Uses of Complex Wavelets in Deep Convolutional Neural Networks

Fergal Cotter

Supervisor: Prof. Nick Kingsbury
Prof. Joan Lasenby

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
PhD

Trinity College July 2019

Declaration

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

Fergal Cotter
July 2019

Acknowledgements

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

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

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

Abstract

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

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

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

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

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

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

Table of contents

List of figures

List of tables

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. From fully autonomous vehicles, to automatic labelling of uploaded videos/images for searching, or facial recognition for identification and security, the uses are far reaching and extremely valuable. The challenge does not lie in finding the right application, but in the difficulty of training a computer to *see*.

There are nuisance variables such as changes in lighting condition, changes in viewpoint and background clutter that do not affect the scene but drastically change the pixel representation of it. Humans, even at early stages of their lives, have little difficulty filtering these out and extracting the necessary amount of information from a scene. So to design a robust system, it makes sense to design it off how *our* brains see.

Unfortunately, vision is a particularly complex system to understand. It has more to it than the simply collecting photons in the eye. An excerpt from a recent Neurology paper **raichle_two_2010** 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 10^4 make it to layer IV of V1 **anderson_directed_2005**, **tor_norretranders_user_1998**. 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**anderson_directed_2005**, **tor_norretranders_user_1998**.

Current digital 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 an image

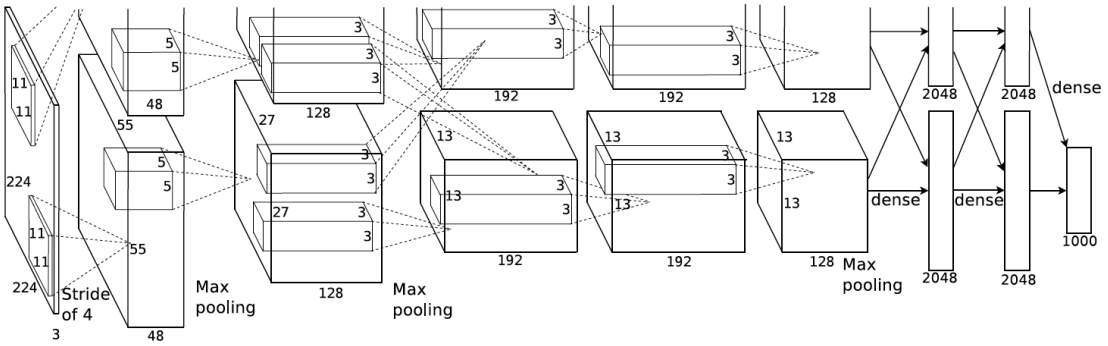


Figure 1.1: **Convolutional Architecture example.** The previous layer's activations are combined with a learned convolutional filter. Note that while the activation maps are 3D arrays, the convolution is only a 2D operation. This means the filters have the same number of channels as the input and produce 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 taken from **krizhevsky__imagenet__2012**.

on the order of magnitude of 10^6 pixels (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. The question now stands before us — what information is kept on entry to the V1 cortex? Hubel and Wiesel revolutionized our understanding of the V1 cortex in the 50s and 60s by studying cats **hubel__receptive__1959**, **hubel__receptive__1962**, macaques and spider monkeys **hubel__receptive__1968**. 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. Continued work on their experiments by Blakemore and Cooper **blakemore__development__1970** showed that these early layers of perception are in fact *learned*. Their experiments kept kittens in darkness and exposed them 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 rods that matched the orientation of their environment.

1.1 Convolutional Neural Networks

The current state of the art in image understanding sysems are Convolutional Neural Networks (CNNs). These are a learned model that stacks many convolutional filters on top of each other 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 architecture for the famous AlexNet **krizhevsky__imagenet__2012**. Inputs are resized to a manageable size, in this case 224×224 pixels. Then 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 a pointwise nonlinear function, or a *nonlinearity*. The activations are then 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 the previously mentioned AlexNet nearly halved the top-5 classification error rate (from 26% to 16%) on the ImageNet Large Scale Visual Recognition Competition (ILSVRC) **russakovsky__imagenet__2014**[1]. In the years since then, their complexity has grown significantly. AlexNet had only 5 conovlutional layers, whereas the 2015 ILSVRC winner ResNet **he__deep__2016** achieved 3.57% top-5 error with 151 convolutional layers (and had some experiments with 1000 layer networks).

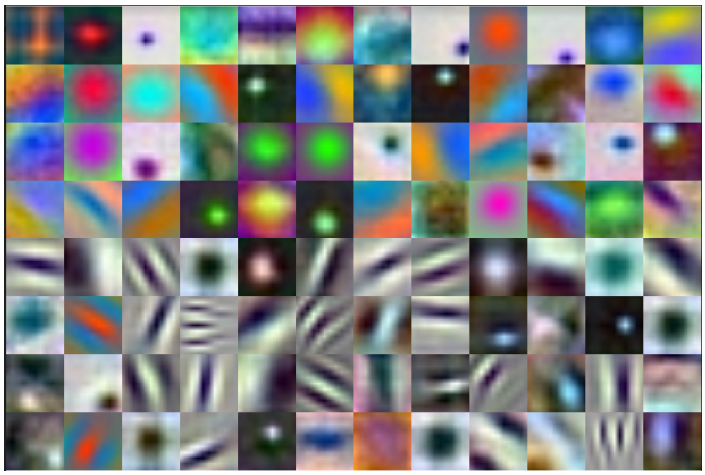
1.2 Issues with CNNs

Despite their success, they 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, *depth* dimension typically much larger than its 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]. We can see in Figure 1.2b that in the conv1 activations, some of the first layer channels are responding to edges or colour information, but as we go deeper to conv2 and conv3, it becomes less and less clear what each activation is responding to.

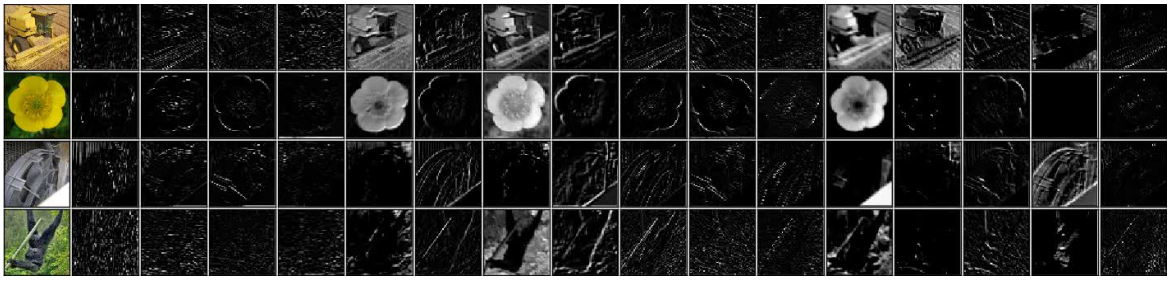
Aside from their lack of interpretability, it takes a long time and a lot of effort to train state of the art CNNs. Typical networks that have won ILSVRC since 2012 have had roughly 100 million parameters and take up to a week to train. This is optimistic and assumes that you already know the necessary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

[1] The previous state of the art classifiers had been built by combining keypoint extractors like SIFT**lowe__distinctive__2004** and HOG**dalal__histograms__2005** with classifiers such as Support Vector Machines**cortes__support-vector__1995** and Fisher Vectors**sanchez__image__2013**, for example **sanchez__high-dimensional__2011**.

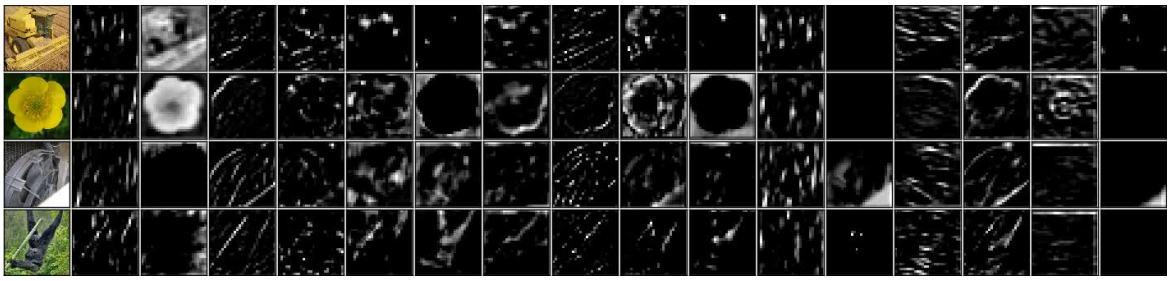
[2] These activations are taken after a specific nonlinearity that sets negative values to 0, hence the large black regions.



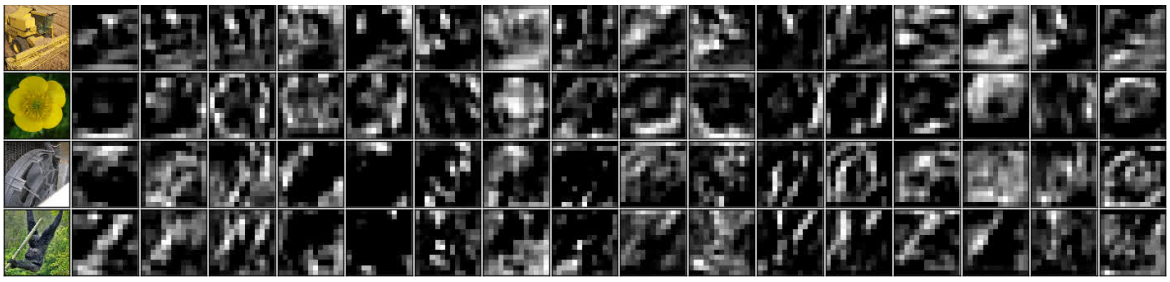
(a) conv1 filters



(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 taken from supplementary material of **krizhevsky_imagenet_2012**.

> 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, which 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.

1.3 An Interesting Result - The Learned Wavelet Transform

The structure of convolutional layers is fairly crude in terms of signal processing - arbitrary taps of an FIR filter are learned typically via gradient descent to minimize either a mean-squared error or cross entropy loss. It is perhaps surprising and motivating then that the filters we saw earlier learned by the first layer of a CNN (Figure 1.2) look like oriented wavelets. From a biological point of view this makes sense, as it matches the earlier mentioned results by Hubel and Wiesel. From a signal processing point of view this also makes sense, as the wavelet transform is a powerful and stable way to split up an image into areas of the frequency domain. However, there was no prior placed on the filters to make them have this similarity to wavelets. There were no constraints on vanishing moments**daubechies_ten_1992** or smoothness, or even for the 'conv1' activations to be sparse; the only constraint they had was an ℓ_2 penalty to avoid large filter taps.

1.4 Project Motivation

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?

However in achieving this, it is important to find ways to have an adequate richness to filtering while reducing the number of parameters needed to specify this. 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 the right basis functions for convolutional filtering in CNNs. We have already seen in the previous section that they would do well in replacing the first layer of a CNN, but can they 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 is important to avoid sensitivity to adversarial or spoofing attacks **szegedy_intriguing_2014** 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 near the same performance by using wavelets as the building blocks for CNNs, and this is the core goal of this thesis.

1.5 ScatterNets

To explore this intuition, 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 **mallat_group_2012**, **bruna_invariant_2013** at the same time as AlexNet. It is a non black box network that can be thought of as a restricted complex valued CNN **bruna_mathematical_2015**. Unlike a CNN, it has predefined convolutional kernels, set to complex wavelet (and scaling) functions. Due to its well-defined structure, it can be analyzed and bounds on its stability to shifts, noise and deformations are found in **mallat_group_2012**.

For a simple task like identifying small handwritten digits the variablities in the data are simple and small and the ScatterNet can easily reduce the problem into a space which a Gaussian SVM can easily solve **bruna_invariant_2013**. 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 large performance gap between this and what a CNN can achieve **oyallon_deep_2015**.

1.6 Layout

This thesis has one literature review chapter and four 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.
- Chapter 3 proposes a change to the core of the ScatterNet. In addition to performance issues with ScatterNets, they are slow and both memory and compute intensive to calculate. This in itself is enough of an issue to prevent them from ever being used as part of deep networks. To overcome this, we change the computation to use the DT$\mathbb{C}$WT **selesnick_dual-tree_2005** instead of Morlet wavelets, achieving a 20 to 30 times speed up.
- 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. While this work is interesting in its ability to learn filters in a completely new way, it does not add any benefit over learning solely in the pixel space.

1.6.1 Contributions

The key contributions of this thesis are:

- Software for wavelets and DT$\mathbb{C}$WT based ScatterNet (described in chapter 3) and publically available at **cotter_pytorch_2018**.
- ScatterNet analysis and visualizations (described in chapter 4). This chapter expands on the paper we presented at MLSP2017 **cotter_visualizing_2017**.
- Invariant Layer/Learnable ScatterNet (described in chapter 5)). This chapter expands on the paper accepted at ICIP2019 **cotter_learnable_2019**. Software available at **cotter_learnable_2019-1**.
- Learning convolutions in the wavelet domain (described in chapter 6). We have published preliminary results on this work to arxiv **cotter_deep_2018** but have expanded on this paper in the chapter. Software available at **cotter_dtcwt_2018**.

1.6.2 Related Research

Readers may also be interested in **singh_scatternet_2018** and **oyallon_analyzing_2017**. In **singh_scatternet_2018 singh_scatternet_2018** 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 an defined way he creates unsupervised feature extractors which can then be used with simple classifiers. In **oyallon_analyzing_2017 oyallon_analyzing_2017** combines a ScatterNet front end to a deep CNN. In chapter 5 we look at a similar design, but allow for changes to the scattering.

Chapter 2

Background

This thesis combines work in several fields. We provide a background for the most important and relevant fields in this chapter. We first introduce the basics of deep learning, before defining the properties of Wavelet Transforms and finally, we introduce the Scattering Transform, the original inspiration for this thesis.

2.1 Supervised Machine Learning

While this subject is general and covered in many places, we take inspiration from **murphy__machine__2012** (chapters 1, 2, 7, 8) and **goodfellow__deep__2016** (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:

$$y = f(x, \theta) + \epsilon \tag{2.1.1}$$

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:

$$p_{model}(y|x, \theta, \sigma^2) = \mathcal{N}(y;\ f(x, \theta), \sigma^2 I) \tag{2.1.2}$$

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:

$$\theta_{MLE} = \arg\max_{\theta} p_{model}(y|x,\theta) \tag{2.1.3}$$

$$= \arg\max_{\theta} \prod_{n=1}^{N} p_{model}(y^{(n)}|x^{(n)},\theta) \tag{2.1.4}$$

$$= \arg\max_{\theta} \sum_{n=1}^{N} \log p_{model}(y^{(n)}|x^{(n)},\theta) \tag{2.1.5}$$

Using the Gaussian regression model from above, this becomes:

$$\theta_{MLE} = \arg\max_{\theta} \sum_{n=1}^{N} \log p_{model}(y^{(n)}|x^{(n)},\theta) \tag{2.1.6}$$

$$= \arg\max_{\theta} \left(-N\log\sigma - \frac{N}{2}\log(2\pi) - \sum_{n=1}^{N} \frac{\left(y^{(n)} - f(x^{(n)},\theta)\right)^2}{2\sigma^2} \right) \tag{2.1.7}$$

$$= \arg\min_{\theta} \frac{1}{N} \sum_{n=1}^{N} \frac{\left(y^{(n)} - f(x^{(n)},\theta)\right)^2}{2} \tag{2.1.8}$$

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 (2.1.2), we have:

$$p_{model}(y|x,\theta) = \text{Ber}(y;\ \sigma(f(x,\theta))) \tag{2.1.9}$$

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:

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.1.10}$$

$$\text{Ber}(y;p) = p^{\mathbb{I}(y=1)}(1-p)^{\mathbb{I}(y=0)} \tag{2.1.11}$$

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 **goodfellow_deep_2016**.

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 $\boldsymbol{z}$ is defined as:

$$\sigma_i(\boldsymbol{z}) = \frac{e^{z_i}}{\sum_{k=1}^{C} e^{z_k}} \tag{2.1.12}$$

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:

$$\text{Cat}(y;\boldsymbol{\sigma}) = \prod_{c=1}^{C} \sigma_c^{\mathbb{I}(y_c=1)} \tag{2.1.13}$$

If we let $\hat{y}_c = \sigma_c(f(x,\theta))$, this makes (2.1.9):

$$p_{model}(y|x,\theta) = \text{Cat}(y;\ \sigma(f(x,\theta))) \tag{2.1.14}$$

$$= \prod_{c=1}^{C}\prod_{n=1}^{N} \left(\hat{y}_c^{(n)}\right)^{\mathbb{I}(y_c^{(n)}=1)} \tag{2.1.15}$$

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

$$\theta_{MLE} = \arg\max_{\theta} \prod_{c=1}^{C}\prod_{n=1}^{N} \left(\hat{y}_c^{(n)}\right)^{y_c^{(n)}} \tag{2.1.16}$$

$$= \arg\max_{\theta} \frac{1}{N}\sum_{n=1}^{N}\sum_{c=1}^{C} y_c^{(n)} \log \hat{y}_c^{(n)} \tag{2.1.17}$$

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

2.1.1 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:

$$\theta_{MAP} = \arg\max_{\theta}\ p(\theta|\mathcal{D},\sigma^2) \tag{2.1.18}$$

$$= \arg\max_{\theta}\ \frac{p(y|x,\theta,\sigma^2)p(\theta)}{p(y|x)} \tag{2.1.19}$$

$$= \arg\max_{\theta}\ \log p(y|x,\theta,\sigma^2) + \log p(\theta) \tag{2.1.20}$$

$$= \arg\min_{\theta}\ \frac{1}{N}\sum_{n=1}^{N} \frac{\left(y^{(n)} - f(x^{(n)},\theta)\right)^2}{2} + \frac{\lambda}{2}||\theta||_2^2 \tag{2.1.21}$$

where $\lambda = \sigma^2/\tau^2$ is the ratio of the observation noise to the strength of the prior **murphy__machine__2012**. 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.

2.1.2 Loss Functions and Minimizing the Objective

It may be useful to rewrite (2.1.18) as an objective function on the parameters $J(\theta)$:

$$J(\theta) = \frac{1}{N}\sum_{n=1}^{N}\frac{\left(y^{(n)} - f(x^{(n)},\theta)\right)^2}{2} + \frac{\lambda}{2}||\theta||_2^2 \tag{2.1.22}$$

$$= L_{data}(y, f(x,\theta)) + L_{reg}(\theta) \tag{2.1.23}$$

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:

$$\theta_{t+1} = \theta_t - \eta\frac{\partial J}{\partial \theta} \tag{2.1.24}$$

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 **boyd__convex__2004**. An example of this is shown in Figure 2.1. In this figure, the step size is chosen with exact line search, i.e.

$$\eta = \arg\min_s f(x + s\frac{\partial f}{\partial x}) \tag{2.1.25}$$

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 **boyd__convex__2004**. 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.

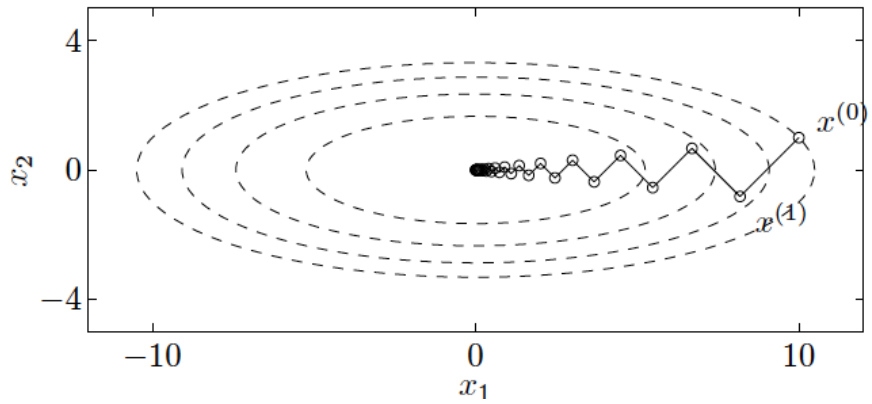


Figure 2.1: **Trajectory of gradient descent in an ellipsoidal parabola.** Some contour lines of the function $f(x) = 1/2\left(x_1^2 + 10x_2^2\right)$ and the trajectory of GD optimization using exact line search. This space has condition number 10, and shows the slow convergence of GD in spaces with largely different eigenvalues. Image taken from **boyd_convex_2004** Figure 9.2.

2.1.3 Stochastic Gradient Descent

Aside from the problems associated the curvature of the function $J(\theta)$, another common issue faced with the gradient descent of (2.1.24) is the cost of computing $\frac{\partial J}{\partial \theta}$. In particular, the first term:

$$L_{data}(y, f(x,\theta)) = \mathbb{E}_{x,y\sim p_{data}}\left[L_{data}(y, f(x,\theta))\right] \tag{2.1.26}$$

$$= \frac{1}{N}\sum_{n=1}^{N} L_{data}\left(y^{(n)}, f(x^{(n)},\theta)\right) \tag{2.1.27}$$

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.

(2.1.26) 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 **wilson_general_2003**. Increasing the batch size to larger values allows you to easily parallelize computation as well as increasing your accuracy for the gradient, allowing you to take larger step sizes **smith_dont_2017**. A good initial starting point is to set the batch size to about 100 samples and increase/decrease from there **goodfellow_deep_2016**.

2.1.4 Gradient Descent and Learning Rate

The step size parameter, η in (2.1.24) is commonly referred to as the learning rate. Choosing the right value for the learning rate is key. Unfortunately, the line search algorithm in (2.1.25) 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' **goodfellow_deep_2016**, but **bottou_stochastic_2012-1**, **montavon_neural_2012** 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 **lecun_efficient_2012** 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 **bottou_stochastic_2012-1**:

$$\sum_{t=1}^{\infty} \eta_t \to \infty \tag{2.1.28}$$

$$\sum_{t=1}^{\infty} \eta_t^2 < \infty \tag{2.1.29}$$

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 **bottou_stochastic_2012-1**.

2.1.5 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.

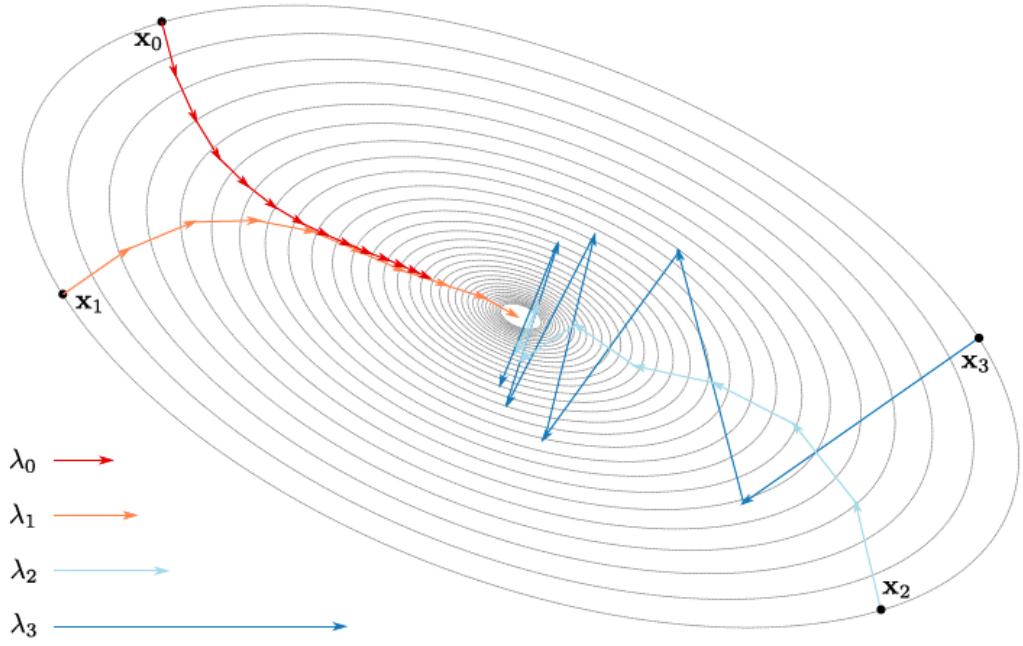


Figure 2.2: **Trajectories of SGD with different initial learning rates.** This figure illustrates the effect the step size has over the optimization process by showing the trajectory for $\eta = \lambda_i$ from equivalent starting points on a symmetric loss surface. Increasing the step size beyond λ_3 can cause the optimization procedure to diverge. Image taken from **Ioannou2017thesis** Figure 2.7.

We can achieve momentum by creating a *velocity* variable v_t and modify (2.1.24) to be:

$$v_{t+1} = \alpha v_t - \eta_k \frac{\partial J}{\partial \theta} \tag{2.1.30}$$

$$\theta_{t+1} = \theta_t + v_{t+1} \tag{2.1.31}$$

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 **kingma_adam:_2014**. There are several other adaptive schemes such as AdaGrad **duchi_adaptive_2011** and AdaDelta **zeiler_adadelta:_2012**, but they are all quite similar, and Adam is often considered the most robust of the three **goodfellow_deep_2016**. 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.

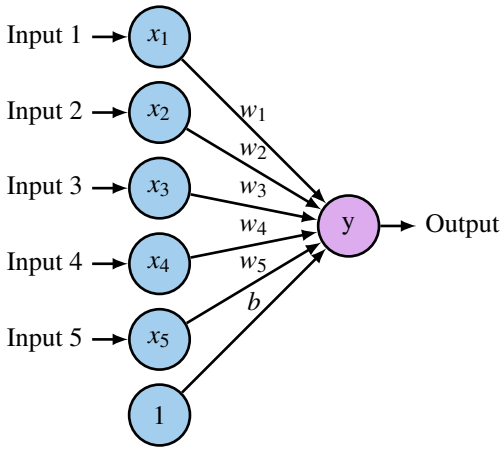


Figure 2.3: **A single neuron.** The neuron is composed of inputs x_i, weights w_i (and a bias term), as well as an activation function. Typical activation functions include the sigmoid function, tanh function and the ReLU

Adam does this by keeping track of the first m_t and second v_t moments of the gradients:

$$g_{t+1} = \frac{\partial J}{\partial \theta} \tag{2.1.32}$$

$$m_{t+1} = \beta_1 m_t + (1-\beta_1) g_{t+1} \tag{2.1.33}$$

$$v_{t+1} = \beta_2 v_t + (1-\beta_2) g_{t+1}^2 \tag{2.1.34}$$

where $0 \leq \beta_1, \beta_2 < 1$. Note the similarity between updating the mean estimate in (2.1.33) and the velocity term in (2.1.30)[1]. The parameters are then updated with:

$$\theta_{t+1} = \theta_t - \eta \frac{m_{t+1}}{\sqrt{v_{t+1}} + \epsilon} \tag{2.1.35}$$

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

2.2 Neural Networks

2.2.1 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

[1] The 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.

generic nonlinearity σ. I.e.

$$y = \sigma(\langle \mathbf{x}, \mathbf{w} \rangle) = \sigma\left(\sum_{i=0}^{D} x_i w_i\right) \tag{2.2.1}$$

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.

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

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.2.2}$$

$$\mathrm{ReLU}(x) = \max(x, 0) \tag{2.2.3}$$

See Figure 2.4 for plots of these. The original Rosenblatt perceptron **rosenblatt__perceptron:__1958** also used the Heaviside function $\mathrm{H}(x) = \mathbb{I}(x > 0)$.

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 **glorot__deep__2011**, **nair__rectified__2010**. 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 **glorot__understanding__2010**. The ReLU, on the other hand, has constant gradient.

2. It promotes sparsity in outputs, by setting them to a hard 0. Studies on brain energy expenditure suggest that neurons encode information in a sparse manner. **lennie__cost__2003** estimates the percentage of neurons active at the same time to be between 1 and 4%. Sigmoid and tanh functions will typically have *all* neurons firing, while the ReLU can allow neurons to fully turn off.

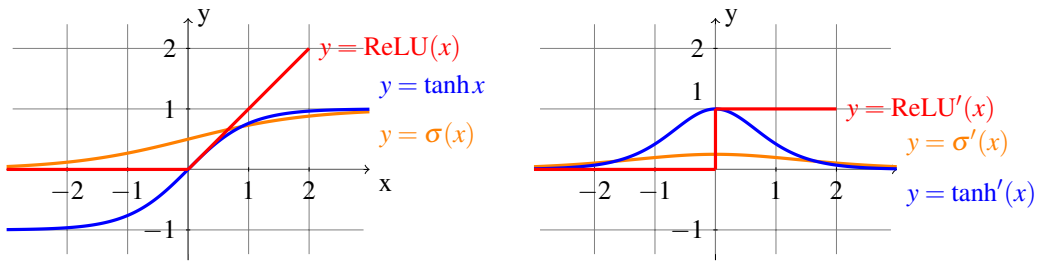


Figure 2.4: **Common Neural Network nonlinearities and their gradients.** The sigmoid, tanh and ReLU nonlinearities are commonly used activation functions for neurons. The tanh and sigmoid have the nice property of being smooth but can have saturation when the input is either largely positive or largely 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.

2.2.2 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 **minsky_perceptrons:_1988**. 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 **hornik_multilayer_1989**, **cybenko_approximation_1989**. This extension is called a multilayer perceptron, or MLP.

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

$$h_i = \sigma\left(\sum_{j=0}^{D} x_j w_{ij}^{(1)}\right) \tag{2.2.4}$$

$$y = \sum_{k=0}^{H} h_k w_k^{(2)} \tag{2.2.5}$$

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 L such fully connected layers, we could rewrite the action of each layer in a recursive fashion:

$$Y^{(l+1)} = W^{(l+1)} X^{(l)} \tag{2.2.6}$$

$$X^{(l+1)} = \sigma\left(Y^{(l+1)}\right) \tag{2.2.7}$$

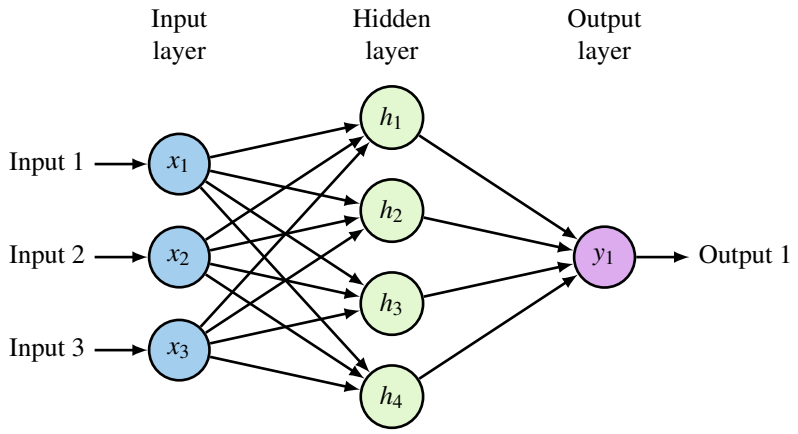


Figure 2.5: **Multi-layer perceptron.** Expanding the single neuron from Figure 2.3 to a network of neurons. The internal representation units are often referred to as the *hidden layer* as they are an intermediary between the input and output.

where W is now a weight matrix, acting on the vector of 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 $Y^{(L)}$ is the network output. Let us say that the output has C nodes, and a hidden layer $X^{(l)}$ has C_l nodes.

2.2.3 Backpropagation

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

The delta rule, initially designed for networks with no hidden layers **widrow_neurocomputing:_1988**, was expanded to what we now consider *backpropagation* in **rumelhart_parallel_1986**. 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 $\frac{\partial J}{\partial w}$ 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 L layers of the network, $W^{(1)}, W^{(2)}, \ldots W^{(L)}$ however, only the final set $W^{(L)}$ are connected to the objective function J.

2.2.3.1 Regression Loss

Let us start with writing down the derivative of J with respect to the network output $Y^{(L)}$ using the regression objective function (2.1.8). As we now have two superscripts, one for the

sample number and one for the layer number, we combine them into a tuple of superscripts.

$$\frac{\partial J}{\partial Y^{(L)}} = \frac{\partial}{\partial Y^{(L)}} \left(\frac{1}{N} \sum_{n=1}^{N_b} \frac{1}{2} \left(y^{(n)} - Y^{(L,n)} \right)^2 \right) \tag{2.2.8}$$

$$= \frac{1}{N} \sum_{n=1}^{N_b} \left(Y^{(L,n)} - y^{(n)} \right) \tag{2.2.9}$$

$$= e \in \mathbb{R} \tag{2.2.10}$$

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

2.2.3.2 Classification Loss

For the classification case (2.1.17), let us keep the output of the network $Y^{(L,n)} \in \mathbb{R}^C$ and define an intermediate value $\hat{y}$ the softmax applied to this vector $\hat{y}_c^{(n)} = \sigma_c \left(Y^{(L,n)} \right)$. Note that the softmax is a vector-valued function going from $\mathbb{R}^C \to \mathbb{R}^C$ so has a Jacobian matrix $S_{ij} = \frac{\partial \hat{y}_i}{\partial Y_j^{(L)}}$ with values:

$$S_{ij} = \begin{cases} \sigma_i(1 - \sigma_j) & \text{if } i = j \\ -\sigma_i \sigma_j & \text{if } i \neq j \end{cases} \tag{2.2.11}$$

Now, let us return to (2.1.17) and find the derivative of the objective function to this intermediate value $\hat{y}$:

$$\frac{\partial J}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}} \left(\frac{1}{N} \sum_{n=1}^{N_b} \sum_{c=1}^{C} y_c^{(n)} \log \hat{y}_c^{(n)} \right) \tag{2.2.12}$$

$$= \frac{1}{N} \sum_{n=1}^{N_b} \sum_{c=1}^{C} \frac{y_c^{(n)}}{\hat{y}_c^{(n)}} \tag{2.2.13}$$

$$= d \in \mathbb{R}^C \tag{2.2.14}$$

Note that unlike (2.2.10), this derivative is vector-valued. To find $\frac{\partial J}{\partial Y^{(L)}}$ 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.:

$$\frac{\partial J}{\partial Y_j^{(L)}} = \sum_{i=1}^{C} \frac{\partial J}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial Y_j^{(L)}} \tag{2.2.15}$$

$$= S_j^T d \tag{2.2.16}$$

where S_j is the jth column of the Jacobian matrix S. It becomes clear now that to get the entire vector derivative for all nodes in $Y^{(L)}$, we must multiply the transpose of the Jacobian

matrix with the error term from (2.2.14):

$$\frac{\partial J}{\partial Y^{(L)}} = S^T d \tag{2.2.17}$$

2.2.3.3 Final Layer Weight Gradient

Let us continue by assuming $\frac{\partial J}{\partial Y^{(L)}}$ is vector-valued as was the case with classification. For regression, it is easy to set $C = 1$ in the following to get the necessary results. For clarity we drop the layer superscript in the intermediate calculations.

