Uses of Complex Wavelets in Deep Convolutional Neural Networks

Fergal Cotter

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of

December 2018

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
December 2018

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 xi

List of tables xiii

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

2 Image Recognition Review 3
- 2.1 Notation . . . 3
- 2.2 Datasets . . . 4
- 2.3 Sparse Dictionary Learning . . . 6
- 2.4 SIFT Detectors . . . 7
- 2.5 Convolutional Neural Networks . . . 8
  - 2.5.1 Input Layer . . . 9
  - 2.5.2 Convolutional Layers . . . 9
  - 2.5.3 Pooling Layers . . . 10
  - 2.5.4 Activation Functions . . . 10
  - 2.5.5 Fully Connected Layers . . . 11
  - 2.5.6 Loss Function . . . 11
  - 2.5.7 Gradient descent vs Stochastic Gradient Descent vs Mini-Batches . . . 13
  - 2.5.8 Backpropagation and the Chain Rule . . . 14
  - 2.5.9 Normalization . . . 14
  - 2.5.10 Other Layers and Trends . . . 18
- 2.6 Visualization Schemes . . . 19

3 Front Ends 23
- 3.1 Fast GPU Implementation . . . 23

3.2 Something . . . 23
3.3 Relu Properties . . . 23
3.4 Activation Statistics . . . 24

4 Conclusion **25**
4.1 Discussion of Attempts so Far . . . 25
4.1.1 Multiscale Scatternets + SVM . . . 25
4.1.2 Reduced Multiscale Scatternets and CNNs . . . 26
4.1.3 Improved Analysis Methods . . . 27
4.2 Future Work . . . 27
4.2.1 Closing the Gap . . . 27
4.2.2 Moving to a new Dataset . . . 28
4.2.3 Improved Visualizations . . . 28
4.2.4 Going Deeper . . . 28
4.2.5 Resiudal Scattering Layers . . . 28
4.2.6 Exploring the Scope of Scatternets . . . 29
4.2.7 Analysing Newer CNN Layers . . . 29
4.2.8 Revisiting Greedy Layer-Wise Training . . . 29
4.3 Timeline . . . 30

List of figures

2.1 Sample images from ImageNet . . . 5
2.2 Sample images from CIFAR-10 . . . 5
2.3 Principal components learned on natural images . . . 6
2.4 Olshausen and Field sparse dictionary basis functions . . . 7
2.5 SIFT descriptor applied to a keypoint . . . 8
2.6 Standard CNN architecture . . . 9
2.7 Tight vs. overlapping pooling . . . 10
2.8 Differences in non-linearities . . . 11
2.9 Dropout Neural Net Model . . . 18
2.10 The residual unit from ResNet . . . 19
2.11 Deconvolution Network Block Diagram . . . 20
2.12 Unpooling operation in a deconvnet . . . 20
2.13 Deconvolution by slices . . . 21
2.14 Visualization of deconvolved features . . . 21

4.1 Comparison of test accuracy for our network vs a two layer CNN . . . 26
4.2 Possible future work with residual mappings . . . 29

List of tables

4.1 Our plan for the remainder of the PhD 31

Chapter 1

Introduction

1.1 Gradients of Complex Activations

1.1.1 Cauchy-Riemann Equations

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

1.1.2 Complex Magntitude

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

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

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

Then the partial derivatives the real and imaginary inputs are:

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

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

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

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

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

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

Chapter 2

Image Recognition Review

Image recognition tools have undergone a revolution in the past decade. Within a significant sector of the research community, previous state of the art models have been all but abandoned in place of newer deep learning methods. While there are too many to explore in detail, we give a brief history of some of the relevant methods in this chapter, outlining the benefits and drawbacks of each.

2.1 Notation

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

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

- **Convolutional networks**
  The input is $x[\mathbf{u}, d]$, with true label y. Intermediate feature maps are $z[\mathbf{u}, d]$ and the output vector is $\hat{y}[c]$, where d indexes the *depth* of the image/feature map (usually 1 or 3 for x, but arbitrary for z) and c indexes the number of classes. The convolutional kernels are $f[\mathbf{u}, d, n]$ and the fully connected weight matrices are W_{ij}, where the new index, n indexes the number of kernels in a layer. The biases for both these layer types are contained within the vectors b.

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

In cases where we need to index a particular sample in a dataset, we use the standard machine learning notation of adding a (i) superscript to the term, i.e., $x^{(i)}$ refers to the i-th input.

It is difficult to adhere strictly to this notation, particularly when including images from other works. Where the notation deviates in a non-obvious way, we will make it clear.

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

  For wavelets, a slight variation on the standard notation is used. The reason for the modification will become clearer when we introduce the DT$\mathbb{C}$WT, which has two filter bank trees, so needs double the notation. The scaling function is still ϕ, the wavelet function is ψ but the filter banks are h_0, g_0 for low pass analysis, h_1, g_1 for high-pass analysis, $\tilde{h}_0, \tilde{g}_0$ for low pass synthesis and $\tilde{h}_1, \tilde{g}_1$ for high-pass synthesis.

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

- **Other**
  A reconstructed signal is deonted with the hat notation: $\hat{f} \approx f$, and normalized signals are denoted with a tilde, e.g.

$$\tilde{z} = \frac{z - \mu}{\sigma}$$

2.2 Datasets

When considering an image recognition model, it is essential to have a set of images to build your model on, and to compare performance to other models. This naturally involves a choice at the outset of your design. Whatever choice is made, it is important to know the benefits of your dataset, its limitations, and most importantly, to remember that your model should still be valid outside of it.

