

INSTITUTE FOR COMPUTER SCIENCE
FREIE UNIVERSITÄT BERLIN



Bachelor's Thesis

**EwaldBlocks: Translating the Idea of Ewald
Summation to Neural Networks for Global
and Local Feature Extraction**

Valentin Wolf

Primary Reviewer: Prof. Dr. Frank Noé
Secondary Reviewer: Prof. Dr. Tim Landgraf
Semester: Summer 2019
Author: Valentin Wolf
Matriculation No.: 5092786
Address: Homuthstr. 7, 12161 Berlin
E-Mail: valentin@valentin-wolf.de
Field of Study: Computer Science B.Sc.

Submission date: 20. August 2019

Statutory Declaration

I herewith declare that I have composed the Bachelor's thesis with the title:

**EwaldBlocks: Translating the Idea of Ewald Summation to Neural
Networks for Global and Local Feature Extraction**

myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned.

The thesis in the same or similar form has not been submitted to any examination body and has not been published. This thesis was not yet, even in part, used in another examination or as a course performance.

Berlin (Germany), 20.08.2019

Valentin Wolf

Abstract

This Bachelor's thesis introduces the EwaldBlock, a component for Neural Networks to efficiently extract local and global features from images. Inspired by the Ewald Summation, a method for efficient computation of long-range interactions in particle systems, EwaldBlocks split their input into high and low frequency components and process them separately. The high frequency part contains local features that can be captured well with standard convolutional layers. Global features remain in the low frequency part of the image. Convolutional layers are unsuited to capture them, as they can only process features as large as their kernel size. EwaldBlocks therefore do pointwise multiplication of the lowest frequency components with learned kernels in the Fourier space. By the circular convolution theorem, this is an approximated convolution with global kernel size. EwaldBlocks are designed to be able to replace convolutional layers in existing Convolutional Neural Networks (CNN) architectures. Experiments show, that the EwaldBlock can improve performance of shallow architectures and that computation in the spectral space could lead to models that are robust against noise perturbations and less biased towards local textures compared to CNNs.

Contents

1	Introduction and Motivation	4
2	Background	5
2.1	Supervised Learning and the Image Classification Task	5
2.2	Neural Networks for Image Classification	6
2.2.1	Introduction to Neural Networks	6
2.2.2	Optimization of Neural Networks	8
2.2.3	Convolutional Neural Networks	10
2.3	Discrete Fourier Transform	12
2.3.1	The Fourier Transform and its Inverse	13
2.3.2	The Discrete Fourier Transform (DFT)	14
2.3.3	Fast Fourier Transform (FFT)	14
2.3.4	Multidimensional DFT	14
2.3.5	Natural Images in the Spectral Domain	15
2.3.6	Conjugate Symmetry from Real Inputs	15
2.3.7	Circular Convolution Theorem	17
2.3.8	Fast Convolution Algorithm	17
2.4	Ewald Summation	18
2.4.1	Signal Split	18
2.4.2	Short-Ranged Interactions	19
2.4.3	Long-Ranged Interactions	19
2.4.4	Combining the Branches	20
3	Related Work	20
4	EwaldBlock	21
4.1	Signal split	21
4.2	Spectral Path	22
4.3	Real Path	23
4.4	Merging paths	25
5	Implementation Details	25
6	Experiments	25
6.1	Benchmark Datasets	25
6.2	Training Scheme	26
6.3	Benchmark Architectures	26
6.4	Performance of EwaldBlocks in Shallow Architectures	27
6.5	Performance of the EwaldBlock in the ResNet Architecture	28
6.6	Robustness Against Noise	28
6.7	Sensitivity Towards High and Low Frequencies	32
6.8	Network Analysis	34
7	Discussion and Outlook	37

1 Introduction and Motivation

Convolutional Neural Networks (CNNs) proved to be effective for learning features from images and inputs with spatial structure. While convolutional layers are highly efficient at extracting local features, they require great network depth to achieve sufficiently large receptive fields for detecting global features. This resulted in the trend for deeper network architectures containing millions of parameters.

Most CNNs architectures follow the same design principle: a high resolution input is processed by many convolutional layers with small kernel sizes, with increasing depth the feature map size is reduced by pooling and the number of output channels of the layer is increased.

While this design scheme achieves outstanding performance, the optimality of it remains questionable:

1. There is no inherent reason why early layers should solely have access to highly local features and later layers only on the global features.
2. Going from details to global shapes is the opposite of the intuitive approach to feature extraction, which would be: first grasp global shapes. Then, if necessary, take a closer look at the details.
3. Recent work suggests that the narrative of CNNs learning global features in deep layers may be wishful thinking. Even deep CNNs that theoretically have a large receptive field in late layers, still only use relatively local information of the input image [4] and remain more sensitive to local textures of an image for classification, compared to humans who more strongly rely on global shapes [12].
4. Early layers of standard CNNs only process a small patch of the input and thus can only process high frequency features of it. This makes their learned features fragile against noise and susceptible to adversarial attacks, where small changes to the input image, unnoticeable to the human eye, can drastically change the models prediction [14].

Finding a way to efficiently learning robust global features in parallel with local features extracted by CNNs could alleviate some of those problems.

Ewald Summation (ES) is a method for efficiently computing interactions in particle systems. It computes short-ranged interactions in the real space and long-ranged or global interactions in the Fourier space. Applying its concept to Neural Networks could be a way to build networks that can efficiently extract global as well as local features. While also resulting in less deep and more parameter efficient network architectures. Inspired by ES, the EwaldBlock is proposed in this thesis. Its key idea is to split the input signal into high and low frequency parts and processes each separately:

1. High frequency components contain local features and can be effectively processed in real space by standard convolutions.

2. Low frequency components are the rough shapes of the input and can be interpreted as global features. Subsection 4.2 explains how these can be efficiently processed by pointwise multiplication with learned kernels in the Fourier domain.

Further, the EwaldBlock allows intercommunication between the two computational paths. This results in larger receptive fields at less depth when using the EwaldBlock. First experiments show, that the use of EwaldBlocks increases performance of shallow architectures.

2 Background

2.1 Supervised Learning and the Image Classification Task

In machine learning there are two types of problem settings: unsupervised and supervised. In the former, only data is given and the goal is to find underlying structure in it (e.g. by clustering). In the latter, a ground truth label y is known for each datapoint x and the task is to find a function that maps datapoints to their respective label. Tasks where some but incomplete supervision of the data exists, are referred to as semi-supervised.

Image classification is in most cases a supervised machine learning task. Training data (X_{train}, Y_{train}) is given, where X are images of different classes and Y their corresponding labels. In the multi-class image classification each image is assigned to exactly one class, whereas in multi-label classification one image can be of multiple classes. The labels Y are considered to be one-hot encoded vectors, i.e.

$$y_{i_n} = \begin{cases} 1 & \text{if } y_i \text{ is of class } n \\ 0 & \text{else} \end{cases} \quad (1)$$

This thesis only considers the multi-class setting, however most concepts will be directly applicable to the multi-label setting.

