Chapter 1

A Learnable ScatterNet: Locally Invariant Convolutional Layers

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

Previous attempts at tying ScatterNets together with CNNs in hybrid networks [1]–[3] have tended to keep the two parts separate, with the ScatterNet forming a fixed front end and the CNN forming a learned backend. We instead look at adding learning between scattering orders, as well as adding learned layers before the ScatterNet.

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

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

In **??** we briefly review convolutional layers and scattering layers before introducing our learnable scattering layers in **??**. In section 1.4 we describe how we implement our proposed layer, and present some experiments we have run in section 1.5 and then draw conclusions about how these new ideas might improve neural networks in the future.

1.1 Related Work

There has been several similar works that look into desigining 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 [4] by Szegedy *et*

al. does this by filtering with different kernel sizes in the first stage, and then combining with a 1×1 convolution in the second stage. Ioannou *et al.* also do something similar by making a first stage with horizontal and vertical filters, and then combining in the second stage again with a 1×1 convolution[5]. But perhaps the most similar works are those that use a first stage with fixed filters, combining them in a learned way in the second stage. Of particular note are:

- "Local Binary Convolutional Neural Networks" [6]. This paper builds a first stage with a small 3×3 kernel filled with zeros, and randomly insert plus and minus ones into it. 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" [7]. 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 [6] 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.

1.2 Recap of Useful Terms

1.2.1 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:

$$y^{(l+1)}(f, \mathbf{u}) = \sum_{c=0}^{C_l - 1} x^{(l)}(c, \mathbf{u}) * h_f^{(l)}(c, \mathbf{u}) \tag{1.2.1}$$

$$x^{(l+1)}(f, \mathbf{u}) = \sigma\left(y^{(l+1)}(f, \mathbf{u})\right) \tag{1.2.2}$$

where $h_f^{(l)}(c, \mathbf{u})$ is the fth filter of the lth layer (i.e. $f \in \{0, \dots, C_{l+1} - 1\}$) 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 $\left\{x^{(l+1)}(f, \mathbf{u})\right\}_{f \in \{0, \dots, C_{l+1} - 1\}, \mathbf{u} \in \mathbb{R}^2}$

1.2.2 Wavelet Transforms

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

$$\psi_{j,\theta}(\mathbf{u}) = 2^{-j}\psi\left(2^{-j}R_{-\theta}\mathbf{u}\right) \tag{1.2.3}$$

where R is the rotation matrix, $1 \leq j \leq J$ indexes the scale, and $1 \leq k \leq K$ indexes θ to give K angles between 0 and π. We copy notation from [8] 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:

$$Wx(c,\mathbf{u}) = \{x(c,\mathbf{u}) * \phi_J(\mathbf{u}), x(c,\mathbf{u}) * \psi_\lambda(\mathbf{u})\}_{\lambda\in\Lambda} \tag{1.2.4}$$

1.2.3 Scattering Transforms

Taking the modulus of the wavelet 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:

$$\tilde{W}x(c,\mathbf{u}) = \{x(c,\mathbf{u}) * \phi_J(\mathbf{u}), |x(c,\mathbf{u}) * \psi_\lambda(\mathbf{u})|\}_{\lambda\in\Lambda} \tag{1.2.5}$$

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:

$$\begin{aligned} U[p]x &= U[\lambda_m]\cdots U[\lambda_2]U[\lambda_1]x & (1.2.6)\\ &= ||\cdots|x * \psi_{\lambda_1}|*\psi_{\lambda_2}|\cdots * \psi_{\lambda_m}| & (1.2.7)\end{aligned}$$

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

$$S[p]x(\mathbf{u}) = U[p]x * \phi_J(\mathbf{u}) \tag{1.2.8}$$

If we define $p + \lambda = (\lambda_1, \ldots, \lambda_m, \lambda)$ then we can combine Equation 1.2.5 and Equation 1.2.6 to give:

$$\tilde{W}U[p]x = \{S[p]x, U[p+\lambda]x\}_\lambda \tag{1.2.9}$$

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 [9], 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:

$$\|S\mathcal{L}_\tau x - Sx\| \leq K_L P H(\tau)\,\|x\| \tag{1.2.10}$$

where $P = \text{length}(p)$ is the scattering order, and $H(\tau)$ is a function of the size of the displacement, derivative and Hessian of τ, $H\tau$ [9]:

$$H(\tau) = 2^{-J}\,\|\tau\|_\infty + \|\nabla\tau\|_\infty \max\left(\log\frac{\|\Delta\tau\|_\infty}{\|\nabla\tau\|_\infty}, 1\right) + \|H\tau\|_\infty \tag{1.2.11}$$

