J + \sum_{\lambda_m} (\hat{U}_m x) e^{j\theta_m} * \tilde{\psi}_{\lambda_m} \quad (4.4.4)$$

4.4.4 The DT \mathbb{C} WT ScatterNet

The combination of the above three stages can be repeated for higher orders. The resulting DeScatterNet is shown in Figure 4.2.

For the DT \mathbb{C} WT ScatterNet (from Algorithm 3.5), this is the same as finding the *gradient* from the corresponding channels in Z .

²We note that this is equivalent to finding the gradient through the magnitude operation, just as the ‘switches’ from [26] are equivalent to taking the gradient of the max pooling layer.

4.5 Visualization with Inverse Scattering

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 52 channel output vector and mask all but the largest value in the channel we are currently examining and mask all values in the other channels. 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](#) for the first and second-order coefficients. For a given output channel, we show the top 9 activations projected independently to pixel space. We also show the patch of pixels in the input image which cause this large output. As there are 12 S_1 coefficients, we randomly choose 3 orientations from S_1^1 and 3 from S_1^2 . Similarly, there are 36 S_2 coefficients, so we randomly choose 16 of these.

The order 0 and 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 [8]. 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 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 ($\theta_1 = \theta_2$). Others, such as the nine in the top left square (first row, first column), and top right square (first row, fourth column) are more sensitive to checker-board like patterns. 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 [26]. Note that while some of the features are edge-like, we also see higher-order shapes like corners, crosses and curves.

These reconstructions show that higher-order features from ScatterNets vary significantly from those learned in CNNs. In many respects, the features extracted from a CNN like VGGNet look preferable for use as part of a classification system.

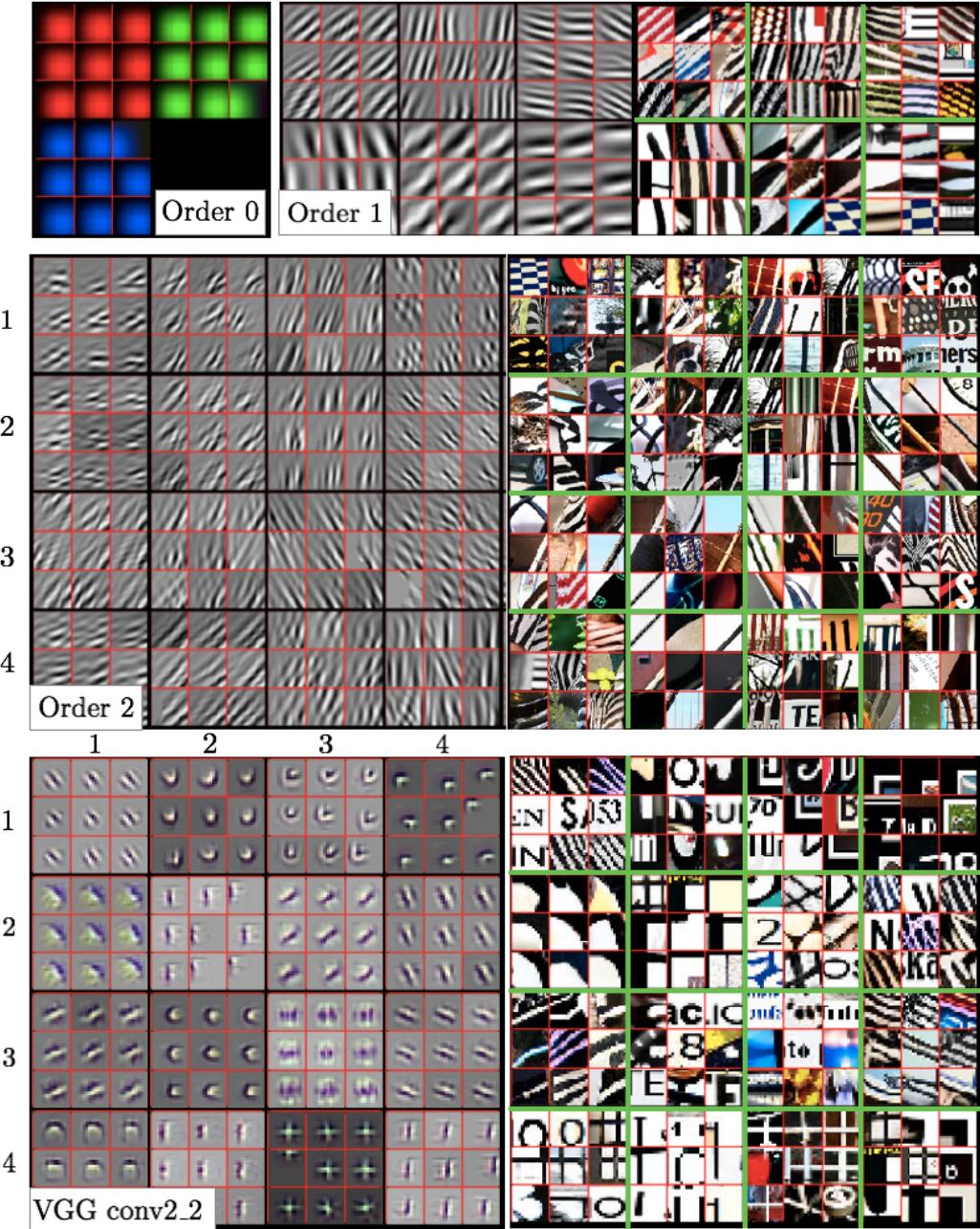


Figure 4.3: Comparison of scattering to convolutional features. Visualization of a random subset of features from S_0 (all 3), S_1 (6 from the 12) and S_2 (16 from the 36) scattering outputs. We record the top 9 activations for the chosen features and project them back to the pixel space. We show them alongside the input image patches which caused the large activations. We also include reconstructions from layer conv2_2 of VGG Net [66] (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 [26]. Image best viewed digitally.

4.6 Channel Saliency

To get another heuristic on the importance of the ScatterNet channels, let us examine the effect on inference scores observed when zeroing out Scattering channels. Zeiler and Fergus [26] and Zhou, Khosla, Lapedriza, *et al.* [134] have done similar studies but over patches of the input image.

We must be careful to occlude with a sensible mask, the S_0x , S_1x and S_2x all have very different probability densities. The authors in [26] occlude with a patch of grey values whereas [134] use random values. Assuming $x \sim \mathcal{N}(0, \sigma^2 I)$ (already a fairly weak assumption), the pdf of S_0x will also be a zero-mean gaussian. However, the distributions of S_1x and S_2x are more complex - the real and imaginary parts of the DT \mathbb{C} WT are sparse but are strongly correlated in energy. Further, after the modulus operation, there is a strong positive bias to all the pdfs until the signal passes through another bandpass filter. Choosing a sensible random mask is therefore difficult, so we instead use a constant mask. Analysis of the datasets showed that zero is very close to the maximum likelihood value for each channel so we occlude channels by simply setting them to zero at every spatial location.

4.6.1 Experiment Setup

We take a network similar to the one from [Table 3.4](#) (2 layers of ScatterNet followed by 4 convolutional layers). Unlike the previous chapter, we use the colour operation described in [subsection 4.3.1](#) so the scattering output has 51 output channels. Further, we set the first convolutional layer after the ScatterNet to have 100 channels for display purposes later.

We train this network on the same 3 datasets - CIFAR-10, CIFAR-100 and Tiny ImageNet, and report the drop in classification scores on the validation set after removing one channel at a time.

We additionally display the weight matrix for the first learned layer of the network trained on Tiny ImageNet. As the scattering output has 51 channels and the first layer of the CNN has 100 channels, this weight matrix has shape: $w \in \mathbb{R}^{100 \times 51 \times 3 \times 3}$ (it is a 3×3 convolution over the 51 channels with 100 filters). This can give us a second perspective on the channel importance by looking at the relative weights of the ScatterNet channels across the 100 filters. We define:

$$A_{c,f}^{rms} = \sqrt{\frac{\sum_{i,j} w[f, c, i, j]^2}{\sum_f \sum_{i,j} w[f, c, i, j]^2}} \quad (4.6.1)$$

This gives us a matrix A^{rms} (for root mean squared) which has columns of unit energy representing the different output channels after conv1. The row values then show how much each scattering channel contributes to each output channel. This is shown in [Figure 4.6](#).

4.6.2 Results

First, we look at Tiny ImageNet in [Figure 4.4](#). Note that when any of the S_0 channels are removed, the validation accuracy drops sharply for all 3 colours. A similar result happens when any of the S_1 channels are zeroed out.

For both the first and second scales of the first-order coefficients, S_1^1 and S_1^2 , there are two channels that seem less important - the second and fifth channels, corresponding to the 45° and 135° wavelets. Often the high-high portion of the first scale coefficients are considered mostly noise, but this does not explain why the 45° and 135° channels for the second scale coefficients are also less important. A possible interesting conclusion to be drawn from this is that the dataset does not have as many important diagonal edges in it as horizontal and vertical edges, and the network has learned this difference in importance.

To test this, we retrain the network but this time rotate the input images randomly 30° clockwise or anti-clockwise in both training and validation. We then rerun the occlusion experiment for all channels and plot the resulting changes in [Figure 4.4b](#). Interestingly, for this network, the 45° and 135° wavelets for S_1^2 are now the most important of the 6, which validates our assumption. The corresponding wavelets for S_1^1 have become more important, but it is likely that they remain less salient because of the effects of the higher bandwidth for the diagonal wavelets.

Comparatively, the S_2 channels have little effect on the classification score when individually masked. The four largest drops in accuracy for S_2 are happening when $\theta_1 = \theta_2 \in \{15^\circ, 75^\circ, 105^\circ, 165^\circ\}$. When we drop channels in S_2 with $\theta_1 \neq \theta_2$, the network performance is not affected very much. Recall that $\theta_1 \neq \theta_2$ corresponds to the ripple-like patterns in [Figure 4.3](#).

We include the same occlusion results for the two CIFAR datasets in [Figure 4.5](#) for completeness, although the insight gained here is the same - the S_2 coefficients are the least important. One notable difference to [Figure 4.4](#) is in the S_1^2 coefficients which have reduced importance in CIFAR. As CIFAR has a smaller input spatial size than Tiny ImageNet this comes as no surprise.

[Figure 4.6](#) shows the size of $A_{c,f}^{rms}$. The columns of the matrix all have unit-norm, so each entry represents how much relative energy comes from each scattering channel (brighter values indicating more energy). Looking across the rows we see how often a scattering output is used for the CNN next layer. Most of the filters are heavily dependent on S_0 , many are dependent on S_1 and only a few take information from S_2 .

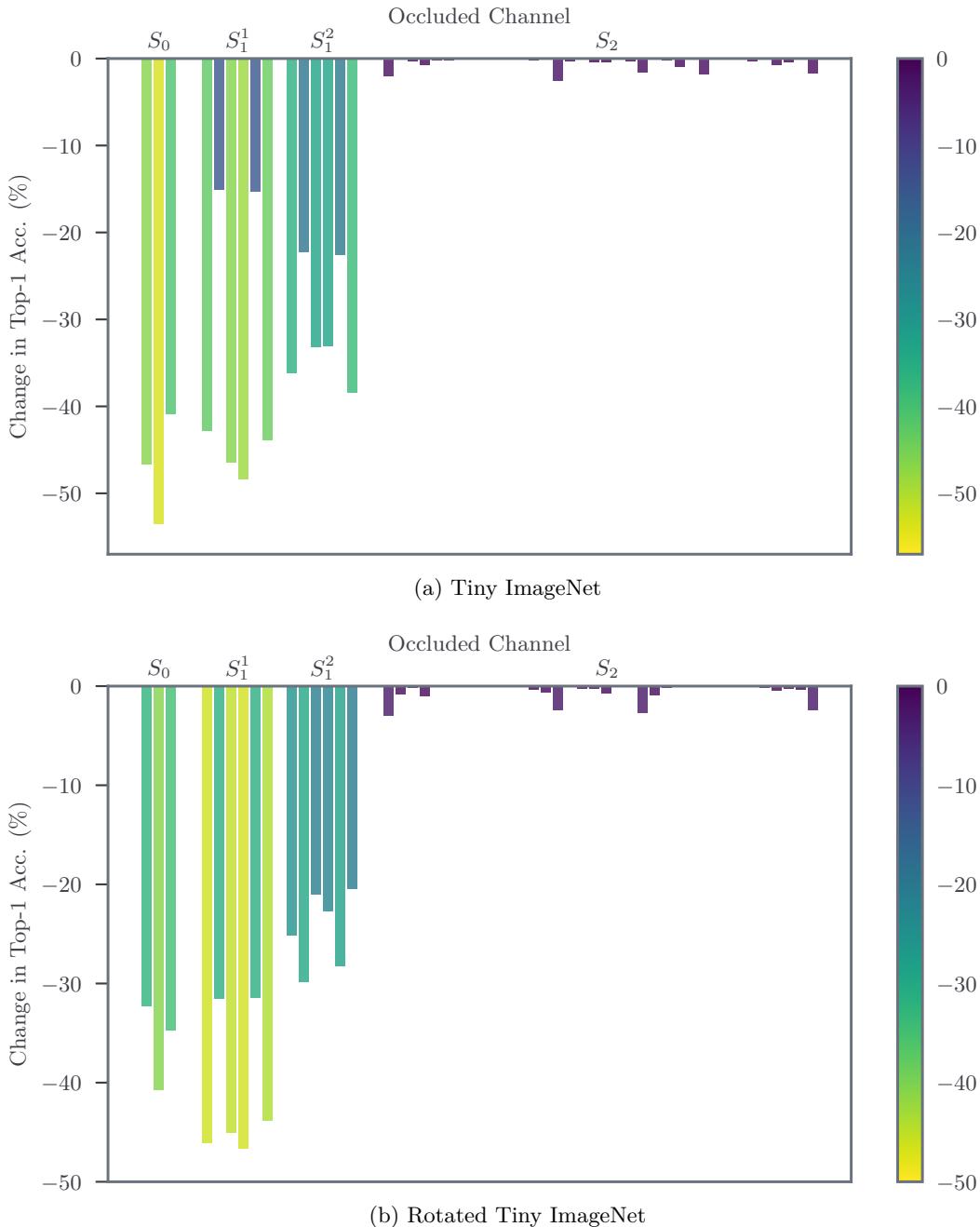


Figure 4.4: Tiny ImageNet changes in accuracy from channel occlusion. Numbers reported are the drop in final classification accuracy when a channel is set to zero. The bars are coloured relative to their magnitude to aid seeing the differences for the S_1 coefficients. (a) When any of the lowpass channels S_0 are removed, the classification accuracy drops sharply, note that the middle channel, corresponding to green, is the most important of the three colours. The first scale, first-order scattering coefficients S_1^1 are slightly more important than the second scale coefficients. The 36 S_2 coefficients have little individual effect on the validation score when removed. (b) The same network trained with input samples rotated by $\pm 30^\circ$. In (a) the second and fifth orientations for both S_1^1 and S_1^2 , corresponding to the 45° and 135° wavelets, are comparatively less important than other orientations at the same scale. This suggests that perhaps the dataset does not have much diagonal information. When rotated this trend changes and the diagonal wavelets at both scales become more important.

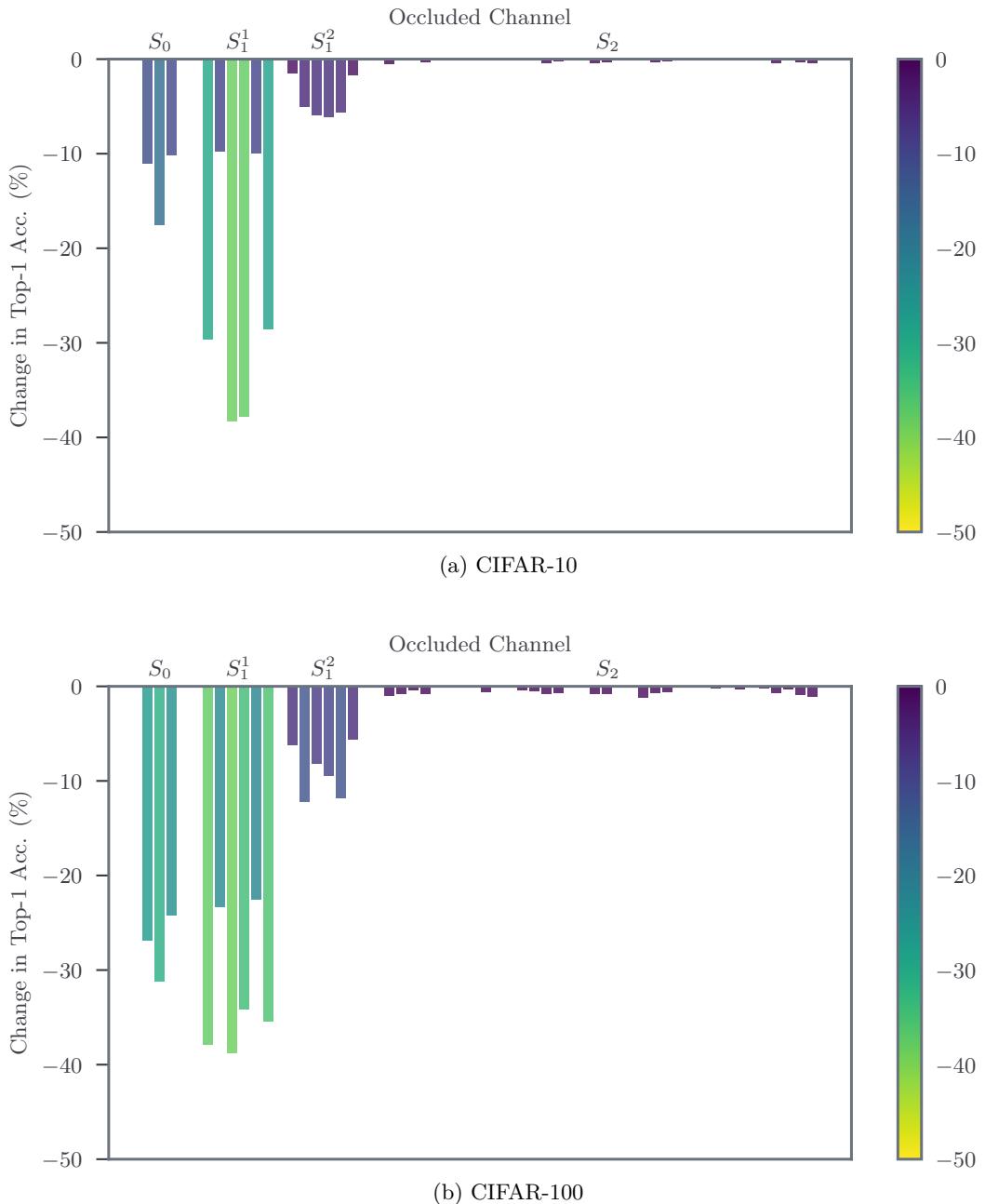


Figure 4.5: CIFAR changes in accuracy from channel occlusion. Numbers reported are the drop in final classification accuracy when a channel is set to zero. The bars are coloured relative to their magnitude to aid seeing the differences for the S_1 coefficients. Unlike Figure 4.4 the S_1^2 coefficients are less important. CIFAR has a smaller image size than Tiny ImageNet so this is not surprising.

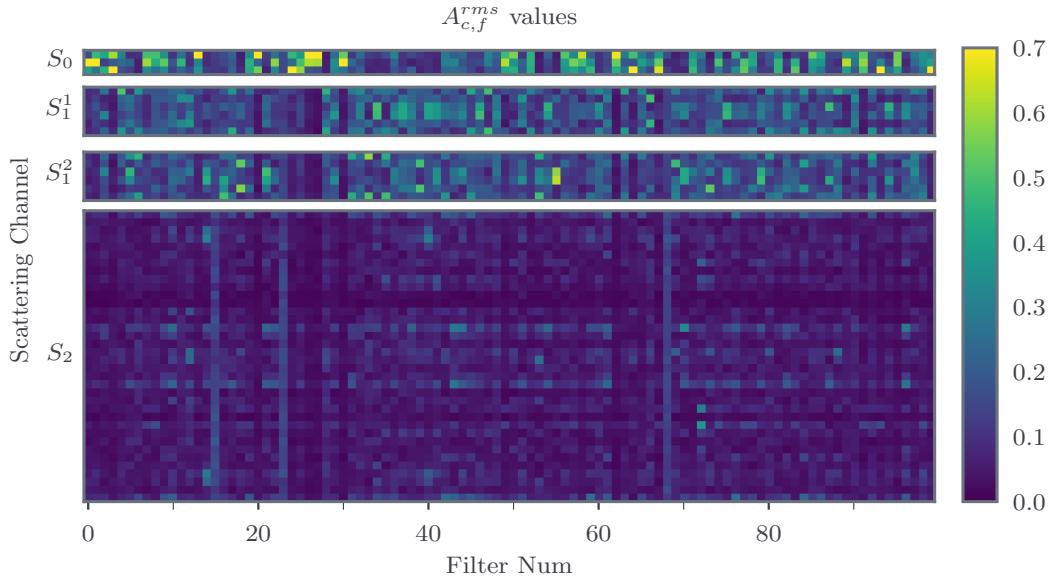


Figure 4.6: **Channel weights for first learned layer in a hybrid ScatterNet-CNN.** A visualization of the matrix A^{rms} from (4.6.1) for a network trained on Tiny ImageNet. The columns of the matrix all have unit norm and represent how much relative energy comes from each scattering output channel. Most of the filters are heavily dependent on S_0 , many are dependent on S_1 and only a few take information from S_2 .

4.7 Corners, Crosses and Curves

As a final part of this chapter, we would like to highlight some of the filters possible by making small modifications to the ScatterNet design. The visualizations shown here are mostly inspirational, as we did not see any marked improvement in using them as a fixed front end for the ScatterNet system. However, they are the basis for the next chapter of work in adding learning in between Scattering layers.

Sifre and Mallat introduced the idea of a ‘Roto-Translation’ ScatterNet in [99]. Invariance to rotation could be made by applying averaging (and bandpass) filters across the K orientations from the wavelet transform *before* applying the complex modulus. Let us call the averaging and bandpass filters they use $h \in \mathbb{C}^K$. We can think of this stage as stacking the K outputs of a complex wavelet transform on top of each other and convolving a filter h_α over all spatial locations of the wavelet coefficients $W_m x$. Let us call the output from filtering across the channel dimension V_m :

$$V_m x(j, \alpha, \mathbf{u}) = \sum_{\theta} W_m x(j, \theta, \mathbf{u}) h_{j, \alpha}(\theta) \quad (4.7.1)$$

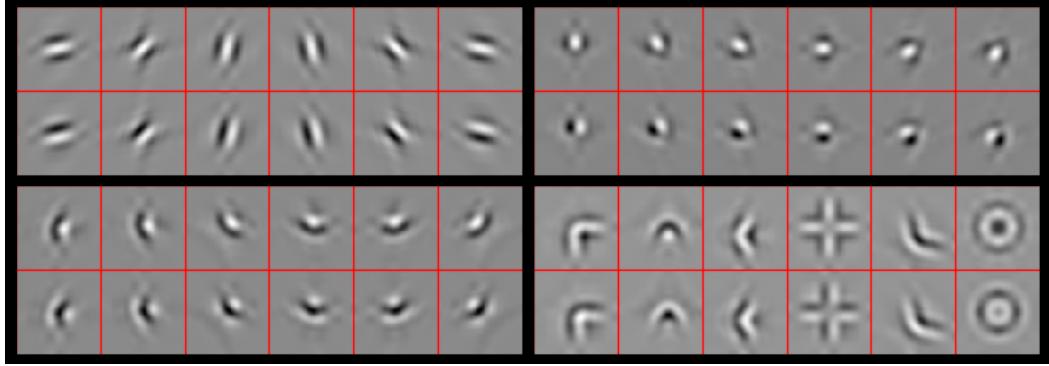


Figure 4.7: Shapes possible by filtering across the wavelet orientations with complex coefficients. All shapes are shown in pairs: the top image is reconstructed from a purely real output, and the bottom image from a purely imaginary output. These ‘real’ and ‘imaginary’ shapes are nearly orthogonal in 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 $12 \times 1 \times 1$ Morlet Wavelet, similar to what was done in the ‘Roto-Translation’ ScatterNet described in [25], [99]. Bottom left - reconstructions from U'_1 made with a more general $12 \times 1 \times 1$ filter, described in Equation 4.7.2. Bottom right - some reconstructions possible by filtering a general $12 \times 3 \times 3$ filter.

We present a variation on this idea by filtering with a more general $h \in \mathbb{C}^{12 \times H \times W}$. We use 12 channels rather than 6, as we use the $K = 6$ 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 4.7 shows some reconstructions from these V coefficients with hand-designed h 's using the above DeScatterNet. All shapes are shown in real and imaginary Hilbert-like pairs; the top row of images in each quadrant are reconstructed from a purely real V , while the bottom row are reconstructed from a purely imaginary V .

In the top left, we display the 6 wavelet filters for reference. In the top right of the figure we see some of the shapes made by using the h 's from the Roto-Translation ScatterNet [25], [99]. The bottom left is where we present some of our novel kernels. These are simple corner-like shapes made by filtering with $h \in \mathbb{C}^{12 \times 1 \times 1}$ where h is set to

$$h = [1, j, j, 1, 0, 0, 0, 0, 0, 0, 0, 0] \quad (4.7.2)$$

The six orientations are made by rolling the coefficients in h along one sample (i.e. $[0, 1, j, j, 1, 0, \dots]$, $[0, 0, 1, j, j, 1, 0, \dots]$, $[0, 0, 0, 1, j, j, 1, 0, \dots] \dots$). The six conjugate orientations then make up the final 6 orientations ($[0, 0, 0, 0, 0, 1, j, j, 1, 0, 0]$, $[0, 0, 0, 0, 0, 0, 0, 1, j, j, 1, 0]$, etc.). Coefficients roll back around (like circular convolution) when they reach the end. The canonical filter is then $[1, j, j, 1]$ where the 90° phase offset of the middle two weights from the outer two allows for nicely continuous ridges of similar intensity around the centre of the corner.

Finally, in the bottom right we see shapes made by $h \in \mathbb{C}^{12 \times 3 \times 3}$. Note that with the exception of the ring-like shape which has 12 non-zero coefficients, all of these shapes were reconstructed with h '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.

The shapes in the bottom row of the bottom right quadrant appear very similar to those in the top row but have nearly zero inner product. This shows one level of invariance of this system, as after taking the complex magnitude, both the top and the bottom shape will activate the network with the same strength. In comparison, for the purely real filters of a CNN if the top shape would cause a large output then the bottom shape would cause near 0 activity.

4.8 Conclusion

This chapter presents a way to investigate what the higher orders of a ScatterNet are responding to - the DeScatterNet described in [section 4.4](#). Using this, we have shown that the second-order of a ScatterNet responds strongly to patterns that are very different 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.

Additionally, we performed occlusion tests to heuristically measure the importance of the individual scattering channels when a ScatterNet is used as a front-end to a CNN. The results of this test reaffirmed the suspicions raised from the visualizations. In particular, many of the second-order Scattering coefficients may not be very useful in a deep classifier. Those that were more useful were typically when the second-order wavelet had the same orientation as the first. We also noted that diagonal orientations appear less important than horizontal and vertical ones, even at coarser scales. This appears to be an artefact of the CIFAR and Tiny ImageNet datasets, as rotating the images by 30° made diagonal edges more important.

Finally, we demonstrated the possible shapes attainable when we filter across orientations with complex mixing coefficients. We believe that this mixing is a key step in the development of improved ScatterNets and wavelets in deep learning systems.

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 [chapter 4](#) 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 combining ScatterNets with CNNs in hybrid networks [34], [39] 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 each 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 closely to 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.

5.1 Chapter Layout

In [section 5.2](#) we discuss related work before briefly reviewing the convolutional layer and scattering notation we will use in [section 5.3](#). We introduce our learnable scattering layers in [section 5.4](#) and describe their properties and implementation in [section 5.5](#). Finally, we

present some experiments we have run in section 5.6, section 5.7 and section 5.8 before drawing conclusions about how these new ideas might improve neural networks in the future.

5.2 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 [135] 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[136]. 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” [137]. This paper builds a first stage with a small 3×3 kernel filled with zeros, and randomly insert ± 1 in several locations, 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” [27]. 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 [137], 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.

5.3 Recap of Useful Terms

5.3.1 Convolutional Layers

Let the output of a CNN at layer l be $x^{(l)}(c, \mathbf{u})$, $c \in \{0, \dots, C_l - 1\}$, $\mathbf{u} \in \mathbb{R}^2$ 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. Recall from (2.4.5) and (2.4.6) that a convolutional layer in a standard CNN is defined by the two operations:

$$y^{(l+1)}(f, \mathbf{u}) = \sum_{c=0}^{C_l-1} x^{(l)}(c, \mathbf{u}) * h_f^{(l)}(c, \mathbf{u}) \quad (5.3.1)$$

$$x^{(l+1)}(f, \mathbf{u}) = \sigma\left(y^{(l+1)}(f, \mathbf{u})\right) \quad (5.3.2)$$

where $h^{(l)}(f, c, \mathbf{u})$ is the f th filter of the l th layer with C_l different point spread functions, and $f \in \{0, \dots, C_{l+1} - 1\}$. σ is a nonlinearity 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.

5.3.2 Wavelet Transforms

Recall from (2.6.26) that:

$$\mathcal{W}x(c, \mathbf{u}) = \{x(c, \mathbf{u}) * \phi_J(\mathbf{u}), x(c, \mathbf{u}) * \psi_\lambda(\mathbf{u})\}_\lambda \quad (5.3.3)$$

Where ψ_λ is a mother wavelet dilated by 2^j , $1 \leq j \leq J$ and rotated by $\theta = \frac{\pi + 2k\pi}{12}$, $0 \leq k < 6$:

$$\psi_{j,\theta}(\mathbf{u}) = 2^{-j}\psi(2^{-j}r_{-\theta}\mathbf{u}) \quad (5.3.4)$$

Define the set of all possible λ s as Λ whose size is $|\Lambda| = JK$.

5.3.3 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:

$$\widetilde{\mathcal{W}}x(c, \mathbf{u}) = \{x(c, \mathbf{u}) * \phi_J(\mathbf{u}), |x(c, \mathbf{u}) * \psi_\lambda(\mathbf{u})|\}_{\lambda \in \Lambda} \quad (5.3.5)$$

The modulus terms are called $U[\lambda]x = |x * \psi_\lambda|$, and the scattering terms are $S[\lambda]x = U[\lambda]x * \phi_J(\mathbf{u})$. In this chapter, we do not use the colour ScatterNet introduced in subsection 4.3.1. Instead, we scatter each colour channel independently.