In current image recognition and classification there are a handful of datasets which are used in the vast majority of literature. In particular, the two most commonly used are ImageNet

Figure 2.1: *Sample images from ImageNet. These images have been cropped to make them square; the raw images vary in size and aspect ratio, but are all at least a few hundred pixels wide and tall.*

Figure 2.2: *Sample images from the 10 classes of CIFAR-10.*

[**?**] (part of the ongoing challenge — ImageNet Large Scale Visual Recognition Competition) and CIFAR-10[**?**]. These are at two ends of our decision spectrum.

ImageNet is a very large dataset, consisting of 1000 classes with a total of 1.4 million images. Each image is typically a few hundred pixels wide and tall, but this does vary image to image. Beyond the size of the dataset and the size of the images, it is also useful for having:

- Objects at varying scales from being only a few pixels in the image, to taking up the entire image.
- Objects with a varying number of instances in the images. I.e., it does not limit an object to only appear once in an image.
- Varying degrees of clutter in the images.
- Varying amounts of texture on the objects of interest.

Some examples are shown in Figure 2.1

On the other hand, CIFAR-10 is a dataset containing only 60000 images from 10 different classes. The images are tiny, all 32×32 pixels[1]. The key benefit of this dataset is its size. Both the images and the training set are small, which means training time is considerably reduced. This can allow us to test many different models and get feedback quickly. Some examples of these images are shown in Figure 2.2. Such small images are appealing to train models on, but when we as humans struggle to see the detail in them, perhaps too much information has been thrown away. On top of this criticism, most of the objects of interest are centred on the image, with little background clutter.

There are a few other datasets that also deserve a mention:

- Pascal-VOC: The state of the art dataset before ImageNet, contains 21738 images in 20 classes. Like ImageNet, the images are typically medium resolution (a few hundred pixels in each direction).

[1] The source images for CIFAR-10 were not necessarily square, but all have been scaled to be *before* including them in the dataset.

Figure 2.3: *Principal components calculated on* 8×8 *image patches from natural images, ordered in increasing variance. Taken from [**?**].*

- Caltech-101 and Caltech-256: Similar image size to ImageNet, but many fewer samples per class (15–30). Caltech is still often used if the goal of the model is training on small amounts of data.

- Microsoft COCO (Common Objects in Context) [**?**]: MS COCO has fewer object categories than ImageNet, and about one quarter of the number of images, but its 328000 images are fully segmented. It is the state of the art dataset currently being used for object detection and localization challenges.

2.3 Sparse Dictionary Learning

The work by **?** in [**?** **?**] show that a system will learn the Gabor-like filters associated with the mammalian V1 cortex, if the constraints of sparsity and minimizing reconstruction error are used.

Introducing sparsity as a constraint may not seem obvious at first. The authors suggest though that it should follow from the intuition that natural images may be described in terms of a small number of structural primitives (e.g. edges, lines)[**?**], which is also supported by the high kurtosis (fourth order statistic) seen in natural images[**?**].

The problem is defined as trying to find ϕ_i[2] that create a dictionary, so that the effective dimensionality of any image can be reduced:

$$\hat{I}(u_1, u_2) = \sum_i a_i \phi_i$$

PCA is a natural starting point to choose to solve this problem, as it attempts to find a set of mutually orthogonal basis functions that capture the directions of maximum variances of the data, and so can maximally reduce the dimensionality for a given reconstruction error, or alternatively, requires the fewest dictionary elements to best represent the data (due to the orthogonality constraint of PCA). The resulting learned dictionary is shown in Figure 2.3. Unfortunately, the shortcomings of this approach are clear. These filters are not at all localized, and further, do not resemble any known cortical receptive fields.

[2]The use of ϕ is deliberate here, to keep it in line with the tradition of it representing an *analysis* function in the wavelet community

Figure 2.4: *Sparse dictionary basis functions found from training on* 16×16 *patches of natural images. Taken from [**?**].*

? instead define a cost function that promotes sparsity as well as the reconstruction error, given by:

$$E = -[\text{preserve information}] - \lambda\,[\text{sparseness of } a_i] \tag{2.3.1}$$

$$= \sum_{u_1,u_2} \left[I(u_1,u_2) - \sum_i a_i \phi_i \right]^2 + \lambda \sum_i S\frac{a_i}{\sigma} \tag{2.3.2}$$

where σ is a scaling constant, and $S(x)$ is a function that promotes sparsity, such as $\exp -x^2$, $\log(1+x^2)$, or $|x|$. Finding a_i and ϕ_i is then done per image patch by holding one constant, and using it to find/update the other. The resulting basis functions are shown in Figure 2.4.

The basis functions learned by this method both resemble the activations found in the V1 cortex, as well as those developed independently by engineers to efficiently code images. This suggests that the V1 cortex may be attempting to compress input into statistically independent and sparse representations.

2.4 SIFT Detectors

SIFT, or Scale Invariant Feature Transforms, were first introduced by **?**] and described in depth in [**?**]. They are currently used extensively for image matching problems. They create a set of features in an image that are designed to be invariant to image scale and rotation, as well as illumination changes.