1.3 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 a wavelet modulus propagator Equation 1.2.5, and let the input to it be $x^{(l)}$. Our proposed output is then:

$$\begin{aligned} y^{(l+1)}(f,\mathbf{u}) &= \sum_{\gamma}\sum_{c=0}^{C-1} |x^{(l)}(c,\mathbf{u}) * \psi_\gamma(\mathbf{u})|\; a_{f,\lambda}(c) \\ &+ \left(\sum_{c=0}^{C-1} x^{(l)}(c,\mathbf{u}) * \phi_J(\mathbf{u})\right) b_f(c) \end{aligned} \tag{1.3.1}$$

Returning to Equation 1.2.5, we define a new index variable γ such that:

$$\tilde{W}[\gamma]x(c,\mathbf{u}) = \begin{cases} x^{(l)}(c,\mathbf{u}) * \phi_J(\mathbf{u}) & \text{if } \gamma = 1 \\ |x^{(l)}(c,\mathbf{u}) * \psi_\gamma(\mathbf{u})| & \text{if } \gamma \in \{2,\ldots,JK+1\} \end{cases} \tag{1.3.2}$$

We do the same for the weights a, b by defining $\tilde{a}_f = \{b_f, a_{f,\lambda}\}_\gamma$ and $\tilde{a}_f[\gamma] = b_f$ if $\gamma = 1$ and $\tilde{a}_f[\gamma] = a_{f,\lambda}$ if $\gamma \in \{2,\ldots JK+1\}$.

We further define $q = (c,\gamma) \in Q$ to combine the channel and index terms. Then we can index both the weights and the wavelet modulus operator by:

$$\tilde{W}x[q] = \tilde{W}[\gamma]x(c,\mathbf{u}) \tag{1.3.3}$$

$$\tilde{a}_f[q] = \tilde{a}_f[\gamma](c) \tag{1.3.4}$$

This simplifies Equation 1.3.1 to be:

$$z^{(l+1)}(q,\mathbf{u}) = \tilde{W}x^{(l)}[q] = \tilde{W}[\gamma]x^{(l)}(c,\mathbf{u}) \tag{1.3.5}$$

$$y^{(l+1)}(f,\mathbf{u}) = \sum_{q\in Q} z^{(l+1)}(q,\mathbf{u})\tilde{a}_f(q) \tag{1.3.6}$$

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

$$Y^{(l+1)}(\mathbf{u}) = AZ^{(l+1)}(\mathbf{u}) \tag{1.3.7}$$

This is very similar to the standard convolutional layer from Equation 1.2.1, except we have replaced the previous layer's x with intermediate coefficients z (with $|Q| = (JK+1)C$ channels), and the convolutions of Equation 1.2.1 have been replaced by a matrix multiply (which can also be seen as a 1×1 convolutional layer). We can then apply Equation 1.2.2 to Equation 1.3.6 to get the next layer's output (or equivalently, the next order scattering coefficients).

Figure 1.1 shows a block diagram for Equations 1.3.2 – 1.3.6.