5.4 Locally Invariant Layer

We propose to mix the terms at the output of each wavelet modulus propagator $\tilde{\mathcal{W}}$. The second term in $\tilde{\mathcal{W}}$, 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 these 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 (5.3.5), and let the input to it be $x^{(l)}$. Our proposed output is then:

$$y^{(l+1)}(f, \mathbf{u}) = \sum_{c=0}^{C-1} b_f(c) \left(x^{(l)}(c, \mathbf{u}) * \phi_J(\mathbf{u}) \right) + \sum_{\lambda \in \Lambda} \sum_{c=0}^{C-1} a_{f,\lambda}(c) \left| x^{(l)}(c, \mathbf{u}) * \psi_\lambda(\mathbf{u}) \right| \quad (5.4.1)$$

$$= \sum_{c=0}^{C-1} \left(b_f(c) \left(x^{(l)}(c, \mathbf{u}) * \phi_J(\mathbf{u}) \right) + \sum_{\lambda \in \Lambda} a_{f,\lambda}(c) \left| x^{(l)}(c, \mathbf{u}) * \psi_\lambda(\mathbf{u}) \right| \right) \quad (5.4.2)$$

Note that an input to the wavelet modulus propagator $\tilde{\mathcal{W}}$ 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 $q \in \mathbb{Z}$ such that:

$$z^{(l+1)}(q, \mathbf{u}) = \begin{cases} x^{(l)}(c, \mathbf{u}) * \phi_J(\mathbf{u}) & \text{if } 0 \leq q < C \\ |x^{(l)}(c, \mathbf{u}) * \psi_\lambda(\mathbf{u})| & \text{if } C \leq q < (JK+1)C \end{cases} \quad (5.4.3)$$

i.e. the lowpass channels are the first C channels of z , the modulus of the 15° ($k=0$) wavelet coefficients with $j=1$ are the next C channels, then the modulus coefficients with $k=1$ and $j=1$ are the third C channels, and so on. This is similar to the view of a ScatterNet we introduced in [Algorithm 3.5](#), but restricted to a single order.

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

$$\tilde{a}_f(q) = \begin{cases} b_f(c) & \text{if } 0 \leq q < C \\ a_{f,\lambda}(c) & \text{if } C \leq q < (JK+1)C \end{cases} \quad (5.4.4)$$

we can then use (5.4.3) and (5.4.4) to simplify (5.4.2), giving:

$$y^{(l+1)}(f, \mathbf{u}) = \sum_{q=0}^{(JK+1)C-1} z^{(l+1)}(q, \mathbf{u}) \tilde{a}_f(q) \quad (5.4.5)$$

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

$$Y^{(l+1)}(\mathbf{u}) = AZ^{(l+1)}(\mathbf{u}) \quad (5.4.6)$$

This equation now looks very similar to the standard convolutional layer from (5.3.1), except we have replaced the previous layer’s x with intermediate coefficients z with $|Q|=$

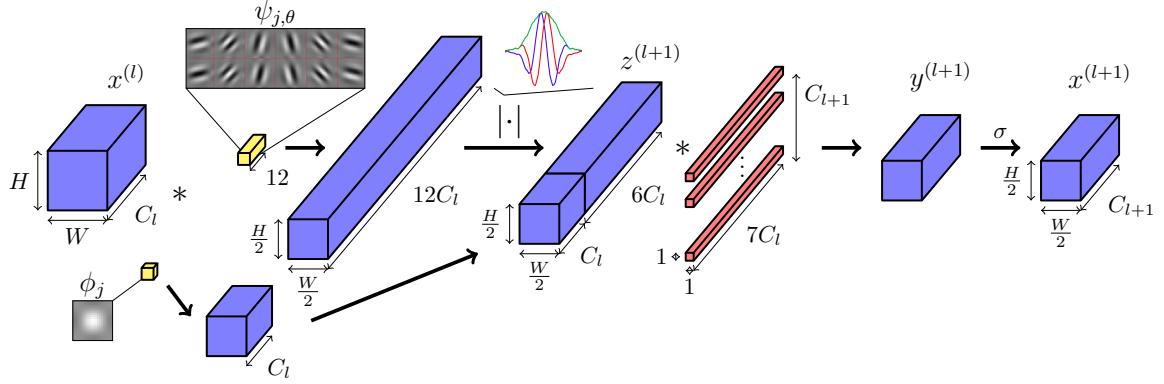


Figure 5.1: **Block Diagram of Proposed Invariant Layer for $j = J = 1$.** Activations are shaded blue, fixed parameters yellow and learned parameters red. Input $x^{(l)} \in \mathbb{R}^{C_l \times H \times W}$ is filtered by real and imaginary oriented wavelets and a lowpass filter and is downsampled. 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) - taking the envelope of the components oscillating in quadrature. These are concatenated with the lowpass activations to give $z^{(l+1)}$. The resulting activations z are mixed across the channel dimension (shown by the convolution with the red filters) 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.

$(JK + 1)C$ channels and the convolutions of (5.3.1) have been replaced by a matrix multiply (which can also be seen as a 1×1 convolutional layer). We can then apply (5.3.2) to (5.4.5) to get the next layer's output:

$$x^{(l+1)}(f, \mathbf{u}) = \sigma(y^{(l+1)}(f, \mathbf{u})) \quad (5.4.7)$$

Figure 5.1 shows a block diagram for this process.

5.4.1 Properties

5.4.1.1 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 [23], [39]. If $C_{l+1} = (JK + 1)C_l$ and A is the identity matrix $I_{C_{l+1}}$, there is no mixing and $y^{(l+1)} = \tilde{\mathcal{W}}x$.

Further, if $\sigma = \text{ReLU}$ as is commonly the case in training CNNs, σ 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 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 $x^{(l+1)} = \widetilde{\mathcal{W}}x$.

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). We can repeat the same process for the next layer (recall the recursive scattering relation we saw in (2.7.12)), 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.

5.4.1.2 Flexibility 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.

5.4.1.3 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 [27] 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 (2.7.23) holds.

5.5 Implementation Details

Again, we use the DTCWT [18] for our wavelet filters due to their fast implementation with separable convolutions which we discuss more in subsection 5.5.3. 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:

$$(((x * \phi_1) \downarrow 2) * \psi_{1,\theta}) \downarrow 2 = (x * \psi_{2,\theta}) \downarrow 4 \quad (5.5.1)$$

5.5.1 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:

$$\#params = (JK + 1)C_l C_{l+1} = 7C_l C_{l+1} \quad (5.5.2)$$

The spatial support of the wavelet filters is typically 5×5 pixels or more, and we have reduced the number of parameters to fewer than $3 \times 3 = 9$ per filter, while producing filters that are significantly larger than this.

5.5.2 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.

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.

5.5.3 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 $2L(1 - 2^{-2J})$ multiplies per input pixel for a J scale decomposition. A DTcWT has 4 DWTs for a 2-D input, so its cost is $8L(1 - 2^{-2J})$, 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 1/4 the

Algorithm 5.1 Locally Invariant Convolutional Layer forward and backward passes

```

1: procedure INVLAYER.FORWARD( $x, A$ )
2:    $yl, yh \leftarrow \text{DTCWT.Forward}(x^l, \text{nlevels} = 1)$ 
3:    $U \leftarrow \text{MAG\_SMOOTH.Forward}(yh)$                                  $\triangleright$  See Algorithm B.3
4:    $yl \leftarrow \text{AVGPOOL2x2}(yl)$                                           $\triangleright$  Downsample lowpass to match U size
5:    $Z \leftarrow \text{CONCATENATE}(yl, U)$                                           $\triangleright$  Concatenate along the channel dim
6:    $Y \leftarrow AZ$                                                         $\triangleright$  Mix
7:   save  $A, Z$                                                   $\triangleright$  For the backwards pass
8:   return  $Y$ 
9: end procedure

1: procedure INVLAYER.BACKWARD( $\frac{\partial L}{\partial Y}$ )
2:   load  $A, Z$ 
3:    $\frac{\partial L}{\partial A} \leftarrow \frac{\partial L}{\partial Y} Z^T$                                  $\triangleright$  Calculate update gradient
4:    $\Delta Z \leftarrow A^T \frac{\partial L}{\partial Y}$ 
5:    $\Delta yl, \Delta U \leftarrow \text{UNSTACK}(\Delta Z)$ 
6:    $\Delta yl \leftarrow \text{AVGPOOL2x2.Backward}(\Delta yl)$ 
7:    $\Delta yh \leftarrow \text{MAG\_SMOOTH.Backward}(\Delta U)$ 
8:    $\frac{\partial L}{\partial x} \leftarrow \text{DTCWT.Backward}(\Delta yl, \Delta yh)$             $\triangleright$  Calculate passthrough gradient
9:   return  $\frac{\partial L}{\partial x}, \frac{\partial L}{\partial A}$ 
10: end procedure

```

spatial area after the first scale and 1/16 the spatial area after the second scale. Restricting ourselves to $J = 1$ as we mentioned in section 5.5, the computational cost is then:

$$\underbrace{\frac{7}{4}C_{l+1}}_{\text{mixing}} + \underbrace{36}_{\text{DTCWT}} \quad \text{multiplies per input pixel} \quad (5.5.3)$$

In most CNNs, C_{l+1} is several dozen if not several hundred, which makes (5.5.3) significantly smaller than $L^2 C_{l+1} = 9C_{l+1}$ multiplies for 3×3 convolutions.

5.5.4 Forward and Backward Algorithm

There are two layer-hyperparameters to choose:

- 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 5.1.

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) Reference Arch with 3×3 convolutions		(b) Invariant Architecture	
Activation Size	Layer Name + Info	Activation Size	Layer Name + Info
$1 \times 28 \times 28$		$1 \times 28 \times 28$	
$7 \times 28 \times 28$	conv1, $w \in \mathbb{R}^{7 \times 1 \times 3 \times 3}$	$7 \times 14 \times 14$	inv1, $A \in \mathbb{R}^{7 \times 7}$
$7 \times 14 \times 14$	maxpool1, 2×2	$49 \times 7 \times 7$	inv2, $A \in \mathbb{R}^{49 \times 49}$
$49 \times 14 \times 14$	conv2 $w \in \mathbb{R}^{49 \times 7 \times 3 \times 3}$	2401×1	unravel
$49 \times 7 \times 7$	maxpool2, 2×2	10×1	project, $w \in \mathbb{R}^{2401 \times 10}$
$49 \times 7 \times 7$	unravel	10×1	reshuffle, $w \in \mathbb{R}^{10 \times 10}$
2401×1	project, $w \in \mathbb{R}^{2401 \times 10}$	10×1	
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 5.6.1](#). 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}

5.6 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

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 the learning rate, lr, was the most important hyperparameter to choose correctly, and had the largest impact on the performance.

Architecture	$\theta = \{\text{lr, mom, wd, a}\}$	accuracy	
		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

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 [55] and only vary the gain value a :

$$A_{ij} \sim U \left[-a \sqrt{\frac{6}{(C_l + C_{l+1})HW}}, a \sqrt{\frac{6}{(C_l + C_{l+1})HW}} \right] \quad (5.6.1)$$

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 [138] 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 [139] 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.

5.6.1 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.

We posit that this is due to the centred nature of the wavelet bases that were used to generate the z and later the y coefficients. As they are all centred at roughly the same location (the phase of the complex wavelet allows for some small spatial separation) 1×1

convolutions may not be capable of building richer shapes by separating the wavelet centres. This effect was seen in the previous chapter when we presented some possible new shapes in [Figure 4.7](#). The bottom right quadrant were shapes made from mixing wavelet coefficients in a 3×3 area, and these were much richer than those attainable by only mixing in a 1×1 area¹

To get spatial separation, we must replace the 1×1 mixing kernel from [\(5.4.5\)](#) ($\tilde{\alpha}_f(q)$) with a 3×3 filter $g_f(q, \mathbf{u})$. We also change the multiply for convolution, giving us:

$$y^{(l+1)}(f, \mathbf{u}) = \sum_{q \in Q} z^{(l+1)}(q, \mathbf{u}) * g_f(q, \mathbf{u}) \quad (5.6.2)$$

If all of the parameters of $g_f(q, \mathbf{u})$ are learned, then the number of parameters in [\(5.5.2\)](#) increases from $7C_l C_{l+1}$ to $63C_l C_{l+1}$. This is quite a lot of parameters for a single layer, and much more than the $9C_l C_{l+1}$ of a standard 3×3 convolution, or even the $25C_l C_{l+1}$ of a larger 5×5 convolution.

However, we can still get spatial separation without having to learn *all* of the coefficients, if we factor g as:

$$g_f(q, \mathbf{u}) = \tilde{\alpha}_f(q) \alpha_f(q, \mathbf{u}) \quad (5.6.3)$$

where $\alpha_f(q, \mathbf{u})$ is an introduced *fixed* kernel, designed to allow mixing of wavelets from neighbouring spatial locations.

We test a range of possible α 's each with varying complexity/overhead:

- (a) **Random Shifts:** 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 setting $\alpha_f(q, \mathbf{u})$ to be a 3×3 kernel with a single 1 and eight 0's.
- (b) **Shifted Gaussians:** Instead of shifting impulses as in the previous option, we can shift a Gaussian kernel by $\{-1, 0, 1\}$ pixel horizontally and $\{-1, 0, 1\}$ pixel vertically, making a smoother filter than in (a).
- (c) **Random Kernels:** 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.
- (d) **DCT Kernels:** We can learn coefficients for the top three 3×3 discrete cosine transform (DCT) coefficients² and sum them to get g . This is equivalent to having three different α 's, and learning three sets of $\tilde{\alpha}_f(q)$. This is a step between the parameterless kernels of (a) - (c) and the nine-fold parameter increase from learning all of g .

¹This comparison is only a guide, however, as in this chapter we are mixing coefficients after taking the modulus, whereas [Figure 4.7](#) were generated by mixing with complex gains *before* the modulus.

²The top three DCT coefficients are the constant, the horizontal and the vertical filters.

5.6.2 Expanded MNIST experiments

We test if the expansions from the previous section improve the MNIST performance. 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](#).

The top two rows are the old results from [Table 5.3](#), the next four rows are the expanded kernels described in the previous section, options (a) - (d). We also include tests for a fully learned g kernel (despite the expense) in the seventh row under ‘Learned 3×3 ’. To compare this option to an equivalent convolutional system, we modify the convolutional architecture from [Table 5.1](#) to use 5×5 convolutions and $C_1 = 10$, $C_2 = 100$ channels and include these results in the final row of [Table 5.4](#) under ‘Wide Convolutional’. For ease of comparison, we list the computational and parameter cost associated with each option alongside the accuracy results.

As expected, adding in random shifts significantly helps the invariant layer for all but the random kernel (c) option. Two systems of note are the shifted impulse (a) system and the fully learned 3×3 kernel 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.

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.

Architecture	$\theta = \{\text{lr, mom, wd, a}\}$	cost		accuracy	
		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 DCT coeffs (d)	$\{1.0, 0.0, 10^{-5}, 1.0\}$	$\frac{7}{3}$	$\frac{7}{4}$	98.1	0.37
Learned 3×3 kernel	$\{0.1, 0.5, 10^{-4}, 1.0\}$	7	$\frac{7}{4}$	99.0	0.12
Wide Convolutional	$\{0.32, 0.5, 10^{-5}, 1.5\}$	7	7	98.7	0.25

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[66] 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		(b) Tiny ImageNet Architecture	
Activation Size	Layer Name + Info	Activation Size	Layer Name + Info
$3 \times 32 \times 32$		$3 \times 64 \times 64$	
$C \times 32 \times 32$	convA, $w \in \mathbb{R}^{C \times 3 \times 3 \times 3}$	$C \times 64 \times 64$	convA, $w \in \mathbb{R}^{C \times 3 \times 3 \times 3}$
$C \times 32 \times 32$	convB, $w \in \mathbb{R}^{C \times C \times 3 \times 3}$	$C \times 64 \times 64$	convB, $w \in \mathbb{R}^{C \times C \times 3 \times 3}$
$C \times 32 \times 32$	pool1, max pooling 2×2	$C \times 64 \times 64$	pool1, max pooling 2×2
$C \times 16 \times 16$	convC, $w \in \mathbb{R}^{2C \times C \times 3 \times 3}$	$C \times 32 \times 32$	convC, $w \in \mathbb{R}^{2C \times C \times 3 \times 3}$
$2C \times 16 \times 16$	convD, $w \in \mathbb{R}^{2C \times 2C \times 3 \times 3}$	$2C \times 32 \times 32$	convD, $w \in \mathbb{R}^{2C \times 2C \times 3 \times 3}$
$2C \times 16 \times 16$	pool2, max pooling 2×2	$2C \times 32 \times 32$	pool2, max pooling 2×2
$2C \times 8 \times 8$	convE, $w \in \mathbb{R}^{4C \times 2C \times 3 \times 3}$	$2C \times 16 \times 16$	convE, $w \in \mathbb{R}^{4C \times 2C \times 3 \times 3}$
$4C \times 8 \times 8$	convF, $w \in \mathbb{R}^{4C \times 4C \times 3 \times 3}$	$4C \times 16 \times 16$	convF, $w \in \mathbb{R}^{4C \times 4C \times 3 \times 3}$
$4C \times 8 \times 8$	avg, 8×8 average pooling	$4C \times 16 \times 16$	pool3, max pooling 2×2
$4C \times 1 \times 1$	fc1, fully connected layer	$4C \times 8 \times 8$	convG, $w \in \mathbb{R}^{8C \times 4C \times 3 \times 3}$
$10 \times 1, 100 \times 1$		$8C \times 8 \times 8$	convH, $w \in \mathbb{R}^{8C \times 8C \times 3 \times 3}$
		$8C \times 1$	avg, 8×8 average pooling
		200×1	fc1, fully connected layer

5.7 Ablation Experiments 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 Table 5.5a. The initial number of channels C we use is 64. Despite this simple design, this reference architecture achieves near state-of-the-art performance for the three datasets (92.6% on CIFAR-10, 72.0% on CIFAR-100 and 59.3% on Tiny ImageNet).

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 only swapping 1 or 2 layers.

The invariant layer naturally downsamples the input. If we need to keep the output resolution the same as the input (i.e. ‘convA’, ‘convF’), we bilinearly interpolate the output of the invariant layer. If we swap out a layer immediately *before* a pooling layer (i.e. ‘convB’, ‘convD’) then we remove the pooling and do not upsample. Similarly, if we swap out a layer immediately *after* a pooling layer (i.e. ‘convC’, ‘convE’) then we remove the pooling layer and do the downsampling with the invariant layer.

Figure 5.2 and Figure 5.3 reports the top-1 classification accuracies for CIFAR-10, CIFAR-100 and Tiny ImageNet. In the table, ‘invX’ means that the ‘convX’ layer from Table 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.

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).

We see improvements for all three datasets for many possible locations for the invariant layer. The best permutations happen 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 when we remove the first max pooling layer and use the natural downsampling of the invariant layer (at either the ‘convB’ or ‘convC’ position). This also worked for the second pooling layer (at the ‘convD’ position), but having an invariant layer at both of these positions (i.e. ‘convBD’ and ‘convCE’) performed slightly worse than using only one.

We recall that the magnitude operation in the ScatterNet effectively demodulates the energy from higher spatial frequencies to lower ones. This may explain why the best place for learnable scattering layers are at positions where you want to downsample in a network.

We also tested the variations from subsection 5.6.1 that added spatial offsets to the gain layer by a fixed kernel and saw similar, if not slightly worse, results than those in Figure 5.2 and Figure 5.3. We did not have time to explore fully as to why this was the case, but we believe that the convolutional backend in the reference architecture (Table 5.5) provided the necessary spatial separation to build rich shapes, something that we did not have in the MNIST experiments.

5.8 A New Hybrid ScatterNet

In the previous section, we examined how the locally invariant layer performs when directly swapped for a convolutional layer in a CNN-based architecture. In this section, we look at how it performs in a hybrid ScatterNet-like network [39]. Recall that a ScatterNet

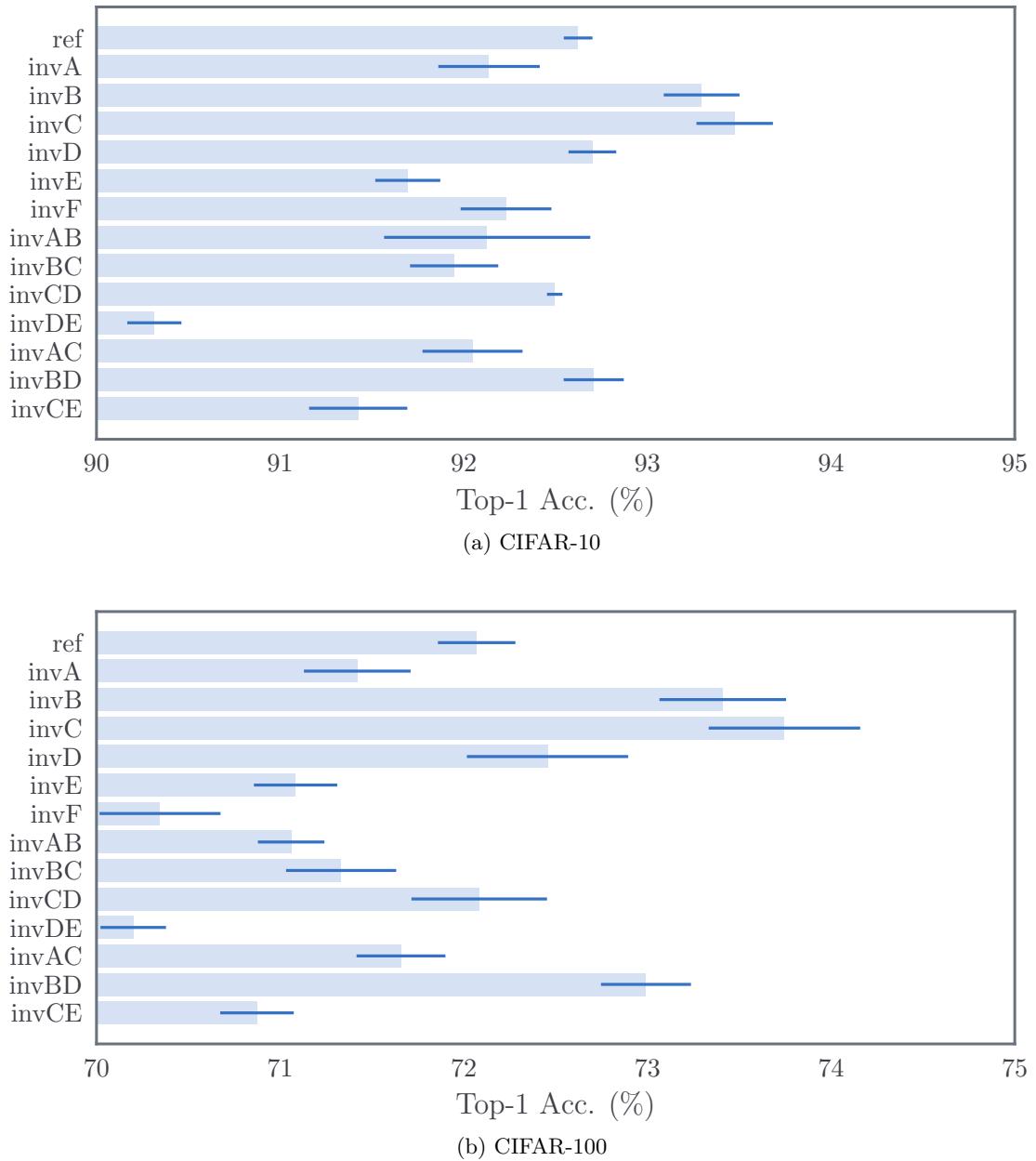


Figure 5.2: Ablation results for the invariant layer. Results for testing VGG like architecture with convolutional and invariant layers on several datasets. These graphs show the average top-1 accuracy on the validation set, and ± 1 standard deviation results (dark blue lines) for 5 runs. An architecture with ‘invX’ means the equivalent convolutional layer ‘convX’ from [Table 5.5](#) was swapped for our proposed layer. The top row is the reference architecture using all convolutional layers. It appears that using a single invariant layer is the best option, and this will improve performance at many possible locations. The best position for the invariant layer appears to be around ‘convC’, which is just after the first pooling layer.

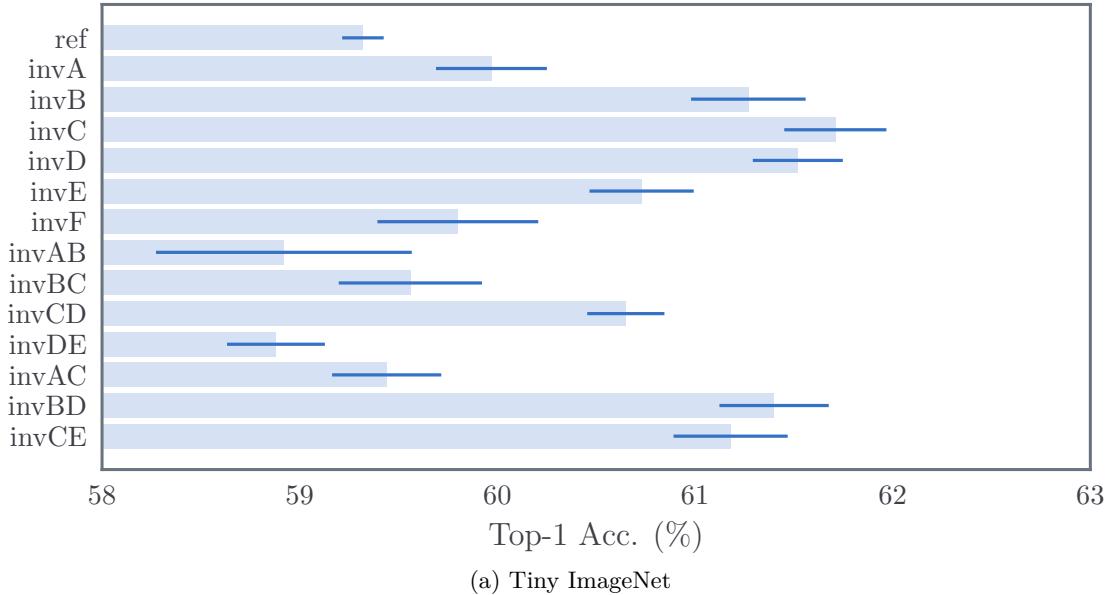


Figure 5.3: **Ablation results for Tiny ImageNet.** These graphs show the average top-1 accuracy on the validation set, and ± 1 standard deviation results (dark blue lines) for 3 runs. Like with CIFAR, it appears gains can be made by using the invariant layer at least once in the network. The best place for it appears to be around the first drop in resolution (which happens between ‘convB’ and ‘convC’).

naturally increases the channel dimension while downsampling the spatial dimensions. A single scattering order for our DTCWT ScatterNet increases the channel dimension seven-fold and reduces the spatial area by four. After two layers of a ScatterNet, the output has 49 times more channels than the input and one-sixteenth the spatial size.

We allow for this natural channel growth, using a ScatterNet front-end before four convolutional layers similar to ‘convC’ to ‘convF’ from [Table 5.5](#). The input to ‘convC’ already has spatial size 8×8 so we do not need the pooling layer between ‘convD’ and ‘convE’. The final classification layer is the same global 8×8 max pooling followed by a fully connected layer. In addition, we use dropout in these later convolutional layers with drop probability $p = 0.3$. See [Table 5.6c](#) for the backend layout.

We compare a ScatterNet with no learning in between scattering orders (ScatNet A in [Table 5.6a](#)) to one with our proposal for a learned mixing matrix (ScatNet B in [Table 5.6b](#)).

We also test the hypothesis from [section 5.7](#) that scattering layers may work better after the first layer of a network. To do this, we put a small learned convolutional layer before the learnable ScatterNet front-end that takes the three colour inputs and outputs 16 new channels. Increasing the ScatterNet input to 16 channels means the default output size would be $16 \times 49 = 784$ channels. This is quite large, so we also use a system that uses a non-square A matrix for the two learnable invariant layers, keeping the output to 147 channels. We call

Table 5.6: **Hybrid ScatterNet models.** Hybrid ScatterNet architectures used for experiments on CIFAR-10 and CIFAR-100. ScatNet A is a regular ScatterNet, and ScatNet B is our proposed learnable ScatterNet. Both ScatNet A and ScatNet B share the same back end, the architecture shown in (c). N_c is the number of output classes; in our experiments, we set the channel multiplier to be $C = 96$.

(a) ScatNet A Front End		(b) ScatNet B Front End	
Layer	Act. Size	Layer	Act. Size
scatA, no w	$3 \times 32 \times 32$	invA, $A \in \mathbb{R}^{21 \times 21}$	$3 \times 32 \times 32$
scatB, no w	$21 \times 16 \times 16$	invB, $A \in \mathbb{R}^{147 \times 147}$	$21 \times 16 \times 16$
	$147 \times 8 \times 8$		$147 \times 8 \times 8$

(c) BackEnd	
convC, $w \in \mathbb{R}^{2C \times 147 \times 3 \times 3}$	$147 \times 8 \times 8$
convD, $w \in \mathbb{R}^{2C \times 2C \times 3 \times 3}$	$2C \times 8 \times 8$
convE, $w \in \mathbb{R}^{4C \times 2C \times 3 \times 3}$	$2C \times 8 \times 8$
convF, $w \in \mathbb{R}^{4C \times 4C \times 3 \times 3}$	$4C \times 8 \times 8$
avg pool 8×8	$4C \times 8 \times 8$
fc1, $4C \times N_c$	$4C$
	N_c

this option ScatNet C (see [Table 5.7a](#)), and the full option with 784 output channels ScatNet D (see [Table 5.7b](#)).

See [Table 5.8](#) for the performance on CIFAR and [Table 5.9](#) for the performance on Tiny ImageNet. For comparison, we have also listed the performance of other architectures on CIFAR as reported by their authors in order of increasing complexity in [Table 5.8](#).

The learnable invariant layer significantly improves the hybrid ScatterNets on all three datasets (see the improvements from ScatNet A to ScatNet B). As we anticipated, adding a learned layer before the learnable ScatterNet improves things further, with ScatNet C and ScatNet D both having improvements on ScatNets A and B.