The task is to learn a function $f_\phi : X \rightarrow Y$ with parameters ϕ that accurately predicts the correct label y for an input x . This function is called “model” or “classifier” and both terms are used interchangeably. The prediction of the model will be denoted as:

$$\hat{y} := f_\phi(x) \text{ and } \hat{Y} := f_\phi(X) \quad (2)$$

The goal is that the learned model generalizes to unseen data, i.e. data that is not in the training data but is of similar nature and comes from the same data distribution. Otherwise, a simple lookup table would suffice to solve the task. To assess the generalization power, the accuracy of the model is evaluated on a separate dataset, the test data (X_{test}, Y_{test}) . In mathematic terms, we want to find parameters ϕ of f_ϕ s.t. the mean accuracy on the test data is maximal:

$$\phi = \arg \max_{\phi} Accuracy(\hat{Y}_{test}, Y_{test}) = \arg \max_{\phi} \frac{1}{|X_{test}|} \sum_{(x_i, y_i) \in (X_{test}, Y_{test})} \mathbb{I}_{y_i = \hat{y}_i} \quad (3)$$

where

$$\mathbb{I}_{y_i=\hat{y}_i} = \begin{cases} 1 & y_i = \hat{y}_i \\ 0 & \text{else} \end{cases} \quad (4)$$

is the indicator function. However, the $Accuracy(\cdot, \cdot)$ metric is discontinuous making it hard to work with. To avoid this, a differentiable loss function \mathcal{L} can be used as a proxy to the model's performance. \mathcal{L} is designed to be small when the accuracy of the model is high and large if not. Using \mathcal{L} , one can rephrase the goal to finding parameters ϕ for the model f_ϕ that minimize the loss \mathcal{L} of the models prediction \hat{y} to the ground truth labels y of the training set:

$$\phi = \arg \min_{\phi} \sum_{x_i \in X_{test}} \mathcal{L}(\hat{y}_{test}, y_{test}) = \arg \min_{\phi} \mathcal{L}(\hat{Y}_{test}, Y_{test}) \quad (5)$$

Common loss functions are the mean squared error (MSE) or the cross entropy loss.

The best performing model is the one with the smallest loss on the test set. If a model is performing well on the training set but is not generalizing well to unseen data, the model is overfitting.

To solve eq. (5), different optimization techniques can be used, as solving for ϕ directly is often unfeasible. The process of finding good parameters is called learning or training. In section 2.2.2, Stochastic Gradient Descent, the go-to optimization algorithm for Neural Networks, will be explained.

2.2 Neural Networks for Image Classification

2.2.1 Introduction to Neural Networks

A Neural Network (NN) is a non-linear parameterized function $f_\phi : X \rightarrow Y$. It consists of $N \geq 2$ interconnected layers L_1, \dots, L_N . The first layer L_1 and last layer L_N are the input and output layers. The layers in between (L_1, \dots, L_{N-1}) are hidden layers. The number of layers N is the depth of the network. A NN is deep, if it contains multiple hidden layers, otherwise it is shallow.

Each layer L_i consists of $H_i \geq 1$ neurons, which are connected to neurons of other layers: a neuron n is connected to neuron m if the output of neuron n depends on the output of neuron m . The output of the n^{th} neuron of the i^{th} layer will be denoted as o_n^i . The output of the input layer L_1 is just the input to the NN, i.e. $o_n^1 := x_n$. Similarly, the output of the output layer L_N is the output of the NN, so $o_n^N := \hat{y}_n$.

In a fully connected or dense layer L_i , every neuron is connected to all neurons of the previous layer L_{i-1} . A schematic of a dense NN with one hidden layer can be seen in Figure 1.

For a dense layer the output of a neuron n is a linear combination over all its inputs, i.e. the outputs of the neurons from the previous layer:

$$o_n^i = \sum_{m=1}^{H_{i-1}} w_{n,m}^i o_m^{i-1} + b_n^i \quad (6)$$

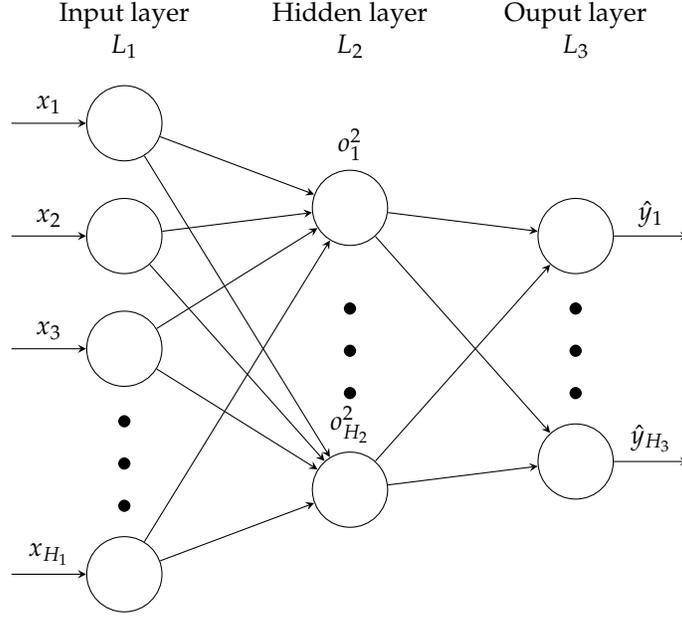


Figure 1: Drawing of a dense Neural Network with one hidden layer

The coefficients of this linear combination are the weights $w_{n,m}^i$ and the bias b_n^i . In the notation, the superscript i refers to the layer and the subscript to the n^{th} neuron with the input being the m^{th} neuron of the previous layer. The weights and biases are the parameters ϕ of the NN and are learned from the data.

Given eq. (6) one can conveniently rewrite the output vector $o^i = \begin{pmatrix} o_1^i \\ \vdots \\ o_{H_i}^i \end{pmatrix}$

of all neurons of layer l_i using matrix notation in the following way:

$$o^i = W^i o^{(i-1)} + b^i \quad (7)$$

with

$$W^i = \begin{pmatrix} w_{0,0}^i & \cdots & w_{0,H_{(i-1)}}^i \\ \vdots & \ddots & \vdots \\ w_{H_i,0}^i & \cdots & w_{H_i,H_{(i-1)}}^i \end{pmatrix} \text{ and } b^i = \begin{pmatrix} b_1^i \\ \vdots \\ b_{H_i}^i \end{pmatrix} \quad (8)$$

where W^i is the weight matrix and b^i the bias vector of layer L_i .

So far, the NN only applies a linear transformations in each layer. No matter how complex is is, it remains a linear model. So, to enable the model to repre-

sent nonlinear functions one adds a non-linearity σ - the activation function - to the output of each layer, redefining it to:

$$x^i = \sigma_i(Wo^{(i-1)} + b^i)$$

In theory, every differentiable nonlinear function could be used as an activation function, however in practice in the multi-class classification task the following is usually done: In all but the last layer a Rectified Linear Unit (ReLU)

$$ReLU(x)_i = \max(0, x_i) \tag{9}$$

is used as the activation function. To the output of the last layer a Softmax function

$$Softmax(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{10}$$

is applied to conveniently convert the predictions to a (pseudo) probability distribution.

2.2.2 Optimization of Neural Networks

The power behind NNs is their remarkable ability to learn highly nonlinear functions. In this section I will introduce Stochastic Gradient Descent, the most common optimization method for Neural Networks and then go on about other important methods for improving learning and reducing the risk of overfitting that are more or less specific to Neural Networks.

Gradient Descent Recall from eq. (5) that the goal is to find a set of parameters ϕ that minimize the loss $\mathcal{L}(\hat{y}, y)$ of the models prediction \hat{y} to the ground truth y of the training set. In NNs solving for ϕ directly is generally unfeasible. But if all computations of the NN and the loss function are differentiable, iterative gradient methods can be used. This optimization process is often called learning or training.

In each iteration, gradient descent computes the gradient of the loss with respect to each parameter and then updates the weights for the next iteration by moving them a small step in the opposite direction of their gradient:

$$\phi_{t+1} = \phi_t - \alpha \frac{\delta \mathcal{L}_t(\hat{Y}, Y)}{\delta \phi_t} \tag{11}$$

where α is the learning rate or step size hyperparameter and ϕ_t are the parameters at iteration t .

Intuitively, gradient descent tries to go downhill in the loss landscape by repeatedly evaluating the local slope and taking a small step downwards. If the learning rate α is chosen small enough, the loss of our model on the training set converges to a minimum. But, if our loss function is not convex - which usually is the case - one is not guaranteed to find a global minimum. Further, if α is too big, the parameter update step could change the weights too much,

overshooting the minimum and possibly increasing instead of decreasing the loss. In practice, gradient descent with carefully chosen learning rates performs very well.