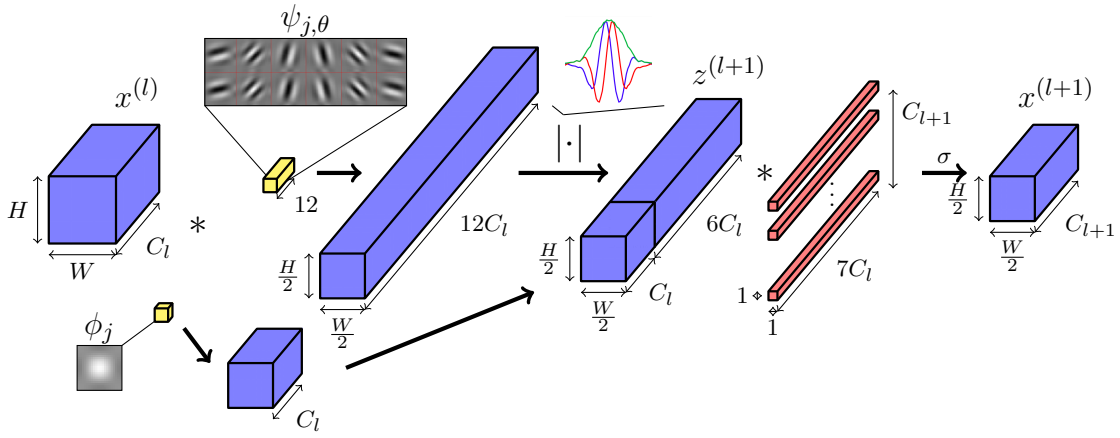

Figure 1.1: Block Diagram of Proposed Invariant Layer for $J = 1$. Activations are shaded blue, fixed parameters yellow and learned parameters red. Input $x^{(l)} \in \mathbb{R}^{C_l \times H \times W}$ is filtered by real and imaginary oriented wavelets and a lowpass filter and is downsampled. The channel 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.

1.3.1 Properties

1.3.1.1 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 [1], [8]. 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 Equation 1.2.6, 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.

1.3.1.2 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.

1.3.1.3 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 [7] find constraints on the mixing weights to make them non-expansive (i.e. Lipschitz constant 1). Further, the ReLU is non-expansive meaning the combination of the two is also non-expansive, so $\|V\mathcal{L}_\tau x - Vx\| \leq \|S\mathcal{L}_\tau x - Sx\|$, and **??** holds.

1.4 Implementation Details

Again, we use the DT$\mathbb{C}$WT [10] for our wavelet filters $\psi_{j,\theta}$ due to their fast implementation with separable convolutions which we will discuss more in subsection 1.4.2. 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^j in each direction for each scale j. 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 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 1.1 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:

$$\big(((x * \phi_1) \downarrow 2) * \psi_{1,\theta}\big) \downarrow 2 = \big(x * \psi_{2,\theta}\big) \downarrow 4 \tag{1.4.1}$$

1.4.1 Memory Cost

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

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

$$\#\text{params} = (JK + 1)C_l C_{l+1} = 7C_l C_{l+1} \tag{1.4.2}$$

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.

1.4.2 Computational Cost

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

As mentioned in subsection 1.4.1, we use the DT$\mathbb{C}$WT for our complex, shift invariant wavelet decomposition. We use the open source Pytorch implementation of the DT$\mathbb{C}$WT [11] as it can run on GPUs and has support for backpropagating gradients.

There is an overhead in doing the wavelet decomposition for each input channel. A regular discrete wavelet transform (DWT) with filters of length L will have $2L\left(1 - 2^{-2J}\right)$ multiplies 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.

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 one first scale and one sixteenth the spatial after the second scale. Restricting ourselves to $J=1$ as we mentioned in section 1.4, the computational cost is then:

$$\underbrace{\frac{7}{4}C_{l+1}}_{\text{mixing}}+\underbrace{\quad 36\quad}_{\text{DT}\mathbb{C}\text{WT}}\quad\text{multiplies per input pixel} \tag{1.4.3}$$

In most CNNs, C_{l+1} is several dozen if not several hundred, which makes Equation 1.4.3 significantly smaller than $L^2C_{l+1}=9C_{l+1}$ multiplies for 3×3 convolutions.

1.4.3 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 1.

1.5 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[12]: 200 classes, 500 images per class, 64×64 pixels per image.

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

1.5.1 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. To minimize the effects of learning from other layers, we build a custom small network, as described in Table 1.1.

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

Algorithm 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

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 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 [13] and only vary the gain value a:

$$A_{ij} \sim U\left[-a\sqrt{\frac{6}{(C_l + C_{l+1})HW}},\ a\sqrt{\frac{6}{(C_l + C_{l+1})HW}}\right] \tag{1.5.1}$$

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 [14] 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 [15] was very helpful in organising parallel distributed training runs. The hyperparameter options are described in Table 1.2, 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 1.3.

Table 1.1: **Architectures for MNIST hyperparameter experiments.** The activation size rows are offset from the layer description rows to convey the input and output shapes. The project layers in both architectures are unlearned, so all of the learning has to be done by the first two layers and the reshuffle layer.