Our proposed ScatNet C and ScatNet D achieve comparable performance with the All Conv, VGG16 and FitNet architectures. The Deep[15] and Wide[70] ResNets perform best, but with very many more multiplies, parameters and layers.

ScatNets C and D perform marginally worse than the ‘invC’ architecture from [section 5.7](#) but recall that the network is quite different in these experiments - their front end is wider and has smaller spatial support. The earlier reduction in spatial size in ScatNet C and D meant that these networks took much less time to train than those in the previous section

Table 5.7: **Hybrid ScatterNet models with convolutional layer first.** The two ScatNet models are similar to the learnable ScatterNet from [Table 5.6b](#) but with a small convolutional layer ('conv0') before it. ScatNet C ensures the same $147 \times 8 \times 8$ output size as the models in [Table 5.6](#) but ScatNet D has a larger output size, allowing for the natural growth of a second-order ScatterNet model from C input channels to $49C$ output channels.

(a) ScatNet C		(b) ScatNet D	
Layer	Act. Size	Layer	Act. Size
conv0, $w \in \mathbb{R}^{16 \times 3 \times 3 \times 3}$	$3 \times 32 \times 32$	conv0, $w \in \mathbb{R}^{16 \times 3 \times 3 \times 3}$	$3 \times 32 \times 32$
invA, $A \in \mathbb{R}^{50 \times 112}$	$16 \times 32 \times 32$	invA, $A \in \mathbb{R}^{112 \times 112}$	$16 \times 32 \times 32$
invB, $A \in \mathbb{R}^{147 \times 350}$	$50 \times 16 \times 16$	invB, $A \in \mathbb{R}^{784 \times 784}$	$112 \times 16 \times 16$
	$147 \times 8 \times 8$		$784 \times 8 \times 8$

(roughly 30 minutes for ScatNets A and B compared to 2 to 4 hours for the networks in [section 5.7](#) using the hardware described in [appendix A](#)).

5.9 Conclusion

In this work, we have proposed a new learnable scattering layer, dubbed the locally invariant convolutional layer, tying together ScatterNets and CNNs.

The invariant layer takes a single scale complex wavelet decomposition of the input. The bandpass coefficients are demodulated by using the complex modulus nonlinearity. The resulting magnitude coefficients are then mixed with the lowpass coefficients with a learnt mixing matrix (or 1×1 convolution).

We tested the invariant layer initially on MNIST and proved that it could achieve comparable performance to a convolutional layer. We did see some issues with using the invariant layer on its own, as the 1×1 convolution may not have been large enough to separate the centres of the wavelets to make the complex shapes necessary. When we used larger kernels, either in the mixing matrix A (see [subsection 5.6.1](#)) or in convolutional layers after the invariant layer (see [section 5.7](#)) the performance improved.

Our ablation studies on a VGG-like CNN showed that the invariant layer can easily be shaped to allow it to drop in the place of a convolutional layer, theoretically saving on parameters and computation (see [section 5.7](#)). 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 early in the network, but *after* the first layer. This is an interesting discovery, as typically other ScatterNet approaches use them as a front end [37], [39]. The invariant layer naturally downsamples the input, so

Table 5.8: **Hybrid ScatterNet top-1 classification accuracies on CIFAR.** N_l is the number of learned convolutional layers, #param is the number of parameters, and #mults is the number of multiplies per $3 \times 32 \times 32$ image. An asterisk indicates that the value was estimated from the architecture description. ScatNet A (described in Table 5.6a) is similar to the hybrid ScatterNet from [39], ScatNet B (Table 5.6b) uses our proposed learnable ScatterNet, and ScatNets C and D (Table 5.7) add a small learned layer *before* scattering.

Arch. Name	Arch. Properties			Top 1 Accuracies (%)	
	N_l	#Mparam	#Mmults	CIFAR-10	CIFAR-100
ScatNet A	4	2.60	165	89.5	68.2
ScatNet B	6	2.64	167	91.5	70.5
ScatNet C	7	2.64	171	92.6	72.7
ScatNet D	7	3.7	251	93.3	73.6
All Conv[68]	8	1.4	281*	92.8	66.3
VGG16[140]	16	138*	313*	91.6	-
FitNet[141]	19	2.5	382	91.6	65.0
ResNet-1001[142]	1000	10.2	4453*	95.1	77.3
WRN-28-10[70]	28	36.5	5900*	96.1	81.2

Table 5.9: **Hybrid ScatterNet top-1 classification accuracies on Tiny ImageNet.**

Arch. Name	Top-1 Acc (%)
ScatNet A	58.1
ScatNet B	60.3
ScatNet C	62.1
ScatNet D	62.6

it works well when replacing a convolutional layer followed by pooling, or a pooling layer followed by a convolution.

We also tested the invariant layer on a hybrid ScatterNet architecture (see [section 5.8](#)). We saw that two invariant layers worked well as the first two layers of a deep CNN, but performed even better when used as the second and third layers, with a small learned convolutional before them. These hybrid ScatterNets downsample the input by a factor of 16 and increasing the channel dimension by 49. The reduced spatial size meant the networks were very quick to train, yet were able to achieve near state-of-the-art performance.

Chapter 6

Learning in the Wavelet Domain

In this final section of our work, 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. With ScatterNets, complex wavelets are used to scatter the energy into different channels (corresponding to the different wavelet subbands), before the complex modulus demodulates the signal to low frequencies. These channels can then be mixed before scattering again (as we saw in the learnable ScatterNet), but successive use of such layers compounds the demodulation of signal energy towards zero frequency. We saw in the previous chapter that as a result, the modulus-based invariant layer worked best at the location of sample rate changes in a CNN. Most modern CNNs operate at only a handful of spatial resolutions, restricting the number of locations it may be useful in.

In this chapter, we introduce the *wavelet gain layer* which starts in a similar fashion to the ScatterNet – by taking the DTCWT of a multi-channel input. Next, instead of taking a complex modulus, we learn a complex gain for each subband in each input channel. A single value here can amplify or attenuate all the energy in one part of the frequency plane. Then, while still in the wavelet domain, we mix the different input channels *by subband* (e.g. all the 15° wavelet coefficients at a given scale are mixed together, but the 75° and 45° coefficients are not). We can then return to the pixel domain with the inverse wavelet transform. The shift-invariant properties of the DTCWT allow the wavelet coefficients to be changed without introducing sampling artefacts.

We also briefly explore the possibility of doing nonlinearities in the wavelet domain. Our ultimate goal is to connect multiple wavelet gain layers together with nonlinearities before returning to the pixel domain. See [subsection 6.2.4](#) for a more detailed description of this.

The proposed wavelet gain layer can be used in conjunction with regular convolutional layers, with a network moving into the wavelet or pixel space and learning filters in one that would be difficult to learn in the other.

Our experiments so far have shown some promise. We are able to learn complex wavelet gains and have found that the ReLU works well as a wavelet nonlinearity. We have found

that the wavelet gain layer works well at the beginning of a CNN but have not yet seen significant improvements for later layers.

6.1 Chapter Layout

We review some related work and notation in section 6.2 before describing the operation of our layer in section 6.3. In section 6.4, we describe some of the preliminary experiments and results we achieve by learning in the wavelet domain but returning to the pixel domain to perform nonlinearities. Section 6.5 describes expansions on this work to include nonlinearities in the wavelet domain and describes the preliminary results we have achieved so far.

6.2 Background

6.2.1 Related Work

Fujieda et. al. use a DWT in combination with a CNN to do texture classification and image annotation [143], [144]. They take a multiscale wavelet transform of the input image, combine the activations at each scale independently with learned weights, and feed these back into the network at locations where the activation resolution matches the subband resolution. The architecture block diagram is shown in Figure 6.1, taken from the original paper. They found that their ‘Wavelet-CNN’ could outperform competitive non-wavelet-based CNNs on both texture classification and image annotation.

Several works also use wavelets in deep neural networks for super-resolution [145] and for adding detail back into dense pixel-wise segmentation tasks [146]. These typically save wavelet coefficients and use them for the reconstruction phase.

In [147], Rippel, Snoek, and Adams parameterize filters in the DFT domain. Rather than having a pixel domain filter $w \in \mathbb{R}^{F \times C \times K \times K}$, they learn a set of Fourier coefficients $\hat{w} \in \mathbb{C}^{F \times C \times K \times \lceil K/2 \rceil}$ (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{w} to obtain w and then convolve this with the input x as a normal CNN would do¹.

Note that an important point should be emphasized about reparameterizing filters in either the wavelet or Fourier domains: many linear transforms of the parameter space will not change parameter updates if a linear optimization scheme is used (for example standard GD, or SGD with momentum). Rippel, Snoek, and Adams mention in their work that this holds

¹The 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.

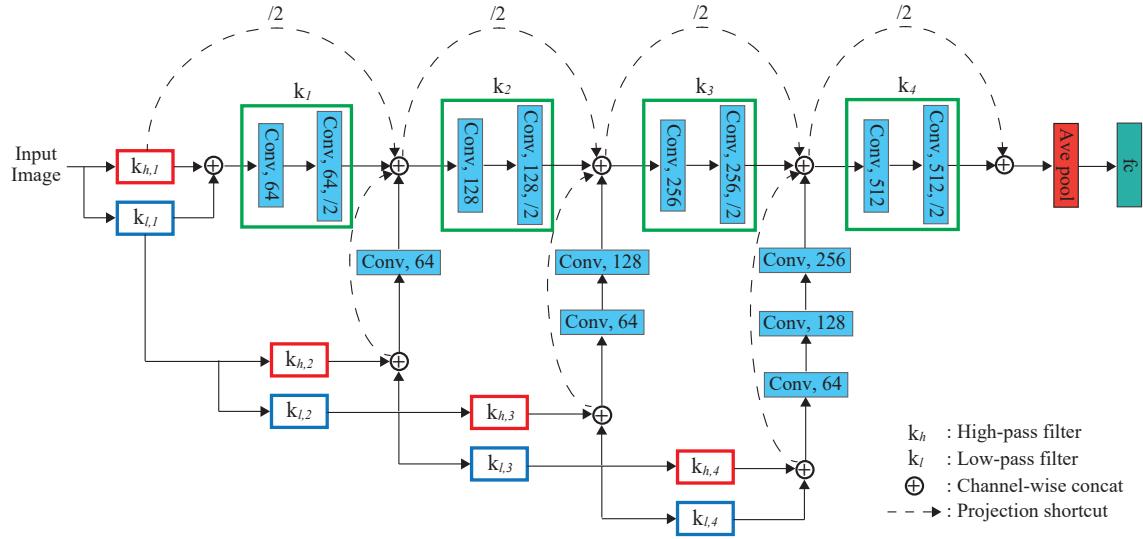


Figure 6.1: **Architecture using the DWT as a frontend to a CNN.** Figure 1 from [144]. Fujieda et. al. take a multiscale wavelet decomposition of the input before passing the input through a standard CNN. They learn convolutional layers independently on each subband and feed these back into the network at different depths, where the resolution of the subband and the network activations match.

for *all* invertible transforms but this is not strictly true, and we prove in [appendix C](#) that it only holds for *tight frames*. We make this point clear as a natural extension to continue the work in [147] would be to parameterize filters in the *wavelet* domain, taking inverse transforms and then doing normal convolution.

While [147] was an inspiration for this chapter, the work we present here is not a reparameterization in the wavelet domain with convolution in the pixel domain. Instead, we learn wavelet filters *and* perform filtering in the wavelet domain too.

6.2.2 Notation

We make use of the 2-D *Z*-transform to simplify our analysis:

$$X(\mathbf{z}) = \sum_{n_1} \sum_{n_2} x[n_1, n_2] z_1^{-n_1} z_2^{-n_2} = \sum_{\mathbf{n}} x[\mathbf{n}] \mathbf{z}^{-\mathbf{n}} \quad (6.2.1)$$

As we are working with three-dimensional and four-dimensional arrays but are only doing convolution in two, we introduce a slightly modified 2-D *Z*-transform which includes the

channel index c and the filter number f :

$$X(c, \mathbf{z}) = \sum_{n_1} \sum_{n_2} x[c, n_1, n_2] z_1^{-n_1} z_2^{-n_2} = \sum_{\mathbf{n}} x[c, \mathbf{n}] \mathbf{z}^{-\mathbf{n}} \quad (6.2.2)$$

$$H(f, c, \mathbf{z}) = \sum_{n_1} \sum_{n_2} h[f, c, n_1, n_2] z_1^{-n_1} z_2^{-n_2} = \sum_{\mathbf{n}} h[f, c, \mathbf{n}] \mathbf{z}^{-\mathbf{n}} \quad (6.2.3)$$

We then define the product of these new Z -transform signals to be the channel-wise convolution. E.g. for the 4-D filter $h[f, c, \mathbf{n}]$ with Z -transform $H(f, c, \mathbf{z})$ and the the 3-D signal $x[c, \mathbf{n}]$ with Z -transform $X(c, \mathbf{z})$, let us call the product of the two Z -transforms:

$$X(c, \mathbf{z}) H(f, c, \mathbf{z}) = \sum_{\mathbf{n}} \left(\sum_{\mathbf{k}} h[f, c, \mathbf{n} - \mathbf{k}] x[c, \mathbf{k}] \right) \mathbf{z}^{-\mathbf{n}} \quad (6.2.4)$$

Recall from [subsection 2.4.1](#) that a typical convolutional layer in a standard CNN gets the next layer's output in a two-step process:

$$y^{(l+1)}[f, \mathbf{n}] = \sum_{c=0}^{C_l-1} x^{(l)}[c, \mathbf{n}] * h^{(l)}[f, c, \mathbf{n}] \quad (6.2.5)$$

$$x^{(l+1)}[f, \mathbf{n}] = \sigma(y^{(l+1)}[f, \mathbf{n}]) \quad (6.2.6)$$

If we define the nonlinearity σ_z to be the action of σ to each z -coefficient in the polynomial $Y(f, \mathbf{z})$, then we can rewrite (6.2.5) and (6.2.6) as:

$$Y^{(l+1)}(f, \mathbf{z}) = \sum_{c=0}^{C_l-1} X^{(l)}(c, \mathbf{z}) H^{(l)}(f, c, \mathbf{z}) \quad (6.2.7)$$

$$X^{(l+1)}(f, \mathbf{z}) = \sigma_z(Y^{(l+1)}(f, \mathbf{z})) \quad (6.2.8)$$

6.2.3 DT \mathbb{C} WT Notation

For this chapter, we will work with lots of DT \mathbb{C} WT coefficients so we define some slightly new notation here.

A J scale 2-D 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 (lp) coefficients.

$$\text{DT}\mathbb{C}\text{WT}_J(x) = \{u_{lp}, u_{j,k}\}_{1 \leq j \leq J, 0 \leq k < 6} \quad (6.2.9)$$

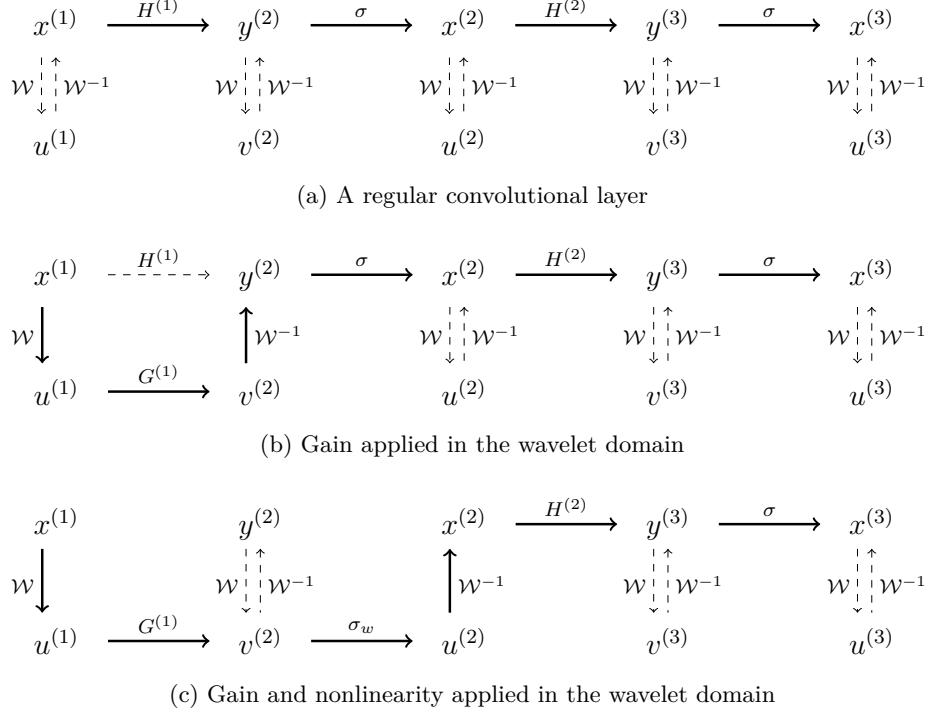


Figure 6.2: Proposed new forward pass in the wavelet domain. Two network layers with some possible options for processing. Solid lines denote the evaluation path and dashed lines indicate relationships. In (a) we see a regular convolutional neural network. We have included the dashed lines to make clear 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 $G^{(1)}$, and take the inverse wavelet transform to give $y^{(2)}$. The dotted line for $H^{(1)}$ indicates that this path is no longer present. Note that there may not be any possible $G^{(1)}$ 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.

Each of these coefficients has size:

$$u_{lp} \in \mathbb{R}^{N \times C \times \frac{H}{2^{J-1}} \times \frac{W}{2^{J-1}}} \quad (6.2.10)$$

$$u_{j,k} \in \mathbb{C}^{N \times C \times \frac{H}{2^J} \times \frac{W}{2^J}} \quad (6.2.11)$$

Recall that the lowpass coefficients are twice as large as in a fully decimated transform due to the interleaving of the four lowpass terms in [Algorithm 3.3](#).

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&CWT coefficients.

6.2.4 Learning in Multiple Spaces

At the beginning of each layer l of a neural network, we have the activations $x^{(l)}$. Naturally, all of these activations have their equivalent wavelet coefficients $u^{(l)}$.

From (6.2.5), convolutional layers also have intermediate activations $y^{(l)}$. Let us discern these from the x coefficients and modify (6.2.9) to say the DT^CWT of $y^{(l)}$ gives $v^{(l)}$.

We now propose the *wavelet gain layer* G . The gain layer G can be used instead of a convolutional layer. It is designed to work on the wavelet coefficients of an activation u , to give wavelet domain outputs v .

This can be seen as breaking the convolutional path in Figure 6.2 and taking a new route to get 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 wavelet-based nonlinearities, σ_{lp} and σ_{bp} for the lowpass and highpass coefficients respectively, to give $u^{(l+1)}$.

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:

1. **How effective is a wavelet gain layer G at replacing a standard convolutional layer H ?**
2. **What are effective wavelet nonlinearities σ_{lp} and σ_{bp} ?**

6.3 The DT^CWT Gain Layer

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

$$g_{lp} \in \mathbb{R}^{C_{l+1} \times C_l \times k_{lp} \times k_{lp}} \quad (6.3.1)$$

$$g_{1,1} \in \mathbb{C}^{C_{l+1} \times C_l \times k_1 \times k_1} \quad (6.3.2)$$

$$g_{1,2} \in \mathbb{C}^{C_{l+1} \times C_l \times k_1 \times k_1} \quad (6.3.3)$$

⋮

$$g_{J,6} \in \mathbb{C}^{C_{l+1} \times C_l \times k_J \times k_J} \quad (6.3.4)$$

where the k_j 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. Importantly, we can select the support size differently for each subband.

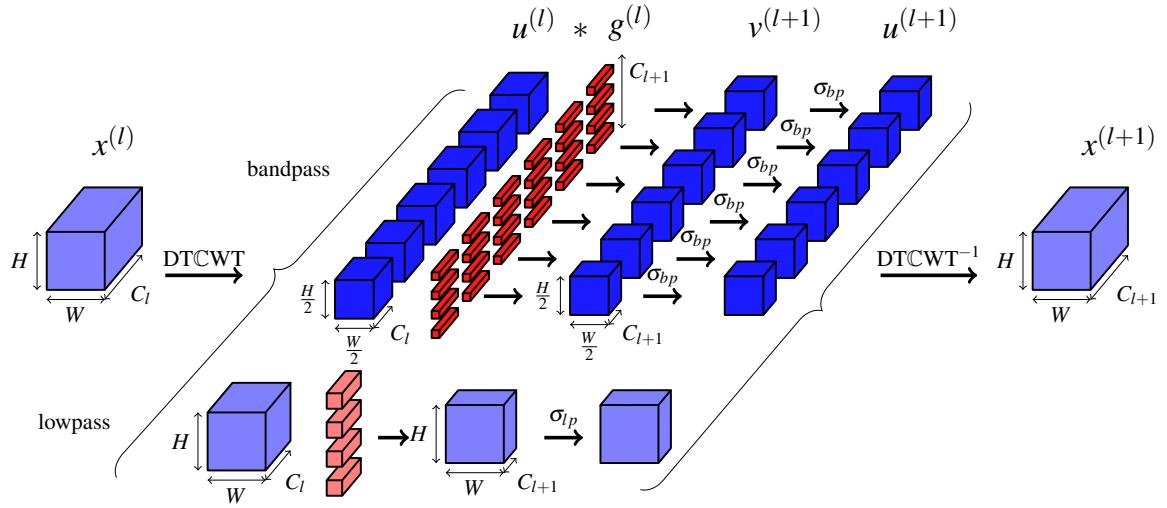


Figure 6.3: **Diagram of the proposed method to learn in the wavelet domain.** Activations are shaded blue and learned parameters red. Deeper shades of blue and red indicate complex-valued activations/weights, and lighter values indicate real-valued activations/weights. The input $x^{(l)} \in \mathbb{R}^{C_l \times H \times W}$ is taken into the wavelet domain (here $J = 1$) and each subband is mixed independently with C_{l+1} sets of convolutional filters. After mixing, a possible wavelet nonlinearity $\sigma_w = (\sigma_{lp}, \sigma_{bp})$ is applied to the subbands, before returning to the pixel domain with an inverse wavelet transform. Note the similarity to the regular convolutional layer in Figure 2.7.

With these gains we define the action of the gain layer $v = Gu$ to be:

$$v_{lp}[f, \mathbf{n}] = \sum_{c=0}^{C_l-1} u_{lp}[c, \mathbf{n}] * g_{lp}[f, c, \mathbf{n}] \quad (6.3.5)$$

$$v_{1,1}[f, \mathbf{n}] = \sum_{c=0}^{C_l-1} u_{1,1}[c, \mathbf{n}] * g_{1,1}[f, c, \mathbf{n}] \quad (6.3.6)$$

$$v_{1,2}[f, \mathbf{n}] = \sum_{c=0}^{C_l-1} u_{1,2}[c, \mathbf{n}] * g_{1,2}[f, c, \mathbf{n}] \quad (6.3.7)$$

\vdots

$$v_{J,6}[f, \mathbf{n}] = \sum_{c=0}^{C_l-1} u_{J,6}[c, \mathbf{n}] * g_{J,6}[f, c, \mathbf{n}] \quad (6.3.8)$$

To avoid ambiguity with complex conjugates we remind ourselves that for complex signals a, b the convolution $a * b$ is defined as $(a_r * b_r - a_i * b_i) + j(a_r * b_i + a_i * b_r)$ (see section E.1).

The action of the gain layer with only a single-scale wavelet transform, $J = 1$, is shown in Figure 6.3.

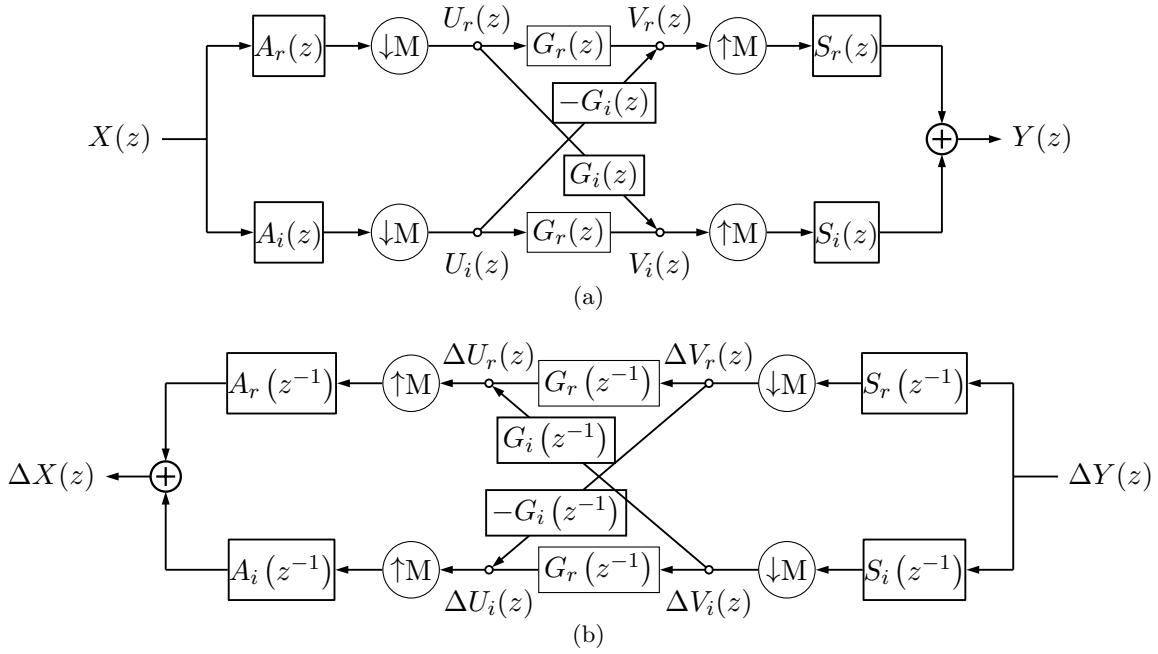


Figure 6.4: **Forward and backward filter bank diagrams for DTCWT gain layer.** Based on Figure 4 in [19]. Ignoring the G gains, the top and bottom paths (through A_r, S_r and A_i, S_i respectively) make up the real and imaginary parts for *one subband* of the dual tree system. Combined, $A_r + jA_i$ and $S_r - jS_i$ make the complex filters necessary to have support on one side of the Fourier domain (see Figure 6.5). 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).

6.3.1 The Output

Due to the shift-invariant properties of the DT \mathbb{C} WT, each wavelet subband has a unique transfer function which is almost free of aliasing (see Theorem D.1). If we do complex convolution of the wavelet coefficients u with gains g as described in (6.3.5) - (6.3.8), then we preserve the shift-invariant properties (see Theorem D.2 and Theorem D.3) and the inverse DT \mathbb{C} WT of the outputs v are free from aliasing.

We can do a complex multiply of the subband coefficients u with gains g by using the filter bank diagram shown in Figure 6.4a. Note that despite the resemblance to many filter bank diagrams for fully decimated DWTs, Figure 6.4a is different. The top rung corresponds to the real part of a subband and the bottom specifies the imaginary part.

We consider the output from a *single* complex subband. The complex gain for this subband is $G = G_r + jG_i$ where j is the square root of negative one. Let us call the analysis filters $A = A_r + jA_i$ and the synthesis filters $S = S_r - jS_i$ (these are normally called H and G ,

but we keep those letters reserved for the CNN and gain layer filters). For a single-channel input in 1-D, the output of this layer with decimation and interpolation is:

$$Y(z) = \frac{2}{M} X(z) [G_r(z^M)(A_r(z)S_r(z) + A_i(z)S_i(z)) + G_i(z^M)(A_r(z)S_i(z) - A_i(z)S_r(z))] \quad (6.3.9)$$

Again, see [appendix D](#) for the derivation which proves that the aliasing terms caused by the downsampling by M are (largely) eliminated.

This expands to the 2-D case with multiple channels as:

$$Y(f, \mathbf{z}) = \sum_{c=0}^{C_l-1} \frac{2}{M} X(c, \mathbf{z}) [G_r(f, c, \mathbf{z}^M)(A_r(\mathbf{z})S_r(\mathbf{z}) + A_i(\mathbf{z})S_i(\mathbf{z})) + G_i(f, c, \mathbf{z}^M)(A_r(\mathbf{z})S_i(\mathbf{z}) - A_i(\mathbf{z})S_r(\mathbf{z}))] \quad (6.3.10)$$

The complex gain G has a real and imaginary part. The real term G_r modifies the subband gain $A_r S_r + A_i S_i$ and the imaginary term G_i modifies its Hilbert Pair $A_r S_i - A_i S_r$. [Figure 6.5](#) shows the contour plots for the frequency support of each of these subbands.

Now we consider the sum of all different subbands. As the operations are all linear, the full output y is simply the sum of all the y 's from individual subbands. The complex gains g adjust the gain and phase of each subband independently. The magnitude of each element controls the amplitude of the frequency response in the region of that subband, while its phase controls the phase of the response and thus modifies the detailed wave-shape (e.g. the locations of its zero crossings).

6.3.2 Backpropagation

Assume we already have access to the quantity $\Delta Y(z)$ (this is the input to the backwards pass). [Figure 6.4b](#) illustrates the backpropagation procedure.

We recall the passthrough and update equations for a convolutional block from [\(2.4.12\)](#) and [\(2.4.15\)](#):

$$\frac{\partial L}{\partial x_{c,k_1,k_2}} = \sum_f \Delta y[f, \mathbf{n}] \star h_c[f, \mathbf{n}] \quad (6.3.11)$$

$$\frac{\partial L}{\partial h_{f,c,k_1,k_2}} = \Delta y[f, \mathbf{n}] \star x[c, \mathbf{n}] \quad (6.3.12)$$

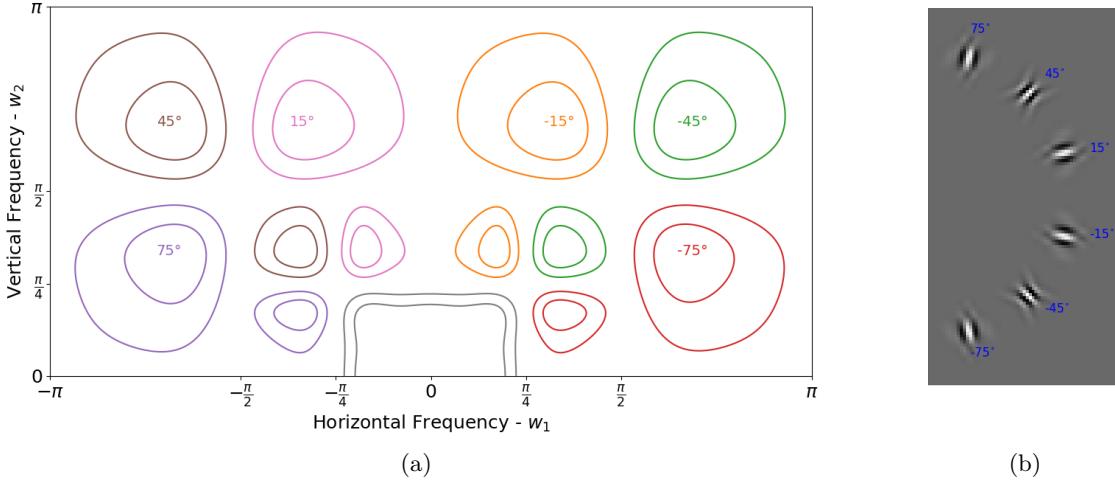


Figure 6.5: DTCWT subbands. (a) -1dB and -3dB contour plots showing the support in the Fourier domain of the 6 subbands of the DT \mathbb{C} WT at scales 1 and 2, and the scale 2 lowpass. These are the product of the single side-band filters $P(\mathbf{z})$ and $Q(\mathbf{z})$ from [Theorem D.1](#), or half of the support of the double side-band filters $A_r S_r + A_i S_i$ and $A_r S_i - A_i S_r$ from [\(6.3.9\)](#). (b) The pixel domain impulse responses for the second scale wavelets. The Hilbert pair for each wavelet is the underlying sinusoid phase shifted by 90 degrees.