The algorithm has two main steps to it:

1. Keypoint detection by finding scale-space extrema — 'blobs' with Laplacian of Gaussian (LoG) filters

2. Generate feature vectors for each keypoint

'Scale' in the first step refers to the width of LoG filters (their σ parameter). They use four scales per octave, so the image is filtered successively by a LoG which has $\sigma_{i+1} = 2^{\frac{1}{3}}\sigma_i$. The increasing width of the LoG means we are successively searching coarser and coarser scales. Maxima are then found by comparing outputs to its neighbours in scale and space, and choosing the max. This point is then checked to see if it has sufficient curvature and rejected if not. If

Figure 2.5: *Keypoint descriptor for a* 8×8 *patch of pixels [**?**]. The left shows gradients being calculated per pixel, then weighted by a smooth gaussian filter. The right shows these gradients put into bins of 8 orientations, and then combined in 4 smaller* 4×4 *pixel areas. Unravelling this would give a* $4 \times 8 = 32$ *long feature vector*

this value is above a set threshold, then that coordinate in scale and space (x, y, σ_i^2) is marked as a keypoint, and given to the feature generator.

To account for rotations across images, once a keypoint is found, its dominant orientation is first determined before generating any features. This is done by calculating the direction with the greatest gradient with a resolution of 10°. Further processing is done relative to this orientation. If there were multiple large orientations (this happens roughly 15% of the time), then multiple feature vectors are created.

Once this has been done, a feature is generated by finding gradients in a patch of pixels around the keypoint in the same scale. These gradients are then put into much coarser bins (usually around 45°), in sub-patches around the keypoint. These are then unravelled to make a low-dimensional vector to represent that keypoint. This is shown more clearly in Figure 2.5.

2.5 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) were initially introduced by **?**] in [**?**]. Due to the difficulty of training and initializing them, they failed to be popular for more than two decades. This changed in 2012, when advancements in pre-training with unsupervised networks [**?**], the use of an improved non-linearity — the Rectified Linear Unit, or ReLU, new regularization methods[**?**], and access to more powerful computers in graphics cards, or GPUs, allowed Krizhevsky, Sutskever and Hinton to develop AlexNet[**?**]. This network nearly halved the previous state of the art's error rate. Since then, interest in them has expanded very rapidly, and they have been successfully applied to object detection [**?**] and human pose estimation [**?**]. It would take a considerable amount of effort to document the details of all the current enhancements and tricks many researches are using to squeeze extra accuracy, so for the purposes of this report we restrict ourselves to their generic design, with some time spent describing some of the more promising enhancements.

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

- Yann LeCun's LeNet-5 [**?**], the state of the art design for postal digit recognition on the MNIST dataset.

Figure 2.6: *Standard CNN architecture. Taken from [**?**]*

- Google's GoogLeNet [**?**] achieved 6.67% top-5 error on ILSVRC2014, introducing the new 'inception' architecture, which uses combinations of 1×1, 3×3 and 5×5 convolutions.

- Oxford's VGG [**?**] — 6.8% and runner up in ILSVRC2014. The VGG design is very similar to AlexNet but was roughly twice as deep. More convolutional layers were used, but with smaller support — only 3×3. These were often stacked directly on top of each other without a non-linearity in between, to give the effective support of a 5×5 filter.

- Microsoft Research's ResNet [**?**] achieved 4.5% top-5 error and was the winner of ILSVRC2015. This network we will talk briefly about, as it introduced a very nice novel layer — the residual layer.

Despite the many variations of CNN architectures currently being used, most follow roughly the same recipe (shown in Figure 2.6):

2.5.1 Input Layer

The input is typically scaled, often removing the mean and dividing by the variance. This helps initialize the network in such a way as to operate within the region of activation of the non-linearities, which are necessary to turn linear convolutions into non-linear functions.

2.5.2 Convolutional Layers

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

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

$$z[u_1, u_2, d''] = b[d''] + \sum_{i=-\frac{H'}{2}}^{\frac{H'}{2}-1} \sum_{j=-\frac{W'}{2}}^{\frac{W'}{2}-1} \sum_{k=0}^{D-1} f[i, j, k, d'']x[u_1 - i, u_2 - j, k] \tag{2.5.1}$$

Figure 2.7: ?? *Tight* 2×2 *pooling with stride 2, vs* **??** *overlapping* 3×3 *pooling with stride 2. Overlapping pooling has the possibility of having one large activation copied to two positions in the reduced size feature map, which places more emphasis on the odd columns.*

The stride is another important feature to note about the convolutional layer. Historically, unit stride was used [**?** **?**], but newer models started to use stride 2 [**?**]. Since then, designers vacillate between whether to keep unit stride or not, with some newer designs using both [**?**].

2.5.3 Pooling Layers

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

While initial designs of max pooling would do it in non-overlapping regions, AlexNet used 3×3 pooling with stride 2 in their breakthrough design, quoting that it gave them an increase in accuracy of roughly 0.5% and helped prevent their network from 'overfitting'. More recent networks will typically employ either this or the original 2×2 pooling with stride 2, see Figure 2.7. A review of pooling methods in [**?**] found them both to perform equally well.