We call the gradient for the final layer weights the *update* gradient. It can be computed by the chain rule again:

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial Y_i}\frac{\partial Y_i}{\partial W_{ij}} + 2\lambda W_{ij} \tag{2.2.18}$$

$$= \frac{\partial J}{\partial Y_i} X_j + 2\lambda W_{ij} \tag{2.2.19}$$

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

$$\frac{\partial J}{\partial W^{(L)}} = \frac{\partial J}{\partial \hat{y}} X^T + 2\lambda W \tag{2.2.20}$$

$$= S^T d \left(X^{(L-1)}\right)^T + 2\lambda W^{(L)} \in \mathbb{R}^{C \times C_{L-1}} \tag{2.2.21}$$

2.2.3.4 Final Layer Passthrough Gradient

We also want to find the *passthrough* gradients of the final layer $\frac{\partial J}{\partial X^{(L-1)}}$. In a similar fashion, we first find the gradient with respect to individual elements in $X^{(L-1)}$ before generalizing to the entire vector:

$$\frac{\partial J}{\partial X_i} = \sum_{j=1}^{C} \frac{\partial J}{\partial Y_j}\frac{\partial Y_j}{\partial X_i} \tag{2.2.22}$$

$$= \sum_{j=1}^{C} \frac{\partial J}{\partial Y_j} W_{j,i} \tag{2.2.23}$$

$$= W_i^T \frac{\partial J}{\partial Y} \tag{2.2.24}$$

$$\tag{2.2.25}$$

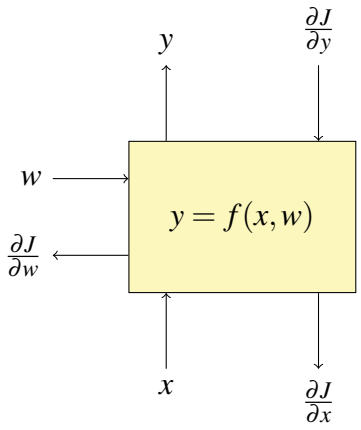


Figure 2.6: **General block form for autograd.** All neural network functions need to be able to calculate the forward pass $y = f(x, w)$ as well as the update and passthrough gradients $\frac{\partial J}{\partial w}, \frac{\partial J}{\partial x}$. Backpropagation is then easily done by allowing data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

where W_i is the ith column of W. Thus

$$\frac{\partial J}{\partial X^{(L-1)}} = \left(W^{(L)}\right)^T \frac{\partial J}{\partial Y^{(L)}} \tag{2.2.26}$$

$$= \left(W^{(L)}\right)^T S^T d \tag{2.2.27}$$

This passthrough gradient then can be used to update the next layer's weights by repeating subsubsection 2.2.3.3 and subsubsection 2.2.3.4.

2.2.3.5 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 $\frac{\partial J}{\partial w}, \frac{\partial J}{\partial x}$ given an input gradient $\frac{\partial J}{\partial y}$. 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.

2.3 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 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 **lecun_gradient-based_1998**, 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 **hubel_receptive_1962**. They did not become popular immediately, and another spatially based keypoint extractor, SIFT **lowe_distinctive_2004**, was the mainstay of detection systems until the AlexNet CNN **krizhevsky_imagenet_2012** won the 2012 ImageNet challenge **russakovsky_imagenet_2014** by a large margin over the next competitors, many of who 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.

2.3.1 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:

$$y[\mathbf{n}] = (x * h)[\mathbf{n}] = \sum_{\mathbf{k}} x[\mathbf{k}]h[\mathbf{n}-\mathbf{k}] \tag{2.3.1}$$

$$= \sum_{k_1,k_2} x[k_1,k_2]h[n_1-k_1,n_2-k_2] \tag{2.3.2}$$

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 its largest 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 other[2], 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 3 dimensional with depth exactly equal to C. The convolution is done only over the two spatial

[2]In 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 **paszke_automatic_2017**. For this reason, we stack channels along the first dimension of our tensors.

dimensions and the C outputs are summed at each pixel location. This makes (2.3.1):

$$y[\mathbf{n}] = \sum_{c=0}^{C-1} \sum_{\mathbf{k}} x[c, \mathbf{k}] h[c, \mathbf{n} - \mathbf{k}] \tag{2.3.3}$$

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 (2.3.3) to give the output $y \in \mathbb{R}^{C_{l+1} \times H \times W}$:

$$y[f, \mathbf{n}] = \sum_{c=0}^{C-1} \sum_{\mathbf{k}} x[c, \mathbf{k}] h_f[c, \mathbf{n} - \mathbf{k}] \tag{2.3.4}$$

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 (2.2.6) and (2.2.7), we can rewrite this for a convolutional layer at depth l with C_l input and C_{l+1} output channels:

$$Y^{(l+1)}[f, \mathbf{n}] = \sum_{c=0}^{C_l - 1} X^{(l)}[c, \mathbf{n}] * h_f^{(l)}[c, \mathbf{n}] \tag{2.3.5}$$

$$X^{(l+1)}[f, \mathbf{n}] = \sigma\left(Y^{(l)}[f, \mathbf{n}]\right) \tag{2.3.6}$$

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

2.3.1.1 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 **lecun_gradient-based_1998**.

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.

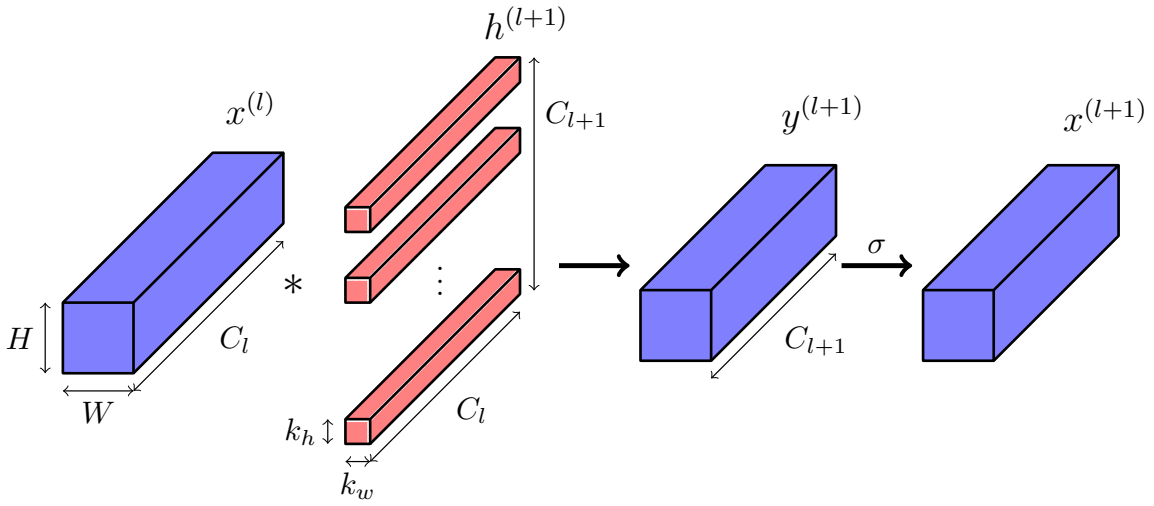


Figure 2.7: **A convolutional layer.** A convolutional layer followed by a nonlinearity σ. The previous layer's activations are convolved with a bank of C_{l+1} filters, each of which has spatial size $k_h \times k_w$ and depth C_l. Note that there is no convolution across the channel dimension. Each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

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.

2.3.1.2 Gradients

To get the update and passthrough gradients for the convolutional layer we will need to expand (2.3.5). Again we will drop the layer superscripts for clarity:

$$Y[f, n_1, n_2] = \sum_{c=0}^{C-1} \sum_{k_1} \sum_{k_2} X[c, k_1, k_2] h_f[c, n_1 - k_1, n_2 - k_2] \tag{2.3.7}$$

The derivative for an activation in Y to a single activation in X is then simply:

$$\frac{\partial Y_{f,n_1,n_2}}{\partial X_{c,k_1,k_2}} = h_f[c, n_1 - k_1, n_2 - k_2] \tag{2.3.8}$$

It is clear from (2.3.7) that a single activation $X[c, n_1, n_2]$ 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:

$$\frac{\partial J}{\partial X_{c,k_1,k_2}} = \sum_f \sum_{n_1} \sum_{n_2} \frac{\partial J}{\partial Y_{f,n_1,n_2}} \frac{\partial Y_{f,n_1,n_2}}{\partial X_{c,k_1,k_2}} \tag{2.3.9}$$

$$= \sum_f \sum_{n_1} \sum_{n_2} \frac{\partial J}{\partial Y_{f,n_1,n_2}} h_f[c, n_1 - k_1, n_2 - k_2] \tag{2.3.10}$$

Now we let $\Delta Y[f, n_1, n_2] = \frac{\partial J}{\partial Y_{f,n_1,n_2}}$ 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 the order of their channel and filter number swapped. Combining these two and substituting into (2.3.10) we get the passthrough gradient for the convolutional layer:

$$\frac{\partial J}{\partial X_{c,k_1,k_2}} = \sum_f \sum_{n_1} \sum_{n_2} \Delta Y[f, n_1, n_2] \tilde{h}_c[f, k_1 - n_1, k_2 - n_2] \tag{2.3.11}$$

$$= \sum_f \Delta Y[f, \mathbf{n}] * \tilde{h}_c[f, \mathbf{n}] \tag{2.3.12}$$

which is the same as (2.3.5). 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:

$$\frac{\partial J}{\partial h_{f,c,k_1,k_2}} = \sum_{n_1} \sum_{n_2} \frac{\partial J}{\partial Y_{f,n_1,n_2}} \frac{\partial Y_{f,n_1,n_2}}{\partial h_{f,c,k_1,k_2}} \tag{2.3.13}$$

$$= \sum_{n_1} \sum_{n_2} \Delta Y[f, n_1, n_2] X[x, n_1 - k_1, n_2 - k_2] \tag{2.3.14}$$

$$= (\Delta Y[f, \mathbf{n}] \star X[c, \mathbf{n}])[k_1, k_2] \tag{2.3.15}$$

where $\star$ is the cross-correlation operation.

2.3.2 Pooling

Pooling layers are common in CNNs where we want to reduce the 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

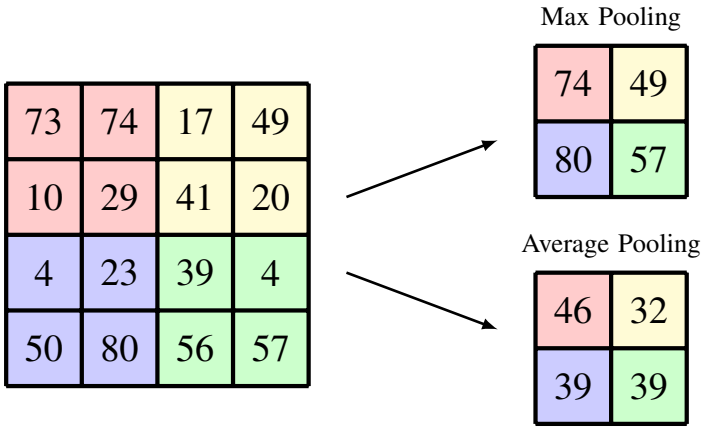


Figure 2.8: **Max vs Average 2×2 pooling.**

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 **mishkin__systematic__2016** found both max and average pooling to perform similarly well. While max pooling was the most popular in earlier state of the art networks **krizhevsky__imagenet__2012**, **simonyan__very__2014**, there has been a recent trend towards using average pooling **huang__densely__2017** or even to do away with pooling altogether in favour of strided convolutions (this idea was originally proposed in **springenberg__striving__2014-3** and used notably in **he__deep__2016**, **xie__aggregated__2017**, **zagoruyko__wide__2016-1**).

2.3.3 Dropout

Dropout is a particularly strong regularization scheme that randomly turns off, or 'zeros out', neurons in a neural network **hinton__improving__2012**, **srivastava__dropout:__2014**. 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 **srivastava__dropout:__2014**.

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 **gal__dropout__2016**.

2.3.4 Batch Normalization

Batch normalization proposed in **ioffe__batch__2015** 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.

$$\mu_c^{(l)} = \frac{1}{N}\sum_{i=0}^{N-1}\sum_{\mathbf{n}} X^{(l,i)}[c,\mathbf{n}] \tag{2.3.16}$$

$$(\sigma_c^{(l)})^2 = \frac{1}{N}\sum_{i=0}^{N-1}\sum_{\mathbf{n}} \left(X^{(l,i)}[c,\mathbf{n}]\right)^2 - (\mu_c^{(l)})^2 \tag{2.3.17}$$

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:

$$Y[c,\mathbf{n}] = \frac{X[c,\mathbf{n}] - \mu_c}{\sigma_c + \epsilon}\gamma_c + \beta_c \tag{2.3.18}$$

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:

$$\frac{\partial J}{\partial X_{c,n_1,n_2}} = \frac{\partial J}{\partial Y_{c,n_1,n_2}}\frac{\gamma}{\sigma+\epsilon} \tag{2.3.19}$$

$$\frac{\partial J}{\partial \beta_c} = \sum_{\mathbf{n}} \frac{\partial J}{\partial Y_{c,\mathbf{n}}} \tag{2.3.20}$$

$$\frac{\partial J}{\partial \gamma_c} = \sum_{\mathbf{n}} \frac{\partial J}{\partial Y_{c,\mathbf{n}}}\frac{X_{c,\mathbf{n}} - \mu_c}{\gamma_c + \epsilon} \tag{2.3.21}$$

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 an increase in weight scale does not change the passthrough gradients and leads to *smaller* update gradients **ioffe_batch_2015**, making the network more resilient to the problems of vanishing and exploding gradients:

$$\begin{aligned}\frac{\partial BN((aW)X)}{\partial X} &= \frac{\partial BN(WX)}{\partial X} \\ \frac{\partial BN((aW)X)}{\partial (aW)} &= \frac{1}{a}\cdot\frac{\partial BN(WX)}{\partial W}\end{aligned} \tag{2.3.22}$$

2.4 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.

2.4.1 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 there are 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 **lecun__modified__1998**.

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 **krizhevsky__learning__2009** and files at **krizhevsky__cifar__2009**.

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 **krizhevsky__learning__2009** and files at **krizhevsky__cifar__2009**.

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 **li__tiny__2017**.

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 **russakovsky__imagenet__2014** and the most reliable source of the data can be found at **stanford__vision__lab__imagenet__2017**.

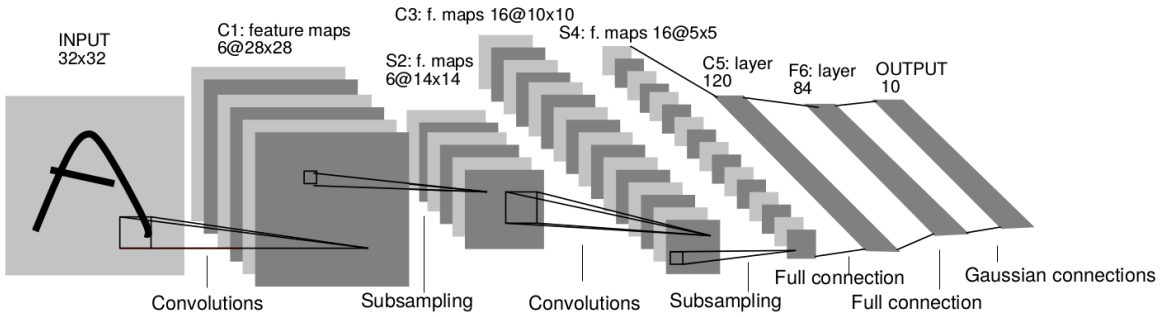


Figure 2.9: **LeNet-5 architecture.** The 'original' CNN architecture used for handwriting recognition. LeNet has 2 convolutional and 3 fully connected layers making 5 parameterized layers. After the second convolutional layer, the $16 \times 5 \times 5$ pixel output is unravelled to a 400 long vector. Image taken from **lecun_gradient-based_1998**.

Several other classification datasets do exist but are not commonly used, such as PASCAL VOC **Everingham15** and Caltech-101 and Caltech-256 **li_fei-fei_learning_2004**[3].

2.4.2 LeNet

LeNet-5 **lecun_gradient-based_1998** 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.

2.4.3 AlexNet

We have already seen AlexNet **krizhevsky_imagenet_2012** in chapter 1. It is arguably one of the most important architectures in the development in CNNs as it was able to experimentally prove that CNNs can be used for complex tasks. This required a few innovations: they used multiple GPUs to do fast processing on large images, they used the ReLU to avoid saturation, and also added dropout to aid generalization. Training of AlexNet on 2 GPUs available in 2012 takes roughly a week.

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

[3]Tiny ImageNet is also not commonly used as it is quite new. We have included it 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.

2.4.4 VGGnet

The Visual Geometry Group (VGG) at Oxford came second in the ILSVRC challenge in 2014 with their VGG-nets **simonyan__very__2014**, but remains an important network for some of the design choices it inspired. Their optimal network 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 its simplicity, compared to the more complex Inception Network **szegedy__going__2015** 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.

2.4.5 The All Convolutional Network

The All Convolutional Network **springenberg__striving__2014-3** 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 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.

2.4.6 Residual Networks

Resiudal 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 **zagoruyko__wide__2016-1**, **xie__aggregated__2017**.

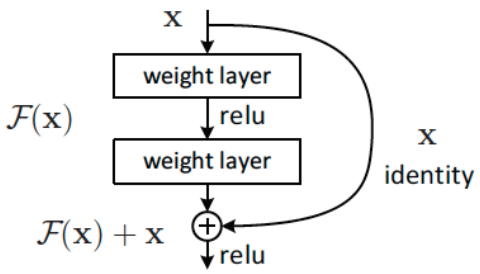


Figure 2.10: **The residual unit from ResNet.** A residual unit. The identity mapping is always present, and the network learns the difference from the identity mapping, $\mathcal{F}(x)$. Taken from **he__deep__2016**.

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 analysis of a ResNet without nonlinearities **bartlett__representing__2018**, **bartlett__gradient__2018** 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.

2.5 The Fourier and Wavelet Transforms

Computer vision is an extremely difficult task. 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 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.

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.

2.5.1 The Fourier Transform

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

$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-j\omega t} dt \tag{2.5.1}$$

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

$$F(\boldsymbol{\omega}) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\mathbf{u}) e^{-j\boldsymbol{\omega}^t \mathbf{u}} d\mathbf{u} = \langle f(\mathbf{u}),\ e^{j\boldsymbol{\omega}^t \mathbf{u}} \rangle \tag{2.5.2}$$

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 $e^{j\boldsymbol{\omega}^t u}$. 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 **antoine_two-dimensional_2004**.

2.5.2 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:

$$\int_{-\infty}^{\infty} \psi(\mathbf{u}) d\mathbf{u} = \Psi(0) = 0 \tag{2.5.3}$$

Any function that has sufficient decay of energy with frequency and satisfies (2.5.3), 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

$$\text{Rotation: } R_\theta x(\mathbf{u}) = x(r_{-\theta}\mathbf{u}) \tag{2.5.4}$$

$$\text{Dilation: } D_a x(\mathbf{u}) = \frac{1}{a} x\left(\frac{\mathbf{u}}{a}\right) \tag{2.5.5}$$

$$\text{Translation: } T_\mathbf{b} x(\mathbf{u}) = x(\mathbf{u} - \mathbf{b}) \tag{2.5.6}$$

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

$$\psi_{\mathbf{b},a,\theta}(\mathbf{u}) = \frac{1}{a} \psi\left(\frac{r_{-\theta}(\mathbf{u} - \mathbf{b})}{a}\right) \tag{2.5.7}$$

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

$$CWT_x(\mathbf{b}, a, \theta) = \int_{-\infty}^{\infty} \psi^*_{\mathbf{b},a,\theta}(\mathbf{u}) x(\mathbf{u}) d\mathbf{u} = \langle \psi_{\mathbf{b},a,\theta}(\mathbf{u}),\ x(\mathbf{u}) \rangle \tag{2.5.8}$$

2.5.2.1 Properties

The CWT has some particularly nice properties, such as *covariance* under the three transformations (2.5.6)-(2.5.4):

$$R_{\theta_0} x \to CWT_x\left(r_{-\theta_0}\mathbf{b}, a, \theta + \theta_0\right) \tag{2.5.9}$$

$$D_{a_0} x \to CWT_x\left(\mathbf{b}/a_0, a/a_0, \theta\right) \tag{2.5.10}$$

$$T_{\mathbf{b}_0} x \to CWT_x\left(\mathbf{b} - \mathbf{b}_0, a, \theta\right) \tag{2.5.11}$$

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.

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

2.5.2.2 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 **holschneider_pointwise_1991**. Assuming these constraints are satisfied, we can recover x from CWT_x.

2.5.2.3 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 $(\mathbf{b}, a, \theta)$ 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 **vetterli_wavelets_2007** and in 2-D we recommend **antoine_two-dimensional_2004**.

2.5.3 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 $(\mathbf{b}, a, \theta)$ 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 $x \in \mathbb{H}$:

$$A||x||^2 \leq \sum_{i \in I} |\langle x, \varphi_i \rangle|^2 \leq B||x||^2 \tag{2.5.12}$$

with A, B called the *frame bounds* **kovacevic_introduction_2008**. 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 more vectors than dimensions, and this will be the case with many expansions we explore in this thesis.

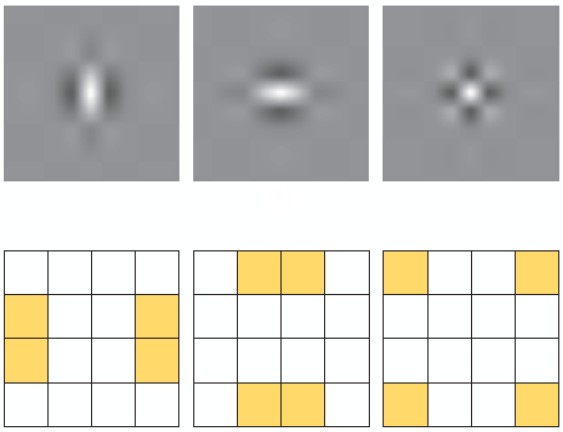


Figure 2.12: **Typical wavelets from the 2D separable DWT.** Top: Wavelet point spread functions for ψ^v (low-high), ψ^h (high-low), and ψ^d (high-high) wavelets. High-high wavelets are in a checkerboard pattern, with no favoured orientation. Bottom: Idealized support of the spectra of each of the wavelets. Image taken from **selesnick_dual-tree_2005**.

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.

2.5.3.1 Inversion and Tightness

(2.5.12) specify the constraints that make a frame representation invertible. The tighter the frame bounds, the more easily it is to invert the signal. This gives us some 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 $\tilde{\Phi} = \{\tilde{\varphi}\}_{i \in I}$ where:

$$\tilde{\varphi}_i = S^{-1}\varphi_i \tag{2.5.13}$$

then from **kovacevic_introduction_2008** we have:

$$x = \sum_{i \in I} \langle x, \varphi_i \rangle \tilde{\varphi}_i = \sum_{i \in I} \langle x, \tilde{\varphi}_i \rangle \varphi_i \tag{2.5.14}$$

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

2.5.4 Discrete Wavelet Transform

(2.5.7) 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:

$$\Psi_{j,l}(t) = \frac{1}{\sqrt{a}}\Psi\left(\frac{t-b}{a}\right) = 2^{-j/2}\Psi\left(2^{-j}t - l\right) \tag{2.5.15}$$

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:

$$\phi(\mathbf{u}) = \Phi(u_1)\Phi(u_2) \tag{2.5.16}$$
$$\psi^h(\mathbf{u}) = \Phi(u_1)\Psi(u_2) \tag{2.5.17}$$
$$\psi^v(\mathbf{u}) = \Psi(u_1)\Phi(u_2) \tag{2.5.18}$$
$$\psi^d(\mathbf{u}) = \Psi(u_1)\Psi(u_2) \tag{2.5.19}$$

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 (2.5.16) – (2.5.19), define the daughter wavelets as:

$$\phi^j_{lm}(\mathbf{u}) = \Phi_{j,l}(u_1)\Phi_{j,m}(u_2) \tag{2.5.20}$$
$$\psi^{h,j}_{lm}(\mathbf{u}) = \Phi_{j,l}(u_1)\Psi_{j,m}(u_2) \tag{2.5.21}$$
$$\psi^{v,j}_{lm}(\mathbf{u}) = \Psi_{j,l}(u_1)\Phi_{j,m}(u_2) \tag{2.5.22}$$
$$\psi^{d,j}_{lm}(\mathbf{u}) = \Psi_{j,l}(u_1)\Psi_{j,m}(u_2) \tag{2.5.23}$$

for $l, m \in \mathbb{Z}$ where l, m define horizontal and vertical translation. We can then get an orthonormal basis with the set $\{\phi^j_{lm}, \psi^{h,j}_{lm}, \psi^{v,j}_{lm}, \psi^{d,j}_{lm}\}_{j,l,m}$. 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.

2.5.4.1 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 **mallat_wavelet_1998**, **coifman_translation-invariant_1995**, but it comes with a heavy computational and memory cost.

2.5.5 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 sinusoids[5]:

$$e^{j\omega t} = \cos(\omega t) + j\sin(\omega t) \tag{2.5.24}$$

whereas the DWT's basis functions are based on only the real sinusoid, $\cos(\omega t)$. 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:

$$\psi_c(t) = \psi_r(t) + j\psi_i(t) \tag{2.5.25}$$

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

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

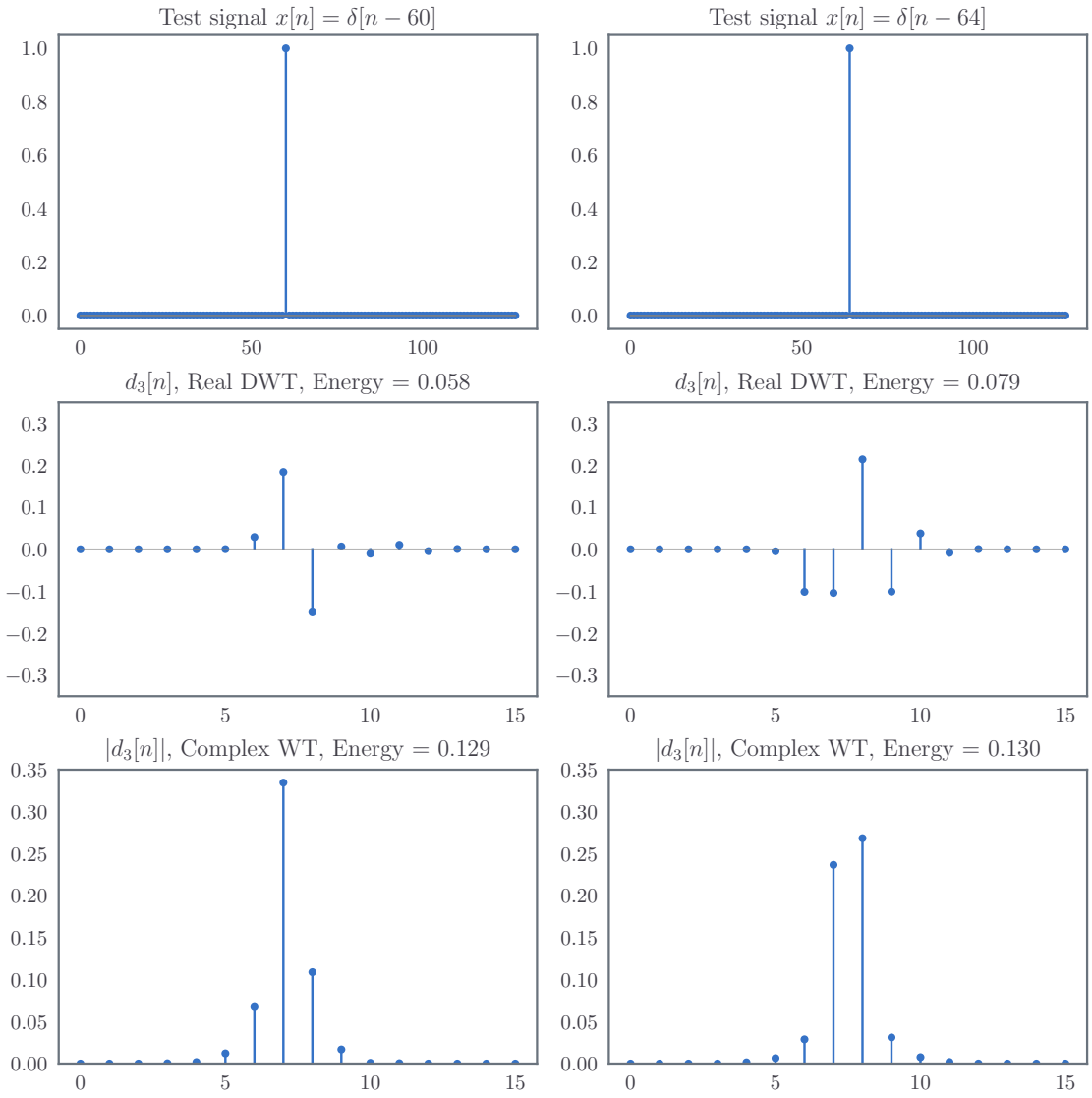


Figure 2.13: **Sensitivity of DWT coefficients to zero crossings and small shifts.** Two impulse signals $\delta(n-60)$ and $\delta(n-64)$ are shown (top), as well as the wavelet coefficients for scale $j=3$ for the DWT (middle) and for the DT$\mathbb{C}$WT (bottom). In the middle row, not only are the coefficients very different from a shifted input, but the energy has almost doubled. As the DWT is an orthonormal transform, this means that this extra energy has come from other scales. In comparison, the energy of the magnitude of the DT$\mathbb{C}$WT coefficients has remained far more constant, as has the shape of the envelope of the output. Image adapted from **selesnick_dual-tree_2005**.

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 (DT$\mathbb{C}$WT) developed by Kingsbury **kingsbury__wavelet__1997**, **kingsbury__dual-tree__1998**, **kingsbury__dual-tree__1998-1**, **kingsbury__image__1999**, **kingsbury__shift__1999**, **kingsbury__dual-tree__2000**, **kingsbury__complex__2001**, **selesnick__dual-tr**

We look at the Morlet wavelet transform because it is used by Mallat et. al. in their scattering transform **bruna__classification__2011**, **bruna__invariant__2013**, **bruna__scattering__2013**, **oyallon__generic__2013**, **oyallon__deep__2015**, **sifre__rotation__2013**, **sifre__rigid-motion__2014**, **sifre__rigid-motion__2014-1**, **sifre__scatnet__2013**. We believe the DT$\mathbb{C}$WT 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

$$\mathcal{W}x = \{x * \phi_J, x * \psi_\lambda\}_\lambda \tag{2.5.26}$$

where $\lambda = (j,k)$ with $j \in \{1,2,\dots J\}$ indexing the scale[6] and $k \in \{0,1,\dots K-1\}$ indexing the orientations of the chosen wavelet transform, whether it be the DT$\mathbb{C}$WT or Morlet transform.

2.5.6 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 (2.5.3) 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 2D):

$$\psi(\mathbf{u}) = \frac{1}{2\pi\sigma^2}(e^{i\mathbf{u}^t\boldsymbol{\xi}} - \beta)e^{-\frac{|\mathbf{u}|^2}{2\sigma^2}} \tag{2.5.27}$$

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

bruna__invariant__2013 **bruna__invariant__2013** 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 makes (2.5.27):

$$\psi(\mathbf{u}) = \frac{\gamma}{2\pi\sigma^2}(e^{i\mathbf{u}^t\boldsymbol{\xi}} - \beta)e^{-\mathbf{u}^t\Sigma^{-1}\mathbf{u}} \tag{2.5.28}$$

[6] We try to number everything from zero in this thesis, but number j from 1 as is the practice in **kingsbury__complex__2001**, **selesnick__dual-tree__2005**.

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 $\sigma = 2, \gamma = 0.5$.