In terms of modified Z -transforms, this becomes:

$$\Delta X(c, \mathbf{z}) = \sum_f \Delta Y(f, \mathbf{z}) H'[c, f, \mathbf{z}^{-1}] \quad (6.3.13)$$

$$\Delta H(f, c, \mathbf{z}) = \sum_f \Delta Y(f, \mathbf{z}) X[c, \mathbf{z}^{-1}] \quad (6.3.14)$$

where H' is H transposed in the filter and channel dimensions, and \mathbf{z}^{-1} indicates the filter was mirror-imaged in the spatial dimension. The simplified, single-channel, single filter, 1-D version of this is:

$$\Delta X(z) = \Delta Y(z) H(z^{-1}) \quad (6.3.15)$$

$$\Delta H(z) = \Delta Y(z) X(z^{-1}) \quad (6.3.16)$$

If H were complex, the first term in [Equation 6.3.15](#) would be $\bar{H}(1/\bar{z})$, but as each individual block in the DT \mathbb{C} WT of [Figure 6.4](#) is purely real, we can use the simpler form $H(z^{-1})$.

We calculate $\Delta V_r(z)$ and $\Delta V_i(z)$ by backpropagating $\Delta Y(z)$ through the inverse DT \mathbb{C} WT. This is the same as doing the forward DT \mathbb{C} WT on $\Delta Y(z)$ with the synthesis and analysis

filters swapped and time-reversed. Then the weight update equations are:

$$\Delta G_r(z) = \Delta V_r(z)U_r(z^{-1}) + \Delta V_i(z)U_i(z^{-1}) \quad (6.3.17)$$

$$\Delta G_i(z) = -\Delta V_r(z)U_i(z^{-1}) + \Delta V_i(z)U_r(z^{-1}) \quad (6.3.18)$$

The passthrough equations have similar form to (6.3.9):

$$\Delta X(z) = \frac{2\Delta Y(z)}{M} [G_r(z^{-M})(A_r(z)S_r(z) + A_i(z)S_i(z)) + G_i(z^{-M})(A_r(z)S_i(z) - A_i(z)S_r(z))] \quad (6.3.19)$$

6.3.3 Examples

Figure 6.6 shows example impulse responses of the DT \mathbb{C} WT gain layer. For comparison, we also show similar ‘impulse responses’ for a gain layer done in the DWT domain². The DWT outputs come from three random variables: a 1×1 convolutional weight applied to each of the low-high, high-low and high-high subbands. The DT \mathbb{C} WT outputs come from twelve random variables, again a 1×1 convolutional weight, but now applied to six complex subbands.

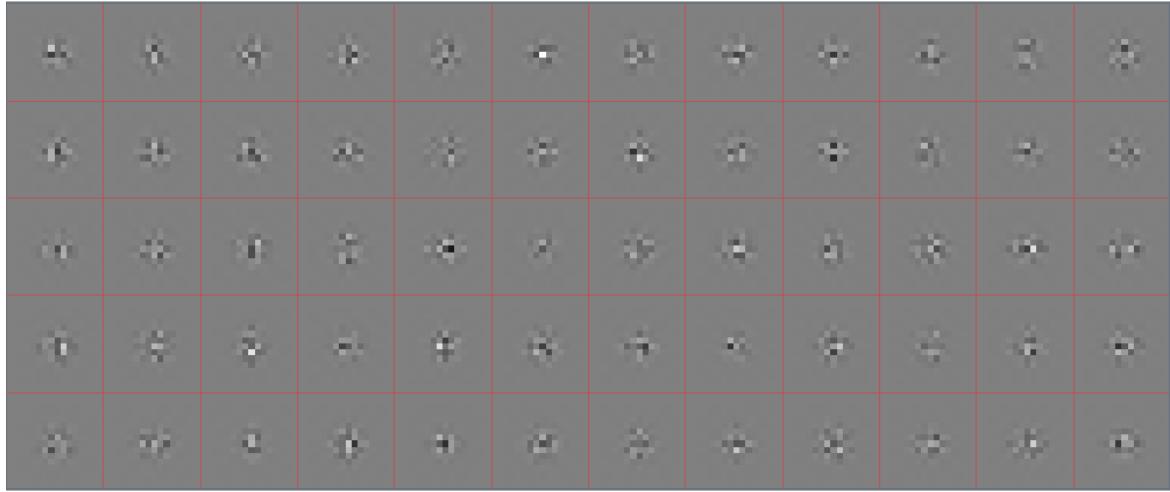
To test the space of generated shapes by a vector gain layer gain g_1 , we generate N random vectors of length 12, with each entry taken from a Gaussian distribution with zero mean and unit variance. We then generate the equivalent point spread functions from (6.3.9) for the N different instances and measure their normalized cross-correlation. We then sort the values and compare the distribution to a set of N random vectors with k degrees of freedom.

Our experiments show that the distribution for the DT \mathbb{C} WT gain layer matches random vectors with roughly 11.5 degrees of freedom (c.f. the 12 variables the layer has). Similarly for the DWT, the normalized cross-correlation matches the distribution for random vectors with roughly 2.8 degrees of freedom (c.f. 3 random variables in the layer). This is particularly reassuring for the DT \mathbb{C} WT as it is showing that there is still representative power despite the redundancy of the transform.

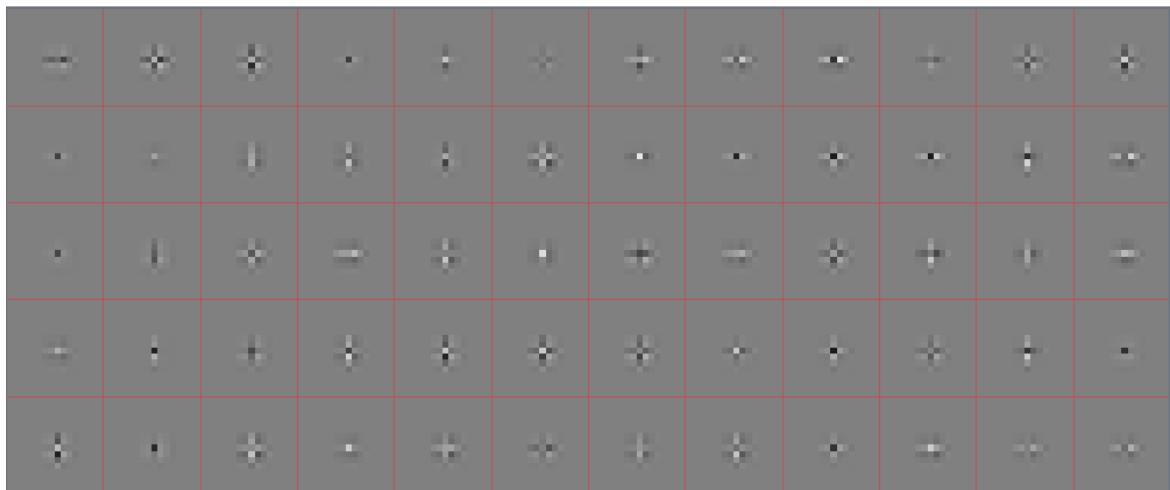
6.3.4 Implementation Details

Before analyzing its performance, we compare the implementation properties of our proposed layer to a standard convolutional layer.

²Modifying DWT coefficients causes a loss of the alias cancellation properties so these are not true impulse responses.



(a)



(b)

Figure 6.6: Example outputs from an impulse input for the proposed gain layers. Example outputs $y = \mathcal{W}^{-1}G\mathcal{W}x$ for an impulse x for the DTCWT gain layer and for a similarly designed DWT gain layer. (a) shows the output y for a DTCWT based system. $g_{lp} = 0$ and g_1 has spatial size 1×1 . The 12 values in g_1 are independently sampled from a random normal of variance 1. The 60 samples come from 60 different random draws of the weights. (b) shows the outputs y when x is an impulse and \mathcal{W} is the DWT with a ‘db2’ wavelet family. Here 3 random numbers are generated for the g_1 coefficients. The strong horizontal and vertical properties of the DWT can clearly be seen in comparison to the much freer DTCWT.

6.3.4.1 Parameter Memory Cost

A standard convolutional layer with C_l input channels, C_{l+1} output channels and kernel size $k \times k$ has $k^2 C_l C_{l+1}$ parameters, with $k = 3$ or $k = 5$ common choices for the spatial size.

$$\#\text{conv params} = k^2 C_l C_{l+1} \quad (6.3.20)$$

We must choose the spatial sizes of both the lowpass and bandpass mixing kernels. In our work, we are somewhat limited in how large we would like to set the bandpass spatial size, as every extra pixel of support requires $2 \times 6 = 12$ extra parameters. For this reason, we almost always set it to have support 1×1 . The lowpass gains are less costly, and we are free to set them to size $k_{lp} \times k_{lp}$ (with $k_{lp} = 1, 3, 5$ in many of our experiments). Further, due to the size of the datasets we test on, we typically limit ourselves initially to only considering a single scale. If we wish, we can decompose the input into more scales, resulting in a larger net area of effect. In particular, it may be useful to do a two-scale transform and discard the first scale coefficients. This does not increase the number of gains to learn but changes the position of the bands in the frequency space.

The number of parameters for the gain layer with $k_{lp} = 1$ is:

$$\#\text{params} = (2 \times 6 + 1) C_l C_{l+1} = 13 C_l C_{l+1} \quad (6.3.21)$$

This is slightly larger than the $9 C_l C_{l+1}$ parameters used in a standard 3×3 convolution, but as Figure 6.6 shows, the spatial support of the full filter is larger than an equivalent one parameterized in the filter domain. If $k_{lp} = 3$ then we would have $21 C_l C_{l+1}$ parameters, slightly fewer than the $25 C_l C_{l+1}$ of a 5×5 convolution.

6.3.4.2 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

$$\#\text{conv floats} = HWC_l \quad (6.3.22)$$

Our layer requires us to save the wavelet coefficients u_{lp} and $u_{j,k}$ for updating the g terms as in (6.3.17) and (6.3.18). For the 4 : 1 redundant DT^CWT, this requires:

$$\#\text{DT}\text{C}\text{WT floats} = 4HWC_l \quad (6.3.23)$$

to be saved for the backwards pass. You can see this difference from the difference in the block diagrams in Figure 6.3.

Note that a single scale DTCWT gain layer requires $16/7$ times as many floats to be saved as compared to the invariant layer of the previous chapter. The extra cost of this comes from two things. Firstly, we keep the real and imaginary components for the bandpass (as opposed to only the magnitude), meaning we need $3HWC_l$ floats, rather than $\frac{3}{2}HWC_l$. Additionally, the lowpass was downsampled in the previous chapter, requiring only $\frac{1}{4}HWC_l$, whereas we keep the full sample rate, costing HWC_l .

If memory is an issue and the computation of the DTCWT is very fast, then we only need to save the $x^{(l)}$ coefficients and can calculate the u 's on the fly during the backwards pass. Note that a two-scale DTCWT gain layer would still only require $4HWC_l$ floats.

6.3.4.3 Computational Cost

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

For the DTCWT, the overhead calculations are the same as in subsection 5.5.3, so we will omit their derivation here. The mixing is however different, requiring complex convolution for the bandpass coefficients, and convolution over a higher resolution lowpass. The bandpass has one quarter spatial resolution at the first scale, but this is offset by the 4 : 1 cost of complex multiplies compared to real multiplies. Again assuming we have set $J = 1$ and $k_{lp} = 1$ then the total cost for the gain layer is:

$$\text{\#mults/pixel} = \underbrace{\frac{6 \times 4}{4} C_{l+1}}_{\text{bandpass}} + \underbrace{C_{l+1}}_{\text{lowpass}} + \underbrace{36}_{\text{DTCWT}} + \underbrace{36}_{\text{DTCWT}^{-1}} = 7C_{l+1} + 72 \quad (6.3.24)$$

which is marginally smaller than the $9C_{l+1}$ of a 3×3 convolutional layer (if $C_{l+1} > 36$).

6.3.4.4 Parameter Initialization

For both layer types we use the Glorot Initialization scheme [55] with $a = 1$:

$$g_{ij} \sim U \left[-\sqrt{\frac{6}{(C_l + C_{l+1})k^2}}, \sqrt{\frac{6}{(C_l + C_{l+1})k^2}} \right] \quad (6.3.25)$$

where k is the kernel size.

6.4 Gain Layer Experiments

Before we explore the possibilities and performance of using a nonlinearity in the wavelet domain, let us present some experiments and results for the wavelet gain layer where we do nonlinearities purely in the spatial domain, as in a conventional CNN layer. This is the first objective in subsection 6.2.4, comparing G to H .

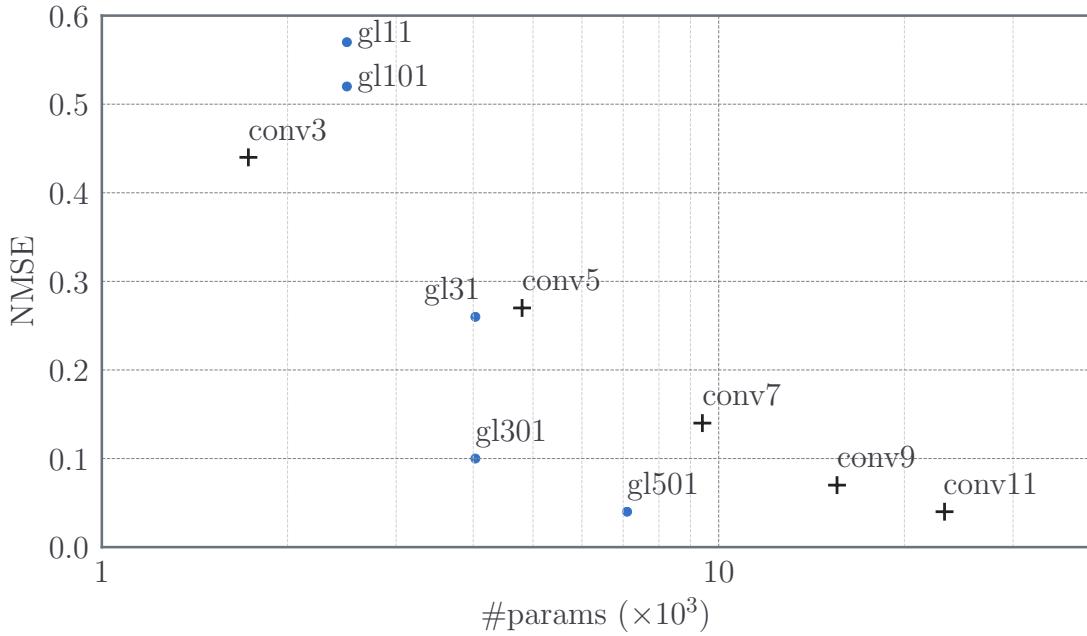


Figure 6.7: **Normalized MSE for conv layer and wavelet gain layer regression.** We minimize (6.4.1) and (6.4.2) on the ImageNet validation set, where the target is made from convolving the input with AlexNet’s first layer filters. This plot shows the final NMSE score compared to the number of learnable parameters. The original conv layer has spatial support 11×11 . The four points labelled ‘convn’ correspond to filters with $n \times n$ spatial support. The four points labelled ‘glabc’ correspond to two-scale gain layers with $a \times a$ support in the lowpass, $b \times b$ spatial support in the first scale, and $c \times c$ spatial support in the second scale. The gain layer can regress to the AlexNet filters quite capably. In this example, it is important to have at least 3×3 lowpass support for the gain layer, and the second scale coefficients are more important than the first scale.

6.4.1 CNN activation regression

One of the early inspirations for using wavelets in CNNs was the visualizations of the first layer filters learned in AlexNet. These 11×11 colour filters (see Figure 1.2a) look very much like a 2-D oriented wavelet transform.

So how well can the gain layer emulate the action of this layer? How would it compare to trying to use a reduced size convolutional kernel to learn the action of the layer?

Let us call the action of our target layer H_0 , a learned convolutional layer H and our gain layer G . We assume that we do not have direct access to H_0 but only the convolved outputs $y = H_0x$. Then, we would like to solve:

$$\arg \min_H (y - Hx)^2, \quad \text{s.t. } h[c, \mathbf{n}] = 0, \forall \mathbf{n} \notin \mathcal{R} \quad (6.4.1)$$

$$\arg \min_G (y - \mathcal{W}^{-1}G\mathcal{W}x)^2, \quad \text{s.t. } g_{j,k}[c, \mathbf{n}] = 0, \forall \mathbf{n} \notin \mathcal{R}' \quad (6.4.2)$$

for some support regions $\mathcal{R}, \mathcal{R}'$. E.g. \mathcal{R} is a 3×3 or 5×5 block, and similarly \mathcal{R}' is a 1×1 region in each subband.

(6.4.1) and (6.4.2) are both regression problems convex in the parameters of H and G , with many possible ways to solve. We are particularly interested in using the gain layer as part of a CNN, so we choose SGD to minimize these distances, using the validation set for ImageNet as the data input-output pair (x, y) . Because of the large size of the input filters, we allow for both a $J = 1$ and $J = 2$ scale gain layer but only learn weights at the lowest frequency bandpass (i.e. for a 2 scale gain layer, we discard the first scale highpass outputs and only learn g_2).

After training, we report the final normalized mean squared errors between the target values y and the estimated outputs \hat{y} , defined as:

$$NMSE = \frac{1}{N} \sum_{n=1}^N \frac{\mathbb{E}[(y - \hat{y})^2]}{\mathbb{E}[y^2]} \quad (6.4.3)$$

The resulting NMSEs are shown in Figure 6.7. A label ‘glab’ indicates a single scale gain layer with $a \times a$ support in the lowpass and $b \times b$ support in the highpass; a label ‘glabc’ indicates a two-scale gain layer with $a \times a$ support in the lowpass, $b \times b$ support in the scale 1 highpass and $c \times c$ support in the scale 2 bandpass gains.

This figure shows several interesting yet unsurprising things. Firstly, bigger lowpass support is very helpful – see the difference between gl101, gl301, and gl501, 3 instances that only vary in the size of the support of their lowpass filter g_{lp} . Additionally, the second scale coefficients appear more useful than the first scale – see the difference between gl31 and gl301, two instances that have the same number of parameters, but gl31 has g_1 with non-zero support and gl301 has g_2 with non-zero support. Compared to the regular convolutional layers, the gain layer is able to achieve the same NMSE with many fewer learned parameters.

This experiment shows that a gain layer is a good representation for a set of filters like the AlexNet first layer. The next section looks at how well it performs at deeper layers.

6.4.2 Ablation Studies

Figure 6.7 is a useful guide on how the gain layer might be placed in a deep CNN. gl11 (a gain layer with a 1×1 lowpass kernel and a 1×1 bandpass kernel at the first scale), gl101 (same as gl11 but no gain at first scale and 1×1 at second scale), and conv3 all achieve similar NMSEs. Additionally, gl31, gl301, and conv5 all achieve similar NMSEs.

6.4.2.1 Small Kernel Ablation

Most modern CNNs are built with small 3×3 kernels, which we believe are not the best use for the gain layer, built from large support wavelets. For this reason, we deviate from

the ablation study done in the previous chapter and build a **shallower** network with **larger** kernel sizes.

For completeness, we also ran ablation tests on the same deeper network with small kernels used in [chapter 5](#) and include the results in [appendix F](#).

6.4.2.2 Large Kernel Ablation

Table 6.1: Ablation base architecture. Reference architecture used for experiments on CIFAR-10, CIFAR-100. The activation size rows are offset from the layer description rows to convey the input and output shapes. Unlike [Table 5.5](#), this architecture is shallower and uses 5×5 convolutional kernels as a base. C is a hyperparameter that controls the network width, we use $C = 64$ for our tests. The Tiny ImageNet architecture is very similar but with larger activation sizes and one more convolutional layer ‘conv4’.

Activation Size	Reference Arch.	Alternate Arch.
$3 \times 32 \times 32$		
$C \times 32 \times 32$	conv1, $w \in \mathbb{R}^{C \times 3 \times 5 \times 5}$	or gain1, $g_{lp} \in \mathbb{R}^{C \times 3 \times 3 \times 3}$, $g_1 \in \mathbb{C}^{C \times 6 \times 3 \times 1 \times 1}$
$C \times 16 \times 16$	pool1, max pool 2×2	
$2C \times 16 \times 16$	conv2, $w \in \mathbb{R}^{2C \times C \times 5 \times 5}$	or gain2, $g_{lp} \in \mathbb{R}^{2C \times C \times 3 \times 3}$, $g_1 \in \mathbb{C}^{2C \times 6 \times C \times 1 \times 1}$
$2C \times 8 \times 8$	pool2, max pool 2×2	
$4C \times 8 \times 8$	conv3, $w \in \mathbb{R}^{4C \times 2C \times 5 \times 5}$	or gain3, $g_{lp} \in \mathbb{R}^{4C \times 2C \times 3 \times 3}$, $g_1 \in \mathbb{C}^{4C \times 6 \times 2C \times 1 \times 1}$
$4C \times 1 \times 1$	avg, 8×8 average pool	
10, 100	fc1, fully connected	

In this experiment, we build a three-layer CNN with 5×5 convolutional kernels, described in [Table 6.1](#). To help differentiate from the small kernel network introduced in the ablation study of the previous chapter, we have labelled the convolutions here ‘conv1’, ‘conv2’ and ‘conv3’ (as opposed to ‘convA’, ‘convB’, ‘convC’, …).

We test the difference in accuracy achieved by replacing each of the three convolution layers with gl31. Although the gain layers with no gain in the first scale and gain in the second scale (e.g. gl301, gl501) performed better than those with only gain in the first scale (e.g. gl31, gl51) in [subsection 6.4.1](#), we saw them perform consistently worse in the following ablation studies. This is not surprising, as a 5×5 convolutional kernel is too small compared to the width of the central lobe of scale 2 wavelets. For ease of presentation, we only show the results from the single scale gain layer gl31.

On the two CIFAR datasets, we train for 120 epochs, decaying learning rate by a factor of 0.2 at 60, 80 and 100 epochs, and for the Tiny ImageNet dataset, we train for 45 epochs, decaying learning rate at 18, 30 and 40 epochs. We set ℓ_2 weight decay of 10^{-4} for the real

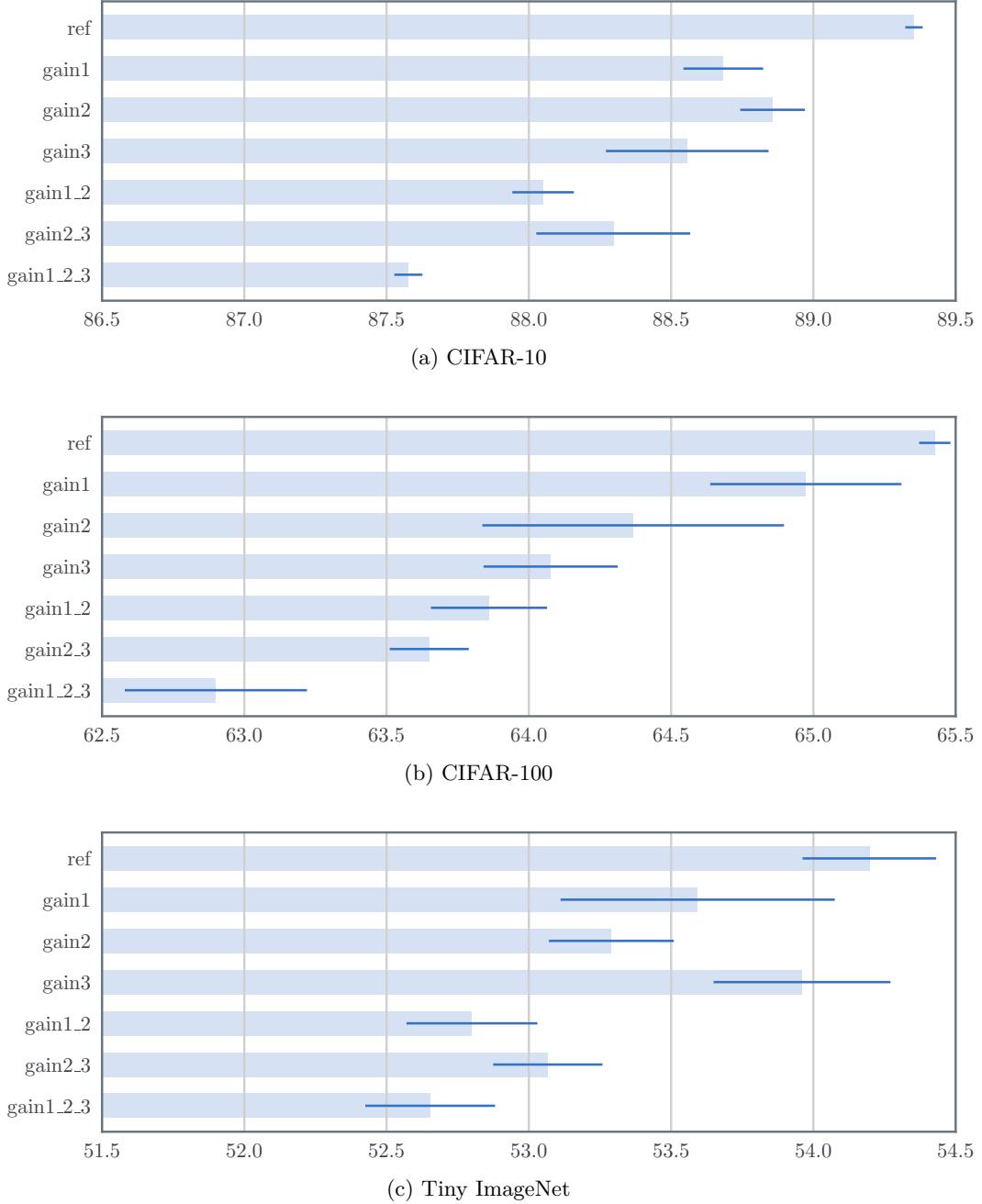


Figure 6.8: **Large kernel ablation results for CIFAR and Tiny ImageNet.** Results showing the percentage classification accuracies obtained by swapping combinations of the three conv layers in the reference architecture from Table 6.1 with gain layers. Results shown are averages of 3 runs with the ± 1 standard deviations shown as dark blue lines. These results show that changing a convolutional layer for a gain layer is possible, but comes with a small accuracy cost which compounds as more layers are swapped.

gains in the system, and ℓ_1 weight decay of 10^{-5} for the complex gains in the system. See section E.2 for information on how we handle regularizing complex gains. The experiment code is available at [33].

The results of various combinations for our three datasets are shown in Figure 6.8. Note that as before, swapping ‘conv1’ with a gain layer is marked by ‘gain1’, and swapping the first two conv layers with two gain layers is marked by ‘gain1_2’ and so forth.

The results are not too promising. Across all three datasets, changing a convolutional layer for a gain layer of similar number of parameters results in a small decrease in accuracy at all depths, and the more layers swapped out the more this degradation compounds.

6.4.3 Network Analysis

It is nonetheless interesting to see that a network with only gain layers (‘gain1_2_3’) can achieve accuracies within a couple of percentage points of a purely convolutional architecture.

In this section, we look at some of the properties of the ‘gain1_2_3’ layer for CIFAR-100 and compare them to the reference architecture.

6.4.3.1 Bandpass Coefficients

When analyzing the ‘gain1_2_3’ architecture, the most noticeable thing is the distribution of the bandpass gain magnitudes. Figure 6.9a shows these for the second gain layer, gain2. Of the $64 \times 128 = 8192$ complex coefficients most have very small magnitude, in particular, the diagonal wavelet gains. The disparity between the diagonal and horizontal/vertical wavelet gain distributions echoes the observation made in subsection 4.6.2, where we saw that occluding diagonal subbands reduced classification performance by less than when occluding horizontal/vertical subbands.

Although the diagonal subbands are particularly small, the horizontal and vertical bands have many coefficients with magnitude very close to zero. This raises an interesting question – how many of these coefficients are important for classification? What if we were to apply a hard thresholding scheme to the weights, would setting some of these values to zero impact the network performance?

We measure the dropoff in accuracy when a hard threshold t is applied to the bandpass gains g_1 for the three gain layers of ‘gain1_2_3’. The resulting sparsity of each layer and the network performance is shown in Figure 6.9b. For example, if we set a hard threshold value $t = 0.4$, only 20% of the gain1 weights, 1% of the gain2 and 0.02% of the gain3 weights remain non-zero, yet the classification accuracy has only dropped by 0.5%.