Stochastic Gradient Descent Gradient descent needs to compute $\hat{Y} = f_{\phi}(X)$ for each weight update step. However, evaluating the model on the whole training set can be computationally expensive. To speed up the computation, Stochastic Gradient Descent (SGD) is generally used when training Neural Networks [3]. In every step or iteration, SGD only computes the gradient for a randomly chosen mini-batch of datapoints (usually 32 to 512) and uses it as an approximation to the real gradient for a weight update step in eq. (11). The training passes over the whole dataset multiple times (each pass is called an epoch). Against first intuition, using small mini-batches does not only speed up convergence in the training process, but also helps to find better minima [23]. A simple method to reduce noise in the mini-batch gradients is to use an additional momentum parameter to keep a moving average over recent gradients and use this instead of the real gradient in each parameter update step. This is called SGD with momentum.

Adaptive Learning Rates In eq. (11) the learning rate α is a hyperparameter which remains fixed during the whole training process and is set equally for every weight.

However, it often makes sense to adjust α over the course of training. A standard procedure is to reduce the learning rate over time: A large learning rate in the beginning can help SGD to not get stuck in a poor local minima, while a small learning rate in the end ensures that the parameters reach the bottom of the minimum around which the parameter have wound up over the course of training.

More complicated but again often used are adaptive methods that utilize gradient information from previous iterations to adapt the step size for each parameter individually. Adaptive methods can speed up the convergence of SGD and make its success less dependent on a good initial learning rate. One example is the often used Adam optimizer [24]. With its name originating from ADaptive Momentum, Adam keeps a moving average of the mean and second moment (i.e. the uncentered variance) of the gradients. The mean is used as in SGD with momentum. Additionally, Adam increases the step size in dimensions where small second moments have been observed in the past and vice versa reduces it when observed second moments were large.

Regularization Overfitting is common when using Neural Networks, especially when the size of the dataset is much smaller than the number of network parameters. There are several techniques to improve generalization, some common ones are:

1. Weight decay: add a term to the loss to penalize large weights

2. Dropout: during training set output of neurons of a layer in the forward pass to zero with a specified probability
3. Input augmentation: apply sensible random transformations (shift, rotate, flip, ...) to the input images to encode invariances that can then be learned directly from the data
4. Early Stopping: stop training when the loss on the test set increases to avoid overfitting

Batch Normalization Another recently proposed and widely used technique to improve the training process of NNs is Batch Normalization (BN) [21]. The idea of BN is to normalize the output of the previous layer to a learned mean and variance. BN improves training speed, makes it possible to train deeper networks and even has a regularization effect. While there is a debate on where to place the batch normalization layer, it is most often applied before the activation function of the layer [30].

Residual Connections When training deep NNs, weights in the early layers of a network become less influential on the total outcome. This results in small gradients for those weights and makes training slow. To mitigate this so called problem of vanishing gradients, Residual Neural Networks (ResNets) have been introduced [15, 17]. These add skip connections or short-cuts that forward a layer's output not only to the next but also to later layers and thereby improve the gradient flow. Skip connections are widely used in many different variations (e.g. [34, 20]) and enable the training of networks with hundreds of layers.

2.2.3 Convolutional Neural Networks

In recent years convolutional Neural Networks (CNNs) have been applied with great success to many different tasks in computer vision. Unlike dense NNs, CNNs make use of spatial information in their input. The typical CNN architecture is designed as follows:

$$input \rightarrow ((conv \times N) \rightarrow pooling) \times M \rightarrow flatten \rightarrow dense \rightarrow sigmoid$$

where *conv* (*pooling*) are convolutional (*pooling*) layers and *dense* are one (or multiple) densely connected layers. In the following paragraphs I will first explain the mentioned building blocks and then give insight on other important concepts of CNNs.

Convolutional Layer A convolutional layer has an input shape of $H_{in} \times W_{in} \times C_{in}$, where H_{in} is the width, W_{in} is the height and C_{in} the number of input channels. If the input to the convolutional layer is a greyscale image, $C_{in} = 1$. If it is

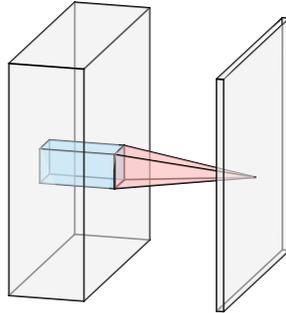


Figure 2: A filter (*blue*) is convolved over the 3D input feature map (*left*) computing its 2D activation map (*right*)

a RGB color image, $C_{in} = 3$. A convolutional layer contains C_{out} filters or kernels each of shape $K \times K \times C_{in}$. K is usually uneven and in most architectures between 3 and 7. The values of the filters are the parameters of the layer and are learned.

The convolutional layer computes, as its name suggests, a discrete convolution of the input and its kernels. This can be imagined as sliding the filter over the input and computing their inner product at each location. For each filter, this gives a 2D output called “activation map”. A schematic of the computation of single activation map is shown in Figure 2. A convolutional layer also has a stride s that determines how many pixels the filter is shifted each time. Generally, $s = 1$ but larger strides can be used to obtain smaller activation maps. Often, zero padding (pre-/appending 0s to the borders of the input) is used, s.t. the convolution can be computed at the edges of the input to yield an output shape equal to the input shape. This is also called “same padding”. Finally, the activation maps of all filters are depth-wise concatenated, giving a final output of shape $H_{out} \times W_{out} \times C_{out}$. W_{out} and H_{out} are dependent on input size, the stride and the padding used, and the number of output channels C_{out} . Analogous to dense layers a (ReLU) non-linearity is usually applied to the output.

The spatial filters enable convolutional layers to extract learned local features of the input. Further, re-using the same filter multiple times at different locations results in two key advantages over dense layers:

1. *Translational invariance*: By shifting the filter, features are detected without regard to their location. This is integral to the success of CNNs, as in many vision tasks shifting the input does not change its content.
2. *Parameter sharing*: The same parameters are used at different locations, making convolutional layers very parameter efficient and less prone to overfit compared to dense layers.

Another cornerstone for the success of convolutional layers are their efficient implementations on graphics processing units (GPUs). These make convolutional layers fast during training and inference, and thus enable it to train very deep CNNs.

Pooling Pooling layers are used to reduce the spatial size of a feature map. These have a stride s and a pooling window of size $K \times K$. The pooling operation works similar to a convolution, but instead of computing a dot product with a kernel, the channel-wise maximum (Max Pooling) or average (Average Pooling) value in the pooling window is used. Often $s = K = 2$ is chosen, reducing both height and width of the input by a factor of 2. Many recent CNN architectures apply Global Average Pooling (GAP) [26] after the last convolutional layer. GAP compresses each feature map to its average.

Receptive Field The receptive field of a neuron is the area in which the network's inputs can influence its output value. The receptive field of a layer refers to the usual size of the receptive fields of its neurons. For dense layers the receptive field is trivially equal to the size of the networks input, as each neuron of the dense layer is connected to all neurons from the previous layer and for any input value there exists a previous layer neuron to which it is connected to.¹ However, as convolutional layers only use local information, the notion of the receptive field becomes interesting. An output neuron of a convolutional layer is only connected to the neurons of the previous layer that are within the layers filter size. So the convolutional layer only has a slightly larger receptive field than the previous convolutional layer. Consider the following example: a network with multiple convolutional layers L_2, L_3, \dots in the beginning. For simplicity let each layer have stride $s = 1$ and some kernel size $K \times K$ with K being uneven, then the receptive field can be calculated with the following recursion:

$$\begin{aligned} \text{receptive_field}(L_2) &= K \\ \text{receptive_field}(L_i) &= K + (\text{receptive_field}(L_{i-1}) - 1) \quad \forall i > 2 \end{aligned}$$

i.e. the receptive field increases the deeper in the network a convolutional layer is. So, for a CNN to be able to extract non local features from the input it has to be deep enough to provide a large enough receptive field in the convolutional networks.