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.

2.5.6.1 Tightness and Invertibility

Recall our definition of the wavelet transform $\mathcal{W}$ from (2.5.26). Assuming the transform is bounded, we can always scale it so that it satisfies Plancherel's equality

$$\|\mathcal{W}x\| = \|x\| \tag{2.5.29}$$

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 (2.5.12) 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 $A(\boldsymbol{\omega})$ to be the coverage each wavelet family has over the frequency plane:

$$A(\boldsymbol{\omega}) = |\Phi_J(\boldsymbol{\omega})|^2 + \sum_{\lambda} |\Psi_\lambda(\boldsymbol{\omega})|^2 \tag{2.5.30}$$

For a unit norm input $||x||^2 = 1$ and scaled wavelets, we can now change (2.5.12) to be:

$$1 - \alpha \leq A(\boldsymbol{\omega}) \leq 1 \tag{2.5.31}$$

If $A(\boldsymbol{\omega})$ 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.

The tightness of the frame is determined by the sampling grid of our wavelets parameters $(\mathbf{b}, a, \theta)$. Common choices for sampling grids for 2-D wavelets are**antoine__two-dimensional__2004**:

- 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 $\theta_k = \frac{k\pi}{K},\ k = \{0, 1, \ldots K-1\}$.
- For the translations, set the offsets $\mathbf{b} = (l2^{j/Q},\ m2^{j/Q}),\ l, m \in \mathbb{Z}$.

2.5.7 The DT$\mathbb{C}$WT

The DT$\mathbb{C}$WT was first proposed by **kingsbury__dual-tree__1998** in **kingsbury__dual-tree__1998**, **kingsbury__dual-tree__1998-1** 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 **kingsbury__image__1999**, **selesnick__dual-tree__2005**. Building on these properties, it been used successfully for denoising and inverse problems **rivaz__bayesian__2001**, **zhang__bayesian__2008**, **zhang__variational__2015**, **miller__image__2008**, texture classification **hatipoglu__texture__1999**, **rivaz__complex__1999**, image registration **loo__motion-estimation-based__2001**, **chen__efficient__2012** and SIFT-style keypoint generation matching **fauqueur__multiscale__2006**, **anderson__determining__2005**, **anderson__rotation-invariant__2006**, **bendale__multiscale__2010**, **ng__robust__2012** amongst many other applications. Compared to Gabor (or Morlet) image analysis, the authors of **selesnick__dual-tree__2005** 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 2.5.6. 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.

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

As in subsection 2.5.5, 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

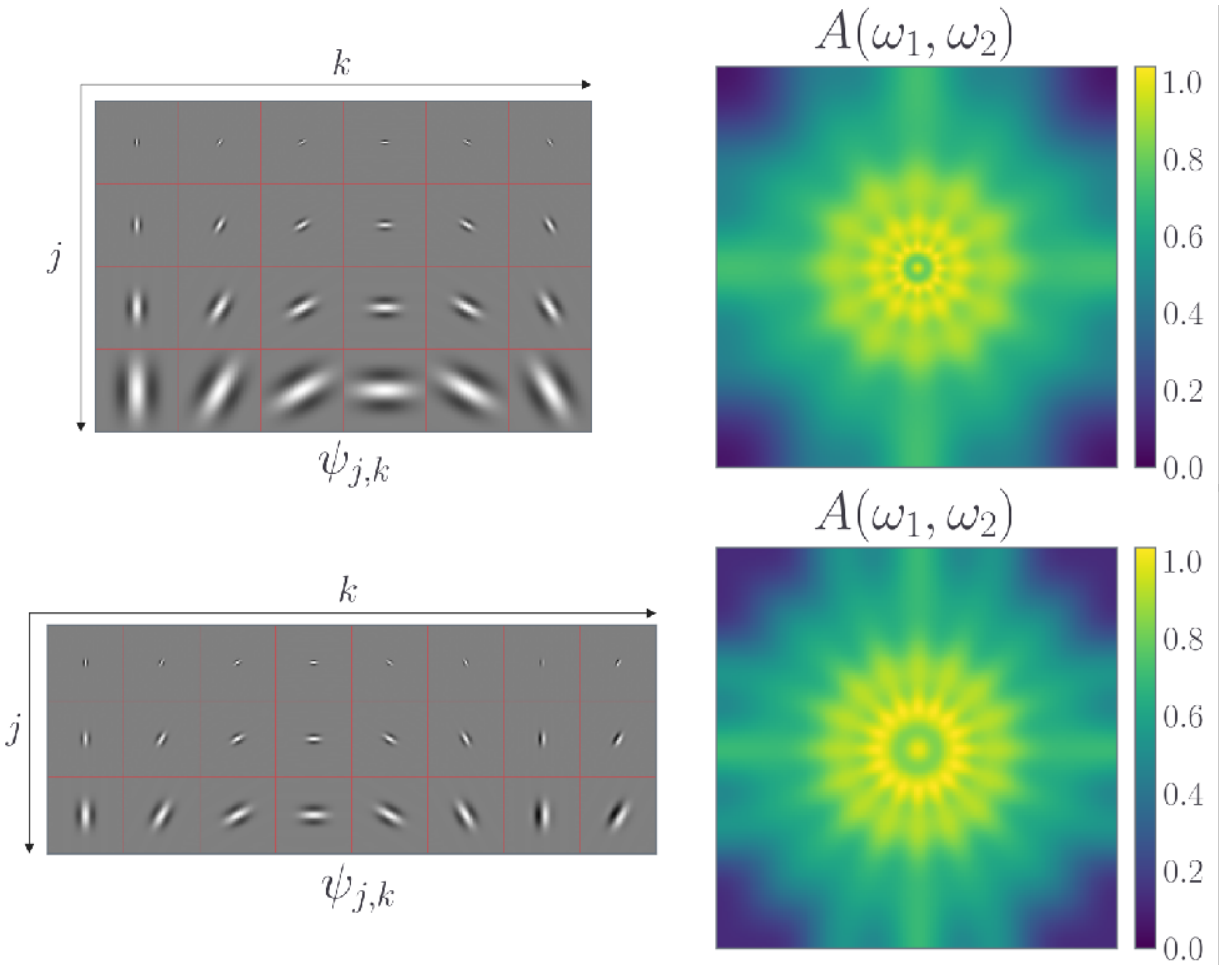


Figure 2.15: **Two Morlet Wavelet families and their tiling of the frequency plane.** For each set of parameters, the point spread functions of the *real* wavelet bases are shown, next to their covering of the frequency plane $A(\boldsymbol{\omega})$. Each square is 45×45 pixels. Top: $J = 3,\ K = 6,\ Q = 1$, 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 adapted from **sifre_rigid-motion_2014-1**.

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 $\tilde{\phi}_i, \tilde{\psi}_i$.

The dilation and wavelet equations for a 1D filter bank implementation are **selesnick__dual-tree__2005**:

$$\phi_r(t) = \sqrt{2}\sum_n h_0(n)\phi_r(2t-n) \tag{2.5.32}$$

$$\psi_r(t) = \sqrt{2}\sum_n h_1(n)\phi_r(2t-n) \tag{2.5.33}$$

$$\phi_i(t) = \sqrt{2}\sum_n g_0(n)\phi_i(2t-n) \tag{2.5.34}$$

$$\psi_i(t) = \sqrt{2}\sum_n g_1(n)\phi_i(2t-n) \tag{2.5.35}$$

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__image__1999** in **kingsbury__image__1999** (and later proved by **selesnick__hilbert__2001** in **selesnick__hilbert__2001**) 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

$$\psi_g(t) \approx \mathcal{H}\{\psi_h(t)\} \tag{2.5.36}$$

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 **kingsbury__complex__2001**, **kingsbury__design__2003**, and in general in **selesnick__dual-tree__2005**. 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.

2.5.7.2 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 2D wavelet transform with oriented wavelets.

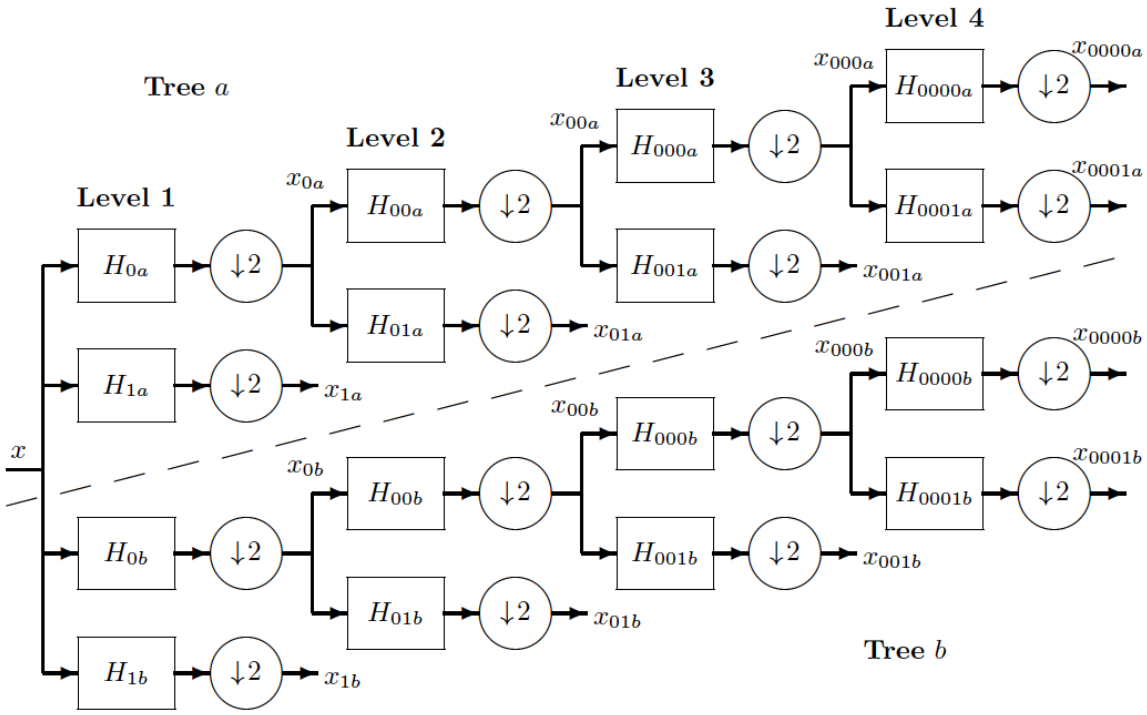


Figure 2.16: **Analysis FBs for the 1-D** DT$\mathbb{C}$WT**.** Top 'tree' forms the real component of the complex wavelet ψ_r, and the bottom tree forms the imaginary (Hilbert pair) component ψ_i. Image taken from **kingsbury_image_1999**.

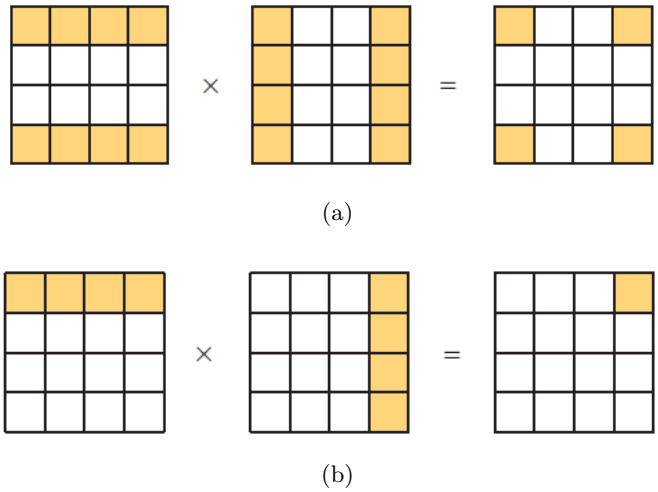


Figure 2.17: **The DWT high-high vs the** DT$\mathbb{C}$WT **high-high frequency support.** (a) The high-high DWT wavelet having a passband in all 4 corners of the frequency plane vs (b) the high-high DT$\mathbb{C}$WT wavelet frequency support only existing in one quadrant. Taken from **selesnick_dual-tree_2005**

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 2D 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:

$$\psi(\mathbf{u}) = \psi_c(u_1)\psi_c^*(u_2) \tag{2.5.37}$$

$$= (\psi_r(u_1) + j\psi_i(u_1))\,(\psi_r(u_2) - j\psi_i(u_2)) \tag{2.5.38}$$

$$= (\psi_r(u_1)\psi_r(u_2) + \psi_i(u_1)\psi_i(u_2)) + j\,(\psi_r(u_1)\psi_i(u_2) - \psi_i(u_1)\psi_r(u_2)) \tag{2.5.39}$$

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 (2.5.37) 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 (2.5.39) 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.

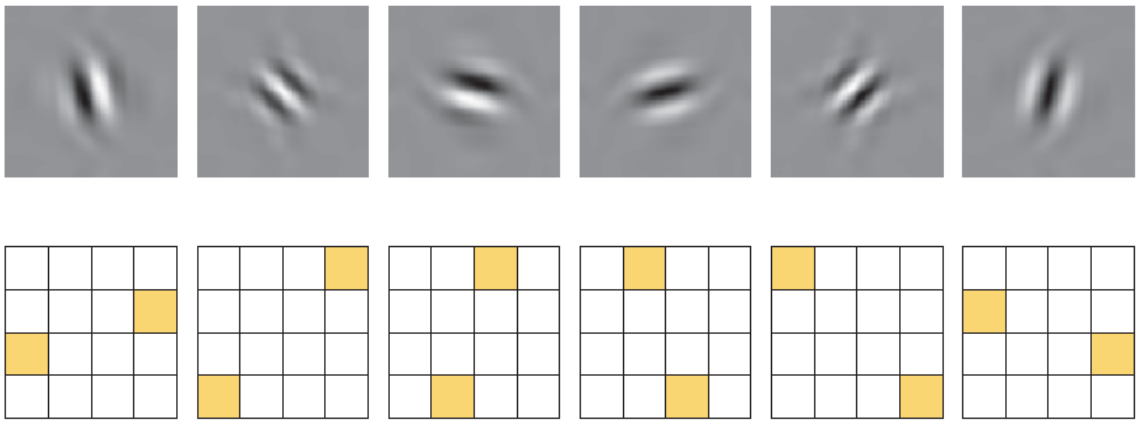


Figure 2.18: **Wavelets from the 2D** DT$\mathbb{C}$WT**. Top:** The six oriented filters in the space domain (only the real wavelets are shown). From left to right these are the $105°, 135°, 165°, 15°, 45°, 75°$ wavelets. **Bottom:** Idealized support of the Fourier spectrum of each wavelet in the 2D frequency plane. Spectra of the the real wavelets are shown — the spectra of the complex wavelets $(\psi_r + j\psi_i)$ only has support in the top half of the plane. Image taken from **selesnick_dual-tree_2005**.

2.5.7.3 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.,

$$\|x\|^2 \approx \|\mathcal{W}x\|^2 \tag{2.5.40}$$

and the implementation is perfectly invertible as $A(\boldsymbol{\omega})$ from (2.5.30) function is unity (or very near unity) $\forall \boldsymbol{\omega} \in [-\pi, \pi] \times [-\pi, pi]$. This is not a surprise, as it is a design constraint in choosing the filters, but nonetheless is important to note.

2.5.8 Summary of Methods

One final comparison to make between the DT$\mathbb{C}$WT and the Morlet wavelets is their frequency coverage. The Morlet wavelets have flexibility at the cost of computational expense and can be made to have tighter angular resolution than the DT$\mathbb{C}$WT. 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.

2.6 ScatterNets

ScatterNets have been a very large influence on our work, as well as being quite distinct from the previous discussions on learned methods. They were first introduced by **bruna__classification__2011** in their work **bruna__classification__2011**, and then were rigorously defined by Mallat in **mallat__group__2012**. Perhaps the clearest explanation of them, and the most relevant to our work is in **bruna__invariant__2013**.

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.

2.6.1 Desirable Properties

2.6.1.1 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:

$$\|\Phi x_c - \Phi x\| \leq C \tag{2.6.1}$$

for some small constant $C > 0$. Note that we may instead want only local translation-invariance and restrict the distance $|\mathbf{c}|$ for which (2.6.1) 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 two images of the same scene taken with small camera disturbances are unlikely to be at integer pixel shifts of each other.

2.6.1.2 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.

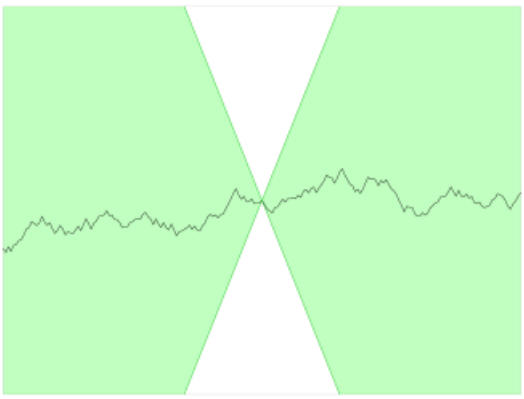


Figure 2.19: **A Lipschitz continuous function.** There is a cone for this function (shown in white) such that the graph always remains entirely outside the cone as it is shifted across. The minimum gradient needed for this to hold is called the 'best Lipschitz constant'.

Alternatively, the modulus squared has derivative everywhere but is not Lipshitz 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.

$$\|\Phi x' - \Phi x\| \leq C\|x' - x\| \tag{2.6.2}$$

2.6.1.3 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.

$$\|\Phi x_\tau - \Phi x\| \leq C_\tau \|x\| \sup_{\mathbf{u}} |\nabla \tau(\mathbf{u})| \tag{2.6.3}$$

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.

2.6.2 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:

$$S[\emptyset]x \triangleq x * \phi_J \left(2^J \mathbf{u}\right) \tag{2.6.4}$$

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 (2.5.26) $\mathcal{W}x = \{x * \phi_J, x * \psi_\lambda\}_\lambda$ 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 location[7]. Unlike the Fourier modulus, the complex wavelet modulus is stable to deformations due to the grouping together of frequencies into dyadic packets **mallat__group__2012**.

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

$$\widetilde{\mathcal{W}}x = \{x * \phi_J,\ |x * \psi_\lambda|\}_\lambda \tag{2.6.5}$$

$$= \{x * \phi_J,\ U[\lambda]x\}_\lambda \tag{2.6.6}$$

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

$$S[\lambda_1]x \triangleq U[\lambda_1]x * \phi_J = |x * \psi_{\lambda_1}| * \phi_J \tag{2.6.7}$$

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.

$$S[\lambda_1, \lambda_2]x \triangleq U[\lambda_2]U[\lambda_1]x \tag{2.6.8}$$

$$= ||x * \psi_{\lambda_1}| * \psi_{\lambda_2}| * \phi_J \tag{2.6.9}$$

[7] Interestingly, the modulus operator can often still be inverted due to the redundancies of the complex wavelet transform **waldspurger__phase__2012**, hence it does not lose any information.

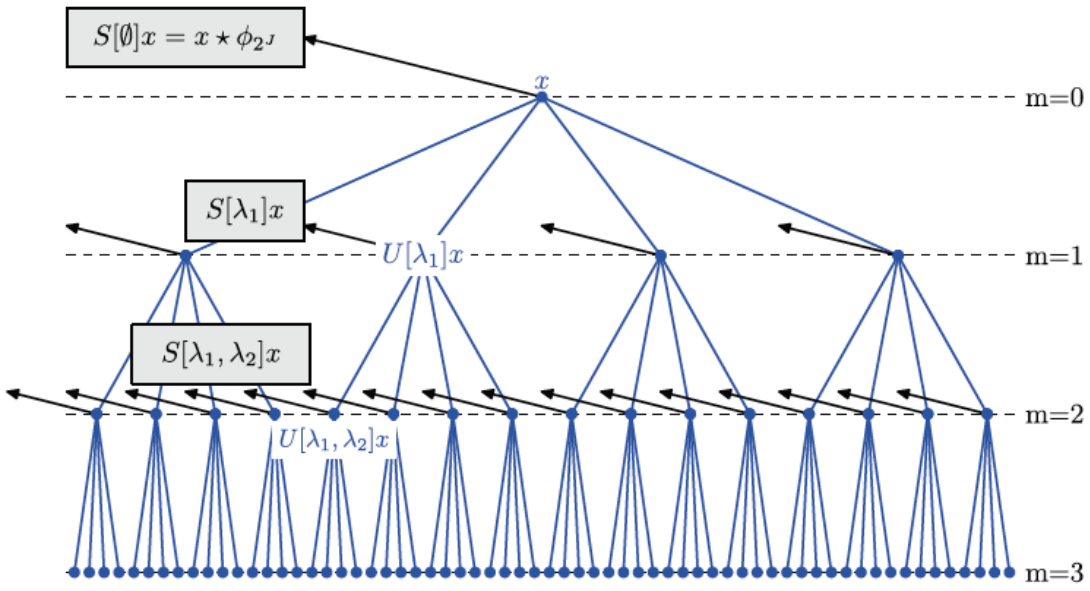


Figure 2.20: **The Scattering Transform.** Scattering outputs are the leftward pointing arrows $S[p]x$, and the intermediate coefficients $U[p]x$ are the centre nodes of the tree. Taken from **bruna_invariant_2013**.

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

$$U[p]x = U[\lambda_m]U[\lambda_{m-1}]\cdots U[\lambda_1]x \tag{2.6.10}$$

$$= ||\cdots|x * \psi_{\lambda_1}|*\psi_{\lambda_2}|\cdots * \psi_{\lambda_m}| \tag{2.6.11}$$

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

$$\widetilde{\mathcal{W}}U[p]x = \{S[p]x,\ U[p+\lambda]x\}_\lambda \tag{2.6.12}$$

which is shown in Figure 2.20

2.6.3 Resulting Properties

For ease, let us define the 'mth 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

$$\|Sx\|^2 = \sum_p \|S[p]x\|^2 \tag{2.6.13}$$

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

$$\|\mathcal{W}x\|^2 = \|x * \phi\|^2 + \sum_{\lambda} \|x * \psi_\lambda\|^2 \tag{2.6.14}$$

then by Plancherel's formula

$$(1-\epsilon)\|x\|^2 \leq \|\mathcal{W}x\|^2 \leq \|x\|^2 \tag{2.6.15}$$

For the Morlet wavelets originally used in **bruna_invariant_2013**, $\epsilon = 0.25$, for the DT$\mathbb{C}$WT $\epsilon \approx 0$ (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).

2.6.3.1 Translation-Invariance

This is proven in section 2.4 of **mallat_group_2012**. 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, $\phi_J = 2^{-J}\phi\left(2^{-J}\mathbf{u}\right)$ it is proven in Appendix B of **mallat_group_2012** that shifting it by $\mathbf{c}$, which we denote as $\mathcal{L}_c$, is Lipschitz continuous:

$$\|\mathcal{L}_c\phi_J - \phi_J\| \leq 2^{-J+2}\|\nabla\phi\|_1|\mathbf{c}| \tag{2.6.16}$$

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

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

$$\begin{aligned}
\|S\mathcal{L}_c x - Sx\| &= \|\mathcal{L}_c A_J U x - A_J U x\| && (2.6.17)\\
&\leq \|\mathcal{L}_c A_J - A_J\|\,\|Ux\| && (2.6.18)\\
&\leq 2^{-J+2}\|\nabla\phi\|_1|\mathbf{c}|\|x\| && (2.6.19)
\end{aligned}$$

2.6.3.2 Stability to Noise

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

$$\left\|\widetilde{\mathcal{W}}x - \widetilde{\mathcal{W}}y\right\| \leq \|x-y\| \tag{2.6.20}$$

We have already shown that S is the repeated application of $\widetilde{\mathcal{W}}$ in (2.6.12), we can then say

$$\|Sx - Sy\| \leq \|x-y\| \tag{2.6.21}$$

making scattering non-expansive and stable to noise. With the DT$\mathbb{C}$WT, both (2.6.20) and (2.6.21) are nearly inequalities as ϵ is close to 0.

2.6.3.3 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 **mallat_group_2012** that:

$$\|S\mathcal{L}_\tau x - Sx\| \leq CP\|x\|(2^{-J}\|\tau\|_\infty + \|\nabla\tau\|_\infty) \tag{2.6.22}$$

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:

$$\|S\mathcal{L}_\tau x - Sx\| \leq CP\|x\|\|\nabla\tau\|_\infty \tag{2.6.23}$$

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

2.6.3.4 Energy Decay

As $m \to \infty$ the invariant coefficients of path length m, U_m, decay towards zero **mallat_group_2012**:

$$\lim_{m\to\infty} U_m = 0 \tag{2.6.24}$$

This is an important property that suggests that we can stop scattering beyond a certain point. Experimental results **bruna_invariant_2013** 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 **bruna_invariant_2013** such as **oyallon_deep_2015**, **oyallon_hybrid_2017**, **oyallon_scaling_2017**, setting $m = 2$ was found to be sufficient.

2.6.3.5 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 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:

$$n(p_0) = 1 \tag{2.6.25}$$

$$n(p_1) = JK \tag{2.6.26}$$

$$n(p_2) = (J-1)K^2 + (J-2)K^2 + \ldots + K^2 \tag{2.6.27}$$

$$= \frac{1}{2}J(J-1)K^2 \tag{2.6.28}$$

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	$N/64$

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 $N2^{-2J}\left(1 + JK + \frac{1}{2}J(J-1)K^2\right)$ 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 wavelet theory, in particular the DT$\mathbb{C}$WT, has any place in deep learning and if it does, quantifying how beneficial it can be. The introduction of more powerful GPUs and fast and popular deep learning frameworks such as PyTorch, Tensorflow and Caffe in the past few years has helped the field of deep learning grow very rapidly. Never before has it been so possible and so accessible to test new designs and ideas for a machine learning algorithm than today. Despite this rapid growth, there has been little interest in building wavelet analysis software in modern frameworks.

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

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

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

This chapter describes how we have built a fast ScatterNet implementation in PyTorch with the DT$\mathbb{C}$WT as its wavelet transform. At the core of the DT$\mathbb{C}$WT is an efficient implementation of the DWT. The result is an open-source library that provides all three (the DWT, DT$\mathbb{C}$WT and DT$\mathbb{C}$WT ScatterNet), available on GitHub as *PyTorch Wavelets* **cotter_pytorch_2018**.

In parallel with our efforts, the original authors of the ScatterNet have improved their implementation, also building it on PyTorch. Our proposed DT$\mathbb{C}$WT ScatterNet is 15 to 35 times faster than their improved implementation (depending on the padding style and wavelet length), while using less GPU memory.

3.1 The Design Constraints

The original authors implemented their ScatterNet in Matlab **oyallon_deep_2015** using a Fourier-domain based Morlet wavelet transform. The standard procedure for using ScatterNets in a deep learning framework up until recently has been to:

1. Pre-scatter a dataset using conventional CPU-based hardware and software and store the features to disk. This can take several hours to several days depending on the size of the dataset and the number of CPU cores available.
2. Build a network in another framework, usually Tensorflow **abadi_tensorflow:_2015** or Pytorch **paszke_automatic_2017**.
3. Load the scattered data from disk and train on it.

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

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

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

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

To achieve all of these goals, we choose to build our software on PyTorch, a popular open source deep learning framework that can do many operations on GPUs with native support for automatic differentiation. PyTorch uses the CUDA and cuDNN libraries for its GPU-accelerated primitives. Its popularity is of key importance, as it means users can build complex networks involving ScatterNets without having to use or learn extra software.

As mentioned earlier, the original authors of the ScatterNet also noticed the shortcomings with their Scattering software, and recently released a new package that can do Scattering in PyTorch called KyMatIO**andreux__kymatio:__2018**, addressing the above design constraints. The key difference between our proposed package and their improved package is the use of the DT$\mathbb{C}$WT as the core rather than Morlet wavelets. While the key focus of this chapter is in detailing how we have built a fast, GPU-ready, deep learning compatible library that can do the DWT, DT$\mathbb{C}$WT, and DT$\mathbb{C}$WT ScatterNet, we also compare the speeds and performance of our package to KyMatIO, as it provides some interesting insights into some of the design choices that can be made with a ScatterNet.

3.2 A Brief Description of Autograd

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

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

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

Note that the pseudo code can be one of three types of functions:

1. Gradient-less code - these are lowlevel functions that can be used for both the forward and backward calculations. E.g. Algorithm 3.1. We name these `NG:<name>`, NG for no gradients.

2. Autograd blocks - the modules as shown in Figure 2.6. These always have a forward and backward pass, and are named `AG:<name>`. E.g. Algorithm 3.2.

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

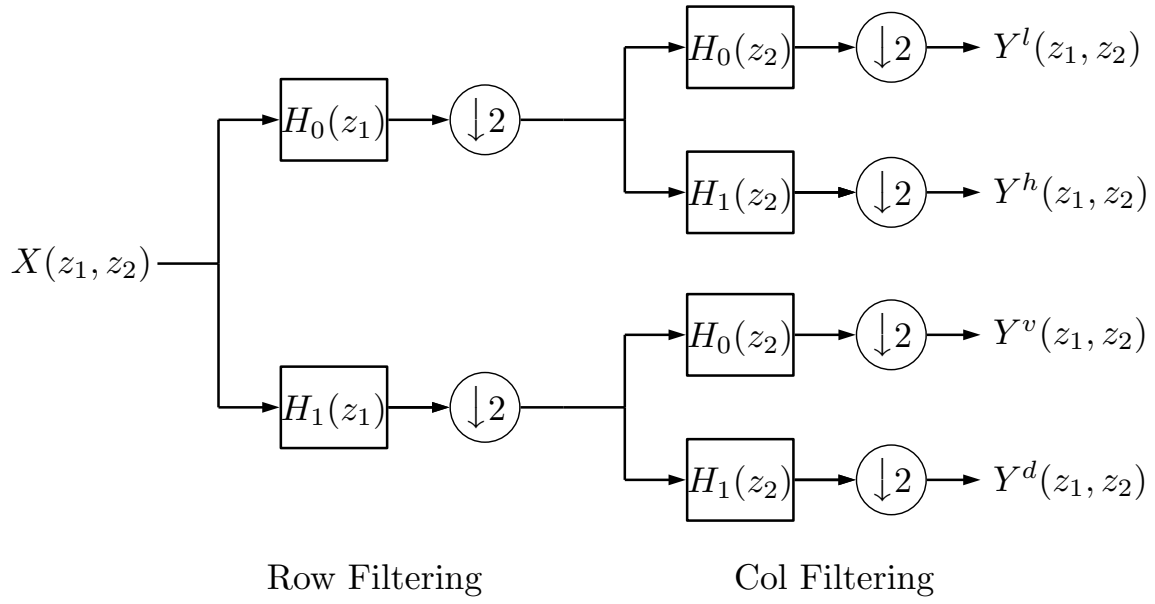


Figure 3.1: **Block Diagram of 2-D DWT.** The components of a filter bank DWT in two dimensions.

3.3 Fast Calculation of the DWT and IDWT

To have a fast implementation of the Scattering Transform, we need a fast implementation of the DT$\mathbb{C}$WT. For a fast implementation of the DT$\mathbb{C}$WT we need a fast implementation of the DWT. Later in our work will we also explore the DWT as a basis for learning, so having an implementation that is fast and can pass gradients through will prove beneficial in and of itself.

There has been much research into the best way to do the DWT on a GPU, in particular comparing the speed of Lifting **sweldens_lifting_1998**, or second generation wavelets, to the direct convolutional methods. **tenllado_parallel_2008**, **galiano_improving_2011** are two notable such publications, both of which find that the convolution based implemetations are better suited for the massively parallel architecture found in modern GPUs. For this reason, we implement a convolutional based DWT.

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

of memory by overwriting the automatic differentiation logic and defining our own backwards pass.

3.3.1 Primitives

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

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

where $H(z^{-1})$ is the Z-transform of the time/space reverse of $H(z)$, $\Delta Y(z) \triangleq \frac{\partial L}{\partial Y}(z)$ is the gradient of the loss with respect to the output, and $\Delta X(z) \triangleq \frac{\partial L}{\partial X}(z)$ is the gradient of the loss with respect to the input. This was proved in subsubsection 2.3.1.2, we have just rewritten it in terms of Z-transforms here.

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

Like Matlab, most deep learning frameworks have an efficient function for doing convolution followed by downsampling. Similarly, there is an efficient function to do upsampling followed by convolution (for the inverse transform). Using the above two properties for the backwards pass of convolution and sample rate changes, we quickly see that the backwards pass of a wavelet transform is simply the inverse wavelet transform with the time reverse of the analysis filters used as the synthesis filters. For orthogonal wavelet transforms, the synthesis filters are the time reverse of the analysis filters so backpropagation can simply be done by calling the inverse wavelet transform on the wavelet coefficient gradients.

3.3.2 The Forward and Backward Algorithms

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

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

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

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

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

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

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

3.3.2.1 The Input

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

3.3.2.2 1-D Filter Banks

Let us assume that the analysis $(h_0,\ h_1)$ and synthesis $(g_0,\ g_1)$ filters are already in the form needed to do column filtering. The necessary steps to do the 1-D analysis and synthesis are described in Algorithm 3.1. We do not need to define backpropagation functions for the `afb1d` and `sfb1d` functions as they are each others backwards step.