This figure shows that despite the high cost of the bandpass gains – $12C_lC_{l+1}$ for a 1×1 gain, very few of these need to be nonzero. This may mean that we can use a low-rank expansion to compress a lot the processing on the bandpass, reducing both its computational and memory cost. For example, it may be sufficient to use a bottleneck layer like the ones

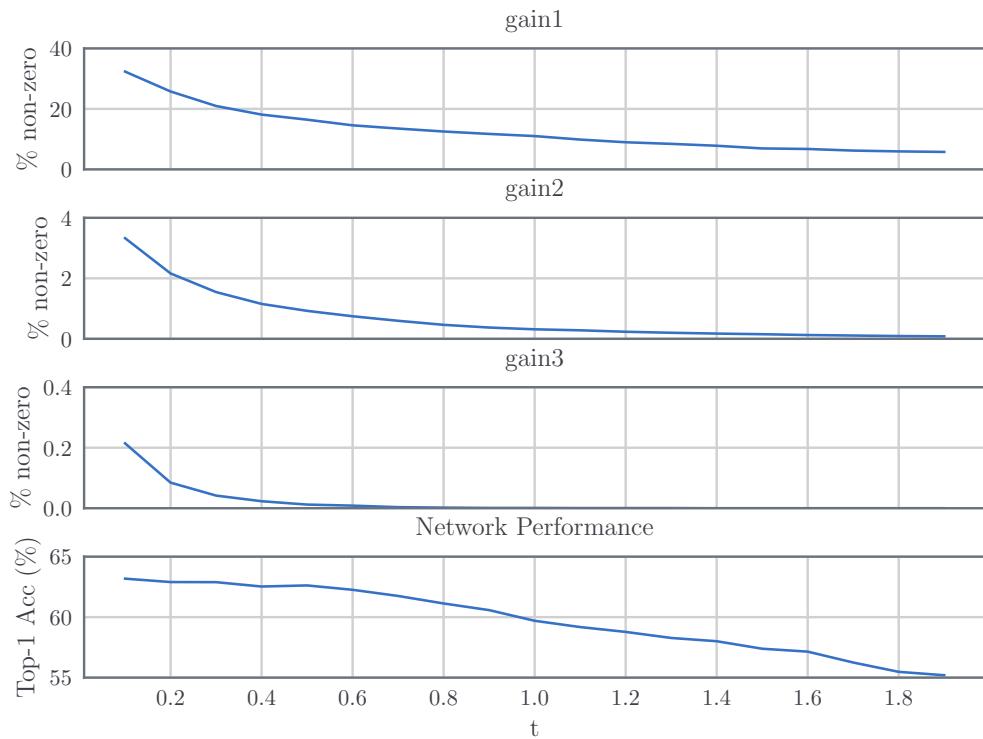
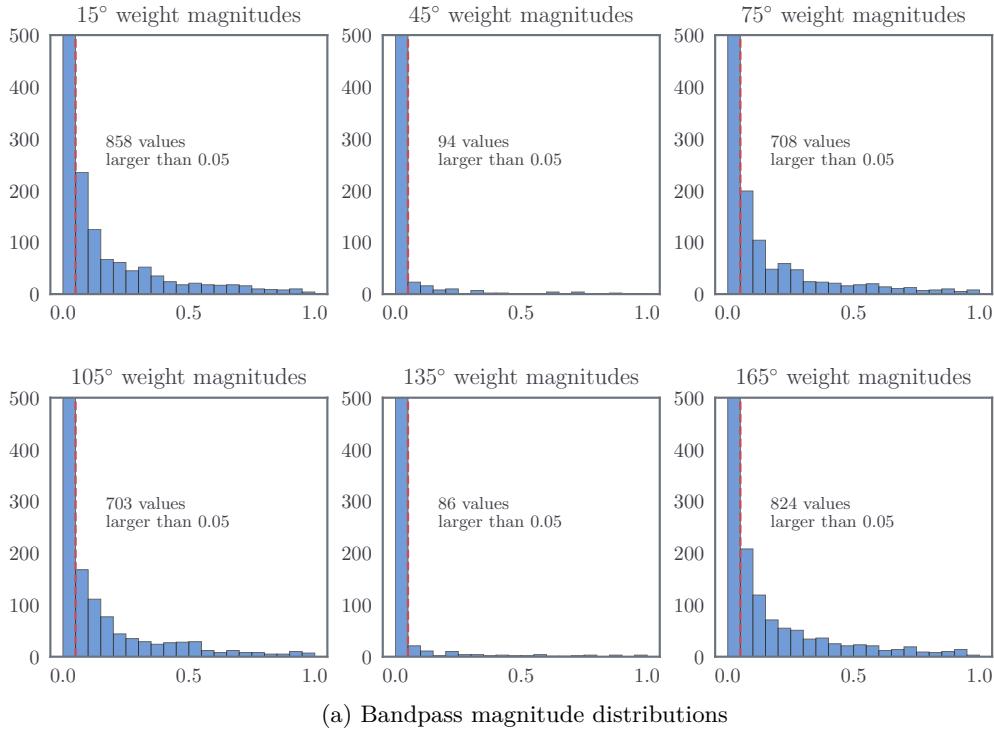


Figure 6.9: Bandpass gain properties for network with only gain layers. (a) shows the distribution of the magnitudes for bandpass coefficients for the second layer (gain2). Each orientation has $128 \times 64 = 8192$ complex weights, most of which are close to 0 (ℓ_1 regularization was used in training); Y axis for each plot has been clipped to 500. The 45° and 135° weights have many fewer large coefficients. (b) shows the increase in sparsity and dropoff in classification accuracy when the weights are hard-thresholded with value t (same threshold applied to all 3 layers). For a threshold value of $t = 0.4$, 80% of the weights in gain1 are 0, 99% of the weights in gain2 are 0, 99.98% of the weights in gain3 are 0, yet classification accuracy is only 0.5% lower than the non-thresholded accuracy.

seen in ResNet [15]. Instead of having a bandpass gain $g \in \mathbb{C}^{C_{l+1} \times C_l \times k \times k}$, we would use a compression layer with shape $g_c \in \mathbb{C}^{M \times C_l \times k \times k}$ followed by an expansion layer with shape $g_e \in \mathbb{R}^{C_{l+1} \times M \times 1 \times 1}$ with $M \ll C_{l+1}$.

6.4.3.2 DeConvolution and Filter Sensitivity

To visualize what the gain layer is responsive to, we build a deconvolutional system similar to the one described in chapter 4. In particular, we present the entire CIFAR-100 validation set to the reference architecture and to the gain1_2_3 architecture, keeping track of what most highly excites each channel. Once we have this information, we present the same image again, storing the ReLU switches and max pooling locations for this same image, then we zero out all but a single value for the given channel, and zero out all other channels, and deconvolve to see the input pattern.

The resulting visualizations for the first two layers are shown in Figure 6.10. We show only the top activation for each filter, rather than the top-9. For the second layer filters, we show only 64 of the 128 filter responses.

It is reassuring to see that despite the performance difference between the reference architecture and the gain1_2_3 architecture, the filters are responding to similar shapes. Note that for both the first and second layer responses, the gain layer has a smoother roll-off at the edges of the visualization, whereas the convolutional architecture has more blocky regions of support.

6.5 Wavelet-Based Nonlinearities

Returning to the goals from subsection 6.2.4, the experiments from the previous section have shown that while it is possible to use a wavelet gain layer (G) in place of a convolutional layer (H), this may come with a small performance penalty. Ignoring this effect for the moment, in this section, we continue with our investigations into learning in the wavelet domain. In particular, is it possible to replace a pixel domain nonlinearity σ with a wavelet-based one σ_w ?

But what sensible nonlinearity should be used? Two particular options are good initial candidates:

1. The ReLU: this is a mainstay of most modern neural networks and has proved invaluable in the pixel domain. Its pseudo nonlinearity ($\text{ReLU}(Ax) = A\text{ReLU}(x)$) makes learning less dependent on signal amplitudes. Perhaps its sparsifying properties will work well on wavelet coefficients too.
2. Thresholding: a technique commonly applied to wavelet coefficients for denoising and compression. Many proponents of compressed sensing and dictionary learning even like to compare soft thresholding to a two-sided ReLU [148], [149].

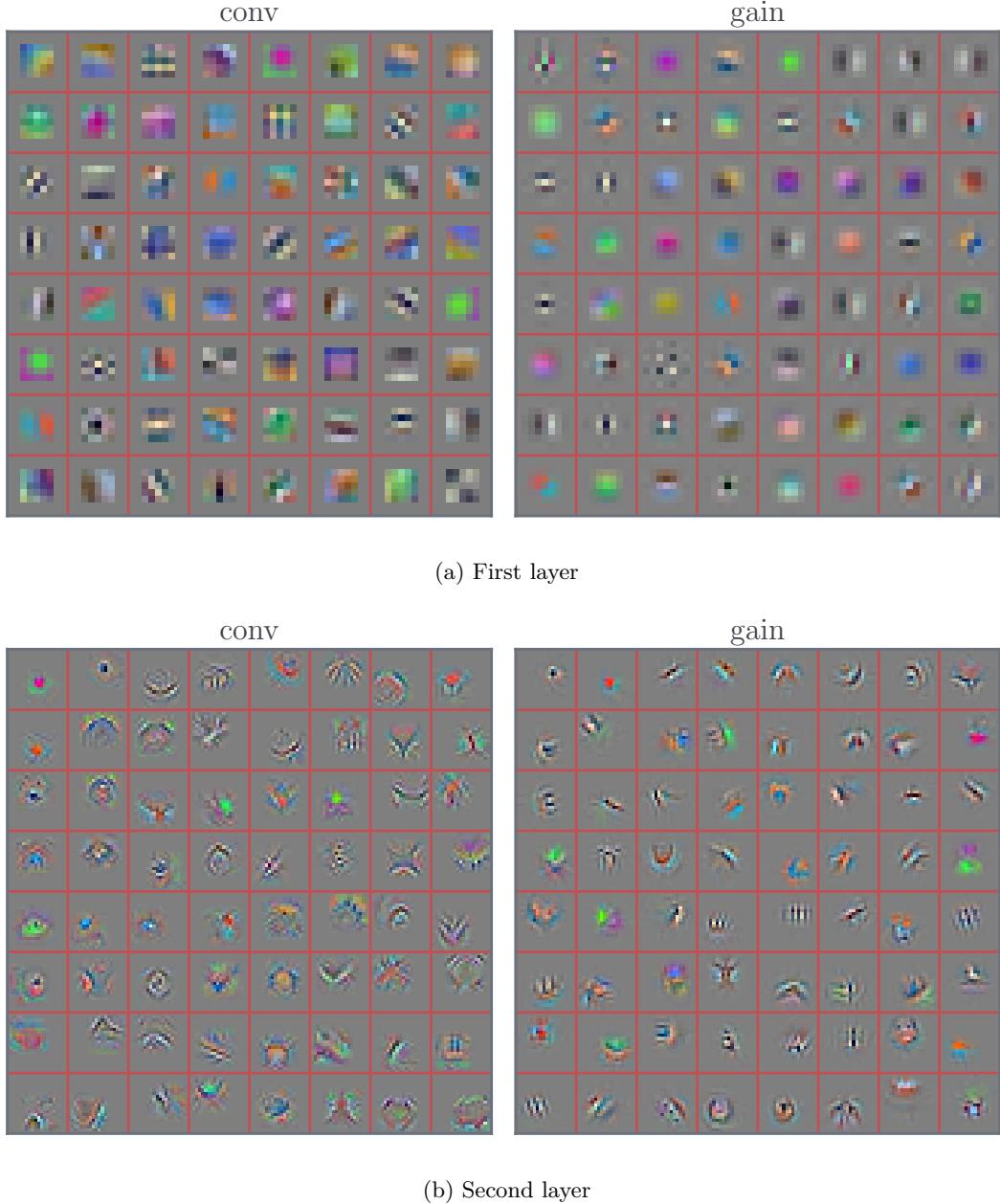


Figure 6.10: Deconvolution reconstructions for the reference architecture and purely gain layer architecture. Visualizations using a DeConvNet method similar to the one described in chapter 4. Here we find the input images in CIFAR-100 validation set that most highly activate each filter. Each image is then re-shown to the network and the meta-information is used to prime the DeConvNet to create the visualizations seen here. The left column has visualizations for the first and second layer filters for the all convolutional method, and the right column has visualizations for the first and second layer filters for the all gain layer method. Note the smoother roll-off at the edge of visualizations in the gain layer compared to the rectangular support regions for the conv layers. Aside from that, the two networks appear to be learning similar shapes.

In this section, we will look at both and see if they improve the gain layer. If they do, it would be possible to connect multiple layers in the wavelet domain, avoiding the necessity to do inverse wavelet transforms after learning.

6.5.1 ReLUs in the Wavelet Domain

Applying the ReLU to the real lowpass coefficients is not difficult, but it does not generalize so easily to complex coefficients. The simplest option is to apply it independently to the real and imaginary coefficients, effectively only selecting one quadrant of the complex plane:

$$u_{lp} = \max(0, v_{lp}) \quad (6.5.1)$$

$$u_j = \max(0, \operatorname{Re}(v_j)) + j\max(0, \operatorname{Im}(v_j)) \quad (6.5.2)$$

Another option is to apply it to the magnitude of the bandpass coefficients. Of course, these are all strictly positive so the ReLU on its own would not do anything. However, they can be arbitrarily scaled and shifted by using a batch normalization layer. Then the magnitude could shift to (invalid) negative values, which can then be rectified by the ReLU.

Dropping the scale subscript j for clarity (we need it for the square root of negative 1), let a bandpass coefficient at a given scale be $v = r_v e^{j\theta_v}$ and define $\mu_r = \mathbb{E}[r_v]$ and $\sigma_r^2 = \mathbb{E}[(r_v - \mu_r)^2]$, then applying batch-normalization and the ReLU to the magnitude of v_j means we get:

$$r_u = \operatorname{ReLU}(BN(r_v)) = \max\left(0, \gamma \frac{r_v - \mu_r}{\sigma_r} + \beta\right) \quad (6.5.3)$$

$$u = r_u e^{j\theta_v} \quad (6.5.4)$$

This also works equivalently on the lowpass coefficients, although v_{lp} can be negative unlike r_v :

$$u_{lp} = \operatorname{ReLU}(BN(v_{lp})) = \max\left(0, \gamma' \frac{v_{lp} - \mu_{lp}}{\sigma_{lp}} + \beta'\right) \quad (6.5.5)$$

6.5.2 Thresholding

For $t \in \mathbb{R}$ and $z = re^{j\theta} \in \mathbb{C}$ the pointwise hard thresholding is:

$$\mathcal{H}(z, t) = \begin{cases} z & r \geq t \\ 0 & r < t \end{cases} \quad (6.5.6)$$

$$= \mathbb{I}(r > t)z \quad (6.5.7)$$

and the pointwise soft thresholding is:

$$\mathcal{S}(z, t) = \begin{cases} (r - t)e^{j\theta} & r \geq t \\ 0 & r < t \end{cases} \quad (6.5.8)$$

$$= \max(0, r - t)e^{j\theta} \quad (6.5.9)$$

Note that (6.5.9) is very similar to (6.5.3) and (6.5.4). We can rewrite (6.5.3) by taking the strictly positive terms γ, σ outside of the max operator:

$$r_u = \max(0, \gamma \frac{r_v - \mu_r}{\sigma_r} + \beta) \quad (6.5.10)$$

$$= \frac{\gamma}{\sigma_r} \max\left(0, r_v - \left(\mu_r - \frac{\sigma_r \beta}{\gamma}\right)\right) \quad (6.5.11)$$

then if $t' = \mu_v - \frac{\sigma_r \beta}{\gamma} > 0$, **doing batch normalization followed by a ReLU on the magnitude of the complex coefficients is the same as soft shrinkage with threshold t' , scaled by a factor $\frac{\gamma}{\sigma_r}$.**

The same analogy does not apply to the lowpass coefficients, as v_{lp} is not strictly positive.

While soft thresholding is similar to batch normalizations and ReLUs, we would also like to test how well it performs as a sparsity-inducing wavelet nonlinearity. To do this, we can:

- Learn the threshold t
- Adapt t as a function of the distribution of activations to achieve a desired sparsity level.

In early experiments, we found that trying to set desired sparsity levels by tracking the standard deviation of the statistics and setting a threshold as a function of it performed very poorly (causing a drop in top-1 accuracy of at least 10%). Instead, we choose to learn a threshold t . We make this an unconstrained optimization problem by changing (6.5.9) to:

$$\mathcal{S}(v, t) = \max(0, r - |t|)e^{j\theta} \quad (6.5.12)$$

Learning a threshold is only possible for soft thresholding, as $\frac{\partial L}{\partial t}$ is not defined for hard thresholding. Like batch normalization, we learn independent thresholds t for each channel.

6.6 Gain Layer Nonlinearity Experiments

Taking the same ‘gain1_2_3’ architecture used for CIFAR-100, we expand the *wavelet gain layer* by including nonlinearities as described in [Algorithm 6.1](#). In this layer, we have three different nonlinearities: the pixel, the lowpass, and the bandpass nonlinearity.

For these experiments, we test over a grid of possible options for these three functions: where:

Algorithm 6.1 The *wavelet gain layer* pseudocode

```

1: procedure WAVEGAINLAYER( $x$ )
2:    $u_{lp}, u_1 \leftarrow \text{DTCWT}(x, \text{nlevels} = 1)$ 
3:    $v_{lp}, v_1 \leftarrow G(u_{lp}, u_1)$                                  $\triangleright$  the normal gain layer
4:    $u_{lp} \leftarrow \sigma_{lp}(v_{lp})$                                  $\triangleright$  lowpass nonlinearity
5:    $u_1 \leftarrow \sigma_{bp}(v_1)$                                  $\triangleright$  bandpass nonlinearity
6:    $y \leftarrow \text{DTCWT}^{-1}(u_{lp}, u_1)$ 
7:    $x \leftarrow \sigma_{pixel}(y)$                                  $\triangleright$  pixel nonlinearity
8:   return  $x$ 
9: end procedure

```

Nonlinearity	Options		
Pixel	None	BN+ReLU	
Lowpass	None	ReLU	BN+ReLU \mathcal{S}
Bandpass	None	ReLU	BN+MagReLU \mathcal{S}

- ‘None’ means no nonlinearity: $\sigma(x) = x$.
- ‘ReLU’ is a ReLU without batch normalization. For real values, is a normal ReLU, for complex values is a ReLU applied independently to real and imaginary parts, i.e. (6.5.2). See section E.3 for equations for the passthrough gradients for this nonlinearity.
- ‘BN+ReLU’ is batch normalization and ReLU (applicable only to real-valued activations) e.g. (6.5.5).
- ‘BN+MagReLU’ applies batch normalization to the magnitude of complex coefficients and then makes them strictly positive with a ReLU. This action is defined in (6.5.3). See section E.5 for information on the passthrough and update equations for this nonlinearity.
- \mathcal{S} is the soft thresholding of (6.5.12) applied to the magnitudes of coefficients with learnable thresholds. See section E.4 for information on the passthrough and update equations for this nonlinearity.

As the pixel nonlinearity has only two options, the results are best displayed as a pair of tables, firstly for no nonlinearity and secondly for the standard batch normalization and ReLU. See Table 6.2 for these two tables.

Digesting this information gives us some useful insights:

1. It is possible to improve on the gain layer from the previous experiments with the right nonlinearities. The previous section’s gain layer corresponds to $\sigma_{lp} = \sigma_{bp} = \text{None}$ and $\sigma_{pixel} = \text{ReLU}$, or the top left entry of Table 6.2b.

Table 6.2: **Different Nonlinearities in the Gain Layer.** Top-1 Accuracies for the ‘gain1_2_3’ network trained on CIFAR-100 using different wavelet and pixel nonlinearities. The rows of the table correspond to different bandpass nonlinearities and the columns correspond to different lowpass nonlinearities. $\sigma_{pixel} = \sigma_{lp} = \sigma_{bp} = \text{None}$ is a linear system (with max pooling). $\sigma_{pixel} = \text{ReLU}$, $\sigma_{lp} = \sigma_{bp} = \text{None}$ is the system used in earlier experiments, which is linear in the wavelet domain and has a nonlinearity in the pixel domain. The best result is highlighted in bold corresponding to $\sigma_{pixel} = \text{None}$, $\sigma_{lp} = \text{ReLU}$ and $\sigma_{bp} = \text{BN + ReLU}$.

		(a) $\sigma_{pixel} = \text{None}$				
		σ_{lp}	None	ReLU	BN+ReLU	\mathcal{S}
σ_{bp}		None	45.0	63.8	64.8	61.9
None		ReLU	42.6	62.8	63.9	61.3
ReLU		BN+MagReLU	48.5	65.3	64.6	62.9
BN+MagReLU		\mathcal{S}	44.6	63.9	63.6	61.7

		(b) $\sigma_{pixel} = \text{BN + ReLU}$				
		σ_{lp}	None	ReLU	BN+ReLU	\mathcal{S}
σ_{bp}		None	62.8	60.7	64.2	63.0
None		ReLU	62.5	60.3	64.2	63.2
ReLU		BN+MagReLU	64.9	61.6	64.7	65.2
BN+MagReLU		\mathcal{S}	63.3	60.9	63.9	63.3

2. Doing a ReLU on the real and imaginary parts of the bandpass coefficients independently (the second row of both tables) almost always performs worse than having no nonlinearity (first row of both tables).
3. The best combination is to have batch normalization and a ReLU applied to the magnitudes of the bandpass coefficients and batch norm and a ReLU applied to either the lowpass or pixel coefficients with no nonlinearity in the pixel domain.

The best accuracy score of 65.3% is now 0.1% lower than the fully convolutional architecture, an improvement from the 62.8% score achieved with only a pixel nonlinearity. This happens when there is no pixel nonlinearity, use a ReLU on the lowpass coefficients and Batch Normalization and a ReLU on the magnitudes of the bandpass coefficients.

6.6.1 Ablation Experiments with Nonlinearities

Now that we have found the best nonlinearity to use for the gain layer, will this improve our ablation study from subsection 6.4.2? To test this, we repeat the same experiment on

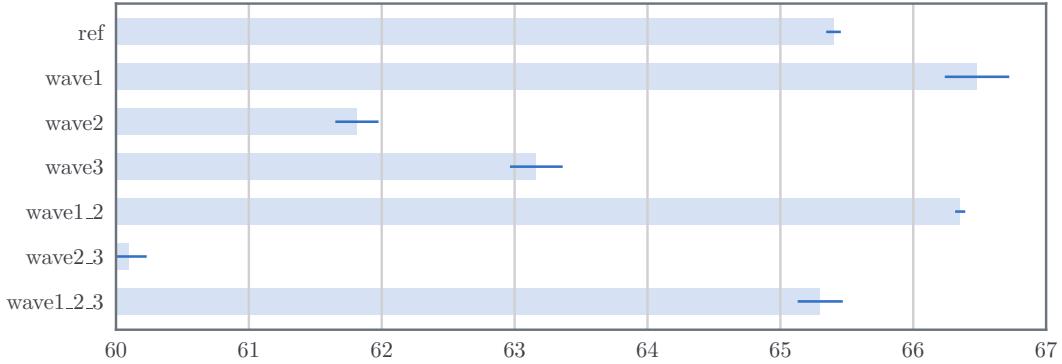


Figure 6.11: **CIFAR-100 Ablation results with the gain layer.** We use the same naming scheme from subsection 6.4.2 but to differentiate between the results from Figure 6.8b we call the options ‘waveX’. Results show the mean of 3 runs with ± 1 standard deviation lines in dark blue. When we add nonlinearities in the wavelet domain, the ablation results change dramatically. It appears that learning in the wavelet domain works best for the first layer of the CNN (wave1), and this improves on the purely convolutional method by a whole percentage point. Replacing the second and third layers degrades performance independently of what was used in the first layer. Swapping the first two layers (wave1_2) performs nearly as well and with a slightly narrower spread.

CIFAR-100 using the newly found nonlinearities in Algorithm 6.1 (i.e., $\sigma_{pixel} = \text{None}$, $\sigma_{lp} = \text{ReLU}$, and $\sigma_{bp} = \text{BN} + \text{ReLU}$).

See Figure 6.11 for the results from these experiments. When we use the wavelet-based nonlinearities, the results change considerably. We see an improvement by 1% when the first layer in the CNN is changed for a gain layer with a nonlinearity, but any other changes degrade performance from this.

6.7 Conclusion

In this chapter, we have presented the novel idea of learning filters by taking activations into the wavelet domain. There, we can apply the proposed wavelet *gain layer* G instead of a pixel-wise convolution. We can return to the pixel domain to apply a ReLU, or stay in the wavelet domain and apply wavelet-based nonlinearities σ_{lp}, σ_{bp} . We have considered the possible challenges this proposes and described how a multi-rate system can learn through backpropagation.

Our experiments have been promising but are still only preliminary. We have shown that the gain layer can learn in an end-to-end system, achieving nearly the same accuracies on CIFAR-10, CIFAR-100 and Tiny ImageNet to the reference system with convolutional layers (Figure 6.8). This is a good start and shows the plausibility of the wavelet gain layer, but more experiments on larger datasets and deeper networks is needed. Despite the slight reduction in performance, we saw some nice properties to the gain layer. Most of the bandpass

gains are near zero ([Figure 6.9](#)), which does not affect training but could offer speedups for inference. Additionally, doing deconvolution to visualize the sensitivity of the filters in a gain layer showed that the system was still learning sensible shapes with nice spatial roll-off properties ([Figure 6.10](#)).

We have searched for good candidates for wavelet nonlinearities, and saw that using a ReLU on the lowpass coefficients, and Batch Normalization and a ReLU on the magnitudes of the bandpass coefficients improved the performance of the gain layer considerably. This is an exciting development and indicates that we may not need to return to the pixel domain at all, possibly eliminating the need for the inverse wavelet transforms used in our experiments (see steps 6 and 7 in [Algorithm 6.1](#)). However, one needs to be careful, as taking the inverse transform followed by taking the forward transform does not necessarily give the same wavelet coefficients due to the redundancy of the DT \mathbb{C} WT, instead projecting onto the range space of the transform. Removing the inverse transform is something we did not have time to fully explore and so we have included it in our future work section.

When we added the nonlinearities to the gain layer we saw that we were able to achieve some improvements in performance over a fully convolutional architecture ([Figure 6.11](#)). The proposed layer worked best at the beginning of the CNN, which matches the intuition for doing this work described in the introduction to the chapter. More research still needs to be done with the gain layer as part of a deeper system.

Chapter 7

Conclusion and Further Work

In this thesis, we have examined the effectiveness of using complex wavelets as basis functions for deep learning models. We summarize the key results we have found before describing possible future extensions of our work. Some of these are natural expansions which we were not able to explore due to time constraints, and some are propositions for tying together interesting pieces of work that were developed towards the end of the project.

7.1 Summary of Key Results

Chapter 3 shows how using the separable spatial implementation of the DTCWT as the chosen wavelets for the original ScatterNet design greatly speeds up computation compared with Fourier domain implementations of complex wavelets (see [Table 3.3](#)). We derived the backpropagation equations for wavelet and scattering layers, aiding the use of ScatterNets as part of deep networks (this was crucial for our work of [chapter 5](#)). As part of this, we tested the performance of the DTCWT-based ScatterNet as a front end to a simple CNN for some small classification tasks, comparing its performance to a Morlet based system. We found that as well as being faster, the performance was often better when using the DTCWT wavelets [Table 3.5](#). When doing these tests, we found that of the wavelet choices available for the DTCWT, those with the fewest taps (and hence wider transition bands and worse stopband rejection) performed the best. This is somewhat surprising and while we were not able to investigate why due to time constraints, it may provide some interesting insights as to what the CNN backend is doing. It also may have been a side effect of the small image sizes used in the classification task.

Chapter 4 builds a visualization tool we call the DeScatterNet. We use this to interrogate the input patches that most highly excite the ScatterNet outputs on a subset of images from ImageNet (see [Figure 4.3](#)). We saw that the second-order scattering coefficients are most highly excited by ripple and checkerboard-like patterns. These were very different to the patterns that most highly excite filters of a CNN. We believe this may explain why ScatterNets

perform well on texture discrimination [23] but less well on natural image classification [25]. We then performed some occlusion tests on a hybrid network with a ScatterNet front end and CNN backend and saw that the CNN was able to operate with little degradation when the second-order scattering coefficients were zeroed out (there was only a small drop in classification accuracy), but it suffered greatly when the zeroth or first-order coefficients were removed. We also found the surprising result that on the datasets we tested, the filters with diagonal edges were less important than their vertical or horizontal counterparts (Figure 4.4). If the input images were rotated by $\pm 30^\circ$ then the diagonal channels became the most important. This echoes the experiments of Blakemore and Cooper[7] who controlled the orientation of edges exposed to kittens in their development stage. Finally, this chapter showed some ways to expand on the ScatterNet network and shows the features possible with these extensions in Figure 4.7. This last section inspires the work of chapter 5 and chapter 6.

Chapter 5 reworks the ScatterNet into individual layers. This redesign allows us to rethink how we want to use wavelets, and we introduce the *learnable ScatterNet* made up of *locally invariant convolutional layers*. Rather than applying the same layer twice to get a second order ScatterNet, we introduce mixing across the output channels, taken after the magnitude operation. The flexibility of the proposed layer means it can be used in a ScatterNet-like system, where the number of output channels grows exponentially with the number of layers, or in a CNN-like system, where the number of output channels remains mostly constant across layers. We experimented with both possibilities, showing that the extra learnability definitely helps the ScatterNet style system (Table 5.8) and CNN style networks (Figure 5.2). The demodulation of energy from the complex modulus means that the proposed locally invariant layer can only be used a few times. In particular, we saw that the layer performed best when used where a CNN would naturally downsample (or pool) the input.

Chapter 6 looks at learning in the wavelet space without taking complex magnitudes. We present the wavelet *gain layer* which takes inputs to the wavelet domain, learns complex gains to attenuate/accentuate different subbands, mixes the subbands across the different channels and offers the ability to return to the pixel domain with the inverse wavelet transform. Our experiments have been promising but are still only preliminary. We show that the *gain layer* can learn in an end-to-end system, achieving nearly the same accuracies on CIFAR-10, CIFAR-100 and Tiny ImageNet to a reference system with only convolutional layers (Figure 6.8). Despite the slight reduction in performance, we saw some nice properties of the gain layer. The bandpass gains were very sparse, needing very few non-zero coefficients and visualizations of what the layers were responding to showed that the gain layer had nice spatial roll-off properties (Figure 6.10). We then found using a ReLU on the lowpass coefficients, and Batch Normalization and a ReLU on the magnitudes of the bandpass coefficients improved the performance of the gain layer considerably (Table 6.2). Using this, we saw that we were able to achieve some improvements in performance over a fully convolutional architecture

(Figure 6.11). The gain layer with nonlinearity seems to work best at the beginning of the CNN but more research still needs to be done with deeper networks.

7.2 Future Work

This thesis has started to examine ways of using wavelets as basis functions for deep learning models. Our research has found some possible approaches which offer some advantage in terms of number of parameters, interpretability and (theoretical) computation time. But there are many things that we were not able to try, and some of these may show that wavelets have a larger benefit than we were able to find.