2.5.4 Activation Functions

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

1. It is less sensitive to initial conditions as the gradients that backpropagate through it will be large even if x is large. A common observation of sigmoid and tanh non-linearities was that their learning would be slow for quite some time until the neurons came out of saturation, and then their accuracy would increase rapidly before levelling out again at a minimum [**?**]. 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. **?**] 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 2.8: *Differences in non-linearities. Green — the* sigmoid *function, Blue — the* tanh *function, and Red — the* ReLU. *The ReLU solves the problem of small gradients outside of the activation region (around $x = 0$) as well as promoting sparsity.*

2.5.5 Fully Connected Layers

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

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

$$z_i = \sum_j W_{ij} x_j \tag{2.5.2}$$

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

2.5.6 Loss Function

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

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

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

$$p_j = \frac{e^{(f(x,w))(j)}}{\sum_{k=1}^{C} e^{(f(x,w))(k)}} \tag{2.5.3}$$

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

$$(y_i,x,w) = \sum_{j=1}^{C} \mathbb{1}\{y_i = j\} \log p_j \tag{2.5.4}$$

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

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

$$(y,x,w) = \sum_{l=1}^{C} \left[\max(0, 1 - \delta(y_i,l) w^T x_i)\right]^p \tag{2.5.5}$$

Using a hinge loss like this introduces extra parameters, which would typically replace the final fully connected layer. The p parameter in Equation 2.5.5 can be used to choose ℓ^1 Hinge-Loss, or ℓ^2 Squared Hinge-Loss.

Regularization

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

$$\mathcal{L} = \mathcal{L}_{data} + \underbrace{\frac{1}{2}\lambda_{fc} \sum_{i=1}^{L_{fc}} \sum_{j} \sum_{k} (w_i[j,k])^2}_{\text{fully connected loss}} + \underbrace{\frac{1}{2}\lambda_c \sum_{i=1}^{L_c} \sum_{u_1} \sum_{u_2} \sum_{d} \sum_{n} (f_i[u_1,u_2,d,n])^2}_{\text{convolutional loss}} \tag{2.5.6}$$

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

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

Empirical Risk vs Expected Risk

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

the expected risk (R_E) of loss:

$$R_E(f(x,w)) = \int (y, f(x,w)) dP(x,y) \tag{2.5.7}$$

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

$$R_{EMP}(f(x,w)) = \frac{1}{N} \sum_{i=1}^{N} (y^{(i)}, f(x^{(i)}, w)) \tag{2.5.8}$$

2.5.7 Gradient descent vs Stochastic Gradient Descent vs Mini-Batches

We can minimize Equation 2.5.8 with *gradient descent* [**?**]. Updates can be made on a generic network parameter w with:

$$w_{t+1} = w_t - \eta E_n w \tag{2.5.9}$$

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

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

$$w_{t+1} = w_t - \eta w \tag{2.5.10}$$

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

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

$$w_{t+1} = w_t - \eta \sum_{n=1}^{N_B} w \tag{2.5.11}$$

2.5.8 Backpropagation and the Chain Rule

With a deep network like most CNNs, calculating Lw may not seem particularly obvious if w is a weight in one of the lower layers. Say we have a deep network, with L layers. We need to define a rule for updating the weights in all L layers of the network, however, only the weights w_L are connected to the loss function, . We assume for whatever function the last layer is that we can write down the derivative of the output with respect to the weights $z_L w_L$. We can then write down the weight-update gradient, w_L with application of the chain rule:

$$w_L = z_L z_L w_L + \underbrace{\lambda w}_{\text{from the reg. loss}} \tag{2.5.12}$$

z_L can be done simply from the equation of the loss function used. Typically this is parameter-less.

Since all of the layers in a CNN are well-defined and differentiable[3] we assume that we can also write down what $z_L z_{L-1}$ is. Repeating this process for the next layer down, we have:

$$w_{L-1} = z_L z_L z_{L-1} z_{L-1} w_{L-1} \tag{2.5.13}$$

We can generalize this easily like so:

$$w_l = z_L \underbrace{\prod_{i=L}^{l+1} z_i z_{i-1}}_{\text{product to } l\text{'s output}} z_l w_l \tag{2.5.14}$$

2.5.9 Normalization

A common practice amongst data scientists is to normalize input data. **?**] state that this is an important first step in training learning networks. The paradigm is typically to remove the mean of the data, (usually) divide by the standard deviation of the data and, if possible, decorrelate or whiten the data with Principle Components Analysis.

Normalization can be dangerous as signals, or dimensions of the signal with low signal to noise ratio, can cause lots of noise to be introduced into the system if normalized naively. However, normalizing the input is still useful if done appropriately.

In multilayer networks such as a CNN, the interface between layers is often thought of as being an input to a smaller network. And so, the data should be normalized once again. Aside from simply subtracting the mean and dividing by the standard deviation, two other methods

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

have become popular recently. The first — Local Response Normalization — was used in AlexNet, and is quite unlike standard normalization. The second — Batch Normalization [**?**] — was introduced more recently, and is a modification of standard normalization to include some learnable parameters.

Standard Normalization

Data in a convolutional network are passed layer to layer in arrays of 'slices'. If the input is RGB, then the input slice would be of size $H \times W \times 3$. In deeper layers, the number of slices typically grows to much larger numbers (it will be equal to the number of filters in the layer below).