3.3.2.3 2-D Transforms and their gradients

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

Algorithm 3.2 2-D DWT and its gradient

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

function AG:DWT:BWD(Δll, Δlh, Δhl, Δhh)
  load h_0^c, h_1^c, h_0^r, h_1^r, $mode$
  h_0^c, $h_1^c \leftarrow$ flip(h_0^c), flip(h_1^c) ▷ flip the filters as in (3.3.1)
  h_0^r, $h_1^r \leftarrow$ flip(h_0^r), flip(h_1^r)
  $\Delta lo \leftarrow$ sfb1d(Δll, Δlh, h_0^c, h_1^c, $mode$, $axis = -2$)
  $\Delta hi \leftarrow$ sfb1d(Δhl, Δhh, h_0^c, h_1^c, $mode$, $axis = -2$)
  $\Delta x \leftarrow$ sfb1d(Δlo, Δhi, h_0^r, h_1^r, $mode$, $axis = -1$)
  return Δx
end function

filter banks. We show how to do do this in Algorithm 3.2. Note that we have allowed for different row and column filters in Algorithm 3.2. Most commonly used wavelets will use the same filter for both directions (e.g. the orthogonal Daubechies family), but later when we use the DT$\mathbb{C}$WT, we will want to have different horizontal and vertical filters.

The inverse transform logic is moved to the appendix Algorithm B.1. An interesting result is the similarity between the two transforms' forward and backward stages. Further, note that the only things that need to be saved are the filters, as seen in Algorithm 3.2.2. These are typically only a few floats, giving us a large saving over relying on autograd.

A multiscale DWT (and IDWT) can easily be made by calling Algorithm 3.2 (Algorithm B.1) multiple times on the lowpass output (reconstructed image). Again, no intermediate activations need be saved, giving this implementation almost no memory overhead.

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

We have built upon previous implementations of the DT$\mathbb{C}$WT, in particular **kingsbury_dtcwt_2003**, **cai_2-d_2011**, **wareham_dtcwt_2014**. The DT$\mathbb{C}$WT gets is name from having two sets of filters, a and b. In two dimensions, we do four multiscale DWTs, called aa, ab, ba and bb, where the pair of letters indicates which set of wavelets is used for the row and column filtering. The twelve bandpass coefficients at each scale are added and subtracted from each other to get the six orientations' real and imaginary components Algorithm B.3. The four lowpass coefficients from each scale can be used for the next scale DWTs. At the final scale, they can be interleaved to get four times the expected decimated lowpass output area.

Algorithm 3.3 2-D DT$\mathbb{C}$WT

function MOD:DTCWT(x, J, $mode$)
  load h_0^a, h_1^a, h_0^b, h_1^b
  ll^{aa}, ll^{ab}, ll^{ba}, $ll^{bb} \leftarrow x$
  for $0 \leq j < J$ **do**
    ll^{aa}, lh^{aa}, hl^{aa}, $hh^{aa} \leftarrow \mathrm{AG:DWT}(ll^{aa}, h_0^a, h_1^a, h_0^a, h_1^a, mode)$
    ll^{ab}, lh^{ab}, hl^{ab}, $hh^{ab} \leftarrow \mathrm{AG:DWT}(ll^{ab}, h_0^a, h_1^a, h_0^b, h_1^b, mode)$
    ll^{ba}, lh^{ba}, hl^{ba}, $hh^{ba} \leftarrow \mathrm{AG:DWT}(ll^{ba}, h_0^b, h_1^b, h_0^a, h_1^a, mode)$
    ll^{bb}, lh^{bb}, hl^{bb}, $hh^{bb} \leftarrow \mathrm{AG:DWT}(ll^{bb}, h_0^b, h_1^b, h_0^b, h_1^b, mode)$
    $yh[j] \leftarrow$ Q2C(
      lh^{aa}, hl^{aa}, hh^{aa},
      lh^{ab}, hl^{ab}, hh^{ab},
      lh^{ba}, hl^{ba}, hh^{ba},
      lh^{bb}, hl^{bb}, hh^{bb})
  end for
  $yl \leftarrow \mathrm{interleave}(ll^{aa}, ll^{ab}, ll^{ba}, ll^{bb})$
  return yl, yh
end function

A requirement of the DT$\mathbb{C}$WT is the need to use different filters for the first scale to all subsequent scales **selesnick_dual-tree_2005**. We have not shown this in Algorithm 3.3 for simplicity, but it would simply mean we would have to handle the $j = 0$ case separately.

3.5 Changing the ScatterNet Core

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

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

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

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

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

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

$$\begin{aligned} \Delta z &= \Delta r \frac{\partial r}{\partial x} + j\Delta r \frac{\partial r}{\partial y} && (3.5.4)\\ &= \Delta r \frac{x}{r} + j\Delta r \frac{y}{r} && (3.5.5)\\ &= \Delta r e^{j\theta} && (3.5.6) \end{aligned}$$

where $\theta = \arctan \frac{y}{x}$. This has a nice interpretation to it as well, as the backwards pass is simply reinserting the discarded phase information. The pseudo-code for this operation is shown in Algorithm 3.4.

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

$$\lim_{x\to 0^-, y\to 0} \frac{\partial r}{\partial x} = -1 \quad (3.5.7)$$

$$\lim_{x\to 0^+, y\to 0} \frac{\partial r}{\partial x} = +1 \quad (3.5.8)$$

$$\lim_{x\to 0, y\to 0^-} \frac{\partial r}{\partial y} = -1 \quad (3.5.9)$$

$$\lim_{x\to 0, y\to 0^+} \frac{\partial r}{\partial y} = +1 \quad (3.5.10)$$

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

$$r_s = \sqrt{x^2 + y^2 + b^2} - b \quad (3.5.11)$$

This keeps the magnitude near zero for small x, y but does slightly shrink larger values. The gain we get however is a new smoother gradient surface. We can then choose the size of b as a hyperparameter in optimization. The partial derivatives now become:

$$\begin{aligned} \frac{\partial r_s}{\partial x} &= \frac{x}{\sqrt{x^2 + y^2 + b^2}} = \frac{x}{r_s} && (3.5.12)\\ \frac{\partial r_s}{\partial y} &= \frac{y}{\sqrt{x^2 + y^2 + b^2}} = \frac{y}{r_s} && (3.5.13) \end{aligned}$$

Algorithm 3.4 Magnitude forward and backward steps

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

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

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

function MOD:DTCWT_SCAT($x,\ J = 2,\ M = 2$)
  $Z \leftarrow x$
  for $0 \le m < M$ **do**
    $yl,\ yh \leftarrow$ DT$\mathbb{C}$WT($Z,\ J = 1,\ mode =$ 'symmetric')
    $S \leftarrow$ avg_pool($yl,\ 2$)
    $U \leftarrow \mathrm{mag}(\mathrm{Re}(yh),\ \mathrm{Im}(yh))$
    $Z \leftarrow$ concatenate($S,\ U,\ axis = 1$) ▷ stack 1 lowpass with 6 magnitudes
  end for
  if $J > M$ **then**
    $Z \leftarrow$ avg_pool($Z,\ 2^{J-M}$)
  end if
  return Z
end function

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

Now that we have the DT$\mathbb{C}$WT and the magnitude operation, it is straightforward to get a DT$\mathbb{C}$WT scattering layer, shown in Algorithm 3.5. To get a multilayer scatternet, we can call the same function again on Z, which would give $S_0,\ S_1$ and U_2 and so on for higher orders.

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

Table 3.1: **Comparison of properties of different ScatterNet packages.** In particular the wavelet backend used, the number of orientations available, the available boundary extension methods, whether it has GPU support and whether it supports backpropagation.

Package	Backend	Orientations	Boundary Ext.	GPU	B
ScatNetLight**oyallon_deep_2015**	FFT-based	Flexible	Periodic	No	N
KyMatIO**andreux_kymatio:_2018**	FFT-based	Flexible	Periodic	Yes	Y
DTℂWT Scat	Separable filter banks	6	Flexible	Yes	Y

3.6 Comparisons

Now that we have the ability to do a DTℂWT based scatternet, how does this compare with the original matlab implementation **oyallon_deep_2015** and the newly developed KyMatIO **andreux_kymatio:_2018**? Table 3.1 lists the different properties and options of the competing packages.

3.6.1 Speed and Memory Use

Table 3.2 lists the speed of the various transforms as tested on our reference architecture appendix A. The CPU experiments used all cores available, whereas the GPU experiments ran on a single GPU. We include two permutations of our proposed scatternet, with different length filters and different padding schemes. Type A uses long filters and uses symmetric padding, and is 15× faster than the Fourier-based KyMatIO. Type B uses shorter filters and the cheaper zero padding scheme, and achieves a 35× speedup over the Morlet backend. Additionally, when compared with version 0.2 of KyMatIO, the DTℂWT based implementation uses 2% of the memory for saving activations for the backwards pass, highlighting the importance of defining the custom backpropagation steps from section 3.3–section 3.5.

3.6.2 Performance

To confirm that changing the ScatterNet core has not impeded the performance of the ScatterNet as a feature extractor, we build a simple Hybrid ScatterNet, similar to **oyallon_hybrid_2017**, **oyallon_scaling_2017**. This puts two layers of a scattering transform at the front end of a deep learning network. In addition to comparing our DTℂWT based scatternet to the Morlet based one, we also test using different wavelets, padding schemes and biases for the magnitude operation. We run tests on

- CIFAR-10: 10 classes, 5000 images per class, 32×32 pixels per image.
- CIFAR-100: 100 classes, 500 images per class, 32×32 pixels per image.

Table 3.2: **Comparison of execution time for the forward and backward passes of the competing ScatterNet Implementations.** Tests were run on the reference architecture described in appendix A. The input for these experiments is a batch of images of size $128 \times 3 \times 256 \times 256$ in 4 byte floating precision. We list two different types of options for our scatternet. Type A uses 16 tap filters and has symmetric padding, whereas type B uses 6 tap filters and uses zero padding at the image boundaries. Values are given to 2 significant figures, averaged over 5 runs.

Package	CPU		GPU	
	Fwd (s)	Bwd (s)	Fwd (s)	Bwd (s)
ScatNetLight**oyallon_deep_2015**	> 200.00	n/a	n/a	n/a
KyMatIO**andreux_kymatio:_2018**	95.00	130.00	3.50	4.50
DT$\mathbb{C}$WT Scat Type A	8.00	9.30	0.23	0.29
DT$\mathbb{C}$WT Scat Type B	3.20	4.80	0.11	0.06

- Tiny ImageNet**li_tiny_2017**: 200 classes, 500 images per class, 64×64 pixels per image.

Table 3.3 details the network layout for CIFAR.

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

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

Our experiment code is available at **cotter_learnable_2019-1**.

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

Before comparing to the Morlet based ScatterNet, we can test test different padding schemes, wavelet lengths and magnitude smoothing parameters (see (3.5.11)) for the DT$\mathbb{C}$WT ScatterNet. We test these over a grid of values described in Table 3.4. The different wavelets have different lengths and hence different frequency responses. Additionally, the 'near_sym_b_bp' wavelet is a rotationally symmetric wavelet with diagonal passband brought in by a factor of $\sqrt{2}$, making it have similar $|\boldsymbol{\omega}|$ to the horizontal and vertical bandpasses.

The results of these experiments are shown in Figure 3.2. Interestingly, for all three datasets the shorter wavelet outperformed the longer wavelets.

Table 3.3: **Hybrid architectures for performance comparison.** Comparison of Morlet based scatternets (Morlet6 and Morlet8) to the DT$\mathbb{C}$WT based scatternet on CIFAR. The output after scattering has $3(K+1)^2$ channels (243 for 8 orientations or 147 for 6 orientations) of spatial size 8×8. This is passed to 4 convolutional layers of width $C = 192$ before being average pooled and fed to a single fully connected classifier. $N_c = 10$ for CIFAR-10 and 100 for CIFAR-100. In the DT$\mathbb{C}$WT architecture, we test different padding schemes and wavelet lengths.

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

3.6.2.2 Results

We use the optimal hyperparameter choices from the previous section, and compare these to morlet based ScatterNet with 6 and 8 orientations. The results of this experiment are shown in Table 3.5. It is promising to see that the DT$\mathbb{C}$WT based scatternet has not only not sped up, but slightly improved upon the Morlet based ScatterNet as a frontend. Interestingly, both with Morlet and DT$\mathbb{C}$WT wavelets, 6 orientations performed better than 8, despite having fewer parameters in conv1.

3.7 Conclusion

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

We have derived the forward and backpropagation functions necessary to do fast and memory efficient DWTs, DT$\mathbb{C}$WTs, and Scattering based on the DT$\mathbb{C}$WT, and have made this code publically available at **cotter_pytorch_2018**. We hope that this will reduce some of the barriers we initially faced in using wavelets and Scattering in deep learning.

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

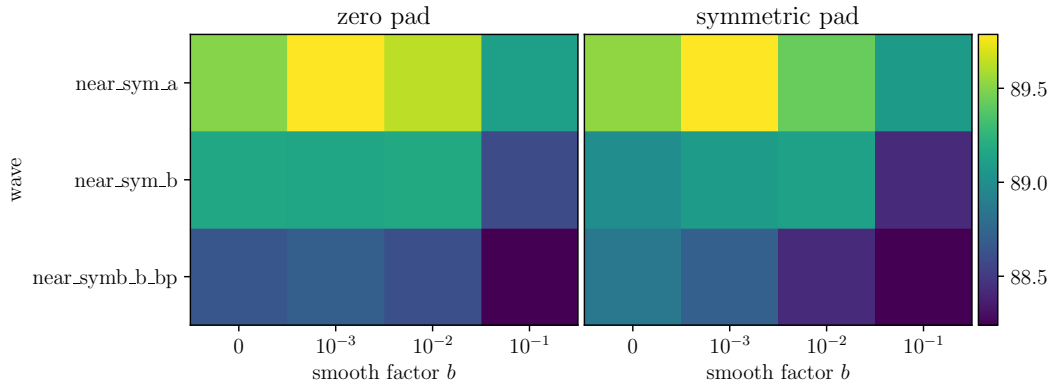
Hyperparameter	Values
Wavelet	near_sym_a 5,7 tap filters, near_sym_b 13,19 tap filters, near_sym_b_bp 13,19 tap filters
Padding Scheme	symmetric zero
Magnitude Smoothing b	0 1e-3 1e-2 1e-1

Table 3.5: **Performance comparison for a** DT$\mathbb{C}$WT **based ScatterNet vs Morlet based ScatterNet.** We report top-1 classification accuracy for the 3 listed datasets as well as training time for each model in hours.

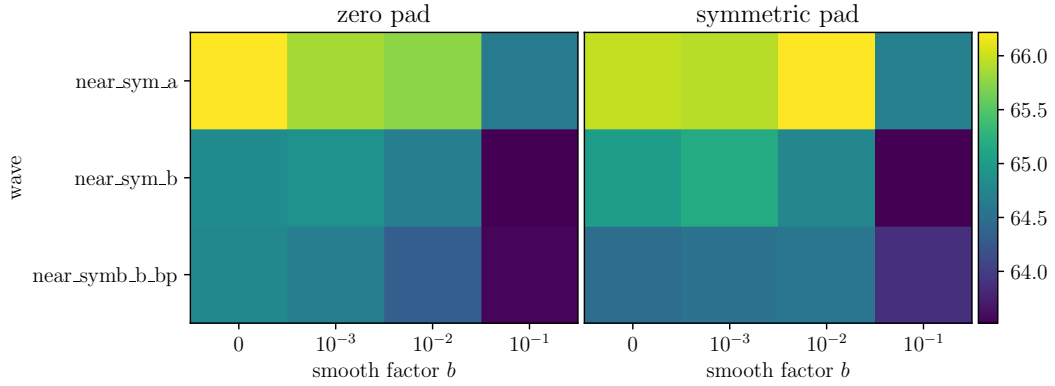
Type	CIFAR-10		CIFAR-100		Tiny ImgNet	
	Acc. (%)	Time (h)	Acc. (%)	Time (h)	Acc. (%)	Time (h)
Morlet8	88.6	3.4	65.3	3.4	57.6	5.6
Morlet6	89.1	2.4	65.7	2.4	57.5	4.4
DT$\mathbb{C}$WT	89.8	1.1	66.2	1.1	57.3	2.7

In parallel with our efforts, the original ScatterNet authors rewrote their package to speed up Scattering. In theory, a spatially separable wavelet transform acting on N pixels has order $\mathcal{O}(N)$ whereas an FFT based implementation has order $\mathcal{O}(N \log N)$. We have shown experimentally that on modern GPUs, the difference is far larger than this, with the DT$\mathbb{C}$WT backend an order of magnitude faster than Fourier-based Morlet implementation **andreux_kymatio:_2018**.

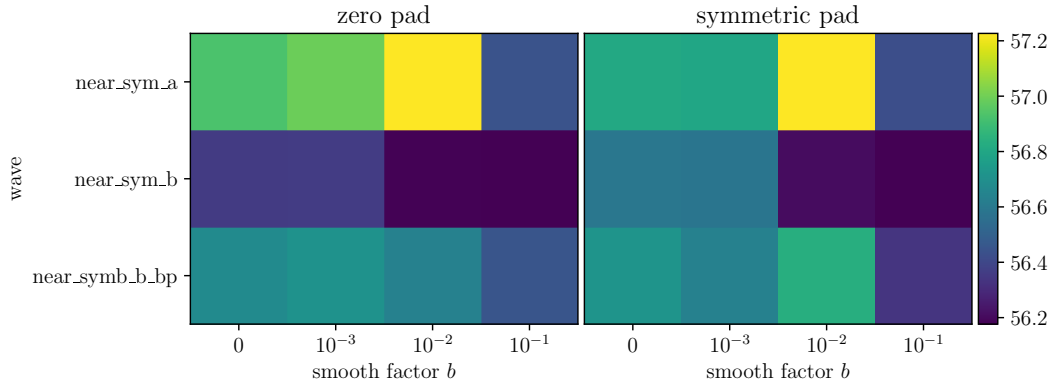
Additionally, we have experimentally verified that using a different complex wavelet core does not have a negative impact on the performance of the ScatterNet as a frontend to Hybrid networks.



(a) CIFAR-10



(b) CIFAR-100



(c) Tiny ImageNet

Figure 3.2: **Hyperparameter results for the** DT$\mathbb{C}$WT **scatternet on various datasets.** Image showing relative top-1 accuracies (in %) on the given datasets using architecture described in Table 3.4. Each subfigure is a new dataset and therefore has a new colour range (shown on the right). Results are averaged over 3 runs from different starting positions. Surprisingly, the choice of options can have a very large impact on the classification accuracy. Symmetric padding is marginally better than zero padding. Surprisingly, the shorter filter (near_sym_a) fares better than its longer counterparts, and bringing in the diagonal subbands (near_sym_b_bp) does not help. Additionally, the smoothing bias may indeed be a good idea for the forward pass as well as aiding the backwards pass, so long as it is less than 0.1.

Chapter 4

Visualizing and Improving Scattering Networks

Deep Convolutional Neural Networks have become the *de facto* solution for image understanding tasks since the rapid development in their growth in recent years. Since AlexNet**krizhevsky__imagenet__2012** in 2012, there have been many new improvements in their design, taking them 'deeper', such as the VGG network in 2014 **simonyan__very__2014**, the Inception Network **szegedy__going__2015**, Residual Networks in 2016 **he__deep__2016** and DenseNets in **huang__densely__2017**. Each iteration has made the inner workings of the model more and more abstract.

Despite their success, Deep Nets are often criticized for being 'black box' methods. Indeed, once you train a CNN, you can view the first layer of filters quite easily (see Figure 1.2a) as they exist in RGB space. Beyond that things get trickier, as the filters have a third, 'depth' dimension typically much larger than its two spatial dimensions, and representing these dimensions with different colours becomes tricky and uninformative.

This has started to become a problem, and while we are happy to trust modern CNNs for isolated tasks, we are less likely to be comfortable with them driving cars through crowded cities, or making executive decisions that affect people directly. 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, has the right to know why **goodman__european__2016**, but also from the bank's — before lending large sums of money, most banks would like to know why the network has given the all clear. 'It has worked well before' is a poor rule to live by.

A recent paper titled 'Why Should I Trust You?' **ribeiro__why__2016** explored this concept in depth. They rated how highly humans trusted machine learning models. Unsurprisingly, a model with an interpretable methodology was trusted more than those which did not have one, even if it had a lower prediction accuracy on the test set. To build trust and to

aid training, we need to probe these networks and visualize how and why they are making decisions.

Some good work has been done in this area. In particular, Zeiler and Fergus **zeiler__visualizing__2014** design a DeConvNet to visualize what input patterns a filter in a given layer is mostly highly activated by. In **mahendran__understanding__2015**, **mahendran__understanding__2015** learn to invert representations by updating a noisy input until its latent feature vector matches a desired target. **simonyan__deep__2014** develop saliency maps by projecting gradients back to the input space and measuring where they have largest magnitude.

In recent years, ScatterNets have been shown to perform well as image classifiers and have received a fair share of attention themselves. 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 **bruna__invariant__2013**, **oyallon__deep__2015**, **singh__dual-tree__2017**, **singh__multi-resolution__201** and can outperform CNNs for classification tasks with reduced training set sizes, e.g. in CIFAR-10 and CIFAR-100 (Table 6 from **oyallon__scaling__2017** and Table 4 from **singh__dual-tree__2017**). They are also near state-of-the-art for Texture Discrimination tasks (Tables 1–3 from **sifre__rotation__2013**). Despite this, there still exists a considerable gap between Scatternets and CNNs on challenges like CIFAR-10 with the full training set (83% vs. 93%). Even considering the benefits of ScatterNets, this gap must be addressed.

While ScatterNets have good theoretical foundation and properties **mallat__group__2012**, it is difficult to understand the second order scattering. In particular, how useful are the second order coefficients for training? How similar are the scattered features to a modern state of the art convolutional network? To answer these questions, this chapter interrogates ScatterNet frontends. Taking inspiration from the interrogative work applied to CNNs, we build a DeScatterNet to visualize what the second order features are. We also heuristically probe a trained Hybrid Network and quantify the importance of the individual features.

We first define the operations that form a ScatterNet in section 4.2. We then introduce our DeScatterNet (section 4.3), and show how we can use it to examine the layers of ScatterNets (using a similar technique to the CNN visualization in **zeiler__visualizing__2014**). We use this analysis tool to highlight what patterns a ScatterNet is sensitive to (section 4.4), 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 peforming an occlusion test 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 4.5. 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 **mallat__group__2012**, 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 4.6.

4.1 Related Work

zeiler_adaptive_2011 first attempted to use 'deconvolution' to improve their learning **zeiler_adaptive_2011**, then later for purely visualization purposes **zeiler_visualizing_2014**. Their method involves monitoring network nodes, seeing what input image causes the largest activity and mapping activations at different layers of the network back to the pixel space using meta-information from these images.

Figure 4.1 shows the block diagram for how deconvolution is done. Inverting a convolutional layer is done by taking the 2D transpose of each slice of the filter. Inverting a ReLU is done by simply applying a ReLU again (ensuring only positive values can flow back through the network). Inverting a max pooling step is a little trickier, as max pooling is quite a lossy operation. **zeiler_adaptive_2011** get around this by saving extra information on the forward pass of the model — switches that store the location of the input that caused the maximum value. This way, on the backwards pass, it is trivial to store activations to the right position in the larger feature map. Note that the positions that did not contribute to the max pooling operation remain as zero on the backwards pass. This is shown in the bottom half of Figure 4.1.

mahendran_understanding_2015 take a slightly different route on deconvolution networks **mahendran_understanding_2015**. 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 **springenberg_striving_2014-3**, the authors design an *all convolutional* model, where they replace the max pooling layers commonly seen in CNNs with a convolutional block with sride 2, i.e. a convolution followed by decimation. They did this by first adding an extra convolutional layer before the max pooling layers, then taking away the max pooling and adding decimation after convolution, noting that removing the max pooling had little effect.

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

Another interrogation tool commonly used is occlusion or perturbation. In **zeiler_visualizing_2014**, **zeiler_visualizing_2014** occlude regions in the input image and measure the impact this has on classification score. In **fong_interpretable_2017**, **fong_interpretable_2017** use gradients to find the minimal mask to apply to the input image that causes misclassification.

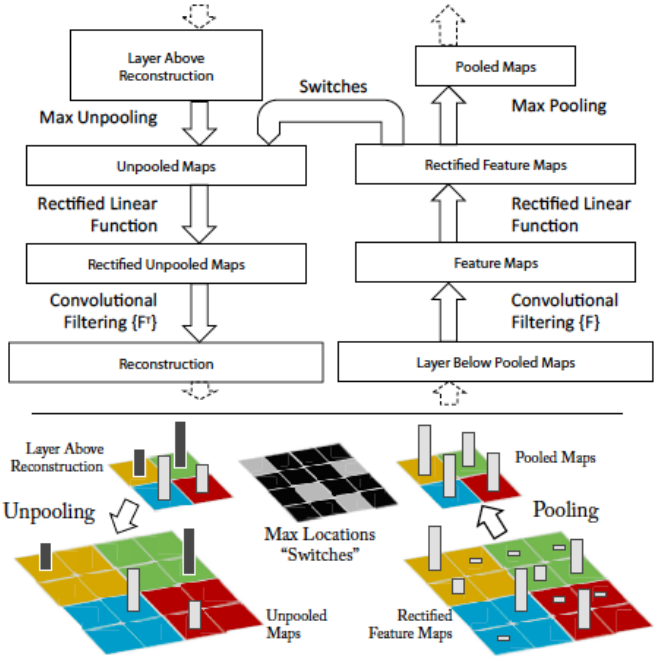


Figure 4.1: **Deconvolution Network Block Diagram.** Note the switches that are saved before the pooled features, and the filters used for deconvolution are the transpose of the filters used for a forward pass. Taken from **zeiler__visualizing__2014**.

4.2 The Scattering Transform

While we have introduced the scattering trasnform 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 FB:

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

This is approximately invariant to translations of up to 2^J pixels[1]. 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:

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

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

for $j_1 \in \{1,2\}, \theta_1 \in \{15°, 45° \ldots, 165°\}$ (we may sometimes index θ with $1 \leq k \leq K$). We will want to retain this information in these coefficients to build a useful classifier.

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

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

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

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

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

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.

Higher order scattering coefficients recover the information lost by averaging U_1, and are defined as:

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

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

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

Previous work shows that for natural images we get diminishing returns after $m = 2$. The second order scattering coefficients are defined only on paths of decreasing frequency **bruna_invariant_2013** as:

$$S_2 x(\lambda_1, \lambda_2, \mathbf{u}) \triangleq ||x * \psi_{\lambda_1}| * \psi_{\lambda_2}| * \phi_J \tag{4.2.8}$$

As we only go on the paths of decreasing frequency $j_1 = 1$, $j_2 = 2$. This then has $6 \times 6 = 36$ output channels per input channel.

Our output is then a stack of these 3 outputs:

$$Sx = \{S_0 x, S_1 x, S_2 x\} \tag{4.2.9}$$

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

[2] In comparison to a system with purely real filters such as a CNN, which would have rapidly varying coefficients for small input shifts **kingsbury_complex_2001**.

4.2.1 Scattering Colour Images

A wavelet transform like the DTℂWT accepts single channel input, while we often work on RGB images. This leaves us with a choice. We can either:

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

2. Define a frequency threshold below which we keep 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 (3.5.11) to now be:

$$r_s = \sqrt{x_r^2 + y_r^2 + x_g^2 + y_g^2 + x_b^2 + y_b^2 + b^2} - b \tag{4.2.10}$$

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 normal.

An alternative to (4.2.10) is to simply combine the colours before scattering into a luminance channel. However we choose to use (4.2.10) 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.

4.3 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 **zeiler_visualizing_2014** 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 usual gradient terms).

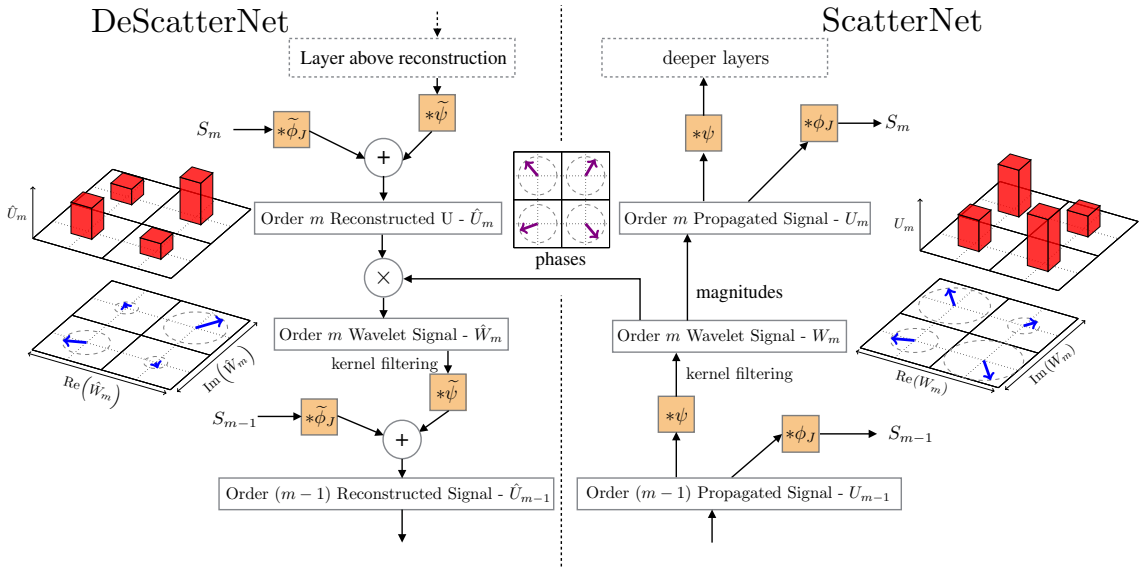


Figure 4.2: **The Descattering Network.** Comprised of a DeScattering layer (left) attached to a Scattering layer (right). We are using the same convention as **zeiler__visualizing__2014** Figure 1 - i.e. the input signal starts in the bottom right hand corner, passes forwards through the ScatterNet (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 non-linearity). For reconstruction, we need to save the discarded phase information and reintroduce it by multiplying it with the reconstructed magnitudes.

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 gives us a good idea of what each output channel is sensitive to, or what it extracts from the input. Note that we do not use any of the log normalization layers described in **oyallon__deep__2015**, **singh__dual-tree__2017**.

4.3.1 Inverting the Low-Pass Filtering

Going from the U coefficients to the S coefficients in the forward pass involved 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, $\widetilde{\phi}_J$ (this is equivalent to the transpose convolution described in **zeiler__visualizing__2014**).

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

This will not recover U as it was on the forward pass, but will recover all the information in U that caused a strong response in S. We note that interpolation usually involves lowpass smoothing of the signal, so this can all be one operation.

4.3.2 Inverting the Magnitude Operation

In the same vein as **zeiler_visualizing_2014**, we face a difficult task in inverting the non-linearity in our system. We lend inspiration from the switches introduced in the DeconvNet; the switches in a DeconvNet save the location of maximal activations so that on the backwards pass activation layers could be unpooled trivially. We do an equivalent operation by saving the phase of the complex activations. On the backwards pass we reinsert the phase to give our recovered W.

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

4.3.3 Inverting the Wavelet Decomposition

Using the 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 $\widetilde{\psi}$:

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

4.4 Visualization with Inverse Scattering

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

Then, in the *reconstruction* phase, we load in the $9 \times C$ images, and scatter them one by one. We take the resulting 52 channel output vector and mask all but a single value (usually the largest) in the channel we are currently examining and 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. This figure shows reconstructed features from the layers of a ScatterNet. For a given output channel, we show the top 9 activations projected independently to pixel space. For the first and second order coefficients, we also show the patch of pixels in the input image which cause this large output. We display activations from various scales (increasing from first row to last row), and random orientations in these scales.