(a) Invariant Architecture

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

(b) Reference Arch with 3×3 convolutions

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

Table 1.2: **Hyperparameter settings for the MNIST experiments.** The weight gain is the term a from Equation 1.5.1. Note that $\log_{10} 3.16 = 0.5$.

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

1.5.1.1 Proposed Expansions

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

To test this hypothesis, we change Equation 1.3.6 to be:

$$y^{(l+1)}(f, \mathbf{u}) = \sum_{q \in Q} z^{(l+1)}(q, \mathbf{u}) * \left(\tilde{a}_f(q)\alpha_f(q, \mathbf{u})\right) \tag{1.5.2}$$

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

(a) We randomly shift each of the $7C$ subbands horizontally by $\{-1, 0, 1\}$ pixels, and vertically

Table 1.3: **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.

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

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 1.4. 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 1.4 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).

1.5.2 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 1.5a. 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

Table 1.4: **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

locally invariant layer, so we restrict our results to swapping 1 or 2 layers. We also report the accuracy when we swap out all of the layers. **??** reports the top-1 classification accuracies for CIFAR-10, CIFAR-100 and Tiny ImageNet. In the table, 'invX' means that the 'convX' layer from 1.5a 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 1.5.1.1.

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

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

1.5.3 Network Comparison

In the previous section, we examined how the locally invariant layer performs when directly swapped in for a convolutional layer in an architecture designed for convolutional layers. In this section, we look at how it performs in a Hybrid ScatterNet-like [1], [2], 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

Table 1.5: **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[16] architecture. C is a hyperparameter that controls the network width, we use $C = 64$ for our initial tests. The activation size rows are offset from the layer description rows to convey the input and output shapes.

(a) CIFAR Architecture

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

(b) Tiny ImageNet Architecture

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

input size. We then use 4 convolutional layers, similar to convC to convF in **??** with $C = 96$. In addition, we use dropout after these later convolutional layers with drop probability $p = 0.3$.

We compare a ScatterNet with no learning in between scattering orders (ScatNet A) to one with our proposal for a learned mixing matrix A (ScatNet B). Finally, we also test the hypothesis seen from subsection 1.5.2 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 1.7 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[17] and Wide[18] ResNets perform best, but with very many more multiplies, parameters and layers.

Table 1.6: Results for testing VGG like architecture with convolutional and invariant layers on several datasets. An architecture with 'invX' means the equivalent convolutional layer 'convX' from **??** was swapped for our proposed layer. The top row is the reference architecture using all convolutional layers. The 'A' architecture is the originally proposed gain layer, the 'B' architecture uses the gainlayer with random shifting of the activations, and the 'C' architecture changes the mixing to be a learned 3×3 kernel acting on the invariant coefficients. Numbers are averages over 5 runs.

	CIFAR-10			CIFAR-100			Tiny ImgNet		
	A	B	C	A	B	C	A	B	C
reference	91.9	-	-	70.3	-	-	59.1	-	-
invA	91.3			69.5			57.7		
invB	91.8			70.7			59.5		
invC	92.3			71.2			59.8		
invD	91.2			70.1			59.3		
invE	91.6			70.0			59.4		
invF	90.5			68.9			57.8		
invA, invB	90.5			68.4			57.9		
invB, invC	91.2			69.1			57.5		
invC, invD	92.1			70.1			59.0		
invD, invE	89.1			67.3			57.5		
invA, invC	90.7			69.6			56.9		
invB, invD	92.7			71.3			59.8		
invC, invE	91.8			70.9			60.2		

Table 1.7: **Hybrid ScatterNet top-1 classification accuracies on CIFAR-10 and CIFAR-100.** N_l is the number of learned convolutional layers, #param is the number of parameters, and #mults is the number of multiplies per $32 \times 32 \times 3$ image. An asterisk indicates that the value was estimated from the architecture description.