For a generic array of feature slices, $z(u_1, u_2, d)$, standard normalization is done across the u_1 and u_2 coordinates. I.e., the mean is:

$$\mu_z(d) = \frac{1}{HW} \sum_{u_1=0}^{H-1} \sum_{u_2=0}^{W-1} z(u_1, u_2, d) \tag{2.5.15}$$

and the variance is:

$$\sigma_z(d)^2 = \frac{1}{HW} \sum_{u_1=0}^{H-1} \sum_{u_2=0}^{W-1} (z(u_1, u_2, d) - \mu_z(d))^2 \tag{2.5.16}$$

Each of which are vectors of size D. These are only point estimates on the distribution of statistics of the feature slice, so an estimate of the true $E[z]$ and $Var[z]$ is taken by averaging the means and variances over the entire *training* dataset. The normalized feature vectors are then:

$$\tilde{z}(u_1, u_2, d) = \frac{z - E[z]}{\sqrt{Var[z]}} \tag{2.5.17}$$

Local Response Normalization

This normalization scheme used in AlexNet was first introduced in [**?**]. It is a form of brightness normalization, as it does not involve subtracting the mean. Further, instead of dividing by the variance of a single feature map[4], they divide by a scaled version of the energy of the same pixel (**u** coordinate) in neighbouring feature maps.

[4] or the energy of the feature map, as we have already stated the mean is not subtracted

I.e., consider the activation $z(u'_1, u'_2, d')$. We compute the energy of the 'local' feature maps for this pixel:

$$\gamma = \sum_{j=d'-n/2}^{d'+n/2} z^2(u'_1, u'_2, j) \tag{2.5.18}$$

Where n is on the order of 10 slices. Then, the normalized pixel value is obtained by:

$$\tilde{z}(u'_1, u'_2, d') = \frac{z(u'_1, u'_2, d')}{(k + \alpha\gamma)^\beta} \tag{2.5.19}$$

Where $k = 2$, $\alpha = 10^{-4}$, and $\beta = 0.75$ in the case of AlexNet. While this formula may seem odd, it is just a normalization scheme that promotes 'competition' or high frequencies across feature maps. An example makes this clear.

Example 2.5.1. *Suppose the activation values are all quite large and uniform across feature maps, say they are all 100.*

In this case, the γ term will be quite large — if $n = 10$, then $\gamma = 10^5$. Then the $\alpha\gamma$ term in the denominator of Equation 2.5.19 will dominate the k term, and the output will be reduced significantly (down to 16.5).

If however, the neighbouring activations were all 0 around a central 100, then the γ term will be smaller, and the output is not reduced as much (down to 43.9).

This technique supposedly draws inspiration from the human visual system, where neuron outputs are reduced if there is a lot of activity in a neighbourhood around them.

Batch Normalization

Batch normalization proposed only very recently in [**?**] is a conceptually simpler technique. Despite that, it has become quite popular and has been found to be very useful. At its core, it is doing what standard normalization is doing, but also introduces two learnable parameters — scale (γ) and offset (β). Equation 2.5.17 becomes:

$$\tilde{z}(u_1, u_2, d) = \gamma \frac{z - E[z]}{\sqrt{Var[z]}} + \beta \tag{2.5.20}$$

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

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

From Equation 2.5.20, let us briefly use the hat notation to represent the standard normalized input: $\hat{z} = (z - E[z])/\sqrt{Var[z]}$, then:

$$\begin{aligned} \tilde{z}^{(i)} &= \gamma \hat{z}^{(i)} + \beta \\ \frac{\partial \mathcal{L}}{\partial \gamma} &= \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \mathcal{L}^{(i)}}{\partial \tilde{z}^{(i)}} \cdot \hat{z}^{(i)} \qquad (2.5.21) \\ \frac{\partial \mathcal{L}}{\partial \beta} &= \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \mathcal{L}^{(i)}}{\partial \tilde{z}^{(i)}} \qquad (2.5.22) \end{aligned}$$

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

$$\begin{aligned} z &= g(BN(Wu + b)) \\ &= g(BN(Wu)) \qquad (2.5.23) \end{aligned}$$

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

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

$$BN(Wu) = BN((aW)u) \qquad (2.5.24)$$

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

$$\begin{aligned} \frac{\partial BN((aW)u)}{\partial u} &= \frac{\partial BN(Wu)}{\partial u} \\ \frac{\partial BN((aW)u)}{\partial (aW)} &= \frac{1}{a} \cdot \frac{\partial BN(Wu)}{\partial W} \qquad (2.5.25) \end{aligned}$$

Figure 2.9: *Dropout Neural Net Model. Left — a standard neural net with 2 hidden layers. Right — an example of a thinned net produced by applying dropout to the left network. Crossed units have been dropped. Taken from [**?**]*

2.5.10 Other Layers and Trends

There are far too many paradigm shifts in the field of CNNs to cover all of them exhaustively, but we describe some other layers/network features that could influence our future work are.

Batch Normalization instead of Dropout

Dropout is a particularly harsh regularization scheme that randomly turns off neurons in a neural network or deep belief network, and all of its ingoing and outgoing connections [**?** **?**]. Each node is retained with a fixed probability p (typically around 0.5 for hidden units, and closer to 1 for input units), and the 'thinned' netowrk is sampled, giving a sample output — see Figure 2.9.

An estimate of the output distribution can be taken by repetitively sampling thinned networks, but a simpler method is typically used at test time — run the entire feedforward network, but multiply the output of each node by its dropout probability p.

This was used with great success in the AlexNet design, but recent state of the art models forgo it in favour of Batch Normalization [**?** **?**]. The paper that introduced Batch Normalization[**?**] states that:

> Removing Dropout from Modified BN-Inception (their model) speeds up training, without increasing overfitting.

No More Max Pooling

Max pooling has been falling out of favour recently, in favour of decimation by a factor of two in each direction [**?**] (this can be achieved efficiently by only performing the convolution on every second pixel, so is often called 'stride 2 convolution'). This is quite interesting, as is something that happens in Scatternets.