2.3 Discrete Fourier Transform

In this subsection, the Discrete Fourier Transform is introduced and relevant properties of it will be explained. Much of the following content and its nota-

¹for highly unusual network architectures this may not hold, but generally it does

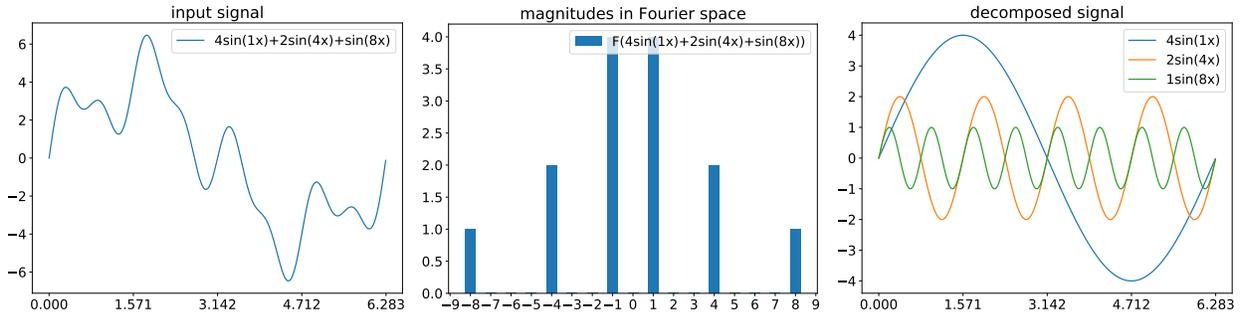


Figure 3: Example of how an input signal can be decomposed into its underlying frequencies with the Fourier transformation: (*left*) the input signal; (*middle*) the magnitudes of the Fourier transformed signal, note the symmetry of the spectral components; (*right*) the decomposed signal received applying the inverse Fourier transform on single magnitudes and their complex conjugate (the symmetric counterpart)

tion is adopted from Osgood, 2013 [32].

2.3.1 The Fourier Transform and its Inverse

The Fourier transform $\mathcal{F} : \mathbb{C} \rightarrow \mathbb{C}$ is a linear transformation that can decompose a signal f into its underlying frequency components (see Figure 3). For a continuous function $f : \mathbb{R} \rightarrow \mathbb{C}$ and any real number $y \in \mathbb{R}$ the Fourier transformation can be written as:

$$\mathcal{F}f(y) = \int_{-\infty}^{+\infty} f(x)e^{-2\pi i x \cdot y} dx \tag{12}$$

where i is the imaginary unit, \cdot the scalar product and dx can be any finite dimensional vector space. Note, that the resulting frequency components are (usually) complex, even if the input signal is strictly real. The magnitude of a frequency component obtained by the Fourier transform is the absolute value of its complex coefficient and its phase the coefficients argument.

The Fourier transform is invertible and with its inverse \mathcal{F}^{-1} being:

$$\mathcal{F}^{-1}[\mathcal{F}f(y)] = f(y) = \int_{-\infty}^{+\infty} [\mathcal{F}f(y)] e^{2\pi i x \cdot y} dy \tag{13}$$

The Fourier transform is often used to analyze signals over time, but it can also be used in other signal spaces (e.g. spatial signals such as images).

2.3.2 The Discrete Fourier Transform (DFT)

In signal processing one usually does not have access to the underlying continuous function f of a signal. Rather, one uses measurements, sampled from the underlying function at discrete but equidistant points in its signal space (e.g. temporal or spatial). Most signals are captured this way in the real world, for example a sensor measuring some value every millisecond or an image sensor measuring the incoming light intensity at equally spaced pixels. For simplicity, consider the case where the signal f is one dimensional and a sequence $x = x_0, x_1, \dots, x_{N-1}$ of N measurements of f is available. Then one can use the discrete Fourier transformation (DFT) F as an approximation to the continuous counterpart:

$$X_k = F(x)_k = F(x_0, x_1, \dots, x_{N-1})_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} k \cdot n} \quad (14)$$

Applying the DFT on x now returns a sequence $X = X_0, X_1, \dots, X_{N-1}$ of the complex valued spectral (or frequency or Fourier) coefficients (or components). Analogous to the continuous case the DFT is invertible:

$$x_k = F^{-1}(X_0, X_1, \dots, X_{N-1})_k = \sum_{n=0}^{N-1} X_n e^{\frac{2\pi i}{N} k \cdot n} \quad (15)$$

2.3.3 Fast Fourier Transform (FFT)

For a sequence of N values, the naive DFT implementation computes N sums each containing N summands which gives a runtime complexity of $O(N^2)$. However, by using a divide and conquer approach and exploiting the periodicity of the complex exponential, it is possible to drastically reduce the computation of the DFT and its inverse to $O(n \log n)$ time. This algorithm is known as the Cooley–Tukey fast Fourier Transform (FFT) algorithm [9], however its details will be omitted here as they are beyond the scope of this thesis. The FFT algorithm is also applicable to the inverse DFT, abbreviated as IFFT for Inverse Fast Fourier Transform.

2.3.4 Multidimensional DFT

The DFT can easily be expanded to multiple dimensions. This is where the Fourier transform gets interesting for image processing. Consider discrete measurements of a two dimensional signal x (e.g. pixel values of an image) of size $M \times N$:

$$x = \begin{pmatrix} x_{0,0} & \cdots & x_{0,N-1} \\ \vdots & \ddots & \vdots \\ x_{M-1,0} & \cdots & x_{M-1,N-1} \end{pmatrix}$$

Then its 2D spectral representation X is:

$$X = \begin{pmatrix} X_{0,0} & \cdots & X_{0,N-1} \\ \vdots & \ddots & \vdots \\ X_{M-1,0} & \cdots & X_{M-1,N-1} \end{pmatrix}$$

with

$$X_{k,l} = F(x)_{k,l} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x_{m,n} e^{-2\pi i(\frac{m \cdot k}{M} + \frac{n \cdot l}{N})} \quad (16)$$

2.3.5 Natural Images in the Spectral Domain

It has been shown that the expected magnitudes of frequency components from natural images decreases by the inverse power law as the frequency increases. Thus, for natural images information is concentrated in the low frequencies, whereas high frequencies encode mostly noise [35]. In Figure 4 a example image and its corresponding frequency spectrum is shown in normal and logarithmic scale. One can see, that the magnitudes in the frequency domain decreases rapidly with increasing frequency. Removing the high frequencies slowly blurs an image while retaining most of its information. At a compression rate of 64 the image in Figure 4 still remains clearly recognizable. At higher compression rates only global shapes remain.

2.3.6 Conjugate Symmetry from Real Inputs

The magnitudes of the spectral representations shown in Figures 3 and 4 are point symmetric. This stems from the input being real which results in conjugate symmetry of the spectral coefficients. Precisely, an input sequence $x = x_0, \dots, x_{N-1}$ is purely real, if and only if its Fourier transformed signal $F(x) = X = X_0, \dots, X_{N-1}$ is even symmetric i.e:

$$\forall k : x_k \in \mathbb{R} \iff \forall j : X_j = X_{-j}^* \pmod{N} \quad (17)$$

where X^* is the complex conjugate of X . Thus when doing computation in the Fourier space of a real signal, it is sufficient just to keep the $\lfloor \frac{N}{2} \rfloor + 1$ first spectral components as the others can be inferred. Further, from eq. (17) follows that X_0 and in the case of an even N also $X_{\frac{N}{2}}$ have to be real valued as:

$$\text{eq. (17)} \implies X_0 = X_0^* \pmod{N} \implies X_0 \in \mathbb{R} \quad (18)$$

$$\text{eq. (17)} \wedge \text{even}(N) \implies X_{\frac{N}{2}} = X_{-\frac{N}{2}}^* \pmod{N} = X_{\frac{N}{2}}^* \implies X_{\frac{N}{2}} \in \mathbb{R} \quad (19)$$