7.2.1 Faster Transforms and More Scales

Firstly, despite our best efforts in making a fast wavelet transform, the speed of a DT^CWT in *Pytorch Wavelets* is slower than we believe it ought to be. A 10×10 convolutional filter with 100 multiplies per input pixel is often twice as quick to compute than the DT^CWT with 36 multiplies per pixel. We limited our design to use high-level cuDNN calls and this was the best we could do with these primitives, and believe that any further speed up would require custom CUDA kernels. The computational time was not a problem for datasets such as CIFAR and Tiny ImageNet, but it did prevent us from testing the wavelet gain layer and invariant layer on ImageNet (see appendix A for some run times). We believe that these layers would perform better with larger images where the extent of the wavelet is not comparable to the size of the image (hence requiring lots of padding).

Another aspect of testing larger images is the benefit of using multiple scales in any system. Our wavelet gain layer only used the first or second scale in our experiments, but the real benefit of decimated wavelet transforms is the speedup they offer by allowing for multiscale approaches. Little research has been done in splitting the input or mid-level activations into multiple scales and learning different filters for the different scales, but some examples include [144], [150].

7.2.2 Expanding Tests on Invariant Layers

We have shown in chapter 5 that the invariant layer can work quite well in a VGG-like CNN as a replacement for a convolutional layer followed by pooling. We would have liked to test this on larger networks, replacing areas of sample rate change with invariant layers, and see how well this generalizes. One common location that has a large sample rate change is in the first layer of CNNs, with networks like AlexNet and ResNet downsampling by a factor of four in each direction after the first layer. Experiments by Oyallon, Belilovsky, and Zagoruyko [39] have looked at replacing this with a fixed ScatterNet, but it would be interesting to see how well this performs with the *learned* ScatterNet we have developed.

7.2.3 Expanding Tests on Gain Layer

We believe that the wavelet gain layer of [chapter 6](#) is the most unique and potentially promising piece of work in this thesis. The results we describe in [Table 6.2](#) were the last experiments we were able to do, and they just start to show some promise for learning in the wavelet domain. There is potentially a lot more research to be done on the wavelet gain layer.

Firstly, the result from [Figure 6.9](#) is very intriguing. This figure shows that a lot of the learned bandpass gains are very near zero, and thresholding them after training does little to affect performance. It would be very interesting to see the performance of the network if the threshold values were set near the end of training and the nonzero gains were allowed to grow to compensate for this. Also, is this a property unique to the wavelet gain layer? Or do all CNNs learn *mostly* lowpass filters, and need only a few bandpass filters? The parameter cost of the proposed gain layer was dominated by the cost of coding these bandpass gains, despite requiring very few of them. It would be an interesting piece of work to try to redesign the gain layer to allow training with many fewer bandpass parameters.

Secondly, the results from [Table 6.2](#) show that the best performing gain layer had no pixel nonlinearity, a ReLU in the lowpass and a ReLU applied to the magnitude of the bandpass. This is very exciting as it opens up the possibility of staying in the wavelet domain longer. In our experiments, we would take inverse transforms and apply a NoOp in the pixel domain before returning to the wavelet domain. This is marginally different to staying in the wavelet domain, as it involves projecting from a redundant space to a non-redundant one. More experiments could be done to investigate whether this projection process is needed, and hence whether significant DT^CWT computation could be saved.

7.2.4 ResNets and Lifting

We briefly mentioned ResNets in [subsection 2.5.6](#) but did not study them in depth in this thesis. Interestingly, there are many similarities between ResNets and second generation wavelets, or the *lifting* framework [121], [151]. In a residual layer, the output is $y = \mathcal{F}(x) + x$ where for a lifting system, the layer is a two-port network defined by:

$$y_1 = \mathcal{F}(x_1) + x_2 \quad (7.2.1)$$

$$y_2 = \mathcal{G}(y_1) + x_1 \quad (7.2.2)$$

[Figure 7.1](#) shows the similarities between the two designs. The works [152], [153] both make the small modifications to the ResNet design to make a lifting style architecture. Gomez, Ren, Urtasun, *et al.* [152] do this to save memory on the activations for backpropagation (you do not need to save meta-information on the forward pass as you can regenerate activations on the backwards pass). Jacobsen, Smeulders, and Oyallon [153] extend on [152] and also explore merging images with linear interpolation in the activation space, reconstructing from

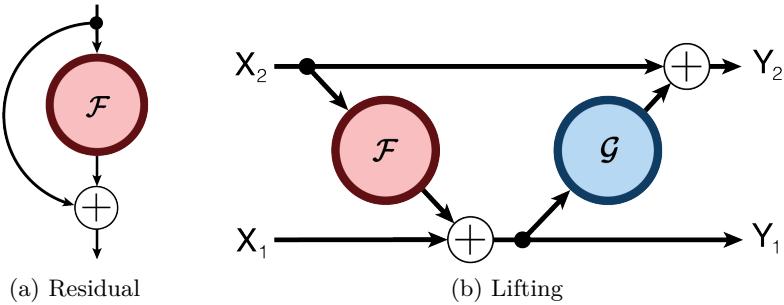


Figure 7.1: **Residual vs lifting layers.** In a residual layer, an input is transformed by a learned function \mathcal{F} and added to itself. Typically $\|\mathcal{F}x\| \ll \|x\|$ meaning the vector y is a small perturbation of the vector x . In a lifting layer, each path is a learned function of the other added to itself. In the original second generation wavelets, \mathcal{F} and \mathcal{G} are FIR filters, but there is no requirement on them to be. Diagrams are taken from [152].

the new latent values. We believe that there are potentially many more benefits to using the lifting design as an extension of our work into learning from basis functions.

7.2.5 Protecting against Attacks

Adversarial examples are starting to become a real concern for CNNs. A classic adversarial example is an image that appears innocuous to a human but has been corrupted with a low energy signal that can completely convince a CNN that the image is something else. Figure 7.2 shows an example of this taken from [22].

While Szegedy, Zaremba, Sutskever, *et al.* [22] show there is a weakness to CNNs being fooled, these adversarial examples were obtained by gradient ascent to the target class. I.e. they had full access to the parameters of the model. Papernot, McDaniel, Goodfellow, *et al.* [154] expand this to show it is possible to develop an attack without knowing the model, treating it as a *black box*. Protecting against these black box attacks has become an arms race in recent years, with measures and counter-measures constantly being developed. An excellent paper reviewing some attacks and defences is [155].

One such concerning black box attack is described in [156], where Engstrom, Tran, Tsipras, *et al.* show that even simple transformations such as translations and rotations are enough to fool many modern CNNs. This is something our invariant layer may be able to protect against.

Alternatively, a recent defence tactic is described in [157]. In this paper, Cisse, Bojanowski, Grave, *et al.* propose to have non-expansive convolutional and pooling layers (Lipschitz constant 1) and have weight matrices that are approximately Parseval tight frames, a result intimately related to the tight framed DTCWT. In addition to finding these networks more robust to adversarial examples, they show that they can train faster and have a better usage of the capacity of the networks.

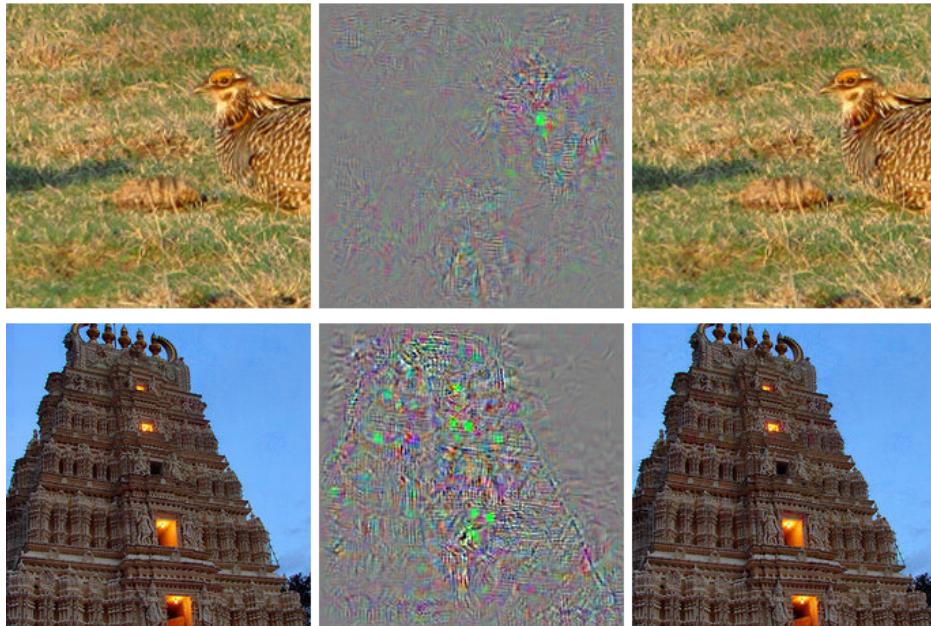


Figure 7.2: Adversarial examples that can fool AlexNet. Two examples of images that were correctly classified on the left, with additive signals in the centre (contrast levels were magnified by 10) and the resulting images on the right. Both of the output images are then predicted to be an ostrich. Images are taken from [22].

We feel that the learning of relatively smooth functions offered by wavelet bases has a potentially large scope for making CNNs more robust to many attacks.

7.2.6 Convolutional Sparse Coding

Recent work on convolutional sparse coding and convolutional dictionary learning [148], [149], [158]–[160] has started to draw many parallels between the structure of modern CNN architectures and the problem of sparse representations. We believe this is a valuable insight and can give a fresh new perspective on how we can better train CNNs.

It would be interesting to try to extend this recent work on convolutional sparse coding to wavelet bases. In [161] Rubinstein, Zibulevsky, and Elad attempted something similar with their double sparsity model, learning to sparsely combine atoms from a fixed base dictionary (they use wavelet, Fourier and DCT base dictionaries), although this was applied to block coding and has not been extended to convolutional sparse coding.

7.2.7 Weight Matrix Properties

Some recent work by Advani and Saxe [162] show that if you ignore the nonlinearities in a neural network, you can analyze the convergence properties by looking at the conditioning of the weight matrix (the ratio of the largest to the smallest singular values). In particular, well

conditioned matrices have better convergence, and the size of the *eigengap* (distance between the smallest non-zero eigenvalue and 0) is important to protect against overfitting.

This is something that we have not enforced or considered in the design of our wavelet-based layers, and it would be an interesting extension.

7.3 Final Remarks

It is our intuition that complex wavelets in a ScatterNet style system perform well in CNNs at locations where we want to reduce the sample rate, as they can nicely demodulate regions of the frequency space to lower frequencies. We also believe that using complex wavelets without taking the complex modulus is beneficial at locations where we want filters with large spatial support, something that is particularly useful in the early layers of CNNs. However, the current trend in CNNs is shifting away from these uses. Modern architectures typically build many layers with very small spatial support filters (usually 3×3 and often 1×1) and lots of mixing and combining of the channels. For example, the recent Wide ResNet [70] (one of the best modern methods), has close to 1000 channels in the later layers.

We believe that the work in this thesis has opened up a rich vein of ideas and a new perspective on modern CNN methods. We have found there to be some performance advantages to redesigning CNNs with complex wavelets as well as other, less measurable advantages, such as the ability to determine that certain orientations and frequency regions are less important than others (see subsection 4.6.2 and subsection 6.4.3) or the ability to have smooth roll-off in the support of filters. There is still much more work to be done; the learning efficiency of CNNs must be improved, as well as a greater understanding of their operation and outputs if they are to be widely used in the future.

Appendix A

Architecture Used for Experiments

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

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

To do hyperparameter search we used the Tune package [139] which we highly recommend, as it makes running trials in parallel very easy.

A.1 Run Times of some of the Proposed Layers

Throughout the main body of the thesis, we derive theoretical computational costs for many of our methods and compare these to convolutional operations. While this is useful to give a rough guide about the cost of our methods, we give experimental values here.

The numbers in the tables are calculated by running the specified input through our layer five times and then averaging the values. Timings were done by using NVIDIA's 'nvprof' command, which allows us to get millisecond timing on kernel execution times.

We test the effect of changing the spatial size for a constant batch and channel size in [Table A.1](#), and we test the effect of changing the channel dimension size for constant batch and spatial size in [Table A.2](#). Our reference is a 10×10 convolutional layer that does not do mixing across the channels. We compare the run time of this operation to each of our layers on an input of size $C \times H \times W$.

Using results from [section 3.5](#) (for the DTcWT and ScatterNet), [subsection 5.5.3](#) (for the invariant layer) and [subsubsection 6.3.4.3](#) (for the gain layer), the *theoretical* computational costs for the tested layers for an input with size $C \times H \times W$ are:

Table A.1: **Run time speeds for different layers with increasing spatial size.** Input size is $32 \times 32 \times H \times H$ where H is the column heading listed below. Run times are in milliseconds, averaged over five runs.

Spatial Size	16	32	64	128	256
Conv10x10	0.2	0.8	6.2	22.4	112
DT ^C WT	0.5	2.0	7.6	29.4	118
DT ^C WT ⁻¹	0.6	2.1	8.1	33.3	123
Scatter	0.6	2.1	8.7	31.8	125
Invariant	0.7	2.4	9.6	37.4	144
Gain ($J = 1$)	1.5	5.7	21.6	80	336

- **10×10 Convolution:** 100 multiplies per input pixel
- **DT^CWT with $J = 1$:** 36 multiplies per input pixel (see [Algorithm 3.3](#))
- **DT^CWT⁻¹ with $J = 1$:** 36 multiplies per input pixel (see [Algorithm B.2](#))
- **DT^CWT ScatterNet with $J = 1$:** 39 multiplies per input pixel (see [Algorithm 3.5](#))
- **Invariant Layer with square A matrix:** $\frac{7}{4}C + 36$ multiplies per input pixel (see [Algorithm 5.1](#))
- **DT^CWT Gain Layer with $J = 1$:** $7C + 72$ multiplies per input pixel (see [Algorithm 6.1](#))

While we were able to create a reasonably fast method for calculating the DT^CWT, it is still slower than what we believe it ought to be, with it often running 1 to 2 times slower than a 10×10 convolution. As it is the core for the other layers in this thesis, these are also affected.

Table A.2: **Run time speeds for different layers with increasing channel size.** Input size is $32 \times C \times 64 \times 64$ where C is the column heading listed below. Run times are in milliseconds, averaged over five runs.

Channel Size	3	10	32	64	128
Conv10x10	2	4.6	15.8	28.4	70
DTCWT	3.2	10.5	30.0	58.6	126
DTCWT ⁻¹	4.1	13.3	37.0	79.4	152
Scatter	3.4	11.0	31.4	65.8	133
Invariant	3.6	11.8	34.6	73.6	164
Gain Layer	9.7	28.4	79.4	158	371

Appendix B

Extra Proofs and Algorithms

We derive proofs for the gradients of decimation, interpolation, and for the forward 2-D DWT. These are needed for [subsection 3.4.3](#).

We have also listed some of the algorithms that are not included in the main text for the interested reader. In particular, the inverse DWT (used in [subsection 3.4.5](#)), the inverse DT \mathbb{C} WT (used in [section 3.5](#)), and the smooth magnitude operation (used in [subsection 3.6.1](#)).

B.1 Gradients of Sample Rate Changes

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

Decimation of a signal x by $M \in \mathbb{Z}$ is defined as:

$$y[n] = x[Mn] \quad (\text{B.1})$$

and interpolation by M as:

$$y[n] = \begin{cases} x[\frac{n}{M}] & n = Mk, k \in \mathbb{Z} \\ 0 & \text{otherwise} \end{cases} \quad (\text{B.2})$$

B.1.1 Decimation Gradient

From (B.1) the gradient $\frac{\partial y_n}{\partial x_k}$ is:

$$\frac{\partial y_n}{\partial x_k} = \begin{cases} 1 & k = Mn \\ 0 & \text{otherwise} \end{cases} \quad (\text{B.3})$$

By the chain rule, $\frac{\partial L}{\partial x_k}$ is:

$$\frac{\partial L}{\partial x_k} = \frac{\partial L}{\partial y_n} \frac{\partial y_n}{\partial x_k} \quad (\text{B.4})$$

$$= \begin{cases} \Delta y[\frac{k}{M}] & k = Mn \\ 0 & \text{otherwise} \end{cases} \quad (\text{B.5})$$

which is interpolating Δy by M from (B.2).

B.1.2 Interpolation Gradient

From (B.2) the gradient $\frac{\partial y_n}{\partial x_k}$ is:

$$\frac{\partial y_n}{\partial x_k} = \begin{cases} 1 & n = Mk \\ 0 & \text{otherwise} \end{cases} \quad (\text{B.6})$$

and then the gradient $\frac{\partial L}{\partial x_k}$ is:

$$\frac{\partial L}{\partial x_k} = \frac{\partial L}{\partial y_n} \frac{\partial y_n}{\partial x_k} \quad (\text{B.7})$$

$$= \begin{cases} \Delta y[n] & n = Mk \\ 0 & \text{otherwise} \end{cases} \quad (\text{B.8})$$

$$= \Delta y[Mk] \quad (\text{B.9})$$

which is decimation of Δy by M .

B.2 Gradient of Wavelet Analysis Decomposition

We mention in subsection 3.4.3 that the gradient of a forward DWT with orthogonal wavelets is just the inverse DWT. This easily follows from applying the chain rule and using the gradients of each of the stages of the DWT (convolution becomes correlation, downsampling becomes upsampling). The equivalent backward pass of Figure 3.1 is shown in Figure B.1. The algorithm for computing this is shown below in Algorithm B.1.

It is interesting to note that for an orthogonal wavelet transform, the synthesis filters are the time reverse of the analysis filters [87, Chapter 3]. This means that the blocks $H_0(z^{-1}), H_1(z^{-1})$ are $G_0(z), G_1(z)$ respectively, and the gradient of the forward DWT is the inverse DWT.

B.3 Extra Algorithms

Below are some algorithms referred to in the text that we have moved here for compactness.

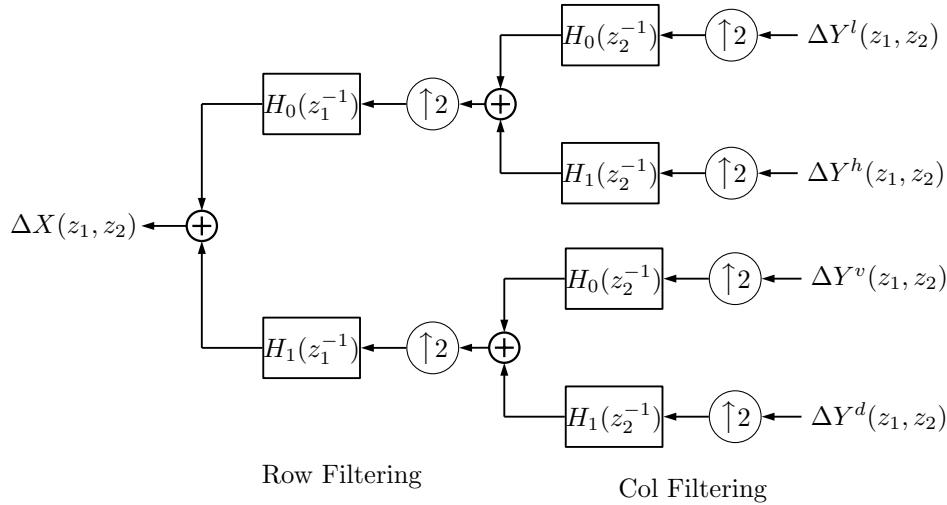


Figure B.1: **Gradient of DWT analysis.** Each block in the forward pass of Figure 3.1 has been swapped for its gradient. The resulting gradient has the same form as the inverse DWT.

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

```

1: function IDWT.FORWARD( $ll, lh, hl, hl, g_0^c, g_1^c, g_0^r, g_1^r, mode$ )
2:   save  $g_0^c, g_1^c, g_0^r, g_1^r, mode$                                  $\triangleright$  For the backwards pass
3:    $lo \leftarrow \text{sfb1d}(ll, lh, g_0^c, g_1^c, mode, axis = 2)$            $\triangleright$  See Algorithm 3.1
4:    $hi \leftarrow \text{sfb1d}(hl, hh, g_0^c, g_1^c, mode, axis = 2)$ 
5:    $x \leftarrow \text{sfb1d}(lo, hi, g_0^r, g_1^r, mode, axis = 3)$ 
6:   return  $x$ 
7: end function

1: function IDWT.BACKWARD( $\Delta y$ )
2:   load  $g_0^c, g_1^c, g_0^r, g_1^r, mode$ 
3:    $\Delta lo, \Delta hi \leftarrow \text{afb1d}(\Delta y, g_0^r, g_1^r, mode, axis = 3)$        $\triangleright$  See Algorithm 3.1
4:    $\Delta ll, \Delta lh \leftarrow \text{afb1d}(\Delta lo, g_0^c, g_1^c, mode, axis = 2)$ 
5:    $\Delta hl, \Delta hh \leftarrow \text{afb1d}(\Delta hi, g_0^c, g_1^c, mode, axis = 2)$ 
6:   return  $\Delta ll, \Delta lh, \Delta hl, \Delta hh$ 
7: end function

```

Algorithm B.2 2-D Inverse DTcwt

```

1: function IDTCWT( $yl$ ,  $yh$ , mode)
2:   load  $g_0^a$ ,  $g_1^a$ ,  $g_0^b$ ,  $g_1^b$                                  $\triangleright$  Load from memory
3:    $ll_r$ ,  $ll_{j_1}$ ,  $ll_{j_2}$ ,  $ll_{j_1j_2} \leftarrow$  deinterleave( $yl$ )
4:   for  $j = J$ ;  $j \geq 1$ ;  $j = j - 1$  do
5:      $x_{1b}, x_{3b}, x_{2b}, x_{2a}, x_{3a}, x_{1a} \leftarrow yh[j]$ 
6:      $lh_r \leftarrow \text{Re}(x_{1a} + x_{1b})$ 
7:      $lh_{j_1} \leftarrow \text{Im}(x_{1a} - x_{1b})$ 
8:      $lh_{j_2} \leftarrow \text{Im}(x_{1a} + x_{1b})$ 
9:      $lh_{j_1j_2} \leftarrow \text{Re}(x_{1b} - x_{1a})$ 
10:     $hl_r \leftarrow \dots$                                       $\triangleright$  Same procedure for  $hl$  and  $hh$ 
11:     $ll_r \leftarrow \text{IDWT}_r(ll_r, lh_r, hl_r, hh_r, g_0^a, g_1^a, g_0^a, g_1^a, mode)$ 
12:     $ll_{j_1} \leftarrow \text{IDWT}_r(ll_{j_1}, lh_{j_1}, hl_{j_1}, hh_{j_1}, g_0^b, g_1^b, g_0^b, g_1^b, mode)$ 
13:     $ll_{j_2} \leftarrow \text{IDWT}_r(ll_{j_2}, lh_{j_2}, hl_{j_2}, hh_{j_2}, g_0^a, g_1^a, g_0^b, g_1^b, mode)$ 
14:     $ll_{j_1j_2} \leftarrow \text{IDWT}_r(ll_{j_1j_2}, lh_{j_1j_2}, hl_{j_1j_2}, hh_{j_1j_2}, g_0^b, g_1^b, g_0^b, g_1^b, mode)$ 
15:   end for
16:   return  $(ll_r + ll_{j_1} + ll_{j_2} + ll_{j_1j_2}) / 2$ 
17: end function

```

Algorithm B.3 Smooth Magnitude

```

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

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

```

Appendix C

Invertible Transforms and Optimization

This Appendix proves that reparameterization of filters with an invertible transform can affect linear methods like SGD. This section is referenced from [subsection 6.2.1](#) in the main thesis.

We initially looked at this problem after seeing the claim in [147] that *any* invertible transform of the parameter space would not change the update equations for linear methods like SGD. In their work, they examine reparameterizing convolutional filters in the DFT space. The filters are taken into the pixel domain with the inverse DFT before regular convolution was applied. We wondered if this also applied to redundant representations like the DT \mathbb{C} WT.

We prove in this section that this statement only holds for tight frames. To prove this we follow some notation and theory from [20].

C.1 Background

Consider a pair of dual frames $\{\Phi, \tilde{\Phi}\}$ where $\tilde{\Phi}$ is the *analysis* operator and Φ is the *synthesis* operator. In $\mathbb{R}^n, \mathbb{C}^n$ $\tilde{\Phi}$ is an $n \times m$ matrix describing the frame change (with $m \geq n$), with the *dual* frame vectors as its columns. Similarly, Φ is an $n \times m$ matrix containing the frame vectors as its columns. The analysis and synthesis operations respectively are:

$$X = \tilde{\Phi}^* x \tag{C.1}$$

$$x = \Phi X \tag{C.2}$$

Where $\tilde{\Phi}^*$ is the Hermitian transpose of $\tilde{\Phi}$. As $\tilde{\Phi}$ is the dual of Φ , $\Phi\tilde{\Phi}^* = I_n$.

We prove that tight frame representations do not affect learning for linear optimizers by induction.

C.2 Proof

Consider a single filter parameterized in the pixel and frame space. In one system, the original filter parameters are updated. In a second system, the frame representation of them are updated. We want to track the evolution of the two filters and compare them when the same data are presented to them. We set them to have the same ℓ_2 regularization rate λ and the same learning rate η .

Let us call the weights at time t are \mathbf{w}_t , the frame-parameterized $\hat{\mathbf{w}}_t$ and we assume that:

$$\hat{\mathbf{w}}_t = \tilde{\Phi}^* \mathbf{w}_t \quad (\text{C.3})$$

It follows from (C.2) and our definition of $\hat{\mathbf{w}}_t$ that:

$$\frac{\partial L}{\partial \hat{\mathbf{w}}_t} = \frac{\partial \mathbf{w}_t}{\partial \hat{\mathbf{w}}_t} \frac{\partial L}{\partial \mathbf{w}_t} = \Phi^* \frac{\partial L}{\partial \mathbf{w}_t} \quad (\text{C.4})$$

After presenting both systems with the same minibatch of samples \mathcal{D} and calculating the gradient $\frac{\partial L}{\partial \mathbf{w}_t}$ we update both parameters:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \left(\frac{\partial L}{\partial \mathbf{w}_t} + \lambda \mathbf{w}_t \right) \quad (\text{C.5})$$

$$= (1 - \eta \lambda) \mathbf{w}_t - \eta \frac{\partial L}{\partial \mathbf{w}_t} \quad (\text{C.6})$$

$$\hat{\mathbf{w}}_{t+1} = (1 - \eta \lambda) \hat{\mathbf{w}}_t - \eta \frac{\partial L}{\partial \hat{\mathbf{w}}_t} \quad (\text{C.7})$$

If we left multiply (C.6) by the analysis operator we get:

$$\tilde{\Phi}^* \mathbf{w}_{t+1} = \tilde{\Phi}^* \left((1 - \eta \lambda) \mathbf{w}_t - \eta \frac{\partial L}{\partial \mathbf{w}_t} \right) \quad (\text{C.8})$$

$$= (1 - \eta \lambda) \hat{\mathbf{w}}_t - \eta \tilde{\Phi}^* \frac{\partial L}{\partial \mathbf{w}_t} \quad (\text{C.9})$$

In general, this does not reduce further. However, if $\tilde{\Phi} = \Phi$ as is the case with tight frames [20], then we can use (C.4) and this last line simplifies to:

$$\tilde{\Phi}^* \mathbf{w}_{t+1} = (1 - \eta \lambda) \hat{\mathbf{w}}_t - \eta \frac{\partial L}{\partial \hat{\mathbf{w}}_t} \quad (\text{C.10})$$

$$= \hat{\mathbf{w}}_{t+1} \quad (\text{C.11})$$

Which shows that they remain related at time $t+1$ given they were related at time t .

This proves the simpler case for SGD, but the same result holds when momentum terms are added due to the linearity of the update equations. This does not hold for the Adam [49]

or Adagrad [50] optimizers, which automatically rescale the learning rates for each parameter based on estimates of the parameter's variance.

We mention in [subsubsection 2.6.7.3](#) that when the DT \mathbb{C} WT uses orthogonal wavelet transforms, as is the case with the q-shift filters [95], then it forms a tight frame. If the biorthogonal filters are used (as is often the case for the first scale of the transform), it does not form a tight frame.

Appendix D

DTCWT Single Subband Gains

This appendix proves that the DTCWT *gain layer* proposed in [chapter 6](#) maintains the shift-invariant properties of the DTCWT.

Recall that with multirate systems, upsampling by M takes $X(z)$ to $X(z^M)$ and down-sampling by M takes $X(z)$ to $\frac{1}{M} \sum_{k=0}^{M-1} X(W_M^k z^{1/M})$ 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 .

D.1 Revisiting the Shift-Invariance of the DT^CWT

It is easiest to prove the shift-invariance of the gain layer by expanding on the shift-invariance of the DTCWT proofs done in [\[19\]](#).

Let us consider one subband of the DTCWT. This includes the coefficients from both tree A and tree B. For simplicity in this analysis we will consider the 1-D DT^CWT without the channel parameter c .

If we only keep coefficients from a given subband and set all the others to zero, then we have a reduced tree as shown in [Figure D.1](#). The output $Y(z)$ is:

$$Y(z) = \frac{1}{M} \sum_{k=0}^{M-1} X(W^k z) [A(W^k z)C(z) + B(W^k z)D(z)] \quad (\text{D.1})$$

where the aliasing terms are formed from the addition of the rotated z transforms, i.e. when $k \neq 0$.

As is standard for filter design in the real DWT, it is possible to make A and C have similar frequency responses. We can also make $A(W^{\pm 2} z)C(z)$ near zero if their stopbands can be made reasonably small. It is not possible however to make the terms $A(W^{\pm 1} z)C(z)$ zero, as the transition band of the shifted analysis filter $A(W^{\pm 1} z)$ overlap with those of the reconstruction filter $C(z)$. This leads us to our first theorem:

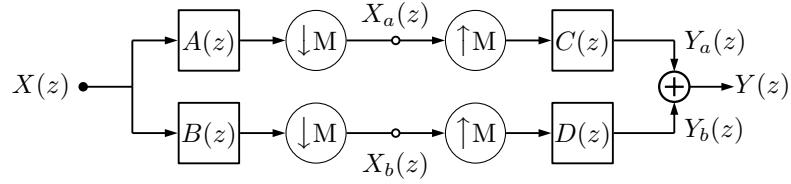


Figure D.1: **Filter bank diagram of 1-D DTCWT.** Note the top and bottom paths are through the wavelet or scaling functions from just level m ($M = 2^m$). Figure based on Figure 4 in [19].