Improved Initialization

Initialization techniques are steadily being improved. Typically, the weights for a convolutional layer are initialized from random Gaussian noise, with a small variance, usually around 0.01. This arbitrary size becomes an issue with deeper designs, as the ability for a network to train is

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

very dependent on its initial conditions. Weights that are too small can often lead to little or slow learning, weights that are too big can lead to divergence.

?] defined the set scale weights should be defined at as a function of the fan-in n_{in} (number of neurons that feed the current neuron) and fan-out n_{out} (number of neurons the current neuron is connected to) of the layers:

$$var(w) = \frac{2}{n_{in} + n_{out}} \tag{2.5.26}$$

?] and **?**] independently showed that forcing orthonormality on weights works much better than this Guassian initialization. I.e., if an output $z = Wx$, then we want $W^T W = I$. Ensuring this condition can allow the network to train in a similar way to the greedy, layer-wise pre-training methods described in [**?**].

Residual Layers

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

To promote the chance of learning the identity mapping, they define a residual unit, shown in Figure 2.10. If a desired mapping is denoted $\mathcal{H}(x)$, instead of trying to learn this, they instead learn $\mathcal{F}(x) = \mathcal{H}(x) - x$.

2.6 Visualization Schemes

Since CNNs have become so good at a variety of tasks, it is more important now than ever to gain more insight into *how* and *what* they are learning. As described in the motivation for the project (**??**), back projecting from the result to the input space can help improve a network's interpretability, and can even help improve its ability to learn.

? first attempted to use 'deconvolution' to improve their learning [**?**], then later for purely visualization purposes [**?**]. Their method involves mapping activations at different layers of the network back to the pixel space

Figure 2.11: *A block diagram view of the model. 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 [**?**].*

Figure 2.12: *Unpooling operation in a deconvnet. On the forward pass, the locations of the maximum values are stored (the centre map with grey squares). This is then used on the backwards pass to put values at the correct location. Figure taken from [**?**].*

Figure 2.11 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. **?** 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 Figure 2.12.

Figure 2.13 gives a more detailed view of how the deconvolution works for convolutional layers.

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

Figure 2.13: *Deconvolution by slices. Visualization of 2 layers of the model showing how to invert a convolutional layer. At layer 2 of the model, there are L feature maps $z_{l,2}$ (top green). Each of these feature maps was made from a different filter. The L different filters are shown below the activations in red — $f^c_{l,2}$. The c superscript on these filters indexes the channel. E.g. a convolutional filter could be $5 \times 5 \times 64$, where the first two indices the spatial support of the filter, and the third index — the 64 — is the fully connected depth aspect of the filter, the c in this case. Each filter is laid out slice by slice. For simplicity, only two slices are shown in the figure. The 2D transpose of this filter is taken and convolved with the feature map $z_{l,2}$. The result of this is $L \times C$ images. For each $c \in \{0 \dots C-1\}$, the c'th output from all L feature maps are summed together to make a pooled map $p_{c,1}$. These C pooled maps are then expanded to make the C feature maps at the layer below (indexed by k in the figure) — $z_{k,2}$. This process then repeats until we return to the input space. Not shown on this diagram are non-linear layers, but these are simple, point-wise operations. It would be trivial to insert them conceptually, by putting one more step, going from an intermediate feature map $z'_{k,1}$ to $z_{k,1}$. This figure was taken from [**?**].*

Figure 2.14: *Visualization of some deconvolved features. 9 coordinates are chosen in the feature map for layer 1 z^1, 16 for the second layer feature map z^2 and 16 for the third layer feature map. The entire dataset is run, and 9 images that made the largest activation at this point are noted. Deconvolution is then done on the feature maps from these 9 images, and the results are shown next to the actual images. Deeper layers are picking up more complex structures. Taken from [**?**].*

Chapter 3

Front Ends

This chapter is about having a wavelet transform/scatternet front end to a deep learning system.

Previous work including Oyallon and Singh have explored ways in which ScatterNets can be improved on for image analysis tasks. This chapter explores some alternate methods we have.

3.1 Fast GPU Implementation

3.2 Something

Where to modify the scatternet? We could look at the difference in training cost when we drop the second order coefficients.

Perhaps just doing the wavelet transform and complex magnitude is enough?

Nick wanted me to move the magnitude before mixing the channels. The argument being that taking the magnitude means the output becomes shift invariant. I.e., we only need to see that there is energy in a wavelet band, and then we can accommodate shifts.

The question is though, how much do we need the phase? Also can we impose some priors on the distribution of subbands for CNN activations? We certainly can for the image statistics.

3.3 Relu Properties

What is the effect of the ReLU on the activations? Show that it sparsifies things.

3.4 Activation Statistics

I want to take the DTCWT of activations throughout a neural network and look at the pdf over the subband energy.

Chapter 4

Conclusion

This chapter aims to logically tie together the results from the previous chapter, outlining what has been promising and what has not been, offering explanations as to why we think that is the case.

4.1 Discussion of Attempts so Far

4.1.1 Multiscale Scatternets + SVM

Much of the beginning of the project was spent understanding and analysing the Scatternet design proposed by Mallat and the proposed DT$\mathbb{C}$WT based multiscale Scatternet. It was found that the DT$\mathbb{C}$WT based Scatternets are much faster than the Mallat based Scatternets (3–4 times faster for tiny images, and 10–15 times faster for medium resolution images). Work by a colleague — **?**] showed that this came at the cost of no appreciable loss of performance.