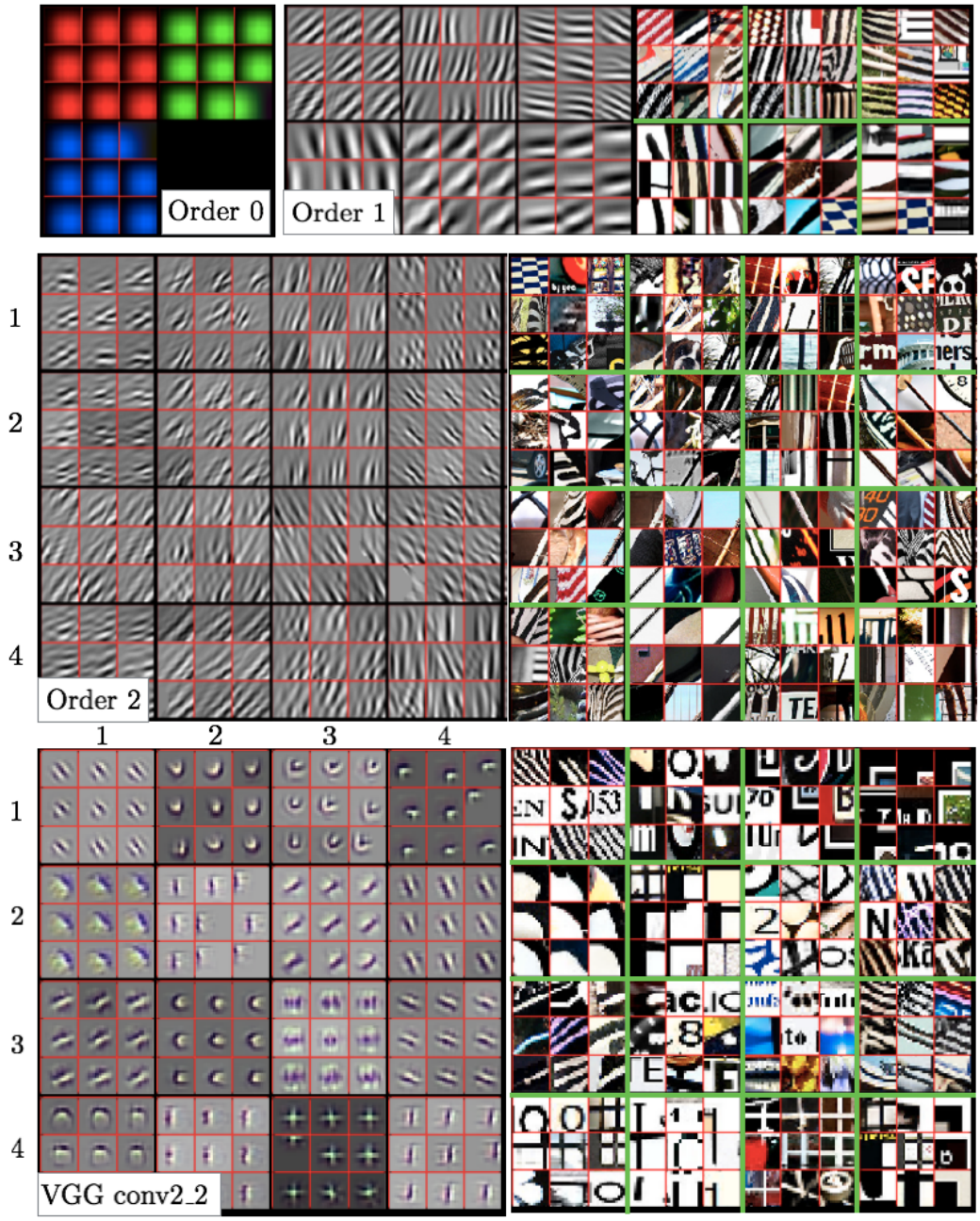


Figure 4.3: **Comparison of Scattering to Convolutional features.** Visualization of a random subset of features from S_0 (all 3), S_1 (6 from the 12) and S_2 (16 from the 36) scattering outputs. We record the top 9 activations for the chosen features and project them back to the pixel space. We show them alongside the input image patches which caused the large activations. We also include reconstructions from layer conv2_2 of VGG Net **simonyan_very_2014**(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 **zeiler_visualizing_2014**. Image best viewed digitally.

The order 1 scattering (labelled with 'Order 1' in Figure 4.3) coefficients look quite similar to the first layer filters from the well known AlexNet CNN **krizhevsky__imagenet__2012**. 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. 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. Indeed, these are activations where the orientation of the wavelet for the first and second order scattering were far from each other (15° and 105° for the first row, first column and 105° and 45° for the first row, fourth column).

For comparison, we include reconstructions from the second layer of the well-known VGG CNN (labelled with 'VGG conv2_2', in Figure 4.3). These were made with a DeconvNet, following the same method as **zeiler__visualizing__2014**. 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 the features extracted from ScatterNets vary significantly from those learned in CNNs after the first order. In many respects, the features extracted from a CNN like VGGNet look preferable for use as part of a classification system.

4.5 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__visualizing__2014**, **zhou__object__2014** have done similar studies but over patches of the input image, with the former using a patch of grey values as the occlusion mask, and the latter using a set of random pixels.

We must be careful to occlude with a sensible mask, the S_0x, S_1x and S_2x all have very different probability densities. 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 show 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.

4.5.1 Experiment Setup

We take a network similar to the one from Table 3.3 but using the colour operation described in subsection 4.2.1 so the scattering output has 51 output channels. We further choose to set $C = 50$ so the conv1 layer has 100 channels, a nice width to display. 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. This gives us a second perspective on the channel importance by looking at the relative weights of the ScatterNet channels passed on to the successive 100 channels. The weight matrix is convolutional filter of size $w \in \mathbb{R}^{100\times 51\times 3\times 3}$. We define:

$$A^{rms}_{c,f} = \sqrt{\frac{\sum_{i,j} w[f,c,i,j]^2}{\sum_f \sum_{i,j} w[f,c,i,j]^2}} \tag{4.5.1}$$

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 4.5.

4.5.2 Discussion

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 datasets. 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 diagonal edges in it as horizontal and vertical edges.

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 happen when $\theta_1 = \theta_2 \in$

$\{15°, 75°, 105°, 165°\}$. When $\theta_1 \neq \theta_2$ we saw the ripple like patterns in Figure 4.3, and we see here that the network has mostly learned to not depend on them.

Figure 4.5 tells a similar tale for the Scattering channels. Here we see directly how much and how little each of the channels is used by the first layer of the network, with the low intensity values in S_2 indicating that the next layer's outputs are less dependent on these coefficients.

We include the same analysis for the two CIFAR datasets in Figure 4.6 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, but with the smaller input spatial size, this comes as no surprise.

4.6 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_rotation_2013 and **oyallon_deep_2015** introduced the idea of a 'Roto-Translation' ScatterNet. 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. Momentarily ignoring the form of the filters they apply, referring to them as $h \in \mathbb{C}^K$, we can think of this stage as stacking the K outputs of a complex wavelet transform on top of each other, and convolving these filters h over all spatial locations of the wavelet coefficients $W_m x$ (this is equivalent to how filters in a CNN are fully connected in depth):

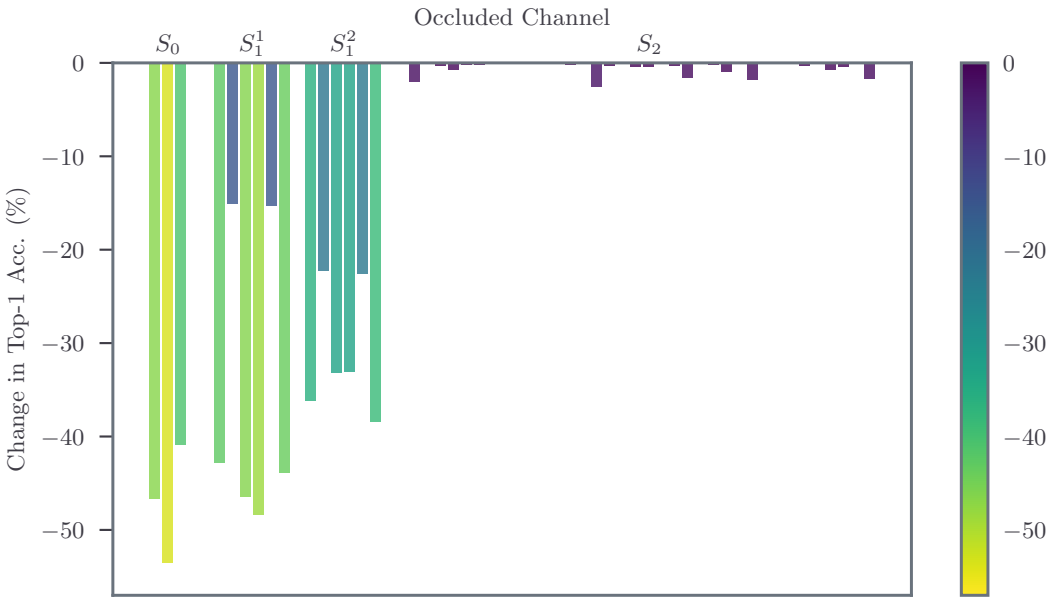
$$V_m x(\lambda, \boldsymbol{u}) = W_m x * h = \sum_{\theta} W_m x(\lambda, \boldsymbol{u}) h(\theta) \tag{4.6.1}$$

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

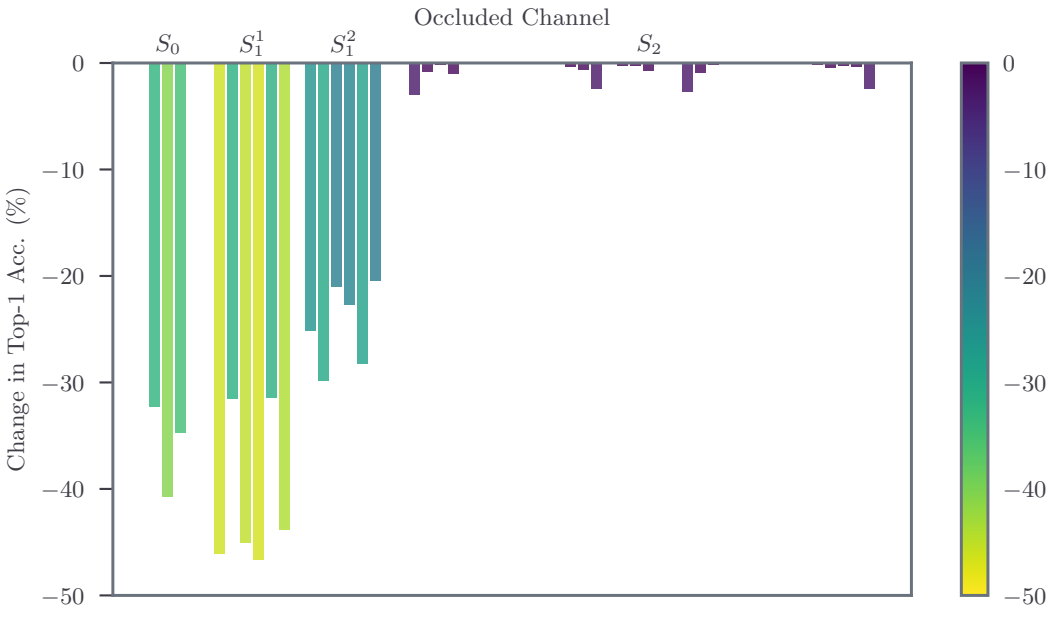
$$U'_m x \triangleq |V_m x| = |W_m x * h| = |U_{m-1} x * \psi_{\lambda_m} * h| \tag{4.6.2}$$

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. Each of the four quadrants show reconstructions from a different class of ScatterNet layer. All shapes are shown in real and imaginary Hilbert-like pairs; the top row of images in each quadrant are reconstructed from a purely real V, while the bottom row are reconstructed from a purely



(a) Tiny ImageNet



(b) Rotated Tiny ImageNet

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 unsurprisingly 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 very little individual effect on the validation score when removed. (b) The same network trained with input samples rotated by $\pm 30°$. 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.

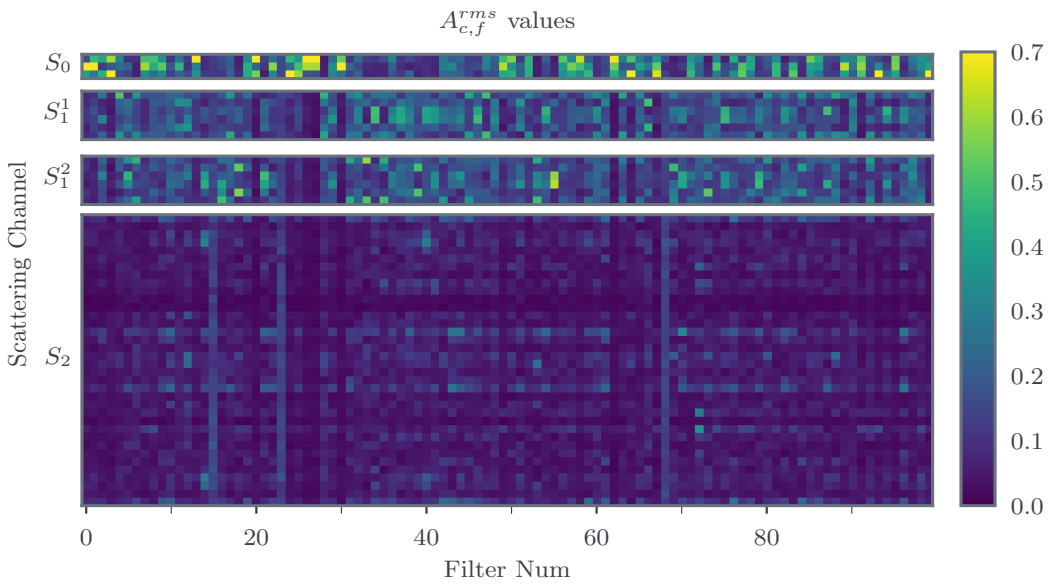


Figure 4.5: **Channel weights for first learned layer.** A visualiztion of the matrix A^{rms} from (4.5.1) for a network trained on Tiny ImageNet. The columns of the matrix all have unit norm and represent how much relative energy comes from each scattering output channel. Most of the filters are heavily dependent on S_0, many are dependent on S_1 and only a few take information from S_2.

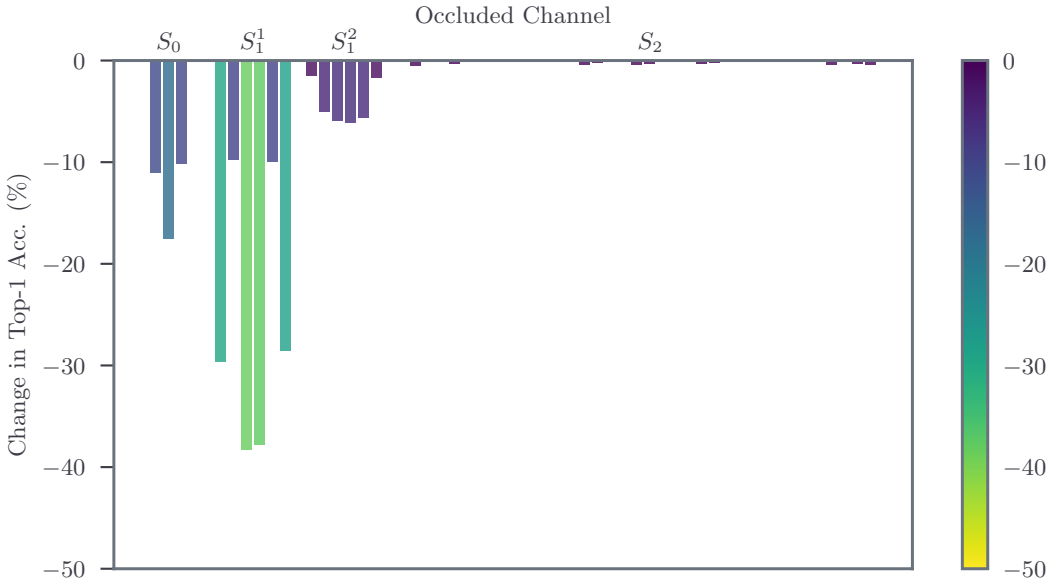
imaginary V. This shows one level of invariance of these filters, as after taking the complex magnitude, both the top and the bottom shape will activate the filter with the same strength. In comparison, for the purely real filters of a CNN, the top shape would cause a large output, and the bottom shape would cause near 0 activity (they are nearly orthogonal to each other).

In the top left, we display the 6 wavelet filters for reference (these were reconstructed from U_1, not V_1). In the top right of the figure we see some of the shapes made by using the h's from the Roto-Translation ScatterNet **sifre_rotation_2013**, **oyallon_deep_2015**. The bottom left is where we present some of our novel kernels. These are simple corner-like shapes made by filtering with $h \in \mathbb{C}^{12\times1\times1}$ where h is set to

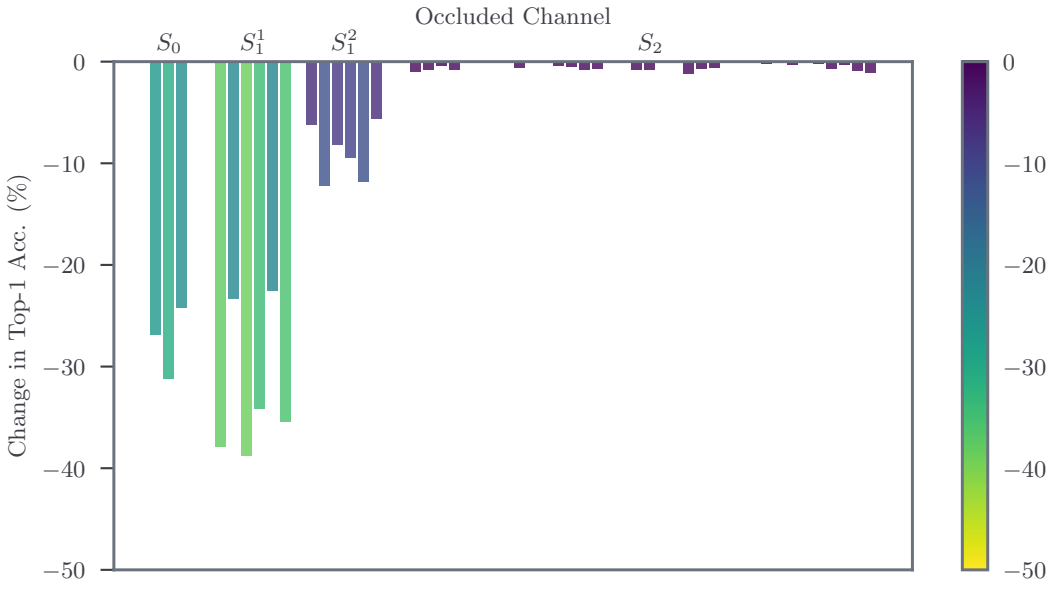
$$h = [1, j, j, 1, 0, 0, 0, 0, 0, 0, 0, 0] \tag{4.6.3}$$

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

Finally, in the bottom right we see shapes made by $h \in \mathbb{C}^{12\times3\times3}$. Note that with the exception of the ring-like shape which has 12 non-zero coefficients, all of these shapes were



(a) CIFAR-10



(b) CIFAR-100

Figure 4.6: **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.

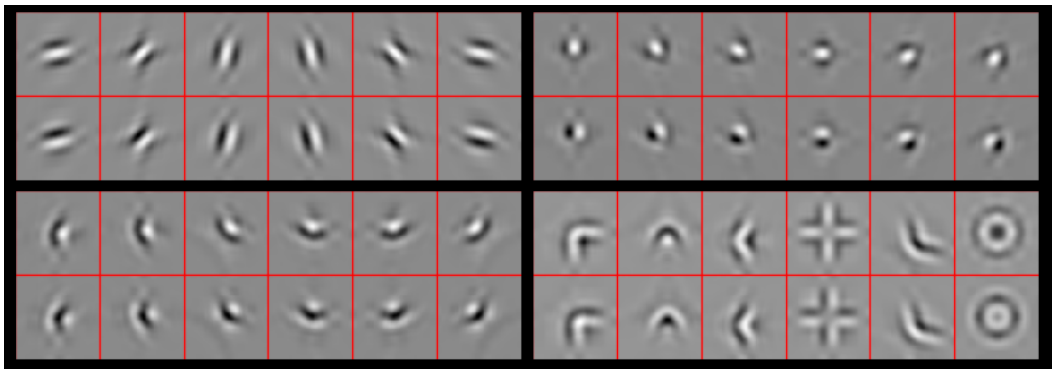


Figure 4.7: **Shapes possible by filtering across the wavelet orientations with complex coefficients.** All shapes are shown in pairs: the top image is reconstructed from a purely real output, and the bottom image from a purely imaginary output. These 'real' and 'imaginary' shapes are nearly orthogonal 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 **sifre_rotation_2013**, **oyallon_deep_2015**. Bottom left - reconstructions from U_1' made with a more general $12 \times 1 \times 1$ filter, described in Equation 4.6.3. Bottom right - some reconstructions possible by filtering a general $12 \times 3 \times 3$ filter.

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.

4.7 Discussion

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

Additionally, we performed occlusion tests to heuristically measure the importance of the individual scattering channels. 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.

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 tying ScatterNets together with CNNs in hybrid networks **oyallon_scaling_2017**, **oyallon_hybrid_2017**, **singh_scatternet_2018** have tended to keep the two parts separate, with the ScatterNet forming a fixed front end and the CNN forming a learned backend. We instead look at adding learning between scattering orders, as well as adding learned layers before the ScatterNet.

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

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

In section 5.2 we briefly review convolutional layers and scattering layers before introducing our learnable scattering layers in section 5.3. In section 5.4 we describe how we implement our proposed layer, and present some experiments we have run in section 5.5 and then draw conclusions about how these new ideas might improve neural networks in the future.

5.1 Related Work

There have been several similar works that look into designing new convolutional layers by separating them into two stages — a first stage that performs a non-standard filtering process, and a second stage that combines the first stage into single activations. The inception layer **szegedy__rethinking__2015** by **szegedy__rethinking__2015** 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__training__2015** 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**ioannou__training__2015**. 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:

- **juefei-xu__local__2016 juefei-xu__local__2016**. This paper builds a first stage with a small 3×3 kernel filled with zeros, and randomly insert plus and minus ones into it keeping a set sparsity level. This builds a very crude spatial differentiator in random directions. The output of the first stage is then passed through a sigmoid nonlinearity before being mixed with a 1×1 convolution. The imposed structure on the first stage was found to be a good regularizer and prevented overfitting, and the combination of the mixing in the second layer allowed for a powerful and expressive layer, with performance near that of a regular CNN layer.

- "DCFNet: Deep Neural Network with Decomposed Convolutional Filters" **qiu__dcfnet:__2018**. 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 **juefei-xu__local__2016** this layer is purely linear. The supposed advantage being that the basis can be truncated to save parameters and make the input less susceptible to high frequency variations. The work found that this layer had marginal benefits over regular CNN layers in classification, but had improved stability to noisy inputs.

5.2 Recap of Useful Terms

5.2.1 Convolutional Layers

Let the output of a CNN at layer l be:

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

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

easily differentiate between the spatial and channel dimensions. A typical convolutional layer in a standard CNN (ignoring the bias term) is:

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

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

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

5.2.2 Wavelet Transforms

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

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

where R is the rotation matrix, $1 \le j \le J$ indexes the scale, and $1 \le k \le K$ indexes θ to give K angles between 0 and π. We copy notation from **bruna_invariant_2013** and define $\lambda = (j,k)$ and the set of all possible λs is Λ whose size is $|\Lambda| = JK$. The wavelet transform, including lowpass, is then:

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

5.2.3 Scattering Transforms

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

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

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

$$U[p]x = U[\lambda_m] \cdots U[\lambda_2]U[\lambda_1]x \tag{5.2.6}$$
$$= ||\cdots|x * \psi_{\lambda_1}| * \psi_{\lambda_2}| \cdots * \psi_{\lambda_m}| \tag{5.2.7}$$

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

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

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

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

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

The resulting scattering coefficients have many nice properties, one of which is stability to diffeomorphisms (such as shifts and warping). From **mallat_group_2012**, if $\mathcal{L}_\tau x = x(\mathbf{u} - \tau(\mathbf{u}))$ is a diffeomorphism which is bounded with $\|\nabla\tau\|_\infty \leq 1/2$, then there exists a $K_L > 0$ such that:

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

where $P = \text{length}(p)$ is the scattering order, and $F(\tau)$ is a function of the size of the displacement, derivative and Hessian of τ, $H(\tau)$ **mallat_group_2012**:

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

5.3 Locally Invariant Layer

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

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

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

Recall that λ is the tuple (j,k) for $1 \le j \le J$, $1 \le k \le K$ and is used to select the bandpass wavelet at scale j and orientation k. Note that an input to the wavelet modulus propagator $\tilde{W}$ with C channels has $(JK+1)C$ output channels – C lowpass channels and JKC modulus bandpass channels. Let us define a new output z with index variable q such that:

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

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

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

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

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

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

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

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

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

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

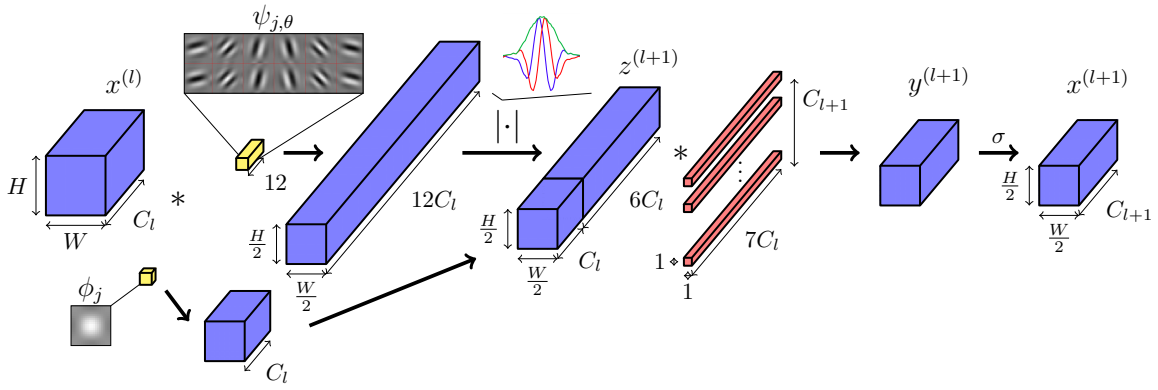


Figure 5.1: **Block Diagram of Proposed Invariant Layer for $j = J = 1$.** Activations are shaded blue, fixed parameters yellow and learned parameters red. Input $x^{(l)} \in \mathbb{R}^{C_l \times H \times W}$ is filtered by real and imaginary oriented wavelets and a lowpass filter and is downsampled.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

Figure 5.1 shows a block diagram for this process.

5.3.1 Properties

5.3.1.1 Recovering the original ScatterNet Design

The first thing to note is that with careful choice of A and σ, we can recover the original translation invariant ScatterNet **bruna_invariant_2013**, **oyallon_scaling_2017**. If $C_{l+1} = (JK+1)C_l$ and A is the identity matrix $I_{C_{l+1}}$, we remove the mixing and then $y^{(l+1)} = \tilde{W}x$.

Further, if $\sigma = \text{ReLU}$ as is commonly the case in training CNNs, it has no effect on the positive locally invariant terms U. It will affect the averaging terms if the signal is not positive, but this can be dealt with by adding a channel dependent bias term α_c to $x^{(l)}$ to ensure it is positive. This bias term will not affect the propagated signals as $\int \alpha_c \psi_\lambda(\mathbf{u}) d\mathbf{u} = 0$. The bias can then be corrected by subtracting $\alpha_c \|\phi_J\|_2$ from the averaging terms after taking the ReLU, then $x^{(l+1)} = \tilde{W}x$.

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

5.3.1.2 Flexibilty of the Layer

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

5.3.1.3 Stability to Noise and Deformations

Let us define the action of our layer on the scattering coefficients to be Vx. We would like to find a bound on $\|V\mathcal{L}_\tau x - Vx\|$. To do this, we note that the mixing is a linear operator and hence is Lipschitz continuous. The authors in **qiu_dcfnet:_2018** 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 (5.2.10) holds.

5.4 Implementation Details

Again, we use the DT$\mathbb{C}$WT **selesnick_dual-tree_2005** for our wavelet filters $\psi_{j,\theta}$ due to their fast implementation with separable convolutions which we will discuss more in subsection 5.4.3. There are two side effects of this choice. The first is that the number of orientations of wavelets is restricted to $K = 6$. The second is that we naturally downsample the output activations by a factor of 2 for each direction for each scale j, giving a 4^j downsampling factor overall. This represents the source of the invariance in our layer. If we do not wish to downsample the output (say to make the layer fit in a larger network), we can bilinearly interpolate the output of our layer. This is computationally cheap to do on its own, but causes the next layer's computation to be higher than necessary (there will be almost no energy for frequencies higher than $f_s/4$).

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

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

5.4.1 Parameter Memory Cost

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

Algorithm 5.1 Locally Invariant Convolutional Layer forward and backward passes

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

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

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

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

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

5.4.2 Activation Memory Cost

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

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

5.4.3 Computational Cost

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

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

Because we are using a decimated wavelet decomposition, the sample rate decreases after each wavelet layer. The benefit of this is that the mixing process then only works on one quarter the spatial size after the first scale and one sixteenth the spatial after the second scale. Restricting ourselves to $J=1$ as we mentioned in section 5.4, the computational cost is then:

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

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

5.4.4 Forward and Backward Algorithm

There are two layer hyperparameters to choose in our layer:

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

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

5.5 Experiments

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

- MNIST: 10 classes, 6000 images per class, 28×28 pixels per image.
- CIFAR-10: 10 classes, 5000 images per class, 32×32 pixels per image.
- CIFAR-100: 100 classes, 500 images per class, 32×32 pixels per image.

- Tiny ImageNet**li__tiny__2017**: 200 classes, 500 images per class, 64×64 pixels per image.

Our experiment code is available at **cotter__learnable__2019-1**.

5.5.1 Layer Introduction with MNIST

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

To minimize the effects of learning from other layers, we build a custom small network, as described in Table 5.1. The first two layers are learned convolutional/invariant layers, followed by a fully connected layer with fixed weights that we can use to project down to the number of output classes. Finally, we add a small learned layer that linearly combines the 10 outputs from the random projection, to give 10 new outputs. This is to facilitate reordering of the outputs to the correct class. This simple network is meant to test the limits of our layer, rather than achieve state of the art performance on MNIST.

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

To simplify the weight variance search, we use Glorot Uniform Initialization **glorot__understanding__2010** and only vary the gain value a:

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

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

We do a grid search over these hyperparameters and use Hyperband **li__hyperband:__2016** 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 **liaw2018tune** 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.

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

(a) Invariant Architecture

Activation Size	Layer Name + Info
$1 \times 28 \times 28$	
	inv1, $A \in \mathbb{R}^{7\times7}$
$7 \times 14 \times 14$	
	inv2, $A \in \mathbb{R}^{49\times49}$
$49 \times 7 \times 7$	
	unravel
2401×1	
	project, $w \in \mathbb{R}^{2401\times10}$
10×1	
	reshuffle, $w \in \mathbb{R}^{10\times10}$
10×1	

(b) Reference Arch with 3×3 convolutions

Activation Size	Layer Name + Info
$1 \times 28 \times 28$	
	conv1, $w \in \mathbb{R}^{7\times1\times3\times3}$
$7 \times 28 \times 28$	
	maxpool1, 2×2
$7 \times 14 \times 14$	
	conv2 $w \in \mathbb{R}^{49\times7\times3\times3}$
$49 \times 14 \times 14$	
	maxpool2, 2×2
$49 \times 7 \times 7$	
	unravel
2401×1	
	project, $w \in \mathbb{R}^{2401\times10}$
10×1	
	reshuffle, $w \in \mathbb{R}^{10\times10}$
10×1	

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

Hyperparameter	Values
Learning Rate (lr)	$\{0.0316,\ 0.1,\ 0.316,\ 1\}$
Momentum (mom)	$\{0,\ 0.25,\ 0.5,\ 0.9\}$
Weight Decay (wd)	$\{10^{-5},\ 3.16\times10^{-5},\ 10^{-4},\ 3.16\times10^{-4}\}$
Weight Gain (a)	$\{0.5,\ 1.0,\ 1.5,\ 2.0\}$

5.5.1.1 Proposed Expansions

The results from the previous section seem to indicate that our proposed invariant layer is a slightly worse substitute for a convolutional layer. However we believe that this is due to the centred nature of the wavelet bases that were used to generate the z and later the y coefficients. A similar effect was seen in the previous chapter in Figure 4.7 where the space of shapes attainable by mixing wavelet coefficients in a 3×3 area was much richer than those attainable by only mixing in a 1×1 area.

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

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

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

		accuracy	
Architecture	$\theta = \{\text{lr}, \text{mom}, \text{wd}, \text{a}\}$	mean	std
Convolutional	$\{0.1,\ 0.5,\ 10^{-5},\ 1.5\}$	97.3	0.29
Invariant	$\{0.032,\ 0.9,\ 3.2\times10^{-5},\ 1.0\}$	96.6	0.26

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

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

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

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

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

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

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

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

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

5.5.2 Layer Comparison with CIFAR and Tiny ImageNet

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

We perform an ablation study where we progressively swap out convolutional layers for invariant layers keeping the input and output activation sizes the same. As there are 6 layers (or 8 for Tiny ImageNet), there are too many permutations to list the results for swapping out all layers for our locally invariant layer, so we restrict our results to swapping 1 or 2 layers. We also report the accuracy when we swap out all of the layers. Table 5.6 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.

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

This network is optimized with stochastic gradient descent with momentum. The initial learning rate is 0.5, momentum is 0.85, batch size $N = 128$ and weight decay is 10^{-4}. For

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**simonyan_very_2014** architecture. C is a hyperparameter that controls the network width, we use $C = 64$ for our initial tests. The activation size rows are offset from the layer description rows to convey the input and output shapes.

(a) CIFAR Architecture

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

(b) Tiny ImageNet Architecture

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

CIFAR-10/CIFAR-100 we scale the learning rate by a factor of 0.2 after 60, 80 and 100 epochs, training for 120 epochs in total. For Tiny ImageNet, the rate change is at 18, 30 and 40 epochs (training for 45 in total).

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

5.5.3 Network Comparison

In the previous section, we examined how the locally invariant layer performs when directly swapped in for a convolutional layer in an architecture designed for convolutional layers. In

Table 5.6: **Ablation results for invariant layer.** Results for testing VGG like architecture with convolutional and invariant layers on several datasets. An architecture with 'invX' means the equivalent convolutional layer 'convX' from Table 5.5 was swapped for our proposed layer. The top row is the reference architecture using all convolutional layers. The 'A' architecture is the originally proposed gain layer, the 'B' architecture uses the gainlayer with random shifting of the activations, and the 'C' architecture changes the mixing to be a learned 3×3 kernel acting on the invariant coefficients. Numbers are averages over 5 runs.