If the output of an inverse Fourier transform is expected to be real, the spectral space signal has to be constrained to adhere to eq. (17) so one has to be careful with computation in the spectral domain. An equivalent of eq. (17) exists for two dimensional real inputs of size $N \times M$:

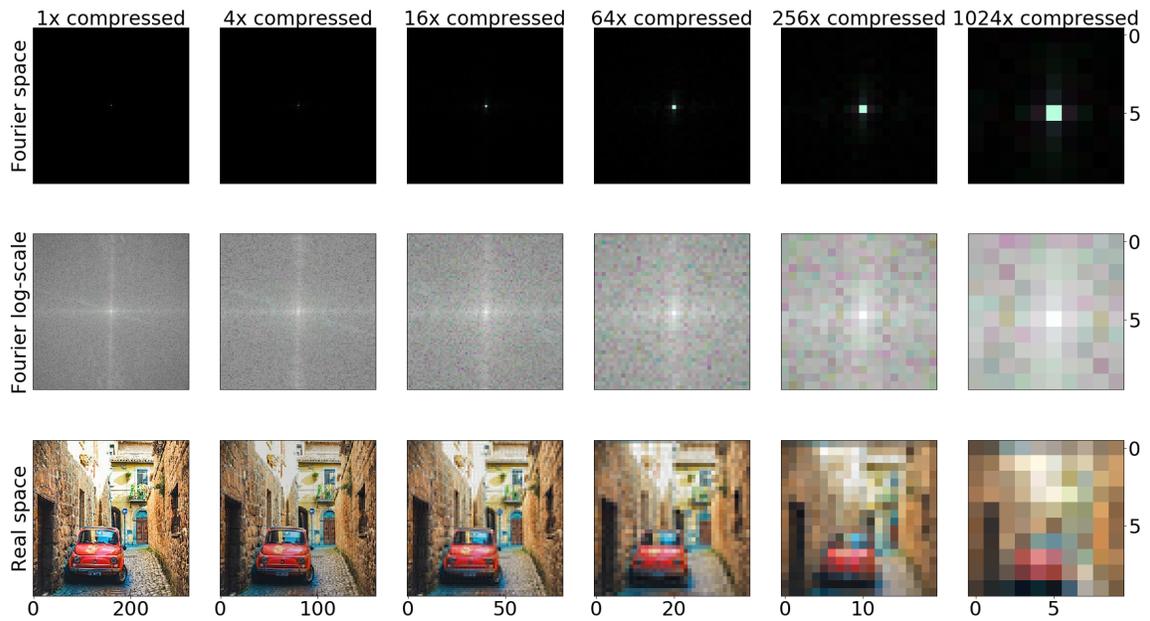


Figure 4: (*first column*) A two dimensional DFT is applied channel-wise to a RGB image of size 320x320 pixels. In the spectral representation is shifted s.t. the 0^{th} frequency component $X_{0,0}$ is in the center of the image. (*subsequent columns*) the spectral coefficient matrix is truncated to $\frac{1}{4}$ the size of the previous column by removing the highest frequency components. The resulting matrix transformed back to the real space yielding a compressed (but blurred) versions of the original image (shown in the last row)

$$\forall k, l : x_{k,l} \in \mathbb{R}^2 \iff \forall i, j : X_{i,j} = X_{-i \pmod N, -j \pmod M}^* \quad (20)$$

and for even N and M implies that $X_{0,0}, X_{\frac{N}{2},0}, X_{0,\frac{M}{2}}, X_{\frac{N}{2},\frac{M}{2}}$ have to be real [33].

2.3.7 Circular Convolution Theorem

Another interesting property of the Fourier transform is the Circular Convolution Theorem. It states that a circular convolution of two signals is equal to a pointwise multiplication of their spectral representations followed by an inverse Fourier transform i.e:

$$f * g = \mathcal{F}^{-1} [\mathcal{F}f \cdot \mathcal{F}g] \quad (21)$$

where $f * g$ is the circular convolution, which for the one dimensional and discrete case is defined as:

$$(f * g)_n = \sum_{k=0}^{M-1} f_k g_{n-k \pmod N} \quad (22)$$

for a kernel f of length M and an input g of length N . However, to compute a convolution by pointwise multiplication, it has to be ensured that the kernel is of the same shape as the input. If this is not the case, one can simply add zero padding to the borders of the shorter sequence.

The circular convolution only differs to a standard convolution by the additional $\pmod N$ in the subscript of g . Intuitively one can picture a convolution, where the kernel wraps around the borders of the input as if the input was cylindrical. For image processing this “wrapping-around” is usually unwanted and one either discards the values at the borders or zero pads the input in all dimensions by half the kernel size.

2.3.8 Fast Convolution Algorithm

By combining the Circular Convolution Theorem with the FFT algorithm one can easily speed up convolutions. For the one dimensional case, the naive circular convolution implementation consists of N sums each with M summands and has a runtime of $O(NM)$. Applying the FFT algorithm on the input and the (padded) kernel, pointwise multiplying and then applying an inverse FFT (IFFT) on the result, gives a faster asymptotic runtime of $O(3 * (N \log N) + N) = O(n \log n)$. For the two dimensional case, the same is applicable and the speedup is from $O(N^2 M^2)$ to $O(N^2 \log^2 N)$.

The fast convolution algorithm has been used for convolutional layers of NNs and has been shown to give a significant speedup for large inputs [29].

2.4 Ewald Summation

Ewald Summation (ES) is a method to efficiently compute particle interactions in cases where long-ranged interactions cannot be efficiently cut off, such as electrostatic charge interactions in periodic systems. ES has a runtime of $O(n^{\frac{3}{2}})$ - a significant speedup compared to the $O(n^2)$ runtime of direct computation [11]. By mapping charges onto a periodic lattice, Particle Mesh Ewald (PME) further speeds up the runtime to $O(n \log n)$ [10]. While the idea of ES does not directly translate to general machine learning, the core concept seems to be well suited for Neural Networks with spatial feature spaces. In the following, I briefly explain the computation steps of ES and propose analogue procedures applicable for NNs. For references on ES, Frankel and Smit, 2001[11] and Noe, 2018 [31] was used.

2.4.1 Signal Split

ES splits the computation of interaction potentials in two parts, one considering short and one considering long-ranged interactions:

$$x(\mathbf{r}) = \underbrace{x_{sr}(\mathbf{r})}_{\text{short-ranged}} + \underbrace{x_{lr}(r)}_{\text{long-ranged}} \quad (23)$$

In ES this is done by adding and subtracting a smeared charge distribution $G_\sigma(r)$ to the initial input :

$$x(\mathbf{r}) = \underbrace{x(\mathbf{r}) - G_\sigma(r)}_{\substack{=: x_{sr}(\mathbf{r}) \\ \text{real space}}} + \underbrace{G_\sigma(r) - \underbrace{\epsilon}_{\substack{\text{self-interaction} \\ \text{correction}}}}_{\substack{=: x_{lr}(\mathbf{r}) \\ \text{Fourier space}}} \quad (24)$$

and thereby splitting the signal. In standard ES, $G_\sigma(r)$ consists of gaussian distributions placed on the point particles, however other choices are possible.

$G_\sigma(r)$ is chosen such that it is smooth and explained by few low frequencies components of the Fourier space. On the other hand, subtracting $G_\sigma(r)$ from the input isolates the particles from global interactions making the signal $x_{sr} := x(\mathbf{r}) - G_\sigma(r)$ short-ranged in real space. In terms of signal processing, x_{lr} is a low-pass filtered input signal and $x_{sr}(r)$ are the residual high frequencies. The short-ranged or high frequency part $x_{sr}(r)$ of the signal is then processed in real space, while the long-ranged or low frequency part $x_{lr}(r)$ of the signal is further processed in Fourier space.