Attempts to use the four layer multiscale Scatternet with linear SVMs on CIFAR-10 data showed that little performance was gained from lots of extra coefficients from the fourth layer. For example, the H4 block could easily be removed with no appreciable difference in performance, but saving one third of the coefficients. Nonetheless, it was difficult to improve this accuracy past 70% without making use of further feature extraction methods (orthogonal least squares was used in both [**?**] and [**?**]).

This shows the multiscale Scatternet alone is not achieving the same quality of feature extraction as a few convolutional layers in a CNN, as simply connecting a classifier on the output achieved a lower test accuracy.

We believe that this was because the initial Scatternet design lacks the fully connectedness in the depth dimension that comes with the CNN. This would allow a network to learn how to add linear combinations of different output activations from the layer below. At the simplest

Figure 4.1: *Comparison of test accuracy for our network vs a two layer CNN. The network used for the Scatternet+CNN design was the reduced Scatternet with data augmentation (see* **??***). The standard CNN reference architecture was used (see* **??***). Our network trains much faster but plateaus earlier than the CNN network. We must find out how to bring the asymptotes of these two curves closer together.*

case, at layer two, this would allow the CNN to have a filter that could combine two oriented wavelets, perhaps at a slight offset from each other, allowing it to detect a corner.

4.1.2 Reduced Multiscale Scatternets and CNNs

Focus switched in the latter half of the year to bringing back in one layer of CNNs with a scaled down multiscale Scatternet, followed by a neural network classifier. Our initial results were quite poor compared to the pure CNN solution — a 10% reduction in accuracy from 87% to 77%. This was still an improvement on the $\sim 69\%$ we were getting with the pure SVM method which, while comforting, does little to meet our project goals. However, subsequent refinement of our methods narrowed this gap significantly and started to show some promising results.

As a side note, poor initial results do not necessarily indicate failings of the Scatternet. There are a huge variety of hyperparameters to choose when designing the convolutional side of our hybrid network, and on top of that, a selection of different extra layers/schemes that can be used to 'normalize' the statistics of the data, all to promote learning. Replacing one layer of an optimized CNN with a Scatternet and getting a reduction in accuracy could be due to the later layers of the CNN now no longer being optimized. In fact, it was very surprising we had such a large reduction in accuracy. The ability for us to eventually narrow the gap only reinforces the necessity of the project, to better understand how we should train CNNs.

Quicker Training Time

Our Scatternet and shallow CNN was able to train much faster (in fewer epochs) than a pure CNN method. We regard this as an early sign that we are on the right track. Unfortunately, our network does plateau earlier than a CNN, so we end up performing worse in the long run — see Figure 4.1.

Data Augmentation

The other interesting thing we have found so far is our model is far less dependent on data augmentation methods. In particular, taking out the random cropping and shifting from the pure

CNN method resulted in a drastic reduction of performance, while it only marginally affected the Scatternet plus CNN performance.

4.1.3 Improved Analysis Methods

One benefit of having set filters in the Scatternet design versus purely learned ones can be seen in the difference between the axes labels of **??** and **??**. We were able to label the slices in the Scatternet design, see that some of the slices were not being used much in the learned CNN, and design experiments that removed these slices, which improved training time and accuracy. This kind of targetted design would be hugely beneficial to the field of CNNs.

Unfortunately, we have not yet fine tuned our visualizations. The figures in **??** are only toy examples at this stage. They show that it is possible to get images like those from the work of **?**] from a subset of Scatternet coefficients.

4.2 Future Work

4.2.1 Closing the Gap

The first layer of a CNN *can* be replaced with a first order Scatternet, with currently a small loss of performance. Getting this result is promising, but it needs further work. If we were to reduce this gap, we would have developed a network that could train *faster* than current methods. While this is not our primary research goal, it will nonetheless be a useful addition to the field.

We need to spend some time researching the cause for the current performance gap. We must revisit the first layers of the CNN and draw inspiration from here. What is missing?

The basis functions for the first order Scatternet are considerably fewer in number than used by the CNN. We are currently using 24 compared to 64 in the pure CNN network. While relying on fewer operations is beneficial, this may be an easy way to increase our final accuracy. Two possibilities immediately come to mind — we could add in more colour channels as currently, the assumption is that only low frequency is necessary for colour, but there certainly are some mid-low level frequency colour filters in both the AlexNet first layer **??**, and the CIFAR-10 first layer **??**. Also, we could increase the number of scales we have per octave from one to two, to get a better coverage of the frequency space.

Unfortunately, both of these additions would mean we would lose perfect reconstruction, but that does not mean we cannot still invert the representations, it just becomes slightly more difficult.

4.2.2 Moving to a new Dataset

We believe that we are not playing wavelets to their strengths by using such small images as the ones found in CIFAR-10. Further, they are very poor representations of the images found in the real world, and it would be lackluster to restrict ourselves to them. Fortunately, there are no shortages in choosing an alternative dataset, with PASCAL-VOC a good candidate due to its similar number of training samples.

This is best done sooner rather than later. It will take time to get used to working on a new dataset, and then some more to fine tune our design. As a first step, we will look at a design like AlexNet, replacing the first layer with a Scatternet, as we have done for CIFAR-10.