Theorem D.1. *The odd k aliasing terms in (D.1) cancel out if the impulse responses of B and D are Hilbert transforms of the impulse responses of A and C respectively.*

Proof. See [19, section 4] for the full proof of this. The full cancellation of aliasing terms for all $k \neq 0$ makes the DTCWT nearly shift-invariant (also see [19, section 7] for the bounds on what ‘nearly’ shift-invariant means). \square

Now, consider the complex filters defined as:

$$P(z) = \frac{1}{2}(A(z) + jB(z)) \quad (\text{D.2})$$

$$Q(z) = \frac{1}{2}(C(z) - jD(z)) \quad (\text{D.3})$$

and define $P^*(z) = \sum_n p^*[n]z^{-n}$ as the Z -transform of p after taking the complex conjugate of the filter taps.

From this, we can rewrite the filters A, B, C and D as:

$$A(z) = P(z) + P^*(z) \quad (\text{D.4})$$

$$B(z) = -j(P(z) - P^*(z)) \quad (\text{D.5})$$

$$C(z) = Q(z) + Q^*(z) \quad (\text{D.6})$$

$$D(z) = j(Q(z) - Q^*(z)) \quad (\text{D.7})$$

Substituting these into (D.1) gives:

$$A(W^k z)C(z) + B(W^k z)D(z) = 2P(W^k z)Q(z) + 2P^*(W^k z)Q^*(z) \quad (\text{D.8})$$

This result is important as it shows that the P^*Q and PQ^* terms cancel out when BD is added to AC , which are the terms that would cause significant aliasing.

Using (D.2) and (D.3) Kingsbury showed that if B is the Hilbert pair of A then P has support only on the right-hand side of the frequency plane. Similarly, if D is the Hilbert pair of C then Q also has support only on the right-hand side of the frequency plane. If P and Q

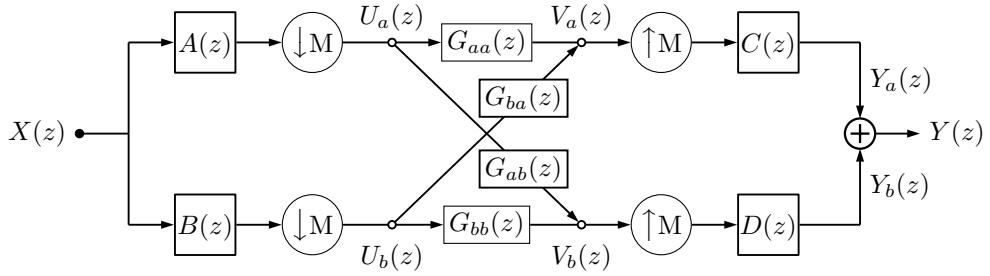


Figure D.2: **Filter bank diagram of 1-D DT-CWT with subband gains.**

are single-sideband, then so are P^* and Q^* , but they now have support only on the left-hand side of the frequency plane.

Given these properties, [19, figure 5] shows that the shifted versions of $P(W^k z)$ have negligible overlap with $Q(z)$ except for $k = 0$ (the wanted term) and $k = \pm 1$ where the transition bands overlap. Similarly, $P^*(W^k z)$ only overlaps with $Q^*(z)$ when $k = 0$ and a small amount for $k = \pm 1$. [19] quantifies the amount of transition band overlap and shows that it is negligible.

This means $A(W^k z)C(z) + B(W^k z)D(z) = 0$ when $k \neq 0$ and (D.1) reduces to:

$$Y(z) = \frac{1}{M}X(z)[A(z)C(z) + B(z)D(z)] \quad (\text{D.9})$$

D.2 Gains in the Subbands

Figure D.2 shows a block diagram of the extension of the above to general gains. This is a two port network with four individual transfer functions. Let the transfer function from U_i to V_j be G_{ij} for $i, j \in \{a, b\}$. Then V_a and V_b are:

$$V_a(z) = U_a(z)G_{aa}(z) + U_b(z)G_{ba}(z) \quad (\text{D.10})$$

$$= \frac{1}{M} \sum_k X(W^k z^{1/M}) [A(W^k z^{1/M})G_{aa}(z) + B(W^k z^{1/M})G_{ba}(z)] \quad (\text{D.11})$$

$$V_b(z) = U_a(z)G_{ab}(z) + U_b(z)G_{bb}(z) \quad (\text{D.12})$$

$$= \frac{1}{M} \sum_k X(W^k z^{1/M}) [A(W^k z^{1/M})G_{ab}(z) + B(W^k z^{1/M})G_{bb}(z)] \quad (\text{D.13})$$

Further, Y_a and Y_b are:

$$Y_a(z) = C(z)V_a(z^M) \quad (\text{D.14})$$

$$Y_b(z) = D(z)V_b(z^M) \quad (\text{D.15})$$

and the output is their sum:

$$Y(z) = Y_a(z) + Y_b(z) \quad (\text{D.16})$$

$$\begin{aligned} &= \frac{1}{M} \sum_{k=0}^{M-1} X(W^k z) \left[A(W^k z)C(z)G_{aa}(z^M) + B(W^k z)D(z)G_{bb}(z^M) + \right. \\ &\quad \left. B(W^k z)C(z)G_{ba}(z^M) + A(W^k z)D(z)G_{ab}(z^M) \right] \end{aligned} \quad (\text{D.17})$$

Theorem D.2. *If we let $G_{aa}(z) = G_{bb}(z) = G_r(z)$ and $G_{ab}(z) = -G_{ba}(z) = G_i(z)$ then the end to end transfer function is shift-invariant.*

Proof. Using the above substitutions, the terms in the square brackets of (D.17) become:

$$G_r(z^M) \left[A(W^k z)C(z) + B(W^k z)D(z) \right] + G_i(z^M) \left[A(W^k z)D(z) - B(W^k z)C(z) \right] \quad (\text{D.18})$$

Theorem D.1 already showed that the G_r terms are shift-invariant and reduce to $A(z)C(z) + B(z)D(z)$. To prove the same for the G_i terms, we follow the same procedure. Using our definitions of A, B, C, D from Theorem D.1 we note that:

$$A(W^k z)D(z) - B(W^k z)C(z) = j \left[P(W^k z) + P^*(W^k z) \right] [Q(z) - Q^*(z)] + \quad (\text{D.19})$$

$$j \left[P(W^k z) - P^*(W^k z) \right] [Q(z) + Q^*(z)] \quad (\text{D.20})$$

$$= 2j \left[P(W^k z)Q(z) - P^*(W^k z)Q^*(z) \right] \quad (\text{D.21})$$

Again, (D.21) shows that the P^*Q and PQ^* cancel out, removing the sources of aliasing. We note that the difference between the G_r and G_i terms is just in the sign of the negative frequency parts, i.e. $AD - BC$ is the Hilbert pair of $AC + BD$. To prove shift invariance for the G_r terms in Theorem D.1, we ensured that $P(W^k z)Q(z) \approx 0$ and $P^*(W^k z)Q^*(z) \approx 0$ for $k \neq 0$. We can use this again here to prove the shift invariance of the G_i terms in (D.18). This completes our proof. \square

Using Theorem D.2, the output is now

$$Y(z) = \frac{2}{M} X(z) [G_r(z^M)(AC + BD) + G_i(z^M)(AD - BC)] \quad (\text{D.22})$$

$$= \frac{2}{M} X(z) [G_r(z^M)(PQ + P^*Q^*) + G_i(z^M)(PQ - P^*Q^*)] \quad (\text{D.23})$$

where we have dropped the z terms on A, B, C, D, P, Q for brevity.

Theorem D.3. *If we treat the two subband coefficients as a complex value $U(z) = U_a(z) + jU_b(z)$ then doing a complex multiply by a gain $G(z) = G_r(z) + jG_i(z)$ maintains shift invariance.*

Proof. This follows from the conditions in [Theorem D.2](#). There we saw that we maintained shift invariance if $G_{aa}(z) = G_{bb}(z) = G_r(z)$ and $G_{ab}(z) = -G_{ba}(z) = G_i(z)$. If we consider V as a complex signal given by $V(z) = V_a(z) + jV_b(z)$, we can see from [Figure D.2](#) that the real and imaginary parts of V are:

$$V_a(z) = G_r(z)U_a(z) - G_i(z)U_b(z) \quad (\text{D.24})$$

$$V_b(z) = G_r(z)U_b(z) + G_i(z)U_a(z) \quad (\text{D.25})$$

which follows the form of a complex multiply. \square

Now if we can assume that our DT \mathbb{C} WT is well designed and extracts frequency bands at local areas, then our complex filter $G(z) = G_r(z) + jG_i(z)$ allows us to modify these passbands (e.g. by simply scaling if $G(z) = C$, or by more complex functions). The phase of the complex gain produces a phase shift of the underlying oscillation in the impulse response of this subband, and thus allows small spatial shifts to be achieved.

[Theorem D.3](#) and [\(D.23\)](#) give us an intuition for the real and imaginary parts of a complex gain G . The real part G_r affects how much of the bandpass gain $PQ + P^*Q^*$ propagates through, and the imaginary part G_i affects how much its Hilbert pair $PQ - P^*Q^*$ propagates.

Appendix E

Complex CNN Operations

This appendix lists some of the forward and backward equations for the complex operations we use in the *gain layer*.

E.1 Convolution

Let us represent the complex input with $\mathbf{x} = \mathbf{x}_r + j\mathbf{x}_i$, where \mathbf{x}_r and \mathbf{x}_i are the real and imaginary parts of x , and x is of shape $\mathbb{C}^{C \times n_1 \times n_2}$. Similarly, we call $\mathbf{y} = \mathbf{y}_r + j\mathbf{y}_i$ the result we get from convolving \mathbf{x} with $\mathbf{h} = \mathbf{h}_r + j\mathbf{h}_i$ where $\mathbf{h} \in \mathbb{C}^{C \times m \times m}$ and so $\mathbf{y} \in \mathbb{C}^{1 \times (n_1+m-1) \times (n_2+m-1)}$. With appropriate zero or symmetric padding, we can make \mathbf{y} have the same spatial shape as \mathbf{x} . Now, consider the full complex convolution to get \mathbf{y} :

$$y[\mathbf{n}] = \sum_{c=0}^{C-1} \sum_{\mathbf{k}} h[c, \mathbf{k}] x[c, \mathbf{n} - \mathbf{k}] \quad (\text{E.1})$$

We can expand this with real and imaginary terms:

$$\begin{aligned} y[\mathbf{n}] &= \sum_{c=0}^{C-1} \sum_{\mathbf{k}} (h_r[c, \mathbf{k}] + jh_i[c, \mathbf{k}]) (x_r[c, \mathbf{n} - \mathbf{k}] + jx_i[c, \mathbf{n} - \mathbf{k}]) \\ &= \sum_{c=0}^{C-1} \sum_{\mathbf{k}} (x_r[c, \mathbf{n} - \mathbf{k}]h_r[c, \mathbf{k}] - x_i[c, \mathbf{n} - \mathbf{k}]h_i[c, \mathbf{k}]) \\ &\quad + j \sum_{c=0}^{C-1} \sum_{\mathbf{k}} (x_r[c, \mathbf{n} - \mathbf{k}]h_i[c, \mathbf{k}] + x_i[c, \mathbf{n} - \mathbf{k}]h_r[c, \mathbf{k}]) \\ y_r[\mathbf{n}] + jy_i[\mathbf{n}] &= \sum_{c=0}^{C-1} ((x_r * h_r) - (x_i * h_i)) [c, \mathbf{n}] + j((x_r * h_i) + (x_i * h_r)) [c, \mathbf{n}] \end{aligned} \quad (\text{E.2})$$

Unsurprisingly, complex convolution is the sum and difference of 4 real convolutions.

We can use this fact to find the update and passthrough gradients for complex convolution.

Update Gradients: We need to find $\frac{\partial L}{\partial h_r}$ and $\frac{\partial L}{\partial h_i}$. From (E.2) we can apply the chain rule and the properties of real convolutions to write:

$$\frac{\partial L}{\partial h_r} = \frac{\partial L}{\partial y_r} \frac{\partial y_r}{\partial h_r} + \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial h_r} \quad (\text{E.3})$$

$$= \Delta y_r \star x_r + \Delta y_i \star x_i \quad (\text{E.4})$$

where \star is the correlation operation (to achieve spatial reversal compared with convolution, see [subsubsection 2.4.1.2](#)). Similarly

$$\frac{\partial L}{\partial h_i} = -\Delta y_r \star x_i + \Delta y_i \star x_r \quad (\text{E.5})$$

Passthrough Gradients: Again with application of the chain rule and the properties of real convolution, we have:

$$\frac{\partial L}{\partial x_r} = \frac{\partial L}{\partial y_r} \frac{\partial y_r}{\partial x_r} + \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial x_r} \quad (\text{E.6})$$

$$= \Delta y_r \star h_r + \Delta y_i \star h_i \quad (\text{E.7})$$

$$\frac{\partial L}{\partial x_i} = -\Delta y_r \star h_i + \Delta y_i \star h_r \quad (\text{E.8})$$

E.2 Regularization

We must be careful with regularizing complex weights. We want to promote the magnitude of the weights to be small but allow the phase to change unrestricted.

To do ℓ_2 regularization we can apply ℓ_2 to the real and imaginary parts independently:

$$\|r\|_2^2 = \left\| \sqrt{x_r^2 + x_i^2} \right\|_2^2 = \frac{1}{2} \sum_{\mathbf{n}} x_r[\mathbf{n}]^2 + x_i[\mathbf{n}]^2 = \frac{1}{2} \sum_{\mathbf{n}} x_r[\mathbf{n}]^2 + \frac{1}{2} \sum_{\mathbf{n}} x_i[\mathbf{n}]^2 = \|x_r\|_2^2 + \|x_i\|_2^2 \quad (\text{E.9})$$

But this does not hold for ℓ_1 regularization as:

$$\|r\|_1 = \left\| \sqrt{x_r^2 + x_i^2} \right\|_1 = \sum_{\mathbf{n}} \sqrt{x_r[\mathbf{n}]^2 + x_i[\mathbf{n}]^2} \neq \|x_r\|_1 + \|x_i\|_1 \quad (\text{E.10})$$

Also, for ℓ_1 regularization the derivatives have a discontinuity at the complex origin as:

$$\frac{\partial \ell_1}{\partial x_r[\mathbf{n}]} = \frac{x_r[\mathbf{n}]}{\sqrt{x_r[\mathbf{n}]^2 + x_i[\mathbf{n}]^2}} \quad (\text{E.11})$$

is not defined when $x_r = x_i = 0$. A similar problem was mentioned in [subsection 3.6.1](#) where we wanted to pass gradients through the magnitude operation of a ScatterNet. Since we do not explicitly care if weights are zero or near zero, we choose to handle this by setting the

gradient at the origin to be 0. It is unlikely our weights will ever be zero with this method, but if they are, we cover the case of dividing by zero.

E.3 ReLU Applied to the Real and Imaginary Parts Independently

If we define a nonlinearity to be $y = \sigma(x)$ where:

$$y = y_r + jy_i = \max(0, x_r) + j\max(0, x_i) \quad (\text{E.12})$$

then the passthrough gradients are:

$$\frac{\partial L}{\partial x_r} = \Delta y_r \mathbb{I}(x_r > 0) \quad (\text{E.13})$$

$$\frac{\partial L}{\partial x_i} = \Delta y_i \mathbb{I}(x_i > 0) \quad (\text{E.14})$$

E.4 Soft Shrinkage

Let $z = x + jy$ and $w = u + jv = \mathcal{S}(z, t)$ where we define the soft shrinkage on a complex number $z = re^{j\theta}$ by a real threshold t as:

$$\mathcal{S}(z, t) = \begin{cases} 0 & r < t \\ (r - t)e^{j\theta} & r \geq t \end{cases} \quad (\text{E.15})$$

This can alternatively be written as:

$$\mathcal{S}(z, t) = \frac{\max(r - t, 0)}{r} z \quad (\text{E.16})$$

$$= gz \quad (\text{E.17})$$

To find the pass through gradients $\frac{\partial L}{\partial x}$, $\frac{\partial L}{\partial y}$ and update equations $\frac{\partial L}{\partial t}$ we need to find all the real and imaginary partial derivatives. We can apply the product rule once we find $\frac{\partial g}{\partial x}$, $\frac{\partial g}{\partial y}$, $\frac{\partial g}{\partial t}$:

$$\frac{\partial g}{\partial x} = \begin{cases} 0 & r < t \\ \frac{r \frac{\partial r}{\partial x} - (r-t) \frac{\partial r}{\partial x}}{r^2} & r \geq t \end{cases} \quad (\text{E.18})$$

$$= \frac{xt \mathbb{I}(g > 0)}{r^3} \quad (\text{E.19})$$

$$\frac{\partial g}{\partial y} = \frac{yt \mathbb{I}(g > 0)}{r^3} \quad (\text{E.20})$$

$$(\text{E.21})$$

$$\frac{\partial g}{\partial t} = \frac{-\mathbb{I}(g > 0)}{r} \quad (\text{E.22})$$

Then from the definition of $w = u + jv$ we have $u = gx$ and $v = gy$, giving us:

$$\frac{\partial u}{\partial x} = g + \frac{x^2 t \mathbb{I}(g > 0)}{r^3} \quad (\text{E.23}) \quad \frac{\partial v}{\partial x} = \frac{xyt \mathbb{I}(g > 0)}{r^3} \quad (\text{E.26})$$

$$\frac{\partial u}{\partial y} = \frac{xyt \mathbb{I}(g > 0)}{r^3} \quad (\text{E.24}) \quad \frac{\partial v}{\partial y} = g + \frac{y^2 t \mathbb{I}(g > 0)}{r^3} \quad (\text{E.27})$$

$$\frac{\partial u}{\partial t} = \frac{-x \mathbb{I}(g > 0)}{r} \quad (\text{E.25}) \quad \frac{\partial v}{\partial t} = \frac{-y \mathbb{I}(g > 0)}{r} \quad (\text{E.28})$$

Putting it all together, our update and passthrough gradients are:

$$\frac{\partial L}{\partial x} = \frac{xt \mathbb{I}(g > 0)}{r^3} (x \Delta u + y \Delta v) + g \Delta u \quad (\text{E.29})$$

$$\frac{\partial L}{\partial y} = \frac{yt \mathbb{I}(g > 0)}{r^3} (x \Delta u + y \Delta v) + g \Delta v \quad (\text{E.30})$$

$$\frac{\partial L}{\partial t} = \frac{-\mathbb{I}(g > 0)}{r} (x \Delta u + y \Delta v) \quad (\text{E.31})$$

These equations are for point-wise application of soft-thresholding. When the same threshold is applied to an entire image, then we sum $\frac{\partial L}{\partial t}$ over all locations.

E.5 Batch Normalization and ReLU Applied to the Complex Magnitude

Again let $z = x + jy = re^{j\theta}$ and $w = u + jv = \text{ReLU}(\text{BN}(r))e^{j\theta}$. In (6.5.11) we showed that this nonlinearity is equivalent to soft-thresholding with threshold $t = \mu_r - \frac{\sigma_r \beta}{\gamma}$ and multiplied by a learned gain γ divided by the tracked standard deviation of r : $\frac{\gamma}{\sigma_r}$.

Let us call the action of this nonlinearity \mathcal{B} , defined by:

$$\mathcal{B}(z, \gamma, \beta) = \begin{cases} 0 & r < t \\ \frac{\gamma}{\sigma_r}(r-t)e^{j\theta} & r \geq t \end{cases} \quad (\text{E.32})$$

$$= \frac{\gamma}{\sigma_r} \frac{\max(r-t, 0)}{r} z \quad (\text{E.33})$$

$$= g'z \quad (\text{E.34})$$

Where g' is now the g from (E.17) scaled by $\frac{\gamma}{\sigma_r}$. It is clear from our new definition of g' that the equations (E.29) - (E.31) are also scaled by $\frac{\gamma}{\sigma_r}$. This immediately gives us the passthrough gradients. For the update equations, we must find some additional information:

$$\frac{\partial t}{\partial \beta} = -\frac{\sigma_r}{\gamma} \quad (\text{E.35})$$

$$\frac{\partial t}{\partial \gamma} = \frac{\sigma_r \beta}{\gamma^2} \quad (\text{E.36})$$

$$\frac{\partial g'}{\partial \beta} = \frac{-\gamma \mathbb{I}(g' > 0)}{\sigma_r r} \frac{\partial t}{\partial \beta} = \frac{\mathbb{I}(g' > 0)}{r} \quad (\text{E.37})$$

$$\frac{\partial g'}{\partial \gamma} = \frac{\mathbb{I}(g' > 0)}{\sigma_r r} \left(r - t - \frac{\partial t}{\partial \gamma} \right) = \frac{\mathbb{I}(g' > 0)}{\sigma_r r} \left(r - t - \frac{\sigma_r \beta}{\gamma^2} \right) \quad (\text{E.38})$$

Therefore, combining these with (E.31) we get:

$$\frac{\partial L}{\partial \beta} = \frac{\mathbb{I}(g' > 0)}{r} (x \Delta u + y \Delta v) \quad (\text{E.39})$$

$$\frac{\partial L}{\partial \gamma} = \frac{\mathbb{I}(g' > 0)}{\sigma_r r} \left(r - t - \frac{\sigma_r \beta}{\gamma^2} \right) (x \Delta u + y \Delta v) \quad (\text{E.40})$$

Appendix F

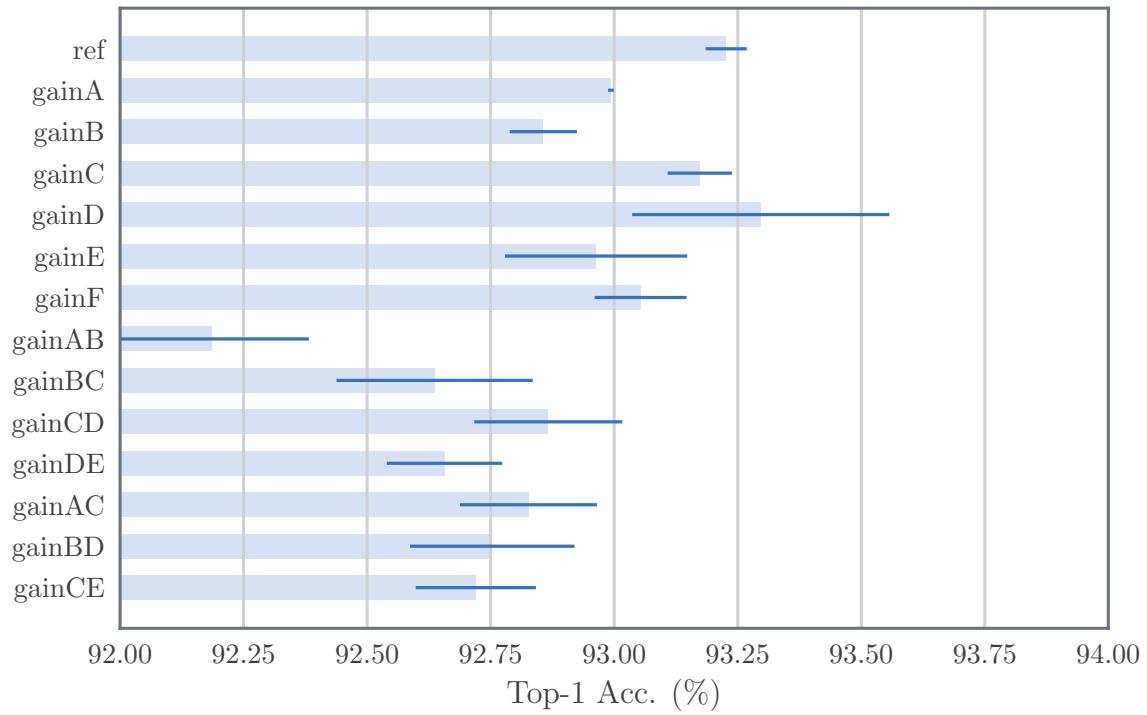
Wavelet Gain Layer Additional Results

This appendix presents some additional results for the gain layer experiments from [subsection 6.4.2](#). Here, we do ablation experiments similar to those done on the invariant layer in [section 5.7](#). In particular, we use the architecture described in [Table 5.5](#) and progressively swap out convolutional layers with gain layers. Again, we run this on CIFAR-10, CIFAR-100 and Tiny ImageNet.

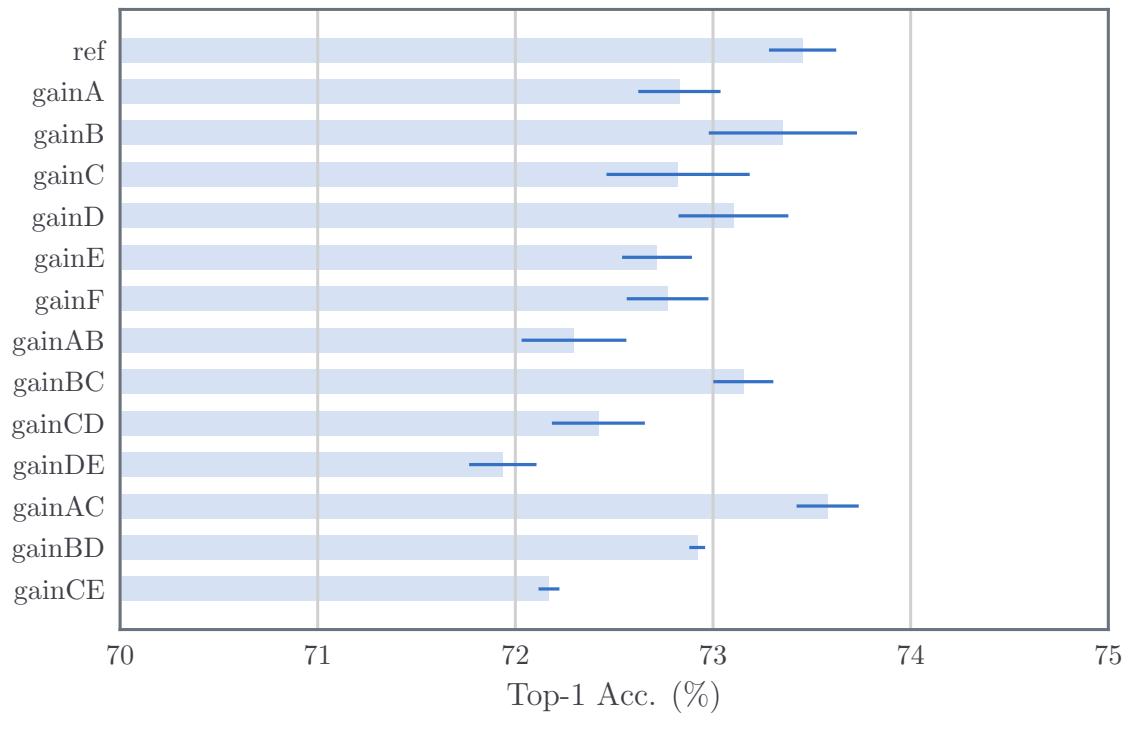
We use the same naming technique as in the previous chapter, calling a network ‘gainX’ means that the ‘convX’ layer was replaced with a wavelet gain layer, but otherwise keeping the rest of the architecture the same. Recall that the CIFAR architectures in [Table 5.5](#) have 6 convolutional layers called ‘convA’ to ‘convF’, and the Tiny ImageNet architectures have 8 convolutional layers called ‘convA’ to ‘convH’.

We train all our networks for 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).

[Figure F.1](#) and [Figure F.2](#) show the results from these experiments for CIFAR and Tiny ImageNet. Just as with the large kernel ablation study in [subsection 6.4.2](#), adding a gain layer typically degrades performance by a small amount.



(a) CIFAR-10



(b) CIFAR-100

Figure F.1: Small kernel ablation results for CIFAR. These graphs show the average and ± 1 standard deviation results for 3 runs. The names on the left represent where the convolutional layer is swapped for a gain layer, with ‘gainAB’ and below indicating two convolutional layers were swapped for gain layers.