	Arch. Properties			Top 1 Accuracies	
Arch. Name	N_l	#Mparam	#Mmults	CIFAR-10	CIFAR-100
ScatNet A	4	2.6	165	89.4	67.0
ScatNet B	6	2.7	167	91.1	70.7
ScatNet C	5	3.7	251	91.6	70.8
ScatNet D	7	4.3	294	93.0	73.5
All Conv[19]	8	1.4	281*	92.8	66.3
VGG16[20]	16	138*	313*	91.6	-
FitNet[21]	19	2.5	382	91.6	65.0
ResNet-1001[17]	1000	10.2	4453*	95.1	77.3
WRN-28-10[18]	28	36.5	5900*	96.1	81.2

ScatNets A to D with 6 layers like convC to convG from **??** after the scattering, achieve $58.1, 59.6, 60.8$ and 62.1% top-1 accuracy on Tiny ImageNet. As these have more parameters and multiplies from the extra layers we exclude them from Table 1.7.

1.6 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 [22]. This invariant layer can easily be shaped to allow it to drop in the place of a convolutional layer, theoretically saving on parameters and computation. However, care must be taken when doing this, as our ablation study showed that the layer only improves upon regular convolution at certain depths. Typically, it seems wise to use the invariant layer right before a sample rate change.

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

References

[1] E. Oyallon, E. Belilovsky, and S. Zagoruyko, "Scaling the Scattering Transform: Deep Hybrid Networks", in *2017 IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017, pp. 5619–5628. arXiv: 1703.08961.

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

[3] A. Singh, "ScatterNet Hybrid Frameworks for Deep Learning", PhD thesis, University of Cambridge, May 2018.

[4] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision", *arXiv:1512.00567 [cs]*, Dec. 2015. arXiv: 1512.00567 [cs].

[5] Y. Ioannou, D. Robertson, J. Shotton, R. Cipolla, and A. Criminisi, "Training CNNs with Low-Rank Filters for Efficient Image Classification", *arXiv:1511.06744 [cs]*, Nov. 2015. arXiv: 1511.06744 [cs].

[6] F. Juefei-Xu, V. N. Boddeti, and M. Savvides, "Local Binary Convolutional Neural Networks", *arXiv:1608.06049 [cs]*, Aug. 2016. arXiv: 1608.06049 [cs].

[7] Q. Qiu, X. Cheng, R. Calderbank, and G. Sapiro, "DCFNet: Deep Neural Network with Decomposed Convolutional Filters", *arXiv:1802.04145 [cs, stat]*, Feb. 2018. arXiv: 1802.04145 [cs, stat].

[8] J. Bruna and S. Mallat, "Invariant Scattering Convolution Networks", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1872–1886, Aug. 2013.

[9] S. Mallat, "Group Invariant Scattering", en, *Communications on Pure and Applied Mathematics*, vol. 65, no. 10, pp. 1331–1398, Oct. 2012.

[10] I. W. Selesnick, R. G. Baraniuk, and N. G. Kingsbury, "The dual-tree complex wavelet transform", *Signal Processing Magazine, IEEE*, vol. 22, no. 6, pp. 123–151, 2005.

[11] F. Cotter, *Pytorch Wavelets*, GitHub fbcotter/pytorch_wavelets, 2018.

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

[13] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks", in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[14] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization", *arXiv:1603.06560 [cs, stat]*, Mar. 2016. arXiv: 1603.06560 [cs, stat].

[15] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A Research Platform for Distributed Model Selection and Training", *arXiv preprint arXiv:1807.05118*, 2018.

[16] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition", *arXiv:1409.1556 [cs]*, Sep. 2014. arXiv: 1409.1556 [cs].

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Identity Mappings in Deep Residual Networks", *arXiv:1603.05027 [cs]*, Mar. 2016. arXiv: 1603.05027 [cs].

[18] S. Zagoruyko and N. Komodakis, "Wide Residual Networks", *arXiv:1605.07146 [cs]*, May 2016. arXiv: 1605.07146 [cs].

[19] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for Simplicity: The All Convolutional Net", *arXiv:1412.6806 [cs]*, Dec. 2014. arXiv: 1412.6806 [cs].

[20] S. Liu and W. Deng, "Very deep convolutional neural network based image classification using small training sample size", in *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, Nov. 2015, pp. 730–734.

[21] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "FitNets: Hints for Thin Deep Nets", *arXiv:1412.6550 [cs]*, Dec. 2014. arXiv: 1412.6550 [cs].

[22] F. Cotter and N. Kingsbury, "Visualizing and Improving Scattering Networks", in *2017 IEEE 27th International Workshop on Machine Learning for Signal Processing (MLSP)*, Sep. 2017, pp. 1–6. arXiv: 1709.01355.