4.2.3 Improved Visualizations

We believe that visualizations are key to unlocking the secrets of CNNs. Our attempts so far at creating visualizations have taught us a great deal, but further work needs to be done. In particular, a recent paper by **?**] has given a detailed comparison and analysis of how visualizations have developed since the work of **?**]. We must spend time to study these works and use them to improve our visualizations method with Scatternets.

4.2.4 Going Deeper

To really indicate a gained insight, we need to be able to replace more than one layer of a CNN.

This does not mean that Scatternets are a poor feature extractor, they have already been proven to be state of the art in texture datasets [**?** **?**], but they do not help us gain insight into CNNs.

As we now have a framework that successfully mimics one layer of a CNN. We believe that the visualization tools that we are currently working on will be the key to unlocking more information about the deeper layers of CNNs, which will in turn allow us to design enhancements to the Scatternets.

4.2.5 Resiudal Scattering Layers

This somewhat continues on from § 4.2.1, but also could lead to a complete rethink of the Scatternet design. We want to investigate to achieve this is inspired from the recent state of the art ResNets — mentioned in § 2.5.10. Residual networks work on the basis that if the ideal mapping $\mathcal{H}(x)$ is close to the identity mapping, then it will be easier to learn $\mathcal{F}(x) := \mathcal{H}(x) - x$ than to learn $\mathcal{H}(x)$ directly. To do this, they construct a residual layer, shown again in **??**.

Figure 4.2: *Possible future work with residual mappings.* **??** *shows the residual layer as used by [***?** *]. These networks started with the assumption that the the identity mapping $f(\cdot) = I$ was a good reference function, and the network should learn deviations from that.* **??** *shows the proposed architecture. Assuming the first order Scatternet (wavelet transform + modulus + averaging) is a good base mapping, improve on it by allowing some deviation from it as a residual.*

Assuming we have developed a near-optimal mapping with a first order Scatternet, S_J^1, then a sensible thing to do is to improve performance by adding in extra flexibility and the ability to better fit the nuances of the training set. We propose to do that with the residual layer shown in **??**.

4.2.6 Exploring the Scope of Scatternets

We believe that applications a designed architecture like the Scatternet is well suited for are unsupervised and low data tasks. More specifically, any method that is difficult to propagate back gradients (due to the lack of labelled data, or to only a few training points). We would like to spend some time analysing what progress can be made in these fields with the knowledge we have.

4.2.7 Analysing Newer CNN Layers

For the moment we have satisfied ourselves with examining architectures like Cuda Convnet and AlexNet, which while high end, are not state of the art. There have been many new layers and designs added in since then, which have taken modern CNNs even further. We have already covered one example, the Residual Unit. Another example is the 'Inception Unit' [**?** **?**], which combines 5×5, 3×3, and 1×1 convolutions of different.

Also, there is a new trend of using stride-2 convolution with downsampling is something we would like to investigate more, as it could lead us to clues about how to use subsampling and scale in a CNN, ideas that fit very naturally in a wavelet based network.

4.2.8 Revisiting Greedy Layer-Wise Training

The work on using unsupervised Deep Belief Networks to progressively train the hidden units of a deep neural network by **?**] has been largely abandoned recently. This is sad to see, as it looked like a promising, well-defined way to train deep networks. We would like to revisit this in the context of a Scatternet, and attempt to use this paradigm.

4.3 Timeline

We set our timeline for the next two years in Table 4.1. The future work we have discussed so far roughly falls into two categories: the first four (§§ 4.2.1–4.2.4) fall on what we can consider the main line of research into designing a learned CNN style network. The second four (§§ 4.2.5–4.2.8) is more speculative research, that we hope will allow us to explore and gain inspiration from similar problems.

Ordering these tasks chronologically would not be very well thought out. For example, successfully achieving § 4.2.4 is the main goal of the PhD after all, so it would be naive to say we believe we can achieve it by the end of the second year. Instead, we believe that the main line of research will take us a long time, and during that time, we must explore the more speculative research. Also, a few of the tasks will need to be revisited at certain points — just as importantly as not being too ambitious, we should not be too lazy. I.e., we should not leave our first attempt at building deeper layers of a CNN until our third year. The same goes for improving our visualizations.

Table 4.1: *Our plan for the remainder of the PhD. The first column lists the months we want to do the work in, and the second column describes the task, and the questions we want to answer in this time.*

Second Year		
Date	Task	Second Task
Oct–Nov	Closing the Gap § 4.2.1	
Dec–Jan	Developing ResNet Layers § 4.2.5	
Feb–Mar	Improved Visualizations Research § 4.2.3	
Apr–May	Moving to a new dataset (preparation) § 4.2.2	
Jun–Jul	Moving to a new dataset (testing) § 4.2.2	Going Deeper § 4.2.4
Aug–Sep	Going Deeper § 4.2.4	Recap on Second Year
Third Year		
Oct–Nov	Exploring the Scope of Scatternets § 4.2.6	
Dec–Jan	Revisiting Greedy Layer-Wise Training § 4.2.8	Analysing Newer CNN Layers § 4.2.7
Feb–Mar	Revisiting Greedy Layer-Wise Training§ 4.2.8	Analysing Newer CNN Layers § 4.2.7
Apr–May	Improved Visualizations Research § 4.2.3	
Jun–Jul	Going Deeper § 4.2.4	
Aug–Sep	Going Deeper § 4.2.4	Recap on Third Year
Fourth Year		
Oct-Dec	Writing up	