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

this section, we look at how it performs in a Hybrid ScatterNet-like **oyallon_hybrid_2017**, **oyallon_scaling_2017**, network.

To build the second order ScatterNet, we stack two invariant layers on top of each other. For 3 input channels, the output of these layers has $3(1+6+6+36) = 147$ channels at $1/16$ the spatial input size. We then use 4 convolutional layers, similar to convC to convF in Table 5.5 with $C = 96$. In addition, we use dropout after these later convolutional layers with drop probability $p = 0.3$.

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

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

For comparison, we have also listed the performance of other architectures as reported by their authors in order of increasing complexity. Our proposed ScatNet D achieves comparable performance with the the All Conv, VGG16 and FitNet architectures. The Deep**he_identity_2016** and Wide**zagoruyko_wide_2016** ResNets perform best, but with very many more multiplies, parameters and layers.

ScatNets A to D with 6 layers like convC to convG from Table 5.5 after the scattering, achieve $58.1, 59.6, 60.8$ and 62.1% top-1 accuracy on Tiny ImageNet. As these have more parameters and multiplies from the extra layers we exclude them from Table 5.7.

5.6 Conclusion

In this work we have proposed a new learnable scattering layer, dubbed the locally invariant convolutional layer, tying together ScatterNets and CNNs. We do this by adding a mixing between the layers of ScatterNet allowing the learning of more complex shapes than the ripples seen in **cotter_visualizing_2017**. This invariant layer can easily be shaped to allow it to drop in the place of a convolutional layer, theoretically saving on parameters and computation. However, care must be taken when doing this, as our ablation study showed that the layer only improves upon regular convolution at certain depths. Typically, it seems wise to use the invariant layer right before a sample rate change.

We have developed a system that allows us to pass gradients through the Scattering Transform, something that previous work has not yet researched. Because of this, we were able to train end-to-end a system that has a ScatterNet surrounded by convolutional layers

Table 5.7: **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 $32 \times 32 \times 3$ image. An asterisk indicates that the value was estimated from the architecture description.

	Arch. Properties			Top 1 Accuracies	
Arch. Name	N_l	#Mparam	#Mmults	CIFAR-10	CIFAR-100
ScatNet A	4	2.6	165	89.4	67.0
ScatNet B	6	2.7	167	91.1	70.7
ScatNet C	5	3.7	251	91.6	70.8
ScatNet D	7	4.3	294	93.0	73.5
All Conv**springenberg__striving__2014-3**	8	1.4	281*	92.8	66.3
VGG16**liu__very__2015**	16	138*	313*	91.6	-
FitNet**romero__fitnets:__2014**	19	2.5	382	91.6	65.0
ResNet-1001**he__identity__2016**	1000	10.2	4453*	95.1	77.3
WRN-28-10**zagoruyko__wide__2016**	28	36.5	5900*	96.1	81.2

and with our proposed mixing. We were surprised to see that even a small convolutional layer before Scattering helps the network, and a very shallow and simple Hybrid-like ScatterNet was able to achieve good performance on CIFAR-10 and CIFAR-100.

Chapter 6

Learning in the Wavelet Domain

In this chapter we move away from the ScatterNet ideas from the previous chapters and instead look at using the wavelet domain as a new space in which to learn. 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 the progressive stages all result in a steady demodulation of signal energy towards zero frequency.

In this chapter we introduce the *wavelet gain layer* which starts in a similar fashion to the ScatterNet – by taking the DT$\mathbb{C}$WT 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 DT$\mathbb{C}$WT allows gains and phases to be changed at will without introducing sampling artifacts.

We also briefly explore the possibility of doing nonlinearities in the wavelet domain. The goal being to ultimately connect multiple wavelet gain layers together with nonlinearities before returning to the pixel domain.

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. However, replacing a convolutional layer with a gain layer degrades performance by a small amount.

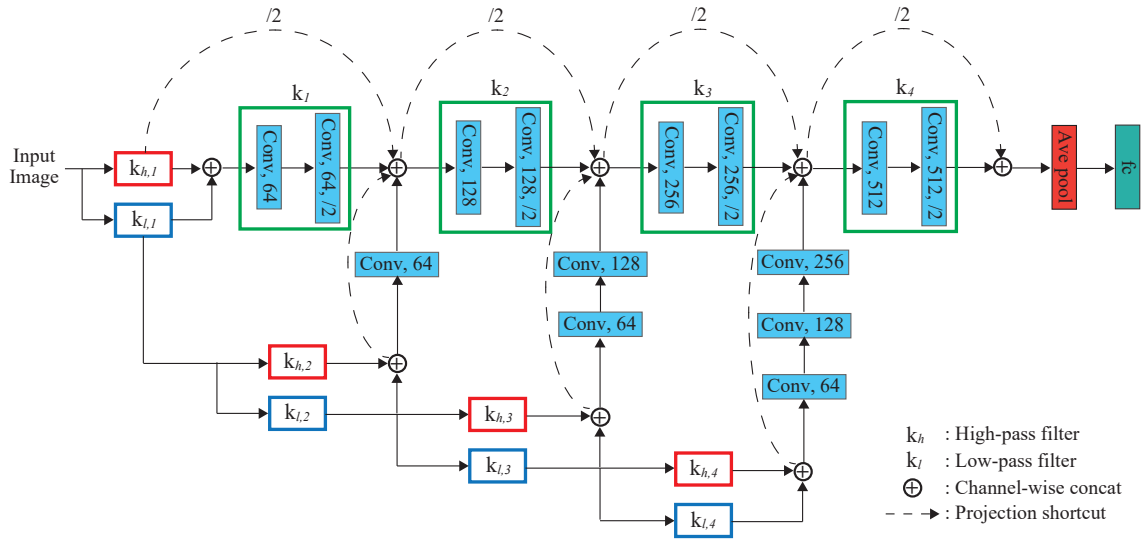


Figure 6.1: **Architecture using the DWT as a frontend to a CNN.** Figure 1 from **fujieda__wavelet__2018**. Fujieda et. al. take a multiscale wavelet decomposition of the input before passing the input through a standard CNN. They learn convolutional layers independently on each subband and feed these back into the network at different depths, where the resolution of the subband and the network activations match.

6.1 Related Work

Fujieda et. al. use a DWT in combination with a CNN to do texture classification and image annotation **fujieda__wavelet__2017**, **fujieda__wavelet__2018**. In particular, they take a multiscale wavelet transform of the input image, combine the actviations at each scale independently with learned weights, and feed these back into the network where the activation resolution size matches the subband resolution. The architecture block diagram is shown in Figure 6.1, taken from the original paper. This work found that their dubbed 'Wavelet-CNN' could outperform competetive non wavelet based CNNs on both texture classification and image annotation.

Several works also use wavelets in deep neural networks for super-resolution **guo__deep__2017** and for adding detail back into dense pixel-wise segmentation tasks **ma__detailed__2018**. These typically save wavelet coefficients and use them for the reconstruction phase.

In **rippel__spectral__2015**, **rippel__spectral__2015** parameterize filters in the DFT domain. Rather than having a pixel domain filter $\mathbf{w} \in \mathbb{R}^{F \times C \times K \times K}$, they learn a set of fourier coefficients $\hat{\mathbf{w}} \in \mathbb{C}^{F \times C \times K \times \lceil K/2 \rceil}$ (the reduced spatial size is a result of enforcing that the inverse DFT of their filter to be real, so the parameterization is symmetric). On the forward

pass of the neural network, they take the inverse DFT of $\hat{\mathbf{w}}$ to obtain $\mathbf{w}$ and then convolve this with the input $\mathbf{x}$ as a normal CNN would do.[1].

Note that an important point should be laboured about reparameterizing filters in either the wavelet or Fourier domains: any invertible linear transform of the parameter space will not change the updates if a linear optimization scheme is used (for example standard gradient descent, or SGD with momentum)[2]. This is proved in appendix C. We make this point clear as a natural extension to continue the work in **rippel_spectral_2015** would be to parameterize filters in the wavelet domain, taking inverse transforms to the pixel domain and doing normal convolution.

The work presented in this chapter does not learn wavelet coefficients for convolutional filters to be applied in the pixel domain, rather we learn wavelet filters *and* do convolution in the wavelet domain too.

6.2 Background and Notation

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

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

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

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

We then define the product of these new Z-transform signals to be the channel-wise convolution. E.g. for the 3-D filter $h[c, \mathbf{n}]$ with Z-transform $H(c, \mathbf{z})$ 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:

$$H(c, \mathbf{z}) X(c, \mathbf{z}) = \sum_{\mathbf{n}} \left(\sum_{\mathbf{k}} h[c, \mathbf{n} - \mathbf{k}] x[c, \mathbf{k}] \right) \mathbf{z}^{-\mathbf{n}} \tag{6.2.3}$$

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

[2]This fact is something that **rippel_spectral_2015** briefly allude to, but do not make clear

Recall from subsection 2.3.1 that a typical convolutional layer in a standard CNN gets the next layer's output in a two-step process:

$$y^{(l+1)}[f,\mathbf{n}] = \sum_{c=0}^{C_l-1} x^{(l)}[c,\mathbf{n}] * h_f^{(l)}[c,\mathbf{n}] \tag{6.2.4}$$

$$x^{(l+1)}[f,\mathbf{n}] = \sigma\left(y^{(l+1)}[f,\mathbf{n}]\right) \tag{6.2.5}$$

If we define the nonlinearity σ_z to be the action of σ to each z-coefficient in the polynomial $Y(c,\mathbf{z})$, then we can rewrite (6.2.4) and (6.2.5) as:

$$Y^{(l+1)}(f,\mathbf{z}) = \sum_{c=0}^{C_l-1} X^{(l)}(c,\mathbf{z})\, H_f^{(l)}(c,\mathbf{z}) \tag{6.2.6}$$

$$X^{(l+1)}(f,\mathbf{z}) = \sigma_z(Y^{(l+1)}(f,\mathbf{z})) \tag{6.2.7}$$

6.2.1 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.

$$\text{DT}\mathbb{C}\text{WT}_J(x) = \{u_{lp}, u_{j,k}\}_{1\le j\le J, 1\le k\le 6} \tag{6.2.8}$$

Each of these coefficients then has size:

$$u_{lp} \in \mathbb{R}^{N\times C\times \frac{H}{2^{J-1}}\times\frac{W}{2^{J-1}}} \tag{6.2.9}$$

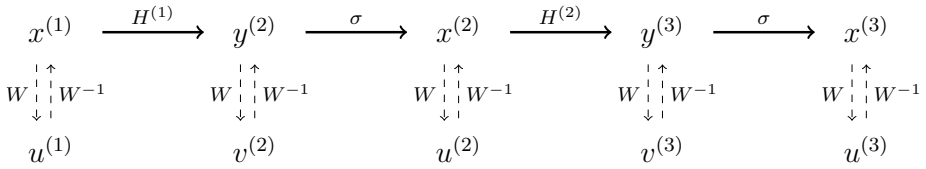
$$u_{j,k} \in \mathbb{C}^{N\times C\times \frac{H}{2^{J}}\times\frac{W}{2^{J}}} \tag{6.2.10}$$

Note that the lowpass coefficients are twice as large as in a fully decimated transform, a feature of the redundancy of the DT$\mathbb{C}$WT and the fact that the lowpass coefficients are most conveniently represented as purely real values, whereas the bandpass ones are most conveniently complex in (6.2.8).

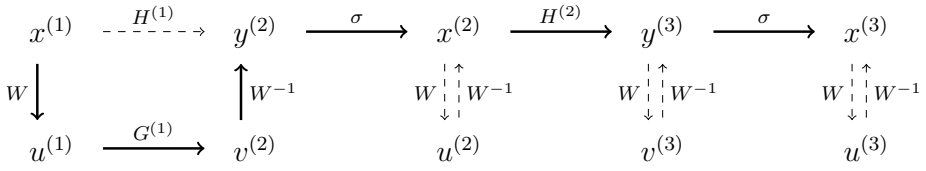
If we ever want to refer to all the subbands at a given scale, we will drop the k subscript and call them u_j. Likewise, u refers to the whole set of DT$\mathbb{C}$WT coefficients.

6.3 Learning in Multiple Spaces

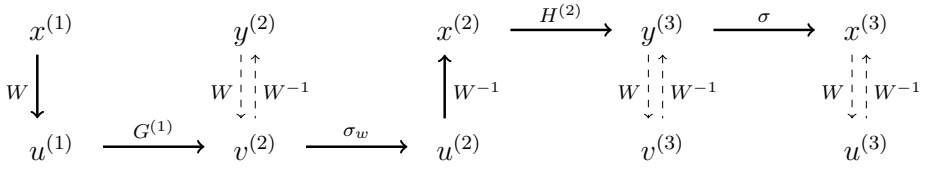
At the beginning of each layer l of a neural network we have the activations $x^{(l)}$. Naturally, all of these activations have their equivalent wavelet coefficients $u^{(l)}$.



(a) A regular convolutional layer



(b) Gains applied in the wavelet domain



(c) Gains and nonlinearities applied in the wavelet domain

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.

From (6.2.4), convolutional layers also have intermediate activations $y^{(l)}$. Let us discern these from the x coefficients and modify (6.2.8) to say the DT$\mathbb{C}$WT of $y^{(l)}$ gives $v^{(l)}$.

We now propose the *wavelet gain layer* G. The name 'gain layer' comes from the inspiration for this chapter's work, as it appears often the first layer of a CNN could be achieved by simply setting gains for different regions in the frequency space of an image.

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 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}?**

6.3.1 The DT$\mathbb{C}$WT 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}, $g_{j,k}$:

$$g_{lp} \in \mathbb{R}^{C_{l+1} \times C_l \times k_{lp} \times k_{lp}} \tag{6.3.1}$$

$$g_{1,1} \in \mathbb{C}^{C_{l+1} \times C_l \times k_1 \times k_1} \tag{6.3.2}$$

$$g_{1,2} \in \mathbb{C}^{C_{l+1} \times C_l \times k_1 \times k_1} \tag{6.3.3}$$

$$\vdots$$

$$g_{J,6} \in \mathbb{C}^{C_{l+1} \times C_l \times k_J \times k_J} \tag{6.3.4}$$

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.

With these gains we define the action of the gain layer $v = Gu$ to be:

$$v_{lp}[f, \mathbf{n}] = \sum_{c=0}^{C_l - 1} u_{lp}[c, \mathbf{n}] * g_{lp}[f, c, \mathbf{n}] \tag{6.3.5}$$

$$v_{1,1}[f, \mathbf{n}] = \sum_{c=0}^{C_l - 1} u_{1,1}[c, \mathbf{n}] * g_{1,1}[f, c, \mathbf{n}] \tag{6.3.6}$$

$$v_{1,2}[f, \mathbf{n}] = \sum_{c=0}^{C_l - 1} u_{1,2}[c, \mathbf{n}] * g_{1,2}[f, c, \mathbf{n}] \tag{6.3.7}$$

$$\vdots$$

$$v_{J,6}[f, \mathbf{n}] = \sum_{c=0}^{C_l - 1} u_{J,6}[c, \mathbf{n}] * g_{J,6}[f, c, \mathbf{n}] \tag{6.3.8}$$

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).

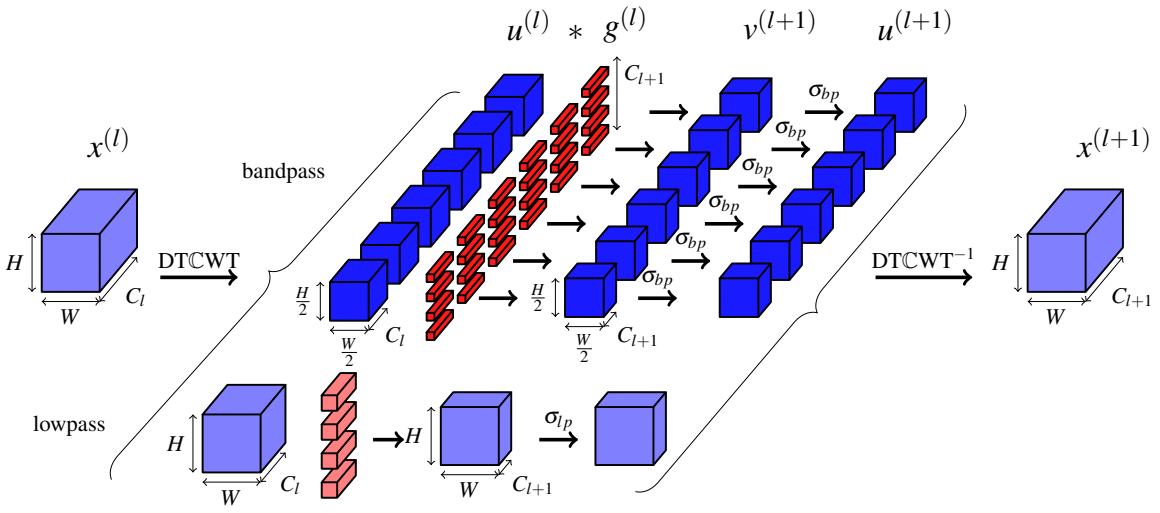


Figure 6.3: **Diagram of proposed method to learn in the wavelet domain.** Activations are shaded blue and learned parameters red. Deeper shades of blue and red indicate complex valued activations/weights, and lighter values indicate real valued activations/weights. The input $x^{(l)} \in \mathbb{R}^{C_l \times H \times W}$ 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 σ_w 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.

The action of the gain layer with only a single-scale wavelet transform, $J = 1$, is is shown in Figure 6.3.

6.3.1.1 The Output

Due to the shift invariant properties of the DTℂWT, 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) - (**??**), then we preserve the shift invariant properties (see Theorem D.2 and Theorem D.3). Then the inverse DTℂWT 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 block diagram shown in Figure 6.4a[3]

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). The gain for a specific subband previously was called $g_{j,k}$ but we here refer to it

[3]Note that despite the resemblance to many block 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.

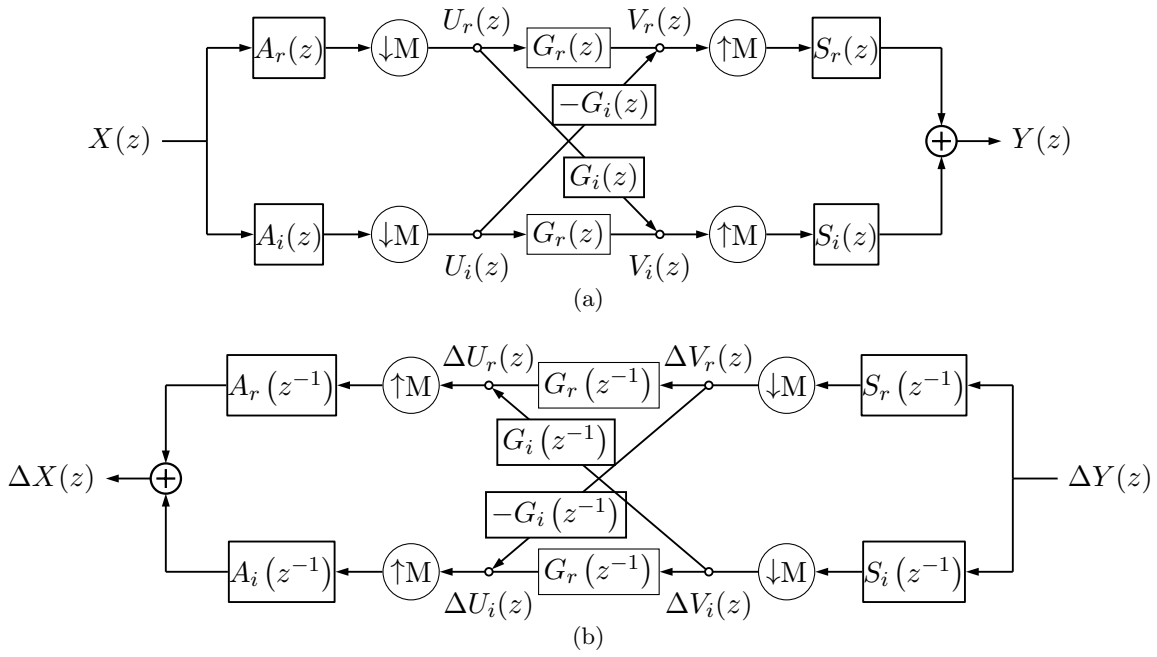


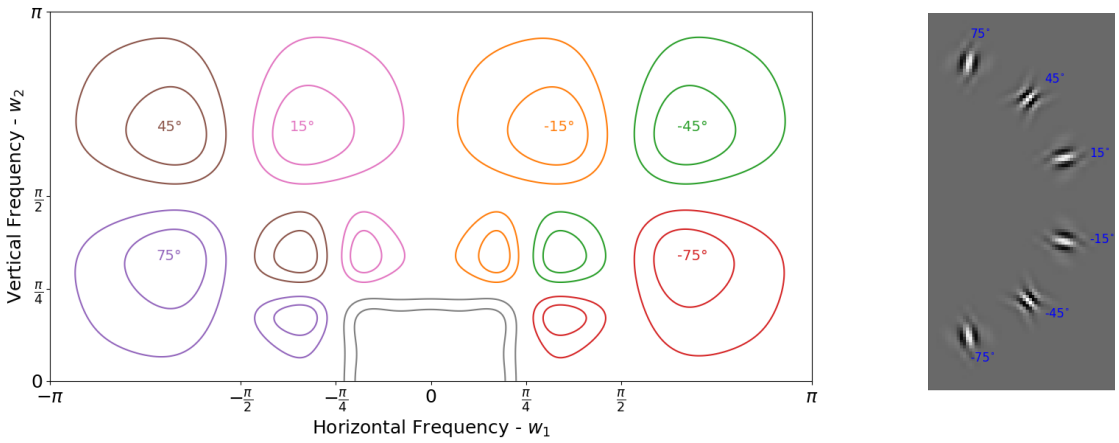
Figure 6.4: **Forward and backward block diagrams for** DT$\mathbb{C}$WT **gain layer.** Based on Figure 4 in **kingsbury_complex_2001**. Ignoring the G gains, the top and bottom paths (through A_r, S_r and A_i, S_i respectively) make up the the real and imaginary parts for *one subband* of the dual 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).

simply as $G = G_r + jG_i$. The output of this layer is:

$$\begin{aligned} Y(z) = \frac{2}{M} X(z) \big[& G_r(z^M) \left(A_r(z) S_r(z) + A_i(z) S_i(z) \right) \\ & + G_i(z^M) \left(A_r(z) S_i(z) - A_i(z) S_r(z) \right) \big] \end{aligned} \tag{6.3.9}$$

Again, see appendix D for the derivation which shows how the aliasing terms caused by the downsampling by M are largely eliminated.

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 show the contour plots for the frequency support of each of these subbands. The complex gain vector g has elements which adjust the gain and phase of each subband independently. The magnitude of each element controls the amplitude of the frequency



(a) (b)

Figure 6.5: DT$\mathbb{C}$WT **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 $P(z)$ and $Q(z)$ 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.

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).

6.3.1.2 Backpropagation

We start with the property that for a convolutional block, the gradient with respect to the input is the gradient with respect to the output convolved with the time reverse of the filter (proved in subsubsection 2.3.1.2). More formally, if $Y(z) = H(z)X(z)$:

$$\Delta X(z) = H(z^{-1})\Delta Y(z) \tag{6.3.10}$$

where $H(z^{-1})$ is the Z-transform of the time/space reverse of $H(z)$, $\Delta Y(z) \triangleq \frac{\partial L}{\partial Y}(z)$ is the gradient of the loss with respect to the output, and $\Delta X(z) \triangleq \frac{\partial L}{\partial X}(z)$ is the gradient of the loss with respect to the input. If H were complex, the first term in Equation 6.3.10 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 $H(z^{-1})$.

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.

Let us 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[4]. Then the weight update equations are:

$$\Delta G_r(z) = \Delta V_r(z)U_r(z^{-1}) + \Delta V_i(z)U_i(z^{-1}) \tag{6.3.11}$$

$$\Delta G_i(z) = -\Delta V_r(z)U_i(z^{-1}) + \Delta V_i(z)U_r(z^{-1}) \tag{6.3.12}$$

The passthrough equations have similar form to (6.3.9):

$$\Delta X(z) = \frac{2\Delta Y(z)}{M}\left[G_r(z^{-M})\left(A_r(z)S_r(z) + A_i(z)S_i(z)\right) + G_i(z^{-M})\left(A_r(z)S_i(z) - A_i(z)S_r(z)\right)\right] \tag{6.3.13}$$

6.3.2 Examples

Figure 6.6 show 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 domain[5]. 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.

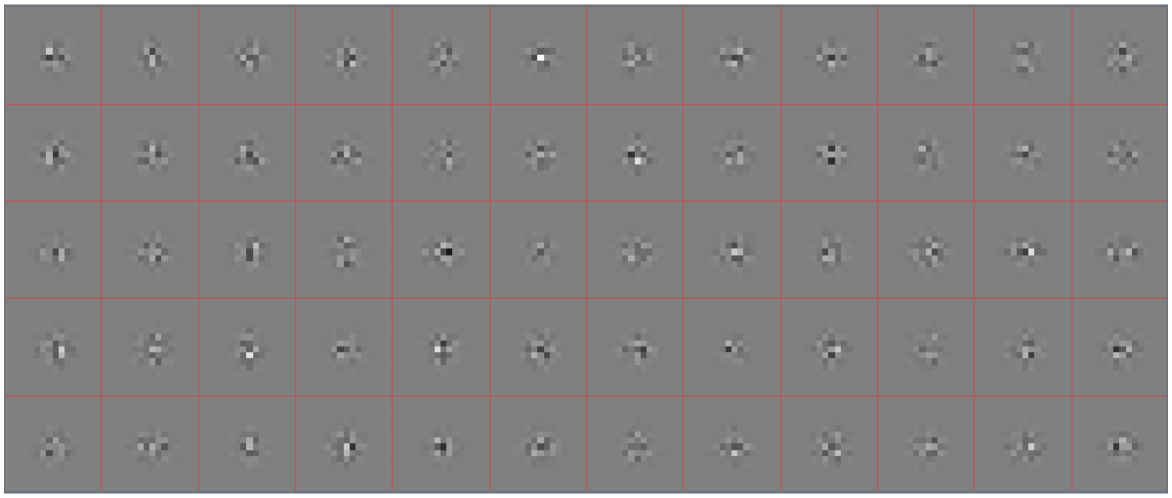
Our experiments show that the distribution for the DT$\mathbb{C}$WT gain layer matches random vectors with roughly 11.5 degrees of freedom (c.f. the 12 variables the layer has). Similarly for the DWT, the the normalized cross-correlation matches the distribution for random vectors with roughly 2.8 degrees 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 representatitve power despite the redundancy of the transform.

6.3.3 Implementation Details

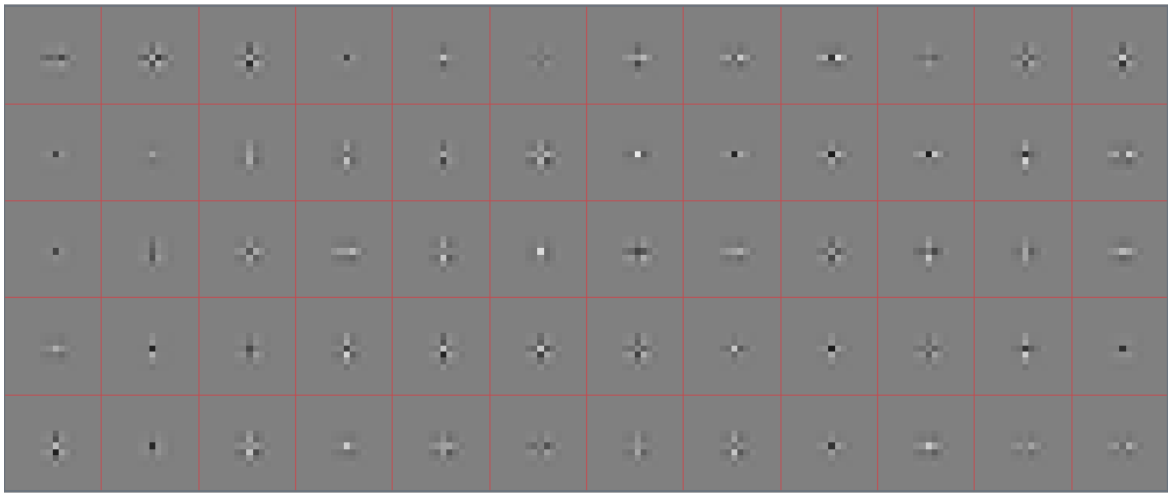
Before analyzing its performance, we compare the implementation properties of our proposed layer to a standard convolutional layer.

[4] An interesting result is that for orthogonal wavelet transforms, $S(z^{-1}) = A(z)$, so the backwards pass of an inverse wavelet transform is equivalent to doing a forward wavelet transform. Similarly, the backwards pass of the forward transform is equivalent to doing the inverse transform.

[5] Modifying DWT coefficients causes a loss of the alias cancellation properties so these are not true impulse response.



(a)



(b)

Figure 6.6: **Example outputs from an impulse input for the proposed gain layers.** Example outputs $y = W^{-1}GWx$ for an impulse x for the DT$\mathbb{C}$WT gain layer and for a similarly designed DWT gain layer. (a) shows the output y for a DT$\mathbb{C}$WT 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 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 DT$\mathbb{C}$WT.

6.3.3.1 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.

$$\#\text{conv params} = k^2 C_l C_{l+1} \tag{6.3.14}$$

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 then:

$$\#\text{params} = (2 \times 6 + 1) C_l C_{l+1} = 13 C_l C_{l+1} \tag{6.3.15}$$

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.

6.3.3.2 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

$$\#\text{conv floats} = HWC_l \tag{6.3.16}$$

Our layers require us to save the wavelet coefficients u_{lp} and $u_{j,k}$ for updating the g terms as in (6.3.11) and (6.3.12). For the $4:1$ redundant DT$\mathbb{C}$WT, this requires:

$$\#\text{DT}\mathbb{C}\text{WT floats} = 4HWC_l \tag{6.3.17}$$

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 DT$\mathbb{C}$WT 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 DT$\mathbb{C}$WT 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 DT$\mathbb{C}$WT gain layer would still only require $4HWC_l$ floats.

6.3.3.3 Computational Cost

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

For the DT$\mathbb{C}$WT, the overhead calculations are the same as in subsection 5.4.3, 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:

$$\#\text{mults/pixel} = \underbrace{\frac{6\times 4}{4}C_{l+1}}_{\text{bandpass}} + \underbrace{C_{l+1}}_{\text{lowpass}} + \underbrace{36}_{\text{DT}\mathbb{C}\text{WT}} + \underbrace{36}_{\text{DT}\mathbb{C}\text{WT}^{-1}} = 7C_{l+1}+72 \tag{6.3.18}$$

which is marginally smaller than the $9C_{l+1}$ of a 3×3 convolutional layer (if $C_{l+1} > 36$).

6.3.3.4 Parameter Initialization

For both layer types we use the Glorot Initialization scheme **glorot_understanding_2010** with $a=1$:

$$g_{ij} \sim U\left[-\sqrt{\frac{6}{(C_l+C_{l+1})k^2}}, \sqrt{\frac{6}{(C_l+C_{l+1})k^2}}\right] \tag{6.3.19}$$

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 perform non-linearities in purely in the spatial domain, as in a conventional CNN layer. This is the first objective in section 6.3, comparing G to H.