As convolutions can act as low-pass filters, a convolutional layer with a parameterized kernel $\kappa_{\text{split}}(\mathbf{r}, \theta)$ could be used to learn an optimal signal split

in a NN:

$$x_{lr}(r) = x(\mathbf{r}) * \kappa_{\text{split}}(\mathbf{r}, \theta) \quad (25)$$

$$x_{sr}(r) = x(\mathbf{r}) - x_{lr}(r) \quad (26)$$

Further, some properties of low-pass filter kernels can be easily enforced on the convolutional kernel $\kappa_{\text{split}}(\mathbf{r}, \theta)$. For example it can be restricted to be symmetric, non-negative or to have a $L1$ norm equal to 1. For even tighter correspondence to standard ES, the kernel can be gaussian with a learned variance.

2.4.2 Short-Ranged Interactions

The short ranged interactions x_{sr} can be calculated by a simple sum over the particles. As short range interactions decay rapidly with distance, the sum can be efficiently truncated to only consider local neighboring particles.

For NNs, we propose to use a single convolutional layer with a small kernel size, as they have proved incredibly effective for extracting local features from their input.

2.4.3 Long-Ranged Interactions

ES Fourier transforms the smeared charge distribution $G_\sigma(r) \simeq x_{lr}$, computes the long-ranged interaction energies in Fourier space and then inverses the transformation. As $G_\sigma(r)$ is periodic and smooth (by choice), it can be represented as a rapidly convergent Fourier series, which in return can be efficiently approximated by few low frequency components of the series. This, and the fact that solving the poisson equation is simple in Fourier space, make it possible to efficiently compute $G_\sigma(r)$. Lastly, as $G_\sigma(r)$ unwantedly takes the self-interaction energy ϵ of the particle r into account, it has to be subtracted afterwards to yield the correct long-range interaction energy x_{lr} .

NNs can process the the low-passed input in a similar fashion:

1. Fourier transform the input
2. Truncate the resulting matrix to only retain the low frequency components prominent in the low-passed signal
3. Process the truncated signal in Fourier space
4. Apply an inverse Fourier transform

Steps (1.), (2.) and (4.) are straight forward, but it remains unclear how (3.) can be done effectively. We propose to pointwise multiply the lowest frequency components of the input with learned kernels. By the Cyclic Convolution Theorem, this is an approximated convolution where only low frequency components of input and kernel are considered. This approximation is beneficial as global features then become more invariant towards local changes. Further details on the proposed computation in Fourier space will be covered in 4.2.

2.4.4 Combining the Branches

ES is calculating a specific value and simply adds the two branches in order to compute the total interaction energies and forces. In machine learning, one is more interested in learning a projection into a multi-dimensional latent space where the task is well separated. Several methods how local and global features can be combined in NNs will be proposed in 4.

3 Related Work

In this chapter, other works on CNNs that make use of the spectral domain or try to learn global features are reviewed.

Rippel et. al [33] show that parameterizing kernels of convolutional kernels in the Fourier space and then transforming them back to spatial space to use them as usual in the computation of a convolution, results in faster convergence during model training. Notably, one can remove the spectral parameterization after the model is trained and simply save the spatial equivalency as the normal kernel weights of the convolutional layers. The authors attribute the speedup to the phenomenon that convolutional kernels are sparse in the spectral domain and state of the art adaptive optimization algorithms like Adam can exploit this to converge faster.

Ballé et. al [2] used spectral parameterization in their end-to-end image compression network and noted that applying a Discrete Cosine Transformation (DCT) instead of a DFT further increased convergence speed. This could also be a very interesting approach to be tried in the future, as efficient *symmetric* convolutions can be implemented using DCT [28]. Symmetric convolutions could be better suited for non-periodic data as it may reduce the amount of padding needed to counteract the wrapping around of the cyclic convolution.

Further, Rippel et. al [33] introduced Spectral Pooling: A FFT is applied to the feature map, the resulting frequency matrix is then truncated to a desired output shape by removing the highest frequency components and then transformed back to spatial space. This makes it possible to slowly reduce feature maps in their size, compared to conventional pooling that have a minimum reduction factor of 4. In order to avoid complex arithmetic and conjugate symmetry constraints Zhang et. al [40] proposed to use the Hartley transform for Spectral Pooling. The concept of Spectral Pooling is used in this work, however with a different use: we truncate the frequency component matrix to efficiently extract global features from the low frequencies.

Trabelsi et. al [36] introduce complex parameterized Neural Networks, an adaption of Batch Normalization and common weight initialization techniques for the complex domain. They further review existing complex valued activation functions and show that their complex parameterized ResNet achieves comparable results to the real counterpart on different datasets.

Other work [38, 6] also exploited the fact that convolutional kernels are highly sparse in their spectral representation to drastically distill and compress

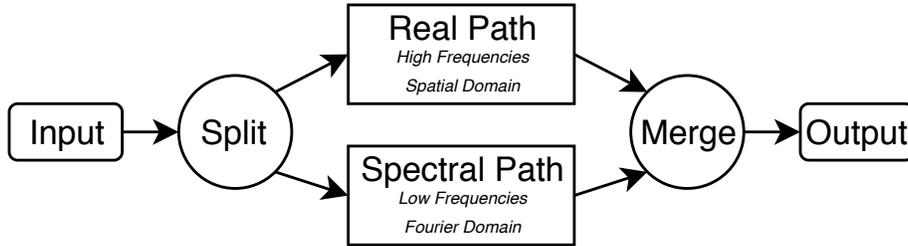


Figure 5: High level concept of Ewald Summation and the proposed Ewald-Block

CNNs.

Multi-scale Representation Learning with multiple computational paths, as it is done in EwaldBlocks, has been explored with CNNs that process feature maps at different spatial sizes in parallel [7, 22, 5, 19, 27, 41, 37]. However, these do not focus on learning global features and none of the works explicitly split the input signal to the frequencies that can actually be captured at the respective scale and kernel size. Chen et al. [7] use an efficient information sharing mechanism between feature maps of different sizes that implicitly up- and downscale between feature maps of different sizes. This could be adapted in further work on the EwaldBlock.

4 EwaldBlock

In this section the EwaldBlock, a component translating the concept of Ewald Summation to NNs for image classification tasks, is introduced. The EwaldBlock applies the ideas from 2.4 to effectively extract local and global features of its input. A schematic of its computation can be seen in Figures 5 and 6. Further, the EwaldBlock is designed to be able to simply replace convolutional layers in existing CNN architectures.

4.1 Signal split

To split the signal into its high and low frequencies, the EwaldBlock uses a channel-wise convolution with a gaussian kernel that has a standard deviation σ and size $K \times K$. The resulting blurred signal of the convolution, containing mainly the input’s low frequency components, is passed to the spectral path. Further, it is subtracted from the original input. This difference contains the high frequency components of the input and is used as input to the real path of the EwaldBlock.

To find a well suited split in each EwaldBlock, σ is kept as a learnable parameter. The size $K \times K$ of the kernel is then set dynamically to be the smallest

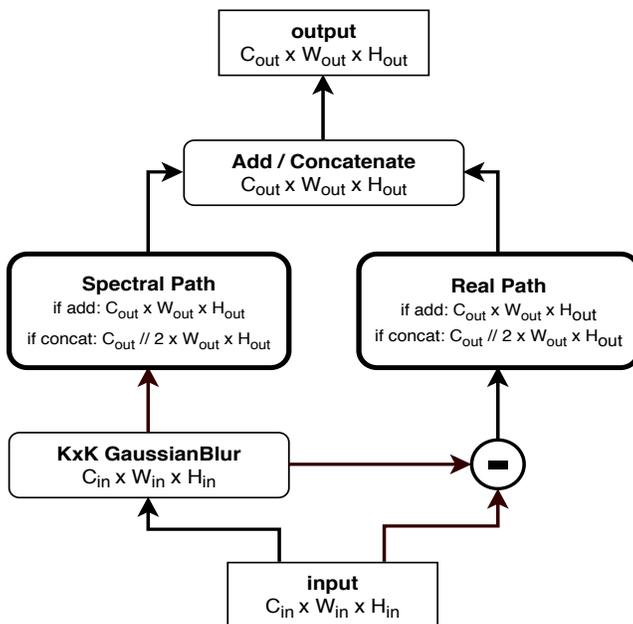


Figure 6: Architecture of the proposed EwaldBlock

uneven integer larger than $2k\sigma$ where k is a hyperparameter. In all experiments $k = 2$, giving the learned gaussian kernels a support of at least 95%. After the size is determined, the kernel is normalized to sum to 1.