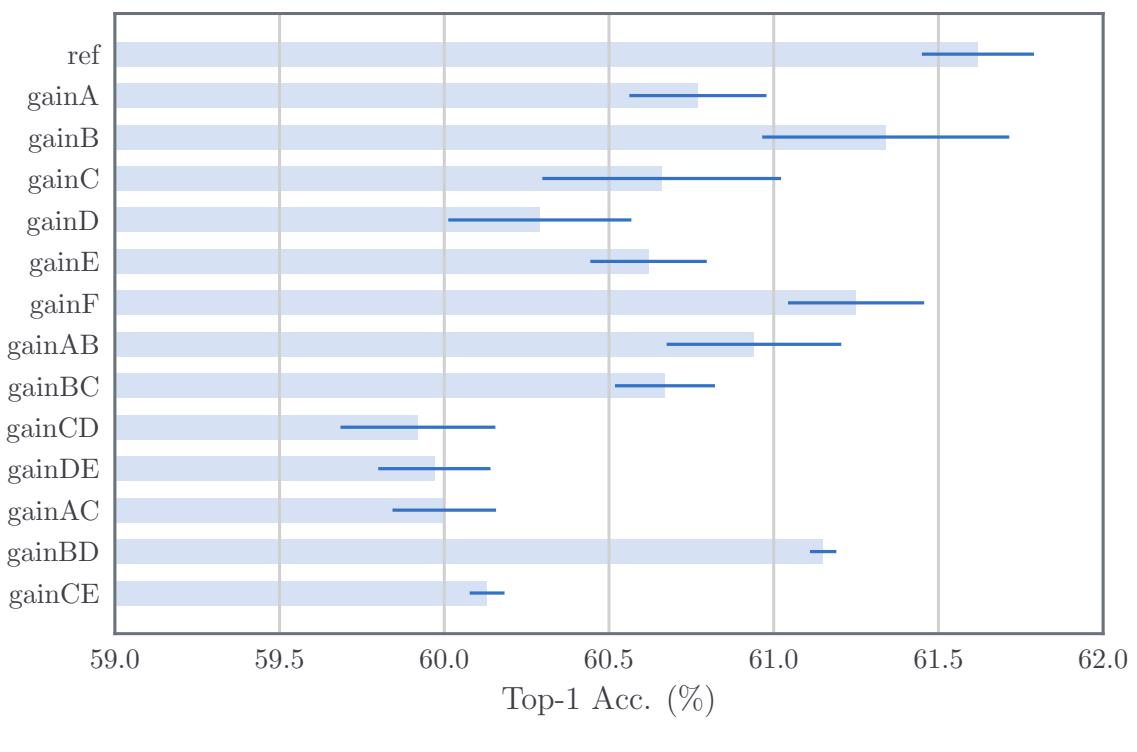


Figure F.2: **Small kernel ablation results for Tiny ImageNet.**

References

- [1] M. E. Raichle, “Two views of brain function,” *Trends in Cognitive Sciences*, vol. 14, no. 4, pp. 180–190, Apr. 2010. pmid: 20206576.
- [2] C. H. Anderson, D. C. Van Essen, and B. A. Olshausen, “Directed Visual Attention and the Dynamic Control of Information Flow,” in *Neurobiology of Attention*, Elsevier, 2005, pp. 11–17.
- [3] Tor Nørretranders, *The User Illusion*. Viking, 1998.
- [4] D. H. Hubel and T. N. Wiesel, “Receptive fields of single neurones in the cat’s striate cortex,” *The Journal of Physiology*, vol. 148, no. 3, pp. 574–591, Oct. 1959. pmid: 14403679.
- [5] ——, “Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex,” *The Journal of Physiology*, vol. 160, pp. 106–154, Jan. 1962.
- [6] ——, “Receptive fields and functional architecture of monkey striate cortex,” *The Journal of Physiology*, vol. 195, no. 1, pp. 215–243, Mar. 1968. pmid: 4966457.
- [7] C. Blakemore and G. F. Cooper, “Development of the Brain depends on the Visual Environment,” *Nature*, vol. 228, no. 5270, pp. 477–478, Oct. 31, 1970.
- [8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” presented at the NIPS, Lake Tahoe, Nevada: Curran Associates, Inc., 2012, pp. 1097–1105.
- [9] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, Apr. 11, 2015.
- [10] D. G. Lowe, “Distinctive Image Features from Scale-Invariant Keypoints,” *International Journal of Computer Vision*, vol. 60, pp. 91–110, 2004.
- [11] N. Dalal and B. Triggs, “Histograms of oriented gradients for human detection,” in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*, vol. 1, Jun. 2005, 886–893 vol. 1.
- [12] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep. 1, 1995.
- [13] J. Sánchez, F. Perronnin, T. Mensink, and J. Verbeek, “Image Classification with the Fisher Vector: Theory and Practice,” *International Journal of Computer Vision*, vol. 105, no. 3, pp. 222–245, Dec. 1, 2013.
- [14] J. Sanchez and F. Perronnin, “High-dimensional signature compression for large-scale image classification,” in *Proceedings of 2011 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Colorado Springs: IEEE, Jun. 2011, pp. 1665–1672.

- [15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas: IEEE, Jun. 2016, pp. 770–778.
- [16] B. Goodman and S. Flaxman, "European Union Regulations on Algorithmic Decision-Making and a "Right to Explanation"," *AI Magazine*, 2017.
- [17] S. Mallat, "Group Invariant Scattering," *Communications on Pure and Applied Mathematics*, vol. 65, no. 10, pp. 1331–1398, Oct. 1, 2012.
- [18] I. W. Selesnick, R. G. Baraniuk, and N. G. Kingsbury, "The dual-tree complex wavelet transform," *Signal Processing Magazine, IEEE*, vol. 22, no. 6, pp. 123–151, 2005.
- [19] N. Kingsbury, "Complex wavelets for shift invariant analysis and filtering of signals," *Applied and Computational Harmonic Analysis*, vol. 10, no. 3, pp. 234–253, May 2001.
- [20] J. Kovacevic and A. Chebira, *An Introduction to Frames*. Hanover, MA, USA: Now Publishers Inc., 2008.
- [21] D. Taubman and M. Marcellin, *JPEG2000 Image Compression Fundamentals, Standards and Practice*. Springer Publishing Company, Incorporated, 2013.
- [22] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *International Conference on Learning Representations (ICLR)*, Banff, Canada, Apr. 14, 2014.
- [23] J. Bruna and S. Mallat, "Invariant Scattering Convolution Networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1872–1886, Aug. 2013.
- [24] M. Tygert, J. Bruna, S. Chintala, Y. LeCun, S. Piantino, and A. Szlam, "A Mathematical Motivation for Complex-Valued Convolutional Networks," *Neural Computation*, vol. 28, no. 5, pp. 815–825, May 2016.
- [25] E. Oyallon and S. Mallat, "Deep Roto-Translation Scattering for Object Classification," in *Proceedings of 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, USA: IEEE, Jun. 2015, pp. 2865–2873.
- [26] M. D. Zeiler and R. Fergus, "Visualizing and Understanding Convolutional Networks," presented at the European Conference on Computer Vision (ECCV), Zurich, Sep. 6, 2014, pp. 818–833.
- [27] Q. Qiu, X. Cheng, R. Calderbank, and G. Sapiro, "DCFNet: Deep Neural Network with Decomposed Convolutional Filters," Feb. 12, 2018. arXiv: 1802.04145 [cs, stat].
- [28] F. Cotter. (2018). *Pytorch Wavelets software*. Python, [Online]. Available: https://github.com/fbcotter/pytorch_wavelets.
- [29] F. Cotter and N. Kingsbury, "Visualizing and Improving Scattering Networks," in *Proceedings of 2017 IEEE International Workshop on Machine Learning for Signal Processing (MLSP)*, Tokyo, Japan: IEEE, Sep. 5, 2017.
- [30] ——, "A Learnable ScatterNet: Locally Invariant Convolutional Layers," Mar. 7, 2019. arXiv: 1903.03137 [cs].
- [31] F. Cotter. (2019). *Learnable ScatterNet software*. Python, [Online]. Available: https://github.com/fbcotter/scatnet_learn.
- [32] F. Cotter and N. Kingsbury, "Deep Learning in the Wavelet Domain," Nov. 14, 2018. arXiv: 1811.06115 [cs].
- [33] F. Cotter. (Nov. 14, 2018). *DTCWT Gainlayer Software*. Python, [Online]. Available: https://github.com/fbcotter/dtcwt_gainlayer/releases/tag/arxiv.

- [34] A. Singh, "ScatterNet Hybrid Frameworks for Deep Learning," PhD Thesis, University of Cambridge, Cambridge, UK, May 2018, 158 pp.
- [35] E. Oyallon, "Analyzing and Introducing Structures in Deep Convolutional Neural Networks," PhD Thesis, Ecole Polytechnique, Paris, France, Oct. 2017.
- [36] A. Singh and N. Kingsbury, "Multi-Resolution Dual-Tree Wavelet Scattering Network for Signal Classification," in *International Conference on Mathematics in Signal Processing*, Birmingham, UK, Dec. 14, 2016.
- [37] A. Singh and N. Kingsbury, "Scatternet hybrid deep learning (SHDL) network for object classification," in *Proceedings of 2017 IEEE International Workshop on Machine Learning for Signal Processing (MLSP)*, Tokyo, Japan: IEEE, Sep. 2017.
- [38] A. Singh and N. Kingsbury, "Generative ScatterNet Hybrid Deep Learning (G-SHDL) Network with Structural Priors for Semantic Image Segmentation," in *Proceedings of 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Calgary, Canada: IEEE, 2018.
- [39] E. Oyallon, E. Belilovsky, and S. Zagoruyko, "Scaling the Scattering Transform: Deep Hybrid Networks," in *International Conference on Computer Vision (ICCV)*, Venice, Italy: IEEE, Oct. 2017, pp. 5619–5628.
- [40] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.
- [41] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.
- [42] S. Boyd and L. Vandenberghe, *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004.
- [43] D. R. Wilson and T. R. Martinez, "The general inefficiency of batch training for gradient descent learning," *Neural Networks: The Official Journal of the International Neural Network Society*, vol. 16, no. 10, pp. 1429–1451, Dec. 2003. pmid: 14622875.
- [44] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le, "Don't Decay the Learning Rate, Increase the Batch Size," Nov. 1, 2017. arXiv: 1711.00489 [cs, stat].
- [45] L. Bottou, "Stochastic Gradient Descent Tricks," in *Neural Networks: Tricks of the Trade: Second Edition*, ser. Lecture Notes in Computer Science, G. Montavon, G. B. Orr, and K.-R. Müller, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 421–436.
- [46] G. Montavon, G. Orr, and K.-R. Müller, *Neural Networks: Tricks of the Trade*, 2nd. Springer Publishing Company, Incorporated, 2012.
- [47] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient BackProp," in *Neural Networks: Tricks of the Trade*, ser. Lecture Notes in Computer Science 7700, G. Montavon, G. B. Orr, and K.-R. Müller, Eds., Springer Berlin Heidelberg, 2012, pp. 9–48.
- [48] Y. A. Ioannou, "Structural Priors in Deep Neural Networks," PhD Thesis, University of Cambridge, Cambridge, UK, Sep. 2017.
- [49] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," Dec. 22, 2014. arXiv: 1412.6980 [cs].
- [50] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, Jul 2011.
- [51] M. D. Zeiler, "ADADELTA: An Adaptive Learning Rate Method," Dec. 22, 2012. arXiv: 1212.5701 [cs].

- [52] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain [J],” *Psychol. Review*, vol. 65, pp. 386–408, Dec. 1, 1958.
- [53] X. Glorot, A. Bordes, and Y. Bengio, “Deep Sparse Rectifier Neural Networks,” in *Proceedings of International Conference on Artificial Intelligence and Statistics (AISTATS)*, Ft. Lauderdale, FL, USA: PMLR, Apr. 2011, pp. 315–323.
- [54] V. Nair and G. E. Hinton, “Rectified Linear Units Improve Restricted Boltzmann Machines,” presented at the Proceedings of the 27th International Conference on Machine Learning (ICML-10), 2010, pp. 807–814.
- [55] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, Sardinia, Italy, May 2010.
- [56] P. Lennie, “The cost of cortical computation,” *Current biology: CB*, vol. 13, no. 6, pp. 493–497, Mar. 18, 2003. pmid: 12646132.
- [57] M. L. Minsky and S. A. Papert, *Perceptrons: Expanded Edition*. Cambridge, MA, USA: MIT Press, 1988.
- [58] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, vol. 2, no. 5, pp. 359–366, Jan. 1, 1989.
- [59] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, pp. 303–314, Dec. 1, 1989.
- [60] B. Widrow and M. E. Hoff, “Neurocomputing: Foundations of Research,” in, J. A. Anderson and E. Rosenfeld, Eds., Cambridge, MA, USA: MIT Press, 1988, pp. 123–134.
- [61] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1,” in, D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, Eds., Cambridge, MA, USA: MIT Press, 1986, pp. 318–362.
- [62] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
- [63] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, Dec. 1, 2015.
- [64] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in PyTorch,” Oct. 28, 2017.
- [65] D. Mishkin, N. Sergievskiy, and J. Matas, “Systematic Evaluation of Convolution Neural Network Advances on the Imagenet,” *Comput. Vis. Image Underst.*, vol. 161, no. C, pp. 11–19, Aug. 2017.
- [66] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” Sep. 4, 2014. arXiv: 1409.1556 [cs].
- [67] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten, “Densely Connected Convolutional Networks,” in *Proceedings of 2017 IEEE Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017, pp. 2261–2269.
- [68] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, “Striving for Simplicity: The All Convolutional Net,” in *Proceedings of the International Conference on Learning Representations (ICLR)*, San Diego, CA, USA, May 2015.

- [69] S. Xie, R. B. Girshick, P. Dollár, Z. Tu, and K. He, “Aggregated Residual Transformations for Deep Neural Networks,” in *Proceedings of 2017 IEEE Computer Vision and Pattern Recognition (CVPR)*, Honolulu, HI, USA: IEEE, Jul. 2017, pp. 5987–5995.
- [70] S. Zagoruyko and N. Komodakis, “Wide Residual Networks,” May 23, 2016. arXiv: 1605.07146 [cs].
- [71] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” Jul. 3, 2012. arXiv: 1207.0580 [cs].
- [72] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.
- [73] Y. Gal and Z. Ghahramani, “Dropout As a Bayesian Approximation: Representing Model Uncertainty in Deep Learning,” in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, (New York, NY, USA), ser. ICML’16, JMLR.org, 2016, pp. 1050–1059.
- [74] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” in *International Conference on Machine Learning*, Lille, France: JMLR, Jul. 2015, pp. 448–456.
- [75] Y. LeCun, C. Cortes, and C. Burges, “Modified NIST database of handwritted digits,” <http://yann.lecun.com/exdb/mnist/>, 1998.
- [76] A. Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” Apr. 2009.
- [77] A. Krizhevsky, V. Nair, and G. Hinton, “CIFAR datasets,” 2009.
- [78] F.-F. Li, “Tiny ImageNet Visual Recognition Challenge,” Stanford cs231n: <https://tiny-imagenet.herokuapp.com/>, 2017.
- [79] Stanford Vision Lab, “ImageNet CLS-LOC,” <https://www.kaggle.com/c/imagenet-object-localization-challenge/data>, 2017.
- [80] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, “The Pascal Visual Object Classes Challenge: A Retrospective,” *International Journal of Computer Vision*, vol. 111, no. 1, pp. 98–136, Jan. 2015.
- [81] Li Fei-Fei, R. Fergus, and P. Perona, “Learning Generative Visual Models from Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories,” in *2004 Conference on Computer Vision and Pattern Recognition Workshop*, Jun. 2004, pp. 178–178.
- [82] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going Deeper With Convolutions,” in *Proceedings of 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, USA: IEEE, Jun. 2015, pp. 1–9.
- [83] P. L. Bartlett, S. N. Evans, and P. M. Long, “Representing smooth functions as compositions of near-identity functions with implications for deep network optimization,” Apr. 13, 2018. arXiv: 1804.05012 [cs, math, stat].
- [84] P. Bartlett, D. Helmbold, and P. Long, “Gradient descent with identity initialization efficiently learns positive definite linear transformations by deep residual networks,” in *International Conference on Machine Learning*, Jul. 3, 2018, pp. 521–530.
- [85] J. Antoine, R. Murenzi, P. Vandergheynst, and S. Ali, *Two-Dimensional Wavelets and Their Relatives*. Cambridge University Press, 2004.

- [86] M. Holschneider and P. Tchamitchian, “Pointwise analysis of Riemann’s “nondifferentiable” function,” *Inventiones mathematicae*, vol. 105, no. 1, pp. 157–175, Dec. 1, 1991.
- [87] M. Vetterli and J. Kovacevic, *Wavelets and Subband Coding*, 2nd ed., ser. Prentice Hall Signal Processing Series. Prentice Hall PTR, 2007.
- [88] S. Mallat, *A Wavelet Tour of Signal Processing*. Academic Press, 1998.
- [89] R. R. Coifman and D. L. Donoho, “Translation-Invariant De-Noising,” in *Wavelets and Statistics*, ser. Lecture Notes in Statistics 103, A. Antoniadis and G. Oppenheim, Eds., Springer New York, 1995, pp. 125–150.
- [90] N. Kingsbury and J. Magarey, “Wavelet Transforms in Image Processing,” in *Signal Analysis and Prediction*, ser. Applied and Numerical Harmonic Analysis, A. Procházka, J. Uhlíř, P. W. J. Rayner, and N. G. Kingsbury, Eds., Boston, MA, USA: Birkhäuser Boston, 1998, pp. 27–46.
- [91] N. Kingsbury, “The Dual-Tree Complex Wavelet Transform: A New Technique For Shift Invariance And Directional Filters,” in *1998 8th International Conference on Digital Signal Processing (DSP)*, Utah: IEEE, Aug. 1998, pp. 319–322.
- [92] ——, “The dual-tree complex wavelet transform: A new efficient tool for image restoration and enhancement,” in *Signal Processing Conference (EUSIPCO 1998), 9th European*, Rhodes, Greece: IEEE, Sep. 1998, pp. 1–4.
- [93] N. Kingsbury, “Image processing with complex wavelets,” *Philosophical Transactions of the Royal Society a-Mathematical Physical and Engineering Sciences*, vol. 357, no. 1760, pp. 2543–2560, Sep. 15, 1999.
- [94] ——, “Shift invariant properties of the dual-tree complex wavelet transform,” in *Proceedings of 1999 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Phoenix, AZ, USA: IEEE, 1999, pp. 1221–1224.
- [95] ——, “A dual-tree complex wavelet transform with improved orthogonality and symmetry properties,” in *2000 IEEE International Conference on Image Processing (ICIP)*, vol. 2, Vancouver, BC: IEEE, Sep. 2000, 375–378 vol.2.
- [96] J. Bruna and S. Mallat, “Classification with scattering operators,” in *Proceedings of 2011 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Colorado Springs: IEEE, Jun. 2011, pp. 1561–1566.
- [97] J. Bruna, “Scattering Representations for Recognition,” Theses, Ecole Polytechnique X, Feb. 2013.
- [98] E. Oyallon, S. Mallat, and L. Sifre, “Generic Deep Networks with Wavelet Scattering,” Dec. 20, 2013. arXiv: 1312.5940 [cs].
- [99] L. Sifre and S. Mallat, “Rotation, Scaling and Deformation Invariant Scattering for Texture Discrimination,” in *Proceedings of 2013 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Portland, Oregon: IEEE, Jun. 2013, pp. 1233–1240.
- [100] L. Sifre and S. Mallat, “Rigid-Motion Scattering for Texture Classification,” Mar. 7, 2014. arXiv: 1403.1687 [cs].
- [101] L. Sifre, “Rigid-Motion Scattering for Image Classification,” PhD Thesis, Ecole Polytechnique, Paris, France, Oct. 2014, 128 pp.
- [102] L. Sifre and J. Anden. (Nov. 2013). *ScatNet Software*. Matlab. version 0.2, [Online]. Available: <https://github.com/scatnet/scatnet/releases/tag/v0.2>.

- [103] P. de Rivaz and N. Kingsbury, "Bayesian image deconvolution and denoising using complex wavelets," in *2001 IEEE International Conference on Image Processing (ICIP)*, vol. 2, Thessaloniki, Greece: IEEE, Oct. 2001, 273–276 vol.2.
- [104] Y. Zhang and N. Kingsbury, "A Bayesian wavelet-based multidimensional deconvolution with sub-band emphasis," in *30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, Vancouver, BC, Canada, Aug. 2008, pp. 3024–3027.
- [105] G. Zhang and N. Kingsbury, "Variational Bayesian image restoration with group-sparse modeling of wavelet coefficients," *Digital Signal Processing*, Special Issue in Honour of William J. (Bill) Fitzgerald, vol. 47, pp. 157–168, Dec. 2015.
- [106] M. Miller and N. Kingsbury, "Image denoising using derotated complex wavelet coefficients," *IEEE Transactions on Image Processing*, vol. 17, no. 9, pp. 1500–1511, Sep. 2008. pmid: 18701390.
- [107] S. Hatipoglu, S. K. Mitra, and N. Kingsbury, "Texture classification using dual-tree complex wavelet transform," in *Seventh International Conference on Image Processing and Its Applications*, Manchester, UK: IEEE, Jul. 1999, pp. 344–347.
- [108] P. de Rivaz and N. Kingsbury, "Complex wavelet features for fast texture image retrieval," in *1999 IEEE International Conference on Image Processing (ICIP)*, vol. 1, Kobe, Japan, Oct. 1999, 109–113 vol.1.
- [109] P. Loo and N. G. Kingsbury, "Motion-estimation-based registration of geometrically distorted images for watermark recovery," in *Proceedings of SPIE - The International Society for Optical Engineering*, P. W. Wong and E. J. Delp III, Eds., Aug. 1, 2001, pp. 606–617.
- [110] H. Chen and N. Kingsbury, "Efficient Registration of Nonrigid 3-D Bodies," *IEEE Transactions on Image Processing*, vol. 21, no. 1, pp. 262–272, Jan. 2012.
- [111] J. Fauqueur, N. Kingsbury, and R. Anderson, "Multiscale keypoint detection using the dual-tree complex wavelet transform," in *2006 International Conference on Image Processing (ICIP)*, Atlanta, GA, USA: IEEE, Oct. 2006, pp. 1625–1628.
- [112] R. Anderson, N. Kingsbury, and J. Fauqueur, "Determining Multiscale Image Feature Angles from Complex Wavelet Phases," in *Image Analysis and Recognition*, ser. Lecture Notes in Computer Science 3656, M. Kamel and A. Campilho, Eds., Springer Berlin Heidelberg, Sep. 28, 2005, pp. 490–498.
- [113] ——, "Rotation-invariant object recognition using edge profile clusters," in *14th European Signal Processing Conference*, Florence, Italy: IEEE, Sep. 2006, pp. 1–5.
- [114] P. Bendale, W. Triggs, and N. Kingsbury, "Multiscale keypoint analysis based on complex wavelets," in *Proceedings of the British Machine Vision Conference*, Aberystwyth, Wales: BMVA Press, 2010, pp. 49–1.
- [115] E. S. Ng and N. G. Kingsbury, "Robust pairwise matching of interest points with complex wavelets," *IEEE Transactions on Image Processing*, vol. 21, no. 8, pp. 3429–3442, 2012.
- [116] I. Selesnick, "Hilbert transform pairs of wavelet bases," *IEEE Signal Processing Letters*, vol. 8, no. 6, pp. 170–173, Jun. 2001.
- [117] N. Kingsbury, "Design of Q-shift complex wavelets for image processing using frequency domain energy minimization," in *2003 IEEE International Conference on Image Processing (ICIP)*, vol. 1, Barcelona, Spain, Sep. 2003.
- [118] I. Waldspurger, A. d'Aspremont, and S. Mallat, "Phase Recovery, MaxCut and Complex Semidefinite Programming," Jun. 1, 2012. arXiv: 1206.0102 [math].

- [119] M. Andreux, T. Angles, G. Exarchakis, R. Leonarduzzi, G. Rochette, L. Thiry, J. Zarka, S. Mallat, J. Andén, E. Belilovsky, J. Bruna, V. Lostanlen, M. J. Hirn, E. Oyallon, S. Zhang, C. Cella, and M. Eickenberg, “Kymatio: Scattering Transforms in Python,” Dec. 28, 2018. arXiv: 1812.11214 [cs, eess, stat].
- [120] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015.
- [121] W. Sweldens, “The Lifting Scheme: A Construction of Second Generation Wavelets,” *SIAM J. Math. Anal.*, vol. 29, no. 2, pp. 511–546, Mar. 1998.
- [122] C. Tenllado, J. Setoain, M. Prieto, L. Piñuel, and F. Tirado, “Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 3, pp. 299–310, Mar. 2008.
- [123] V. Galiano, O. Lopez, M. Malumbres, and H. Migallon, “Improving the discrete wavelet transform computation from multicore to GPU-based algorithms,” in *Proceedings of the 11th International Conference on Computational and Mathematical Methods in Science and Engineering*, Jun. 26–30, 2011, pp. 544–555.
- [124] N. Kingsbury. (2003). *DTCWT Software*. Matlab. version 4.3, [Online]. Available: <http://sigproc.eng.cam.ac.uk/Main/NGK>.
- [125] S. Cai, K. Li, and I. Selesnick. (Nov. 2011). *2-D Dual-Tree Wavelet Transform*. Matlab, [Online]. Available: <http://eeweb.poly.edu/iselesni/WaveletSoftware/dt2D.html>.
- [126] R. Wareham, C. Shaffrey, and N. Kingsbury. (2014). *DTCWT*. Python, [Online]. Available: <https://pypi.org/project/dtcwt/>.
- [127] S. Boyd, L. Xiao, and A. Mutapcic, *The Subgradient Method*, Letter, 2003.
- [128] M. T. Ribeiro, S. Singh, and C. Guestrin, ““Why Should I Trust You?”: Explaining the Predictions of Any Classifier,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, vol. 1, New York, NY, USA, Feb. 16, 2016.
- [129] A. Mahendran and A. Vedaldi, “Understanding Deep Image Representations by Inverting Them,” in *Proceedings of 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, USA: IEEE, Jun. 2015.
- [130] K. Simonyan, A. Vedaldi, and A. Zisserman, “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps,” *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.
- [131] A. Singh and N. Kingsbury, “Dual-Tree Wavelet Scattering Network with Parametric Log Transformation for Object Classification,” in *Proceedings of 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, New Orleans: IEEE, Mar. 2017, pp. 2622–2626.
- [132] M. Zeiler, G. Taylor, and R. Fergus, “Adaptive deconvolutional networks for mid and high level feature learning,” in *2011 IEEE International Conference on Computer Vision (ICCV)*, Nov. 2011, pp. 2018–2025.

- [133] R. Fong and A. Vedaldi, "Interpretable Explanations of Black Boxes by Meaningful Perturbation," *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 3449–3457, Oct. 2017.
- [134] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Object Detectors Emerge in Deep Scene CNNs," Dec. 21, 2014. arXiv: 1412.6856 [cs].
- [135] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," Dec. 1, 2015. arXiv: 1512.00567 [cs].
- [136] Y. Ioannou, D. Robertson, J. Shotton, R. Cipolla, and A. Criminisi, "Training CNNs with Low-Rank Filters for Efficient Image Classification," Nov. 20, 2015. arXiv: 1511.06744 [cs].
- [137] F. Juefei-Xu, V. N. Bodetti, and M. Savvides, "Local Binary Convolutional Neural Networks," Aug. 22, 2016. arXiv: 1608.06049 [cs].
- [138] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization," Mar. 21, 2016. arXiv: 1603.06560 [cs, stat].
- [139] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A Research Platform for Distributed Model Selection and Training," *arXiv preprint arXiv:1807.05118*, 2018.
- [140] S. Liu and W. Deng, "Very deep convolutional neural network based image classification using small training sample size," in *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, Nov. 2015, pp. 730–734.
- [141] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "FitNets: Hints for Thin Deep Nets," Dec. 19, 2014. arXiv: 1412.6550 [cs].
- [142] K. He, X. Zhang, S. Ren, and J. Sun, "Identity Mappings in Deep Residual Networks," in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., ser. Lecture Notes in Computer Science, Springer International Publishing, 2016, pp. 630–645.
- [143] S. Fujieda, K. Takayama, and T. Hachisuka, "Wavelet Convolutional Neural Networks for Texture Classification," Jul. 23, 2017. arXiv: 1707.07394 [cs].
- [144] ——, "Wavelet Convolutional Neural Networks," May 20, 2018. arXiv: 1805.08620 [cs].
- [145] T. Guo, H. S. Mousavi, T. H. Vu, and V. Monga, "Deep Wavelet Prediction for Image Super-Resolution," in *Proceedings of 2017 IEEE Computer Vision and Pattern Recognition (CVPR)*, Honolulu, HI, USA: IEEE, Jul. 2017, pp. 1100–1109.
- [146] L. Ma, J. Stückler, T. Wu, and D. Cremers, "Detailed Dense Inference with Convolutional Neural Networks via Discrete Wavelet Transform," Aug. 6, 2018. arXiv: 1808.01834 [cs].
- [147] O. Rippel, J. Snoek, and R. P. Adams, "Spectral Representations for Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., Curran Associates, Inc., 2015, pp. 2440–2448.
- [148] V. Petyan, Y. Romano, J. Sulam, and M. Elad, "Theoretical Foundations of Deep Learning via Sparse Representations: A Multilayer Sparse Model and Its Connection to Convolutional Neural Networks," *IEEE Signal Processing Magazine*, vol. 35, no. 4, pp. 72–89, Jul. 2018.

- [149] V. Petyan, Y. Romano, and M. Elad, “Convolutional Neural Networks Analyzed via Convolutional Sparse Coding,” *Journal of Machine Learning Research*, vol. 18, no. 83, pp. 1–52, 2017.
- [150] E. Haber, L. Ruthotto, and E. Holtham, “Learning Across Scales - Multiscale Methods for Convolution Neural Networks,” in *31st Conference on Artificial Intelligence (AAAI)*, San Francisco, CA, USA, Feb. 2017.
- [151] I. Daubechies and W. Sweldens, “Factoring wavelet transforms into lifting steps,” *Journal of Fourier Analysis and Applications*, vol. 4, no. 3, pp. 247–269, May 1, 1998.
- [152] A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse, “The reversible residual network: Backpropagation without storing activations,” in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Curran Associates, Inc., 2017, pp. 2214–2224.
- [153] J.-H. Jacobsen, A. Smeulders, and E. Oyallon, “I-RevNet: Deep Invertible Networks,” Feb. 20, 2018. arXiv: 1802.07088 [cs, stat].
- [154] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, “Practical Black-Box Attacks against Machine Learning,” in *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security*, Abu Dhabi, UAE, Apr. 2, 2017, pp. 506–519.
- [155] N. Carlini and D. Wagner, “Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods,” May 20, 2017. arXiv: 1705.07263 [cs].
- [156] L. Engstrom, B. Tran, D. Tsipras, L. Schmidt, and A. Madry, “A Rotation and a Translation Suffice: Fooling CNNs with Simple Transformations,” in *NIPS 2017 Workshop on Machine Learning and Computer Security*, Long Beach, CA, USA, Dec. 8, 2017.
- [157] M. Cisse, P. Bojanowski, E. Grave, Y. Dauphin, and N. Usunier, “Parseval Networks: Improving Robustness to Adversarial Examples,” in *Proceedings of the 34th International Conference on Machine Learning*, (Sydney, NSW, Australia), ser. ICML’17, JMLR, 2017, pp. 854–863.
- [158] J. Liu, C. Garcia-Cardona, B. Wohlberg, and W. Yin, “Online convolutional dictionary learning,” in *2017 IEEE International Conference on Image Processing (ICIP)*, Sep. 2017, pp. 1707–1711.
- [159] ——, “First and Second-Order Methods for Online Convolutional Dictionary Learning,” *SIAM Journal on Imaging Sciences*, vol. 11, no. 2, pp. 1589–1628, Jan. 1, 2018.
- [160] V. Petyan, Y. Romano, M. Elad, and J. Sulam, “Convolutional Dictionary Learning via Local Processing,” in *Proceedings of 2017 IEEE International Conference on Computer Vision (ICCV)*, Venice, Italy, Oct. 2017, pp. 5306–5314.
- [161] R. Rubinstein, M. Zibulevsky, and M. Elad, “Double Sparsity: Learning Sparse Dictionaries for Sparse Signal Approximation,” *IEEE Transactions on Signal Processing*, vol. 58, no. 3, pp. 1553–1564, Mar. 2010.
- [162] M. S. Advani and A. M. Saxe, “High-dimensional dynamics of generalization error in neural networks,” Oct. 10, 2017. arXiv: 1710.03667 [physics, q-bio, stat].