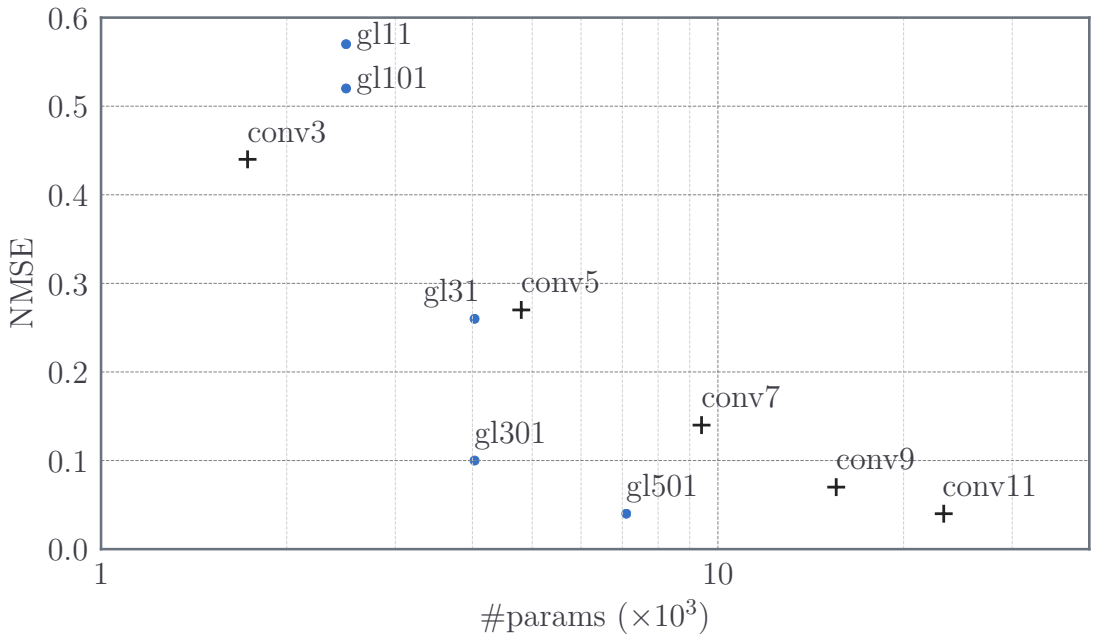


Figure 6.7: **Normalized mean squared error for conv layer and wavelet gain layer regression..** We minimize of (6.4.1) and (6.4.2) on the ImageNet validation set, where the target is made from convolving the input with AlexNet's first layer filters. This plot shows the final MSE score compared to the number of learnable parameters. The original conv layer has spatial support 11×11, and the equivalent number of parameters is shown as a blue dotted line. The four points labelled 'conv$n$' 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, our convolutional layer H and our gain layer G. Let $\|H\|_2$, $\|G\|_2$ be the ℓ_2 norm of the weights for each layer. We would like assume that we do not have direct access to H_0 but only the convolved outputs $Y = H_0 X$. Then, we

would like to solve:

$$\arg\min_{H} (Y - HX)^2, \quad \text{s.t. } h[c, \mathbf{n}] = 0, \ \forall \mathbf{n} \notin \mathcal{R} \tag{6.4.1}$$

$$\arg\min_{G} (Y - W^{-1}GWX)^2, \quad \text{s.t. } g_{j,k}[c, \mathbf{n}] = 0, \ \forall \mathbf{n} \notin \mathcal{R}' \tag{6.4.2}$$

for some support regions $\mathcal{R}, \mathcal{R}'$. E.g. $\mathcal{R}$ could be a 3×3 or 5×5 block, and similarly $\mathcal{R}'$ could define a desired support for each gain 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 not worried with the optimization procedure chosen here, but of the final distances $\|Y - HX\|$ and $\left\|Y - W^{-1}GWx\right\|$ (or equivalently, their squares). We choose to find H and G by gradient descent, using the validation set for ImageNet as the data input-output pair (X, Y). After 3–5 epochs, both H and G typically settle into their global minimum. 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:

$$NMSE = \frac{1}{N} \sum_{n=1}^{N} \frac{\mathbb{E}\left[(Y - \hat{Y})^2\right]}{\mathbb{E}[Y^2]} \tag{6.4.3}$$

The resulting NMSEs are shown in Figure 6.7 . A label 'gl*ab*' indicates a single scale gain layer with $a \times a$ support in the lowpass and $b \times b$ support in the highpass; a label 'gl*abc*' 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 things 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 gl310 and gl301, two instances that have the same number of parameters, but gl310 has g_1 with non-zero support, and gl301 has g_2 with non-zero support.

6.4.2 Ablation Studies

Figure 6.7 is a useful guide on how the gainlayer might be placed in a deep CNN. gl110 (a gain layer with a 1×1 lowpass kernel and a 1×1 bandpass kernel at the first scale), gl101 (same as gl110 but no gain at first scale and 1×1 at second scale), and conv3 all achieve similar MSEs. Additionally gl310, gl301, and conv5 all achieve similar MSEs.

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 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 and Tiny ImageNet. 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.

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, $g_{lp} \in \mathbb{R}^{C \times 3 \times 3 \times 3}$, $g_1 \in \mathbb{C}^{C \times 6 \times 3 \times 1 \times 1}$
$C \times 32 \times 32$	batchnorm + relu		
$C \times 16 \times 16$	pool1, max pool 2×2		
$2C \times 16 \times 16$	conv2, $w \in \mathbb{R}^{2C \times C \times 5 \times 5}$	*or*	gain2, $g_{lp} \in \mathbb{R}^{2C \times C \times 5 \times 5}$, $g_1 \in \mathbb{C}^{2C \times 6 \times C \times 1 \times 1}$
$2C \times 16 \times 16$	batchnorm + relu		
$2C \times 8 \times 8$	pool2, max pool 2×2		
$4C \times 8 \times 8$	conv3, $w \in \mathbb{R}^{4C \times 2C \times 5 \times 5}$	*or*	gain3, $g_{lp} \in \mathbb{R}^{4C \times 2C \times 5 \times 5}$, $g_1 \in \mathbb{C}^{4C \times 6 \times 2C \times 1 \times 1}$
$4C \times 8 \times 8$	batchnorm + relu		
$4C \times 1 \times 1$	avg, 8×8 average pool		
10, 100	fc1, fully connected		

In this experiment, we build a 3 layer CNN with 5×5 convolutional kernels, described in Table 6.1. To help differentiate with the small kernel network introudeced 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 then test the difference in accuracy achieved by replacing each of the three convolution layers with gl310[6]. On the two CIFAR datasets, we train for 120 epochs, decaying learning

[6] Although the gain layers with no gain in the first scale and gain in the second scale performed better than those with gain gl310 and gl510 in subsection 6.4.1, we saw them perform consistently worse in the following ablation studies. For ease of presentation, we have shown only the results from the single scale gain layer.

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 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 **cotter_dtcwt_2018**.

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' for CIFAR-100 and compare them to the reference architecture.

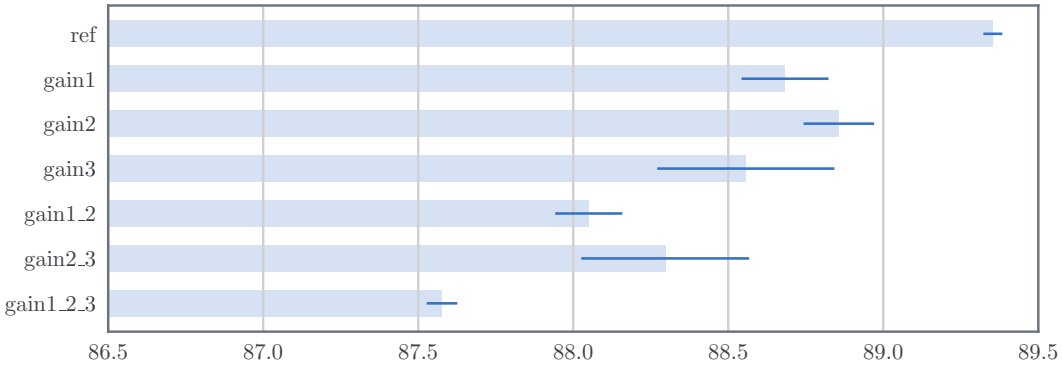
6.4.3.1 Bandpass Coefficients

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. 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 0 impact the entire network accuracy?

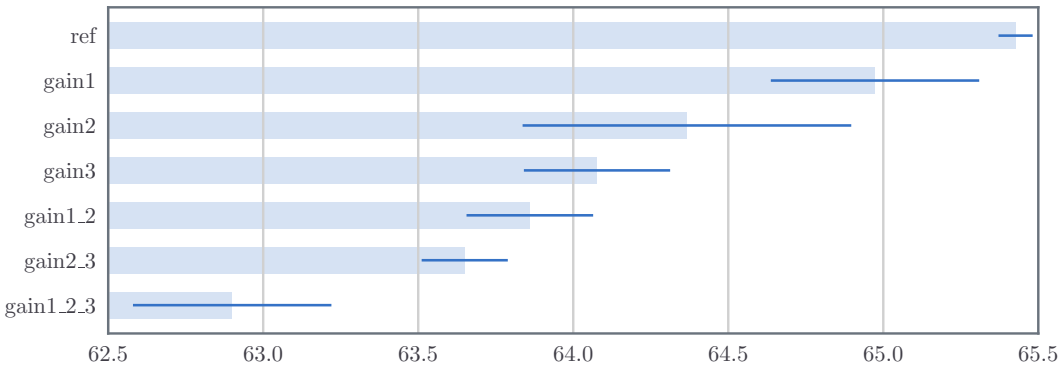
We measure the dropoff in accuracy when a hard threshold t is applied to the bandpass gains g_1 for the three gain layers of 'gain1_2_3'. The resulting sparsity of each layer and the network performance is shown in Figure 6.9b. This figure shows that despite the high cost of the bandpass gains – $12C_lC_{l+1}$ for a 1×1 gain, very few of these need to be nonzero.

6.4.3.2 DeConvolution and Filter Sensitivity

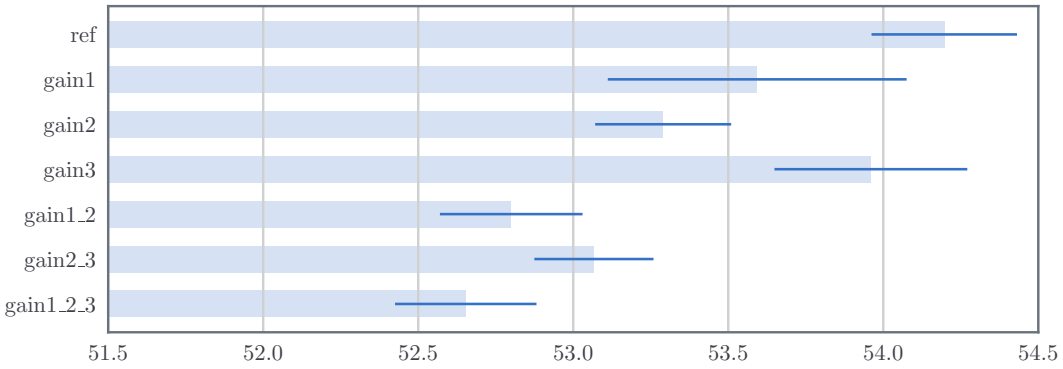
To visualize what the gain layer is responsive to, we build a deconvolutional system similar to the one described in chapter 4. In particular, we present the entire CIFAR-100 validation set to the reference architecture and to the gain1_2_3 architecture, keeping track of what most highly excites each 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.



(a) CIFAR-10

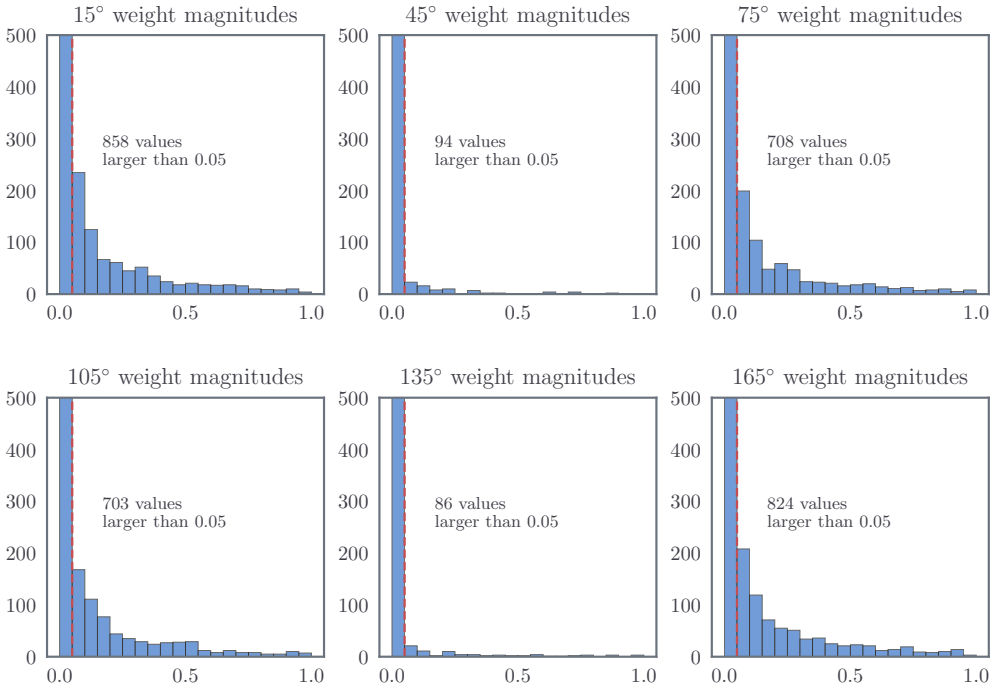


(b) CIFAR-100

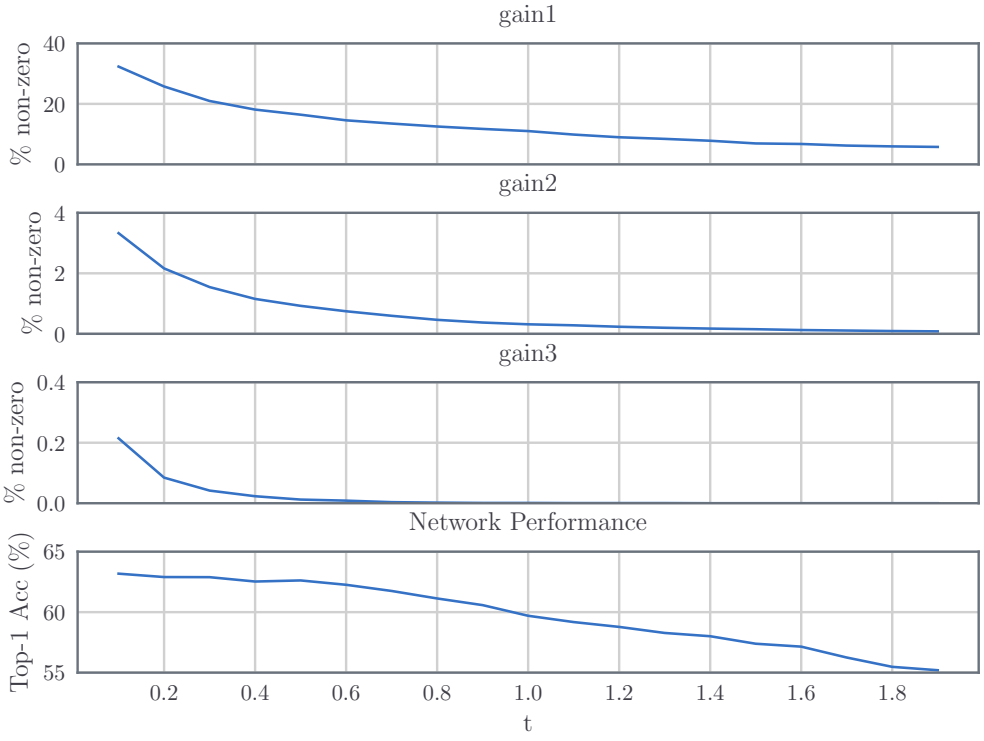


(c) Tiny ImageNet

Figure 6.8: **Large kernel ablation results 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.



(a) Bandpass Magnitude Distributions



(b) Accuracy Dropoff from Thresholding

Figure 6.9: **Bandpass Gain Properties.** (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 or near 0. 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 and classification accuracy is 0.5% lower than the non-thresholded accuracy.

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 still responding to 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 square reconstructions.

6.5 Wavelet Based Nonliearities

Returning to the goals from section 6.3, 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 to use? 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. 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 **papyan_theoretical_2018**, **papyan_convolutional_2016**.

In this section we will look at each, see if they add to the gain layer, and see if they open the possibility of having multiple layers in the wavelet domain.

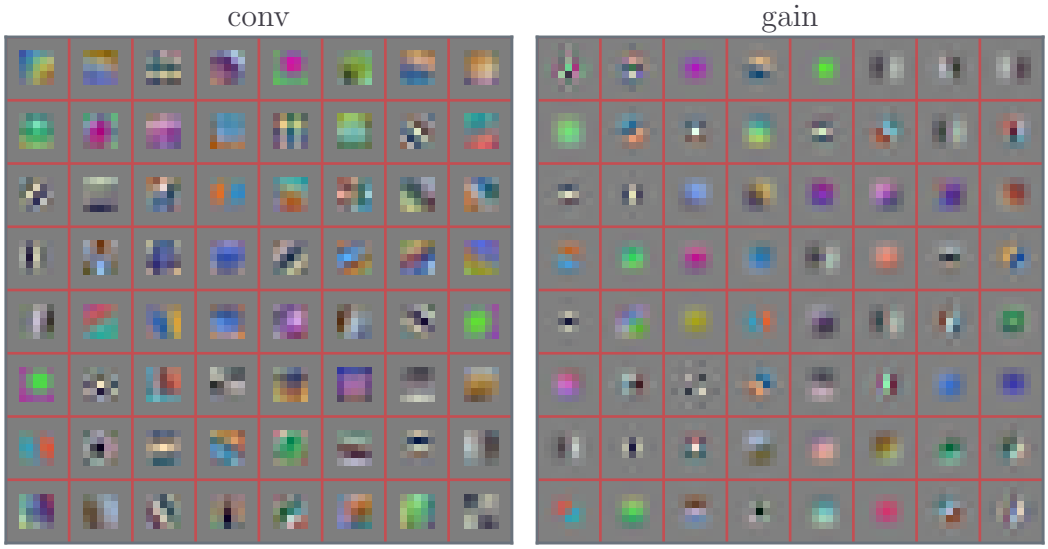
6.5.1 ReLUs in the Wavelet Domain

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:

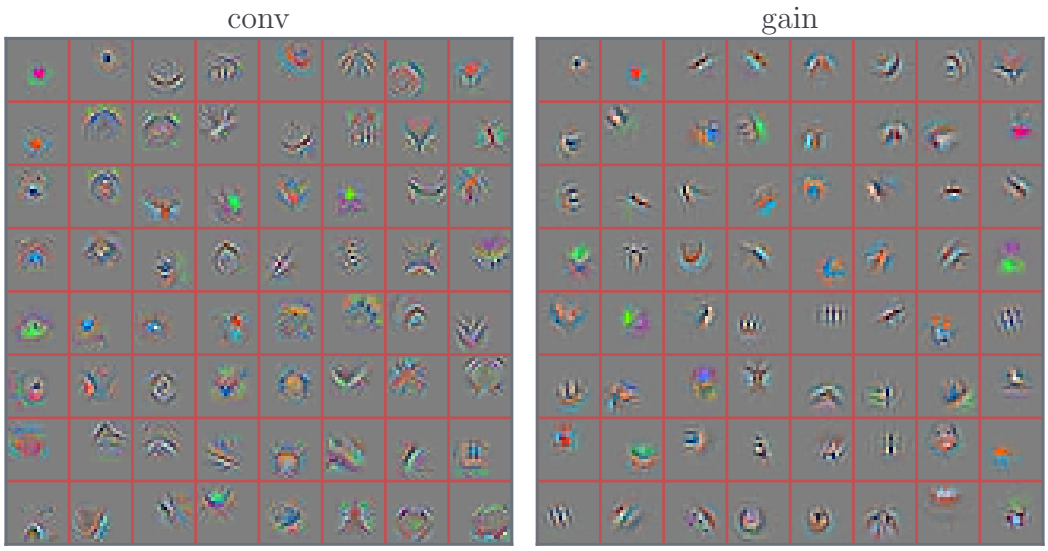
$$u_{lp} = \max(0,\ v_{lp}) \tag{6.5.1}$$

$$u_j = \max(0,\ \mathrm{Re}\,(v_j)) + j\max(0,\ \mathrm{Im}\,(v_j)) \tag{6.5.2}$$

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,



(a) First layer



(b) Second layer

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 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.

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, let a bandpass coefficient at a given scale be $v = r_v e^{j\theta_v}$ and define $\mu_r = \mathbb{E}[r_j]$ 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:

$$r_u = \text{ReLU}(BN(r_v)) = \max(0,\ \gamma\frac{r_v - \mu_r}{\sigma_r} + \beta) \tag{6.5.3}$$

$$u = r_u e^{j\theta_v} \tag{6.5.4}$$

This also works equivalently on the lowpass coefficients although v_{lp} can be negative unlike r_v:

$$u_{lp} = \text{ReLU}(BN(v_{lp})) = \max(0, \gamma'\frac{v_{lp} - \mu_{lp}}{\sigma_{lp}} + \beta') \tag{6.5.5}$$

6.5.2 Thresholding

For $t \in \mathbb{R},\ z = re^{j\theta} \in \mathbb{C}$ the pointwise hard thresholding is defined as:

$$\mathcal{H}(z,t) = \begin{cases} z & r \geq t \\ 0 & r < t \end{cases} \tag{6.5.6}$$

$$= \mathbb{I}(r > t)z \tag{6.5.7}$$

The pointwise soft thresholding is:

$$\mathcal{S}(z,t) = \begin{cases} (r-t)e^{j\theta} & r \geq t \\ 0 & r < t \end{cases} \tag{6.5.8}$$

$$= \max(0, r-t)e^{j\theta} \tag{6.5.9}$$

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:

$$r_u = \max(0, \gamma\frac{r_v - \mu_r}{\sigma_r} + \beta) \tag{6.5.10}$$

$$= \frac{\gamma}{\sigma_r}\max\left(0, r_v - \left(\mu_r - \frac{\sigma_r\beta}{\gamma}\right)\right) \tag{6.5.11}$$

then if $t' = \mu_v - \frac{\sigma_r\beta}{\gamma} > 0$, doing batch normalization followed by a ReLU on the magnitude of the 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 they are similar to batch normalizations and ReLUs, we would also like to test how well thresholding performs as a wavelet nonlinearity. To do this, we can:

Algorithm 6.1 The *wave layer* pseudocode

procedure WAVELAYER(x)
  $u_{lp},\ u_1 \leftarrow \text{DT}\mathbb{C}\text{WT}(x, \text{nlevels} = 1)$
  $v_{lp},\ v_1 \leftarrow G(u_{lp},\ u_1)$ ▷ the normal wave gain layer
  $w_{lp} \leftarrow \sigma_{lp}(v_{lp})$ ▷ lowpass nonlinearity
  $w_1 \leftarrow \sigma_{bp}(v_1)$ ▷ bandpass nonlinearity
  $y \leftarrow \text{DT}\mathbb{C}\text{WT}^{-1}(w_{lp},\ w_1)$
  $x \leftarrow \sigma_{pixel}(y)$ ▷ pixel nonlinearity
  return x
end procedure

- 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 a desired sparsity level 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:

$$\mathcal{S}(v,t) = \max(0, r - |t|)e^{j\theta} \tag{6.5.12}$$

Learning a threshold is only possible for soft thresholding, as $\frac{\partial L}{\partial t}$ is not defined for hard thresholding.

6.5.3 Non-Linearity Experiments

Taking the same 'gain1_2_3' architecture used for CIFAR-100, we expand the *wave gain layer* into one bigger layer dubbed the *wave layer*, described in Algorithm 6.1. In the wave layer, we have 3 different nonliearities: the pixel, the lowpass and the bandpass nonlinearity.

For these experiments, we test over a grid of possible options for these three functions:

Nonlinearity	Values			
Pixel	None	BN+ReLU		
Lowpass	None	ReLU	BN+ReLU	$\mathcal{S}$
Bandpass	None	ReLU	BN+MagReLU	$\mathcal{S}$

Where:

- 'None' means no nonlinearity – $\sigma(x) = x$.
- 'BN+ReLU' is batch normalization and ReLU (applies only to real valued activations) e.g. (6.5.5).

Table 6.2: **Different Nonlinearities in the Gain Layer.** Top-1 Accuracies for '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. $\sigma_{pixel} = \sigma_{lp} = \sigma_{bp} =$ None is a linear system (with max pooling). $\sigma_{pixel} =$ ReLU, $\sigma_{lp} = \sigma_{bp} =$ 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 $\sigma_{pixel} =$ None, $\sigma_{lp} =$ ReLU and $\sigma_{bp} =$ BN + ReLU.

(a) $\sigma_{pixel} =$ None

σ_{bp} \ σ_{lp}	None	ReLU	BN+ReLU	$\mathcal{S}$
None	45.0	63.8	64.8	61.9
ReLU	42.6	62.8	63.9	61.3
BN+MagReLU	48.5	**65.3**	64.6	62.9
$\mathcal{S}$	44.6	63.9	63.6	61.7

(b) $\sigma_{pixel} =$ BN + ReLU

σ_{bp} \ σ_{lp}	None	ReLU	BN+ReLU	$\mathcal{S}$
None	62.8	60.7	64.2	63.0
ReLU	62.5	60.3	64.2	63.2
BN+MagReLU	64.9	61.6	64.7	65.2
$\mathcal{S}$	63.3	60.9	63.9	63.3

- 'ReLU' is a ReLU without batch normalization. Can be applied to the real and imaginary parts of a complex activation independently i.e. (6.5.2). See section E.3 for equations for the passthrough gradients for this nonlinearity.

- '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. See section E.4 for information on the passthrough and update equations for this nonlinearity.

As the pixel nonliearity has only two options, the results are best displayed as a pair of tables, firstly for no nonlinearity and secondly for the 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 gainlayer 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.

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.

6.5.4 Ablation Experiments with Nonlinearities

Now that we have found the best nonlinearity to use for the *wave layer*, will this improve our ablation study from subsection 6.4.2? To test this, we repeat the same experiment on CIFAR-100 only, but use the *wave layer* described in Algorithm 6.1, with $\sigma_{pixel} = \text{None}$, $\sigma_{lp} = \text{ReLU}$, 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 wave layer, but any other changes degrade performance.

6.6 Conclusion

In this chapter we have presented the novel idea of learning filters by taking activations into the wavelet domain. In the wavelet domain then we can apply the proposed *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 multirate 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 more wavelet gain layers 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

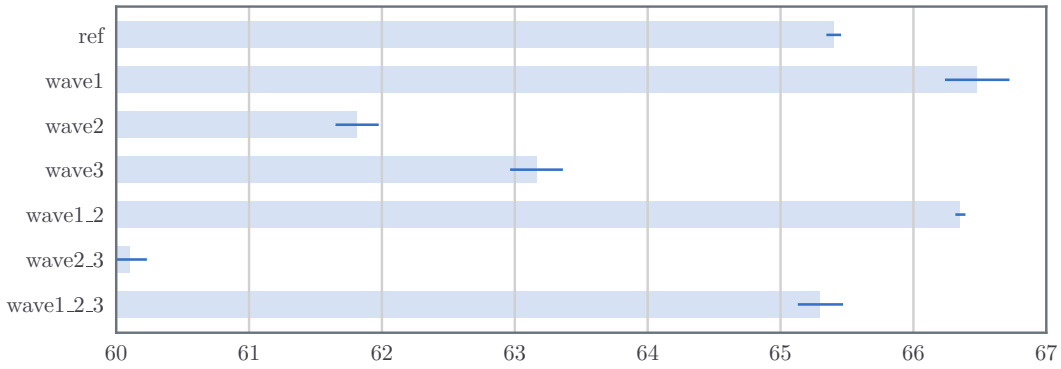


Figure 6.11: **CIFAR-100 Ablation results with the *wave layer*.** We use the same naming scheme from subsection 6.4.2 but to differentiate between the results from Figure 6.8b we call the options 'waveX' (for *wave layer* vs *gain layer*). When we add nonlinearities in the wavelet domain, the ablation results from 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 narrower spread.

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, possibily 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 to make the *wave layer*, we saw that we were able to achieve some improvements in performance over a fully convolutional architecture (Figure 6.11). The *wave 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 wave layer as part of a deeper system.

Chapter 7

Conclusion

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 natural extensions that we were not able to explore due to time constraints, and some 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 DT$\mathbb{C}$WT as the chosen wavelets for the original ScatterNet design greatly speeds up computation (see Table 3.2). We were also able to derive the backpropagation equations for wavelet and scattering layers, aiding the use of ScatterNets as part of deep networks. As part of this, we tested out the performance of the DT$\mathbb{C}$WT based ScatterNet as a front end to a simple CNN for some small classification tasks, and compared its performance to the Morlet based system. We found that as well as being faster, the performance was often better when using the DT$\mathbb{C}$WT wavelets Table 3.5. When doing these tests, we found that of the wavelet choices available for the DT$\mathbb{C}$WT, those with the fewest taps (and largest stopband gain and transition bandwidth) 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 patters that most highly excite filters of a CNN. We believe this may explain why ScatterNets perform well on texture discrimination **bruna_invariant_2013**

but less well on natural image classification **oyallon_deep_2015**. 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 handle 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 zeroed out. We also found the surprising result that on the datasets we tested, diagonal edges were less important than their vertical or horizontal counterparts (Figure 4.4). If the input images were rotated by $\pm 30°$ then the diagonal channels became the most important. This echoes the experiments of **blakemore_development_1970blakemore_development_1970** who controlled the orinetation 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 5.7) and can help a CNN (Table 5.6). 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 of returning 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 the reference system with only convolutional layers (Figure 6.8). Despite the slight reduction in performance, we saw some nice properties to 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). The gains and nonlinearity together make the *wave layer*. Using this, we saw that we were able to achieve

some improvements in performance over a fully convolutional architecture (Figure 6.11). The wave layer seems to work best at the beginning of the CNN but more research still needs to be done with deeper systems.

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$\mathbb{C}$WT in *Pytorch Wavelets* is slower than it could be. A 10×10 convolutional filter with 100 multiplies per input pixel is roughly twice as quick to compute as a DT$\mathbb{C}$WT 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 **haber_learning_2017**, **fujieda_wavelet_2018**.

7.2.2 Expanding Tests on Invariant Layer

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 place 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_scaling_2017** 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 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 than staying in the wavelet domain, as it involves projecting from a redundant space to a non-redundant one. More experiments could be done where this projection is not done.

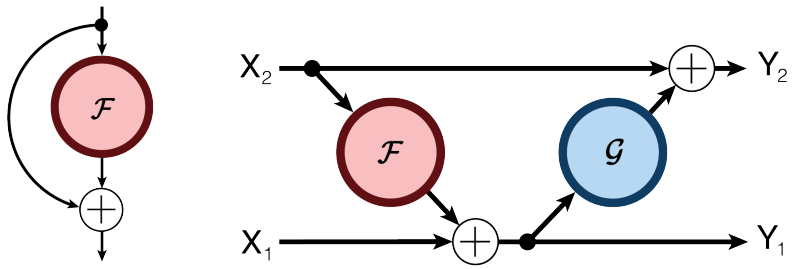
7.2.4 ResNets and Lifting

We briefly mentioned ResNets in subsection 2.4.6 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 **sweldens__lifting__1998**, **daubechies__factoring__1998**. 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:

$$y_1 = \mathcal{F}(x_1) + x_2 \tag{7.2.1}$$

$$y_2 = \mathcal{G}(y_1) + x_1 \tag{7.2.2}$$

Figure 7.1 shows the similarities between the two designs. The works **gomez__reversible__2017**, **jacobsen__i-revnet:__2018** both make the small modifications to the ResNet design to make a lifting style architecture. **gomez__reversible__2017 gomez__reversible__2017** do



(a) Residual (b) Lifting

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 taken from **gomez__reversible__2017**.

this to save memory on the activations for backpropagation (you do not need to save meta-information on the forwards pass as you can regenerate activations on the backwards pass). **jacobsen__i-revnet:__2018 jacobsen__i-revnet:__2018** extend on **gomez__reversible__2017** and also explore merging images with linear interpolation in the activation space, reconstructing from 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 **szegedy__intriguing__2014**.

While **szegedy__intriguing__2014 szegedy__intriguing__2014** 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__practical__2017 papernot__practical__2017** 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 an arms race in recent years, with measures and counter-measures constantly being developed. An excellent paper review some attacks and defences is **carlini__adversarial__2017**.

One such concerning black box attack is described in **engstrom__rotation__2017**, where **engstrom__rotation__2017** show that even simple transformations such as translations and rotations are enough to fool many modern CNNs. This is something our invariant layer would naturally be able to protect against.

Alternatively, a recent defence tactic is described in **cisse__parseval__2017**. In this paper, **cisse__parseval__2017** propose to have non-expansive convolutional and pooling

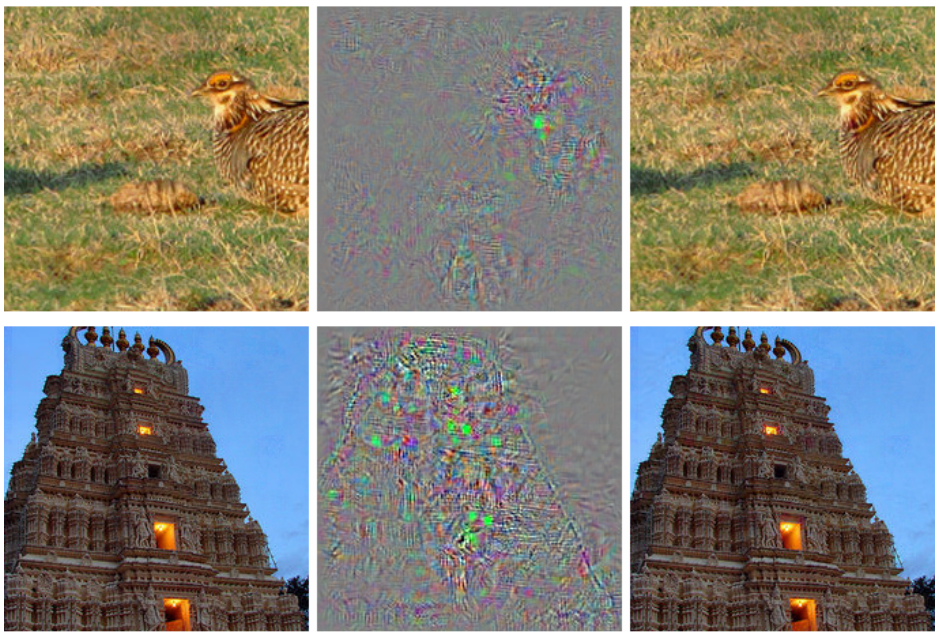


Figure 7.2: **Adversarial examples that can fool AlexNet.** Two examples of images that were correctly classified on the left, with additive signals in the centre (scale magnified by 10) and the resulting images on the right. Both of the output images are then predicted to be an ostrich. Images taken from **szegedy__intriguing__2014**.

layers (Lipschitz constant 1) and have weight matrices that are approximately Parseval tight frames, a result intimately related to the tight framed DT$\mathbb{C}$WT. 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.