4.2 Spectral Path

Figure 7 and 8 outline the computation done in the spectral path. At first a Fast Fourier Transformation (FFT) is applied. The resulting matrix containing the complex valued frequency components is then truncated at a set cutoff c retaining only the most long ranged frequencies. Due to conjugate symmetry as introduced in eq. (20), half of the matrix is redundant and can be disregarded. This leaves a matrix of shape $C_{in} \times (2c - 1) \times c$. In this matrix, $2c - 1$ values² remain redundant (the imaginary part of 0^{th} component and the $c - 1$ complex numbers in the first column, where the bottom half is symmetric to the top half) and can also be disregarded in the computation. This gives

$$2 * (2c - 1) * c - (2c - 1) = (4c^2 - 2c) + 1 = (2c - 1)^2 \quad (27)$$

unique values for each of input channel.

²complex numbers are counted as two values or parameters due to them being stored as two floats

This truncated frequency matrix is then pointwise multiplied and channel-wise aggregated with C_{out} learnable kernels of equal size. Resulting in

$$C_{out} * C_{in} * (2c - 1)^2 \quad (28)$$

unique learnable parameters for the spectral path of the EwaldBlock. For reference, the number of unique learnable parameters at a cutoff c is equal to the number of parameters of a convolutional layer with a kernel size $(2c - 1) \times (2c - 1)$. The computation of the pointwise multiplication is independent of the initial input size $N \times N$. Thus only

$$C_{out} * C_{in} * (2c - 1)^2 \quad (29)$$

multiplications are needed for a pointwise multiplication kernel with cutoff c . Compared to a convolution that needs

$$N^2 * C_{out} * C_{in} * K^2 \quad (30)$$

multiplications for a convolutional kernel of size K . Thus increasing the cutoff c and thereby the number of parameters, has less impact on the total runtime, compared to increasing the kernel size of a convolutional layer.

After pointwise multiplying, we zero pad the output to match the desired output shape $H_{out} \times W_{out}$ and use an IFFT to convert the feature map back to the spatial space. Before back transforming, one has to ensure that the matrix adheres to the conjugate symmetry constraints explained in eq. (18) and eq. (19).

By the circular convolution theorem eq. (21), the computation proposed in the spectral path is equal to a circular convolution with an approximated input and kernel (as only their low frequency components are used) with a kernel size equal to the size of the input. To mitigate the wrapping-around behavior of the circular convolution, additional zero padding of half the input size has to be added to the borders of the input before the whole computation of the spectral path and then be removed in the end. In the following, this padding will be referred to as spectral padding. Spectral padding increases the total number of frequency components and thereby reduces the expressivity of each. Thus, the cutoff frequency c has to be increased by half to retain the same amount of information as without padding. Unfortunately, a larger c results in a significant increase in the number of parameters of the spectral kernel, as its size depends quadratically on c .

4.3 Real Path

The real path is simply a convolutional layer with a small kernel size, as those have been shown to be exceptionally well suited for local feature extraction.

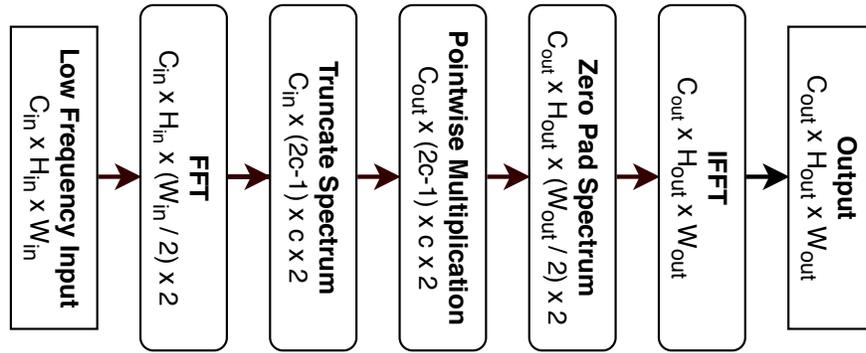


Figure 7: Computation flow for the spectral path with output shapes of each component

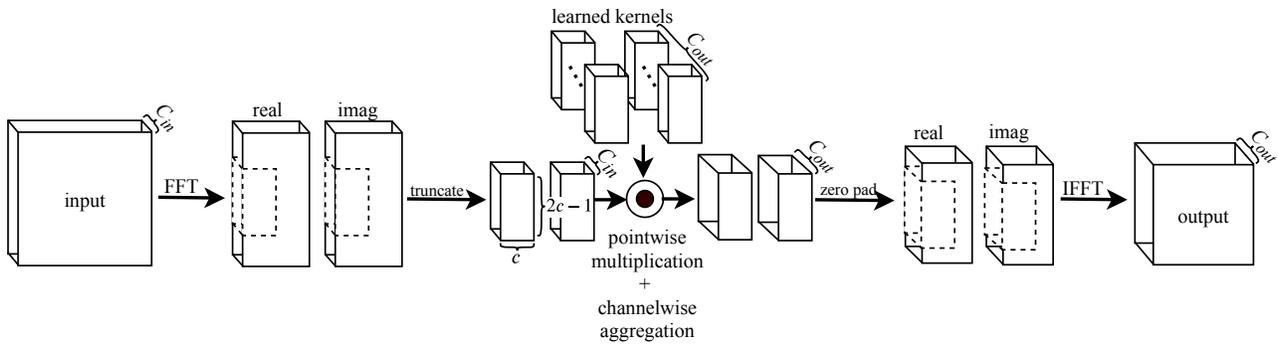


Figure 8: Schematic of spectral path computation

4.4 Merging paths

The last question that arises is how the two paths can be joined in a sensible way. There are multiple ways that come to mind:

1. simply add the outputs of both paths
2. only concatenate the outputs
3. concatenate the outputs and then apply a 1×1 convolution to linearly combine the learned features and compress (or even expand) the number of channels of the output as desired

For simplicity the EwaldBlock just concatenates the computational paths. However, more sophisticated techniques that allow information exchange without having to upsample the spectral path, similar to the way Chen et al. [7] have used, could be of interest in the future.

5 Implementation Details

In this section, details of my implementation will be mentioned. Everything was implemented using Keras [8] and TensorFlow [1]. Models were trained on the NVIDIA GTX 980 GPUs of the allegro cluster or on the free GPUs available on Google Colab³ and Kaggle⁴.

Signal split: For the gaussian kernel convolutions we initialize $\sigma = 1$. Further, we add symmetric padding to reduce vignetting at the borders of the output. Due to the symmetry of the gaussian kernel we also use a separated convolution for efficiency i.e. we apply two 1D convolutions instead of one 2D convolution.

Spectral Path: The weights of the pointwise multiplication kernels are initialized in the following way: Real valued kernels, with height and width equal to the output shape of the EwaldBlock, are initialized by some weight initialization scheme. We then apply a FFT to the kernel and truncate the output at the cutoff frequency c of the EwaldBlock and use those as the initial weights during training. So the kernels are initialized with the low frequency approximation of an initialized convolutional kernel with a size equal to the size of the output.

6 Experiments

6.1 Benchmark Datasets

MNIST contains black and white images of handwritten digits from 0 to 9. Images are greyscale and have a size of 28×28 pixels. The dataset is split into train and test set of size 50k and 10k respectively.