We feel that the smooth learning offered by wavelet basis functions has a potentially large scope for making CNNs more robust to many attacks.

7.2.6 Convolutional Sparse Coding

Recent on convolutional sparse coding and convolutional dictionary learning **liu__online__2017**, **liu__first__2017**, **papyan__convolutional__2016**, **papyan__convolutional__2017**, **papyan__theoretical_** 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 an interesting piece of work to try to extend this recent work on convolutional sparse coding to wavelet bases. **rubinstein__double__2010** 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 coding.

7.2.7 Weight Matrix Properties

Some recent work by **advani_high-dimensional_2017** **advani_high-dimensional_2017** 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.

Appendix A

Architecture Used for Experiments

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

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

To do hyperparameter search we used the Tune package **liaw2018tune** which we highly recommend, as it makes running trials in parallel very easy.

A.1 Run Times of some of the Propsed Layers

.

Appendix B

Forward and Backward Algorithms

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

B.1 Gradients of Sample Rate Changes

Define decimation of a signal X by M as:

$$Y[n] = X[Mn] \tag{B.1}$$

and interpolation by M as:

$$Y[n] = \begin{cases} X[\frac{n}{M}] & n = Mk,\ k \in \mathbb{Z} \\ 0 & \text{otherwise} \end{cases} \tag{B.2}$$

B.1.1 Decimation Gradient

From (B.1) the gradient $\frac{\partial Y_n}{\partial X_k}$ is:

$$\frac{\partial Y_n}{\partial X_k} = \begin{cases} 1 & k = Mn \\ 0 & \text{otherwise} \end{cases} \tag{B.3}$$

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

$$\frac{\partial J}{\partial X_k} = \frac{\partial J}{\partial Y_n}\frac{\partial Y_n}{\partial X_k} \tag{B.4}$$

$$= \begin{cases} \Delta Y[\frac{k}{M}] & k = Mn \\ 0 & \text{otherwise} \end{cases} \tag{B.5}$$

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:

$$\frac{\partial Y_n}{\partial X_k} = \begin{cases} 1 & n = Mk \\ 0 & \text{otherwise} \end{cases} \tag{B.6}$$

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

$$\frac{\partial J}{\partial X_k} = \frac{\partial J}{\partial Y_n}\frac{\partial Y_n}{\partial X_k} \tag{B.7}$$

$$= \begin{cases} \Delta Y[n] & n = Mk \\ 0 & \text{otherwise} \end{cases} \tag{B.8}$$

$$= \Delta Y[Mk] \tag{B.9}$$

which is decimation of ΔY by M.

B.2 Extra Algorithms

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

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

function AG:IDWT:BWD(δy)
    load g_0, g_1, $mode$
    $g_0,\ g_1 \leftarrow \text{flip}(g_0),\ \text{flip}(g_1)$ ▷ flip the filters as in (3.3.1)
    $\Delta lo,\ \Delta hi \leftarrow \text{afb1d}(\delta y,\ g_0,\ g_1,\ mode,\ axis = -2)$
    $\Delta ll,\ \Delta lh \leftarrow \text{afb1d}(\Delta lo,\ g_0,\ g_1,\ mode,\ axis = -1)$
    $\Delta hl,\ \Delta hh \leftarrow \text{afb1d}(\Delta hi,\ g_0,\ g_1,\ mode,\ axis = -1)$
    return Δll, Δlh, Δhl, Δhh
end function

Algorithm B.2 Smooth Magnitude

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

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

Algorithm B.3 Q2C

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

Appendix C

Invertible Transforms and Optimization

This Appendix proves the assertion made in section 6.1, namely that:

> Any invertible linear transform of the parameter space will not change the updates if a linear optimization scheme is used (for example standard gradient descent, or SGD with momentum).

We prove the case for the DFT as the invertible transform, but it can easily be extended to other transforms so long as (C.3) and (C.4) hold.

C.1 Background

We define the DFT as the orthonormal version, i.e.

$$U_{ab} = \frac{1}{\sqrt{N}} \exp\{\frac{-2j\pi ab}{N}\}$$

then call $X = \text{DFT}\{x\}$. In matrix form the 2-D DFT is then **petrou_image_2010**:

$$X = \text{DFT}\{x\} = UxU \tag{C.1}$$

$$x = \text{DFT}^{-1}\{X\} = U^* X U^* \tag{C.2}$$

The gradients of these linear operators are:

$$\frac{\partial L}{\partial X} = U \frac{\partial L}{\partial x} U = \text{DFT}\left\{\frac{\partial L}{\partial x}\right\} \tag{C.3}$$

$$\frac{\partial L}{\partial x} = U^* \frac{\partial L}{\partial X} U^* = \text{DFT}^{-1}\left\{\frac{\partial L}{\partial X}\right\} \tag{C.4}$$

C.2 Analysis

Consider a single filter parameterized in the DFT and spatial domains presented with the exact same data and with the same ℓ_2 regularization λ, and learning rate η. Let the spatial filter at time t be $\mathbf{w}_t$ and the Fourier-parameterized filter be $\hat{\mathbf{w}}_t$, and let

$$\hat{\mathbf{w}}_1 = \mathrm{DFT}\{\mathbf{w}_1\} \tag{C.5}$$

We need to prove that presenting the data to each system in the same order results in the same outputs for all time t.

Proof. Assume that $\hat{\mathbf{w}}_t = \mathrm{DFT}\{\mathbf{w}_t\}$ for a given timestep t.

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:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \left(\frac{\partial L}{\partial \mathbf{w}_t} + \lambda \mathbf{w}_t \right) \tag{C.6}$$

$$= (1 - \eta\lambda)\mathbf{w}_t - \eta \frac{\partial L}{\partial \mathbf{w}_t} \tag{C.7}$$

$$\hat{\mathbf{w}}_{t+1} = \hat{\mathbf{w}}_t - \eta \left(\frac{\partial L}{\partial \hat{\mathbf{w}}_t} + \lambda \hat{\mathbf{w}}_t \right) \tag{C.8}$$

$$= (1 - \eta\lambda)\hat{\mathbf{w}}_t - \eta \frac{\partial L}{\partial \hat{\mathbf{w}}_t} \tag{C.9}$$

We can then compare the effect the new parameters would have on the next minibatch by calculating $\mathrm{DFT}^{-1}\{\hat{\mathbf{w}}_{t+1}\}$. We then get:

$$\mathrm{DFT}^{-1}\left\{\hat{\mathbf{w}}_{t+1}\right\} = \mathrm{DFT}^{-1}\left\{(1 - \eta\lambda)\hat{\mathbf{w}}_t - \eta \, \frac{\partial L}{\partial \hat{\mathbf{w}}_t}\right\} \tag{C.10}$$

$$= (1 - \eta\lambda)\mathbf{w}_t - \eta \, \mathrm{DFT}^{-1}\left\{ \frac{\partial L}{\partial \hat{\mathbf{w}}_t} \right\} \tag{C.11}$$

$$= (1 - \eta\lambda)\mathbf{w}_t - \eta \frac{\partial L}{\partial \mathbf{w}_t} \tag{C.12}$$

$$= \mathbf{w}_{t+1} \tag{C.13}$$

where we have used (C.4) for the second last line. Since the IFT of $\hat{\mathbf{w}}$ is the same as $\mathbf{w}$ at timestep $t+1$ given they are the same at timestep t, and we have set them equal to each other at timestep 1, by induction they will be the same for all time. □

We have proved the simpler case for simple SGD, but the same result holds when momentum terms are added. This does not however hold for the Adam **kingma_adam:_2014** or Adagrad **duchi_adaptive_2011** optimizers, which automatically rescale the learning rates for each parameter based on estimates of the parameter's variance.

Appendix D

DT$\mathbb{C}$WT Single Subband Gains

This appendix proves that the DT$\mathbb{C}$WT *gain layer* proposed in chapter 6 maintains the shift invariant properties of the DT$\mathbb{C}$WT.

Recall that with multirate systems, upsampling by M takes $X(z)$ to $X(z^M)$ and downsampling by M takes $X(z)$ to $\frac{1}{M}\sum_{k=0}^{M-1} X(W_M^k z^{1/M})$ 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$\mathbb{C}$WT

It is easiest to prove the shift invariance of the gain layer by expanding on the shift invariance of the DT$\mathbb{C}$WT proofs done in **kingsbury_complex_2001**.

Let us consider one subband of the DT$\mathbb{C}$WT. This includes the coefficients from both tree A and tree B. For simplicity in this analysis we will consider the 1-D DT$\mathbb{C}$WT 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:

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

where the aliasing terms are formed from the addition of the rotated z transforms, i.e. when $k \neq 0$.

Theorem D.1. *The aliasing terms for* (D.1) *cancel out if the impulse responses of B and D are Hilbert transforms of the impulse responses of A and C respectively, and the system in Figure D.1 is nearly shift invariant.*

Proof. See section 4 of **kingsbury_complex_2001** for the full proof of this, and section 7 for the bounds on what 'nearly' shift invariant means. □

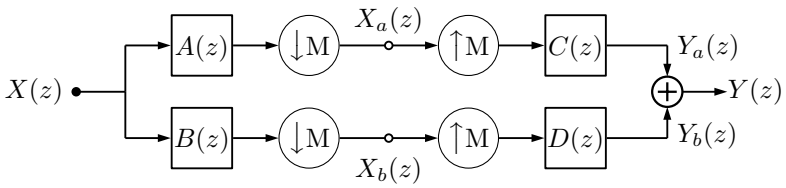


Figure D.1: **Block Diagram of 1-D** DTℂWT**.** Note the top and bottom paths are through the wavelet or scaling functions from just level m ($M = 2^m$). Figure based on Figure 4 in **kingsbury_complex_2001**.

Consider the complex filters defined as:

$$P(z) = \frac{1}{2}(A(z) + jB(z)) \tag{D.2}$$

$$Q(z) = \frac{1}{2}(C(z) - jD(z)) \tag{D.3}$$

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:

$$A(z) = P(z) + P^*(z) \tag{D.4}$$

$$B(z) = -j(P(z) - P^*(z)) \tag{D.5}$$

$$C(z) = Q(z) + Q^*(z) \tag{D.6}$$

$$D(z) = j(Q(z) - Q^*(z)) \tag{D.7}$$

Substituting these into (D.1) gives:

$$A(W^k z)C(z) + B(W^k z)D(z) = 2P(W^k z)Q(z) + 2P^*(W^k z)Q^*(z) \tag{D.8}$$

Using (D.8), Kingsbury show 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 are single side band, then so are P^* and Q^*, but they now have support only on the left hand side of the frequency plane.

Given these properties, 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 $k = \pm 1$. **kingsbury_complex_2001** quantify the amount of transition band overlap and show it is negligible.

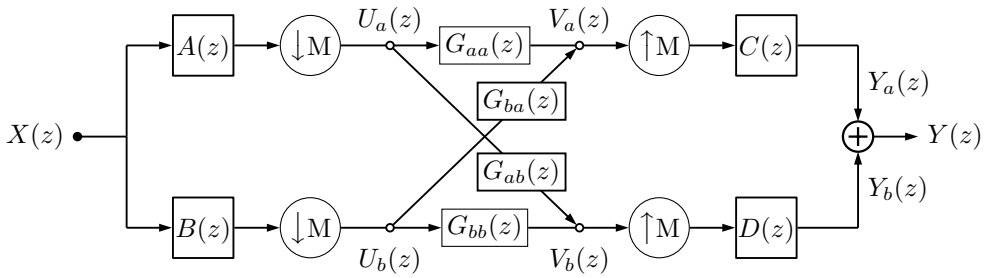


Figure D.2: **Block Diagram of 1-D** DT$\mathbb{C}$WT **with subband gains.**

This means $A(W^k z)C(z) + B(W^k z)D(z) = 0$ when $k \neq 0$ and (D.1) reduces to:

$$Y(z) = \frac{1}{M} X(z) \left[A(z)C(z) + B(z)D(z)\right] \tag{D.9}$$

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 fucntion from U_i to V_j be G_{ij} for $i, j \in \{a, b\}$. Then V_a and V_b are:

$$\begin{aligned}
V_a(z) &= U_a(z)G_{aa}(z) + U_b(z)G_{ba}(z) && \text{(D.10)}\\
&= \frac{1}{M}\sum_k X(W^k z^{1/M}) \left[A(W^k z^{1/M})G_{aa}(z) + B(W^k z^{1/M})G_{ba}(z)\right] && \text{(D.11)}\\
V_b(z) &= U_a(z)G_{ab}(z) + U_b(z)G_{bb}(z) && \text{(D.12)}\\
&= \frac{1}{M}\sum_k X(W^k z^{1/M}) \left[A(W^k z^{1/M})G_{ab}(z) + B(W^k z^{1/M})G_{bb}(z)\right] && \text{(D.13)}
\end{aligned}$$

Further, Y_a and Y_b are:

$$\begin{aligned}
Y_a(z) &= C(z)V_a(z^M) && \text{(D.14)}\\
Y_b(z) &= D(z)V_b(z^M) && \text{(D.15)}
\end{aligned}$$

Then the output is:

$$\begin{aligned}
Y(z) &= Y_a(z) + Y_b(z) && \text{(D.16)}\\
&= \frac{1}{M}\sum_{k=0}^{M-1} X(W^k z)\Big[A(W^k z)C(z)G_{aa}(z^M) + B(W^k z)D(z)G_{bb}(z) + \\
&\qquad B(W^k z)C(z)G_{ba}(z^M) + A(W^k z)D(z)G_{ba}(z)\Big] && \text{(D.17)}
\end{aligned}$$

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:

$$\begin{aligned} A(W^k z)D(z)-B(W^k z)C(z) &= j\left[P(W^k z)+P^*(W^k z)\right][Q(z)-Q^*(z)]+ && \text{(D.19)}\\ & \quad j\left[P(W^k z)-P^*(W^k z)\right][Q(z)+Q^*(z)] && \text{(D.20)}\\ &= 2j\left[P(W^k z)Q(z)-P^*(W^k z)Q^*(z)\right] && \text{(D.21)} \end{aligned}$$

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. □

Using Theorem D.2, the output is now

$$\begin{aligned} Y(z) &= \frac{2}{M}X(z)\left[G_r(z^M)(AC+BD)+G_i(z^M)(AD-BC)\right] && \text{(D.22)}\\ &= \frac{2}{M}X(z)\left[G_r(z^M)(PQ+P^*Q^*)+G_i(z^M)(PQ-P^*Q^*)\right] && \text{(D.23)} \end{aligned}$$

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:

$$\begin{aligned} V_a(z) &= G_r(z)U_a(z)-G_i(z)U_b(z) && \text{(D.24)}\\ V_b(z) &= G_r(z)U_b(z)+G_i(z)U_a(z) && \text{(D.25)} \end{aligned}$$

which follows the form of a complex multiply. □

Now we know can assume that our DTℂWT 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. Theorem D.3 and (D.23) give us an intuiton for the real and imaginary parts of a complex gain G. The real part G_r affects how much of the bandpass gain $PQ + P^*Q^*$ propagetes 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* and *wave layer*.

E.1 Convolution

Let us represent the complex input with $\mathbf{x} = \mathbf{x}_r + j\mathbf{x}_i$, which is of shape $\mathbb{C}^{C\times n_1\times n_2}$. We call $\mathbf{y} = \mathbf{y}_r + j\mathbf{y}_i$ the result we get from convolving $\mathbf{x}$ with $\mathbf{h} = \mathbf{h}_r + j\mathbf{h}_i$, 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 $\mathbf{y}$:

$$y[\mathbf{n}] = \sum_{c=0}^{C-1}\sum_{\mathbf{k}} h[c,\mathbf{k}]x[c,\mathbf{n}-\mathbf{k}] \tag{E.1}$$

We can expand this with real and imaginary terms:

$$\begin{aligned}
y[\mathbf{n}] &= \sum_{c=0}^{C-1}\sum_{\mathbf{k}} (h_r[c,\mathbf{k}] + jh_i[c,\mathbf{k}])(x_r[c,\mathbf{n}-\mathbf{k}] + jx_i[c,\mathbf{n}-\mathbf{k}]) \\
&= \sum_{c=0}^{C-1}\sum_{\mathbf{k}} (x_r[c,\mathbf{n}-\mathbf{k}]h_r[c,\mathbf{k}] - x_i[c,\mathbf{n}-\mathbf{k}]h_i[c,\mathbf{k}]) \\
&\quad + j\sum_{c=0}^{C-1}\sum_{\mathbf{k}} (x_r[c,\mathbf{n}-\mathbf{k}]h_i[c,\mathbf{k}] + x_i[c,\mathbf{n}-\mathbf{k}]h_r[c,\mathbf{k}]) \\
y_r[\mathbf{n}] + jy_i[\mathbf{n}] &= \sum_{c=0}^{C-1} \left((x_r * h_r) - (x_i * h_i)\right)[c,\mathbf{n}] + \left((x_r * h_i) + j(x_i * h_r)\right)[c,\mathbf{n}]
\end{aligned} \tag{E.2}$$

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:

$$\frac{\partial L}{\partial h_r} = \frac{\partial L}{\partial y_r}\frac{\partial y_r}{\partial h_r} + \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial h_r} \tag{E.3}$$

$$= \Delta y_r \star x_r + \Delta y_i \star x_i \tag{E.4}$$

where $\star$ is the correlation operation (see subsubsection 2.3.1.2). Similarly

$$\frac{\partial L}{\partial h_i} = -\Delta y_r \star x_i + \Delta y_i \star x_r \tag{E.5}$$

Passthrough Gradients: Again with application of the chain rule and the properties of real convolution, we have:

$$\frac{\partial L}{\partial x_r} = \frac{\partial L}{\partial y_r}\frac{\partial y_r}{\partial x_r} + \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial x_r} \tag{E.6}$$

$$= \Delta y_r \star h_r + \Delta y_i \star h_i \tag{E.7}$$

$$\frac{\partial L}{\partial x_i} = -\Delta y_r \star h_i + \Delta y_i \star h_r \tag{E.8}$$

E.2 Regularization

We must be careful with regularizing complex weights. We want to set some of the weights to 0, and let the remaining ones evolve to whatever phase they please. To do this, either use the L-2 norm on the real and imaginary parts independently, or be careful about using the L-1 norm. This is because we really want to be penalising the magnitude of the complex weights, r and:

$$\|r\|_2^2 = \left\|\sqrt{x_r^2 + x_i^2}\right\|_2^2 = \sum_{\mathbf{n}} x_r[\mathbf{n}]^2 + x_i[\mathbf{n}]^2 = \sum_{\mathbf{n}} x_r[\mathbf{n}]^2 + \sum_{\mathbf{n}} x_i[\mathbf{n}]^2 = \|x_r\|_2^2 + \|x_i\|_2^2 \tag{E.9}$$

But for ℓ_1 regularization:

$$\|r\|_1 = \left\|\sqrt{x_r^2 + x_i^2}\right\|_1 = \sum_{\mathbf{n}} \sqrt{x_r[\mathbf{n}]^2 + x_i[\mathbf{n}]^2} \neq \|x_r\|_1 + \|x_i\|_1 \tag{E.10}$$

Note that for ℓ_1 regularization the derivatives have a singularity at the complex origin as:

$$\frac{\partial \ell_1}{\partial x_r[\mathbf{n}]} = \frac{x_r[\mathbf{n}]}{\sqrt{x_r[\mathbf{n}]^2 + x_i[\mathbf{n}]^2}} \tag{E.11}$$

is not defined when $x_r = x_i = 0$. A similar problem was mentioned in section 3.5 where we wanted to pass gradients through the magnitude operation of a ScatterNet. We choose to

use the same solution here and define a smooth ℓ_1 regularization:

$$\|r_s\|_1 = \left\| \sqrt{x_r^2 + x_i^2 + b} \right\|_1 \tag{E.12}$$

where b is a small positive number.

E.3 ReLU applied to the real and imaginary parts independently

If we define a nonlinearity to be $y = \sigma(x)$ where:

$$y = y_r + jy_i = \max(0, x_r) + j\max(0, x_i) \tag{E.13}$$

then the passthrough gradients are:

$$\frac{\partial L}{\partial x_r} = \Delta y_r \mathbb{I}(x_r > 0) \tag{E.14}$$

$$\frac{\partial L}{\partial x_i} = \Delta y_i \mathbb{I}(x_i > 0) \tag{E.15}$$

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:

$$\mathcal{S}(z,t) = \begin{cases} 0 & r < t \\ (r-t)e^{j\theta} & r \geq t \end{cases} \tag{E.16}$$

This can alternatively be written as:

$$\mathcal{S}(z,t) = \frac{\max(r-t,0)}{r} z \tag{E.17}$$

$$= gz \tag{E.18}$$

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 $\frac{\partial g}{\partial x}, \frac{\partial g}{\partial y}, \frac{\partial g}{\partial t}$:

$$\frac{\partial g}{\partial x} = \begin{cases} 0 & r < t \\ \frac{r\frac{\partial r}{\partial x} - (r-t)\frac{\partial r}{\partial x}}{r^2} & r \geq t \end{cases} \tag{E.19}$$

$$= \frac{xt\mathbb{I}(g>0)}{r^3} \tag{E.20}$$

$$\frac{\partial g}{\partial y} = \frac{yt\mathbb{I}(g>0)}{r^3} \tag{E.21}$$

$$\tag{E.22}$$

$$\frac{\partial g}{\partial t} = \frac{-\mathbb{I}(g>0)}{r} \tag{E.23}$$

Then from the definition of $w = u + jv$ we have $u = gx$ and $v = gy$, giving us:

$$\frac{\partial u}{\partial x} = g + \frac{x^2 t\mathbb{I}(g>0)}{r^3} \tag{E.24}$$

$$\frac{\partial u}{\partial y} = \frac{xyt\mathbb{I}(g>0)}{r^3} \tag{E.25}$$

$$\frac{\partial u}{\partial t} = \frac{-x\mathbb{I}(g>0)}{r} \tag{E.26}$$

$$\frac{\partial v}{\partial x} = \frac{xyt\mathbb{I}(g>0)}{r^3} \tag{E.27}$$

$$\frac{\partial v}{\partial y} = g + \frac{y^2 t\mathbb{I}(g>0)}{r^3} \tag{E.28}$$

$$\frac{\partial v}{\partial t} = \frac{-y\mathbb{I}(g>0)}{r} \tag{E.29}$$

Putting it all together, our update and passthrough gradients are:

$$\frac{\partial L}{\partial x} = \frac{xt\mathbb{I}(g>0)}{r^3}(x\Delta u + y\Delta v) + g\Delta u \tag{E.30}$$

$$\frac{\partial L}{\partial y} = \frac{yt\mathbb{I}(g>0)}{r^3}(x\Delta u + y\Delta v) + g\Delta v \tag{E.31}$$

$$\frac{\partial L}{\partial t} = \frac{-\mathbb{I}(g>0)}{r}(x\Delta u + y\Delta v) \tag{E.32}$$

These equations are for point-wise application of soft-thresholding. When 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:

$$\mathcal{B}(z,\gamma,\beta) = \begin{cases} 0 & r < t \\ \frac{\gamma}{\sigma_r}(r-t)e^{j\theta} & r \geq t \end{cases} \tag{E.33}$$

$$= \mathbb{I}(r > t)\frac{\gamma(r-t)}{\sigma r}z \tag{E.34}$$

$$= g'z \tag{E.35}$$

It is clear from our new definition of g' that the equations (E.30) - (E.32) are now scaled by $\frac{\gamma}{\sigma}$. This immediately gives us the passthrough gradients. For the update equations, we must find some additional information:

$$\frac{\partial t}{\partial \beta} = -\frac{\sigma}{\gamma} \tag{E.36}$$

$$\frac{\partial t}{\partial \gamma} = \frac{\sigma\beta}{\gamma^2} \tag{E.37}$$

$$\frac{\partial g'}{\partial \beta} = \frac{-\gamma\mathbb{I}(g' > 0)}{\sigma r}\frac{\partial t}{\partial \beta} = \frac{\mathbb{I}(g' > 0)}{r} \tag{E.38}$$

$$\frac{\partial g'}{\partial \gamma} = \frac{\mathbb{I}(g' > 0)}{\sigma r}\left(r - t - \frac{\partial t}{\partial \gamma}\right) = \frac{\mathbb{I}(g' > 0)}{\sigma r}\left(r - t - \frac{\sigma\beta}{\gamma^2}\right) \tag{E.39}$$

Therefore, combinining these with (E.32) we get:

$$\frac{\partial L}{\partial \beta} = \frac{\mathbb{I}(g' > 0)}{r}(x\Delta u + y\Delta v) \tag{E.40}$$

$$\frac{\partial L}{\partial \gamma} = \frac{\mathbb{I}(g' > 0)}{\sigma r}\left(r - t - \frac{\sigma\beta}{\gamma^2}\right)(x\Delta u + y\Delta v) \tag{E.41}$$

Appendix F

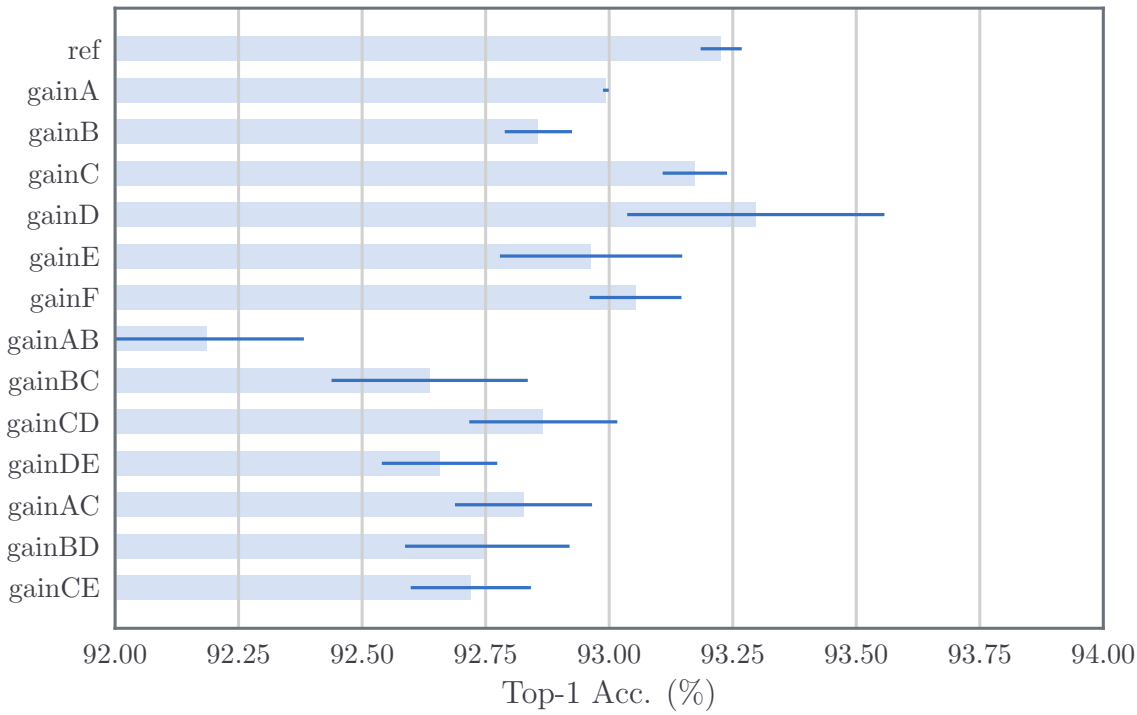
GainLayer Additional Results

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 subsection 5.5.2. 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.

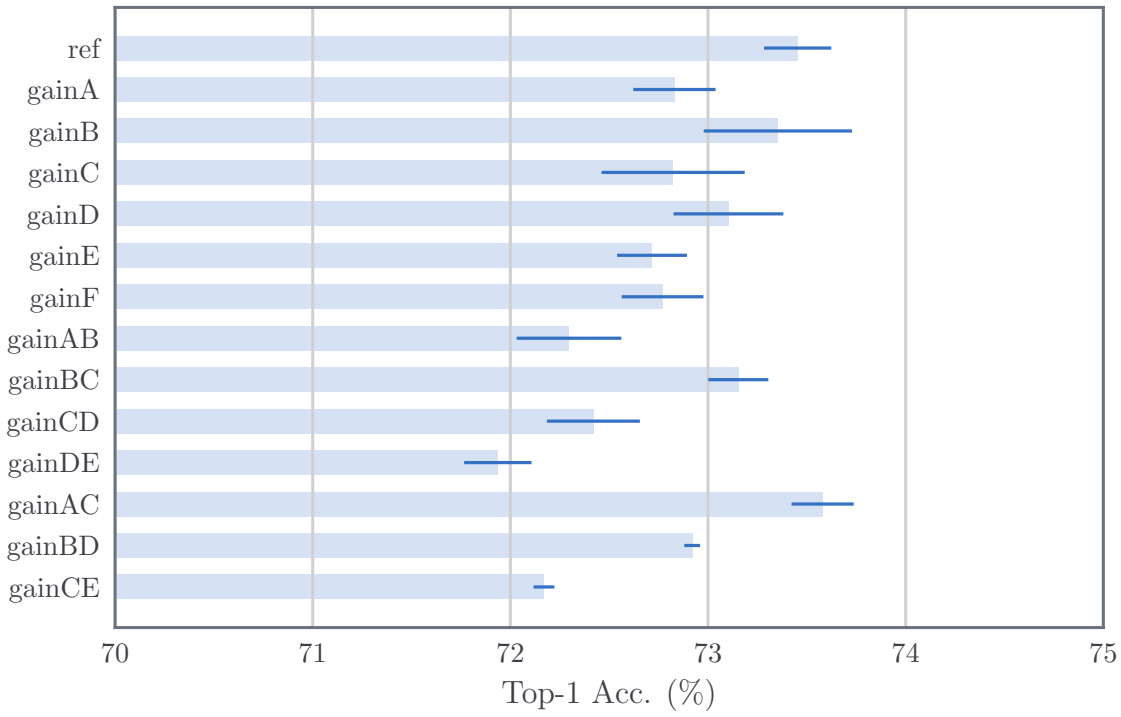
We use the same naming technique as in the previous chapter, calling a network 'gainX' means that the 'convX' layer was replaced with a wavelet gain layer, but otherwise keeping the rest of the architecture the same. Recall that the CIFAR architectures in Table 5.5 have 6 convolutional layers called '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 gainlayer typically degrades performance by a small amount.

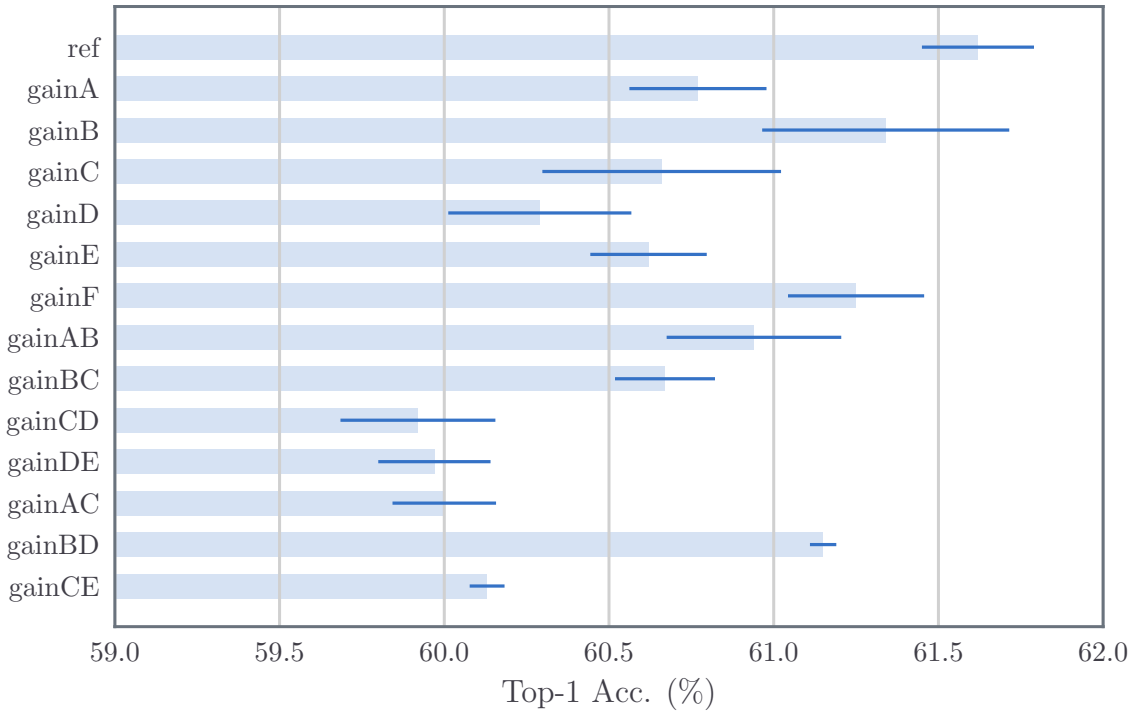


(a) CIFAR-10



(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.



(a) Tiny ImageNet

Figure F.2: **Small kernel ablation results for Tiny ImageNet.**