³<https://colab.research.google.com/>

⁴<https://www.kaggle.com/>

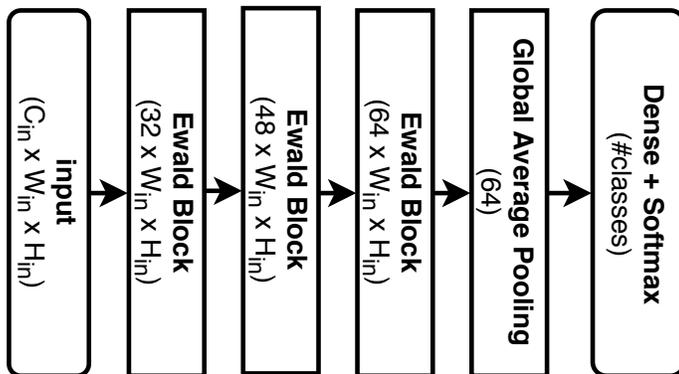


Figure 9: Architecture of the EwaldNet3 network

CIFAR10 [25] contains 60k real world RGB images of 10 different classes with 6000 images per class. Images are sized $32 \times 32 \times 3$ and split into a training set of size 50k and a test set of size 10k.

6.2 Training Scheme

The Adam optimizer [24] with hyperparameters $\beta_1 = 0.9$, and $\beta_2 = 0.999$ and a batch size of 32 was used for all experiments. The initial learning rate of Adam was set to 0.001 and was decreased by a factor of 10 after 10 epochs of stagnation (i.e. when the loss on the test set did not improve). A weight decay with $\alpha = 0.0001$ was used on all convolutional and pointwise multiplication kernel weights. Spectral padding was always used in the spectral path of the EwaldBlock. The mean of all datasets was removed and for CIFAR10 simple data augmentation techniques (horizontal flips and random shifts) were applied.

6.3 Benchmark Architectures

EwaldNet3 As a basic check, we built the small EwaldNet3, a network of 3 stacked EwaldBlocks followed by a Global Average Pooling layer as shown in Figure 9. Convolutions with a kernel size 3×3 and *same* padding are used in the real path of each EwaldBlock. Further, each EwaldBlock is followed by a batch normalization layer and a $LeakyReLU(x) = \begin{cases} 0.3x & \text{if } x < 0 \\ x & \text{else} \end{cases}$ activation function.

EwaldNet3 is designed to evaluate the feature extraction capabilities of proposed spectral path and the interplay between the local and global features extracted. If the spectral path is deactivated, the last convolutional layer only has a receptive field of size 9×9 making the real path intentionally limited to extracting local features. Further, the global pooling in the end makes sure no

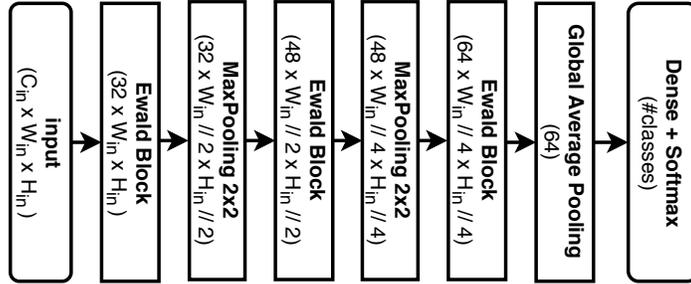


Figure 10: Architecture of the EwaldLeNet network

location information is propagated to the dense layer. Glorot uniform initialization [13] was used for weight initialization.

EwaldLeNet In addition we introduce the EwaldLeNet. Its architecture is identical to EwaldNet3 but additional Max Pooling is added after the first and second EwaldBlocks (see Figure 10). Analogous to the EwaldNet3, the layer order is EwaldBlock->BatchNorm->LeakyReLU->Pooling. With the pooling layers, the last convolution of the real path has an effective receptive field of 18×18 (if the spectral path is disabled).

EwaldResNet We further evaluate EwaldBlocks in the EwaldResNet. For this the ResNetv2 architecture [17] is adapted by replacing all 3×3 convolutions with EwaldBlocks. The EwaldResNet has a width multiplier w . For $w = 1$ the number of channels after concatenation is equal to the number of channels in the original ResNet architecture, i.e. the convolutions real path in the EwaldBlocks in the EwaldResNet use half the channels of the original. For $w = 2$ the same 3×3 convolution in the real path is applied and the spectral path uses an equal amount of output channels. Everything else is kept equal to the ResNetv2 implementation from Keras at keras.io/examples/CIFAR10_resnet/. He normal initialization [16] was used for all kernel weights.

6.4 Performance of EwaldBlocks in Shallow Architectures

To test if EwaldBlocks can improve feature extraction, we train the EwaldNet3 and the EwaldLeNet in several different variants. To indicate what parts of the EwaldBlocks were activated, the following labeling is used:

<i>real</i>	real path is used
<i>spectral</i> ($c = x$)	spectral path with cutoff $c = x$ is used
<i>gaussian split</i>	gaussian split with learnable σ is used

For a fair comparison, if both paths are enabled, the output channels of each path are halved to keep the number of output channels after the concatenation

of the paths equal to the networks using only one path. Training occurred as described in Section 6.2 on MNIST and CIFAR10 for 10 epochs. Results are averaged over 5 runs and are reported in Table 1. Plots of the training progression of the EwaldNet3 are shown in Figures 11 and 12 and of the EwaldLeNet in Figures 13 and 14.

EwaldNet3 with both paths of the EwaldBlock enabled, achieves significantly better performance on CIFAR10 and MNIST. This indicates that the model is able to exploit global features and is less limited by the small receptive field size of the real path. The EwaldLeNet with both paths achieves the best performance on CIFAR10, but remains slightly worse on MNIST than the ‘*real + gaussian split*’ model.

Using cutoff of $c = 4$ in the spectral path generally showed an increase in performance over $c = 3$. Initial tests with $c = 2$ showed no significant performance gains. This is expected as with spectral padding very little information remains at $c = 2$. However, this also means that the spectral path needs more parameters to be effective compared to the convolution of the real path. But, as discussed in Subsection 4.2, an increase in parameters in the pointwise multiplication only has a small impact on runtime.

Further, models using the spectral path, converged faster as they attain higher performance after fewer epochs in the experiments. However, the current implementation of the EwaldBlocks makes the models still significantly slower to train. Much of the overhead stems from the gaussian kernel convolution of the split, such that more efficient alternatives have to be considered in the future to split the signal.

Models using only the spectral path still learn discriminative features. However, their performance remains poor.

Surprisingly, the ‘*real + gaussian split*’ model outperforms the ‘*real*’ model in all experiments. As shown in Table 3, a ‘*real + gaussian split*’ model trained on CIFAR10 learned to keep σ relatively small in its first EwaldBlock and thereby removes low frequencies from the input image. Why this improves performance is unclear and remains an open question.

6.5 Performance of the EwaldBlock in the ResNet Architecture

The models were trained for 200 epochs on CIFAR10 as described in Section 6.2. Results are reported in Table 2 and training progression is shown in Figure 15. However, results have to be treated with caution, as only one run was done for this experiment. Accuracy and loss to converge faster on both train and test set when EwaldBlocks are used, especially with $w = 2$. However, EwaldResNets overfit more to the training set and their final performance on the test set remains worse compared to the normal ResNet.

6.6 Robustness Against Noise

Relying on global features for image classification should improve the robustness against noise in the input data. To test this, the EwaldNet3s and EwaldLeNets

	real	spectral	c	gaussian split	#params	s/epoch	MNIST	CIFAR10
EwaldNet3	✓	✓	3	✓	91k	44	98.65(±0.2)	72.95(±1.1)
	✓	✓	4	✓	152k	44	98.68(±0.2)	72.32(±1.6)
	-	✓	3	✓	140k	43	94.44(±0.1)	52.99(±0.3)
	-	✓	4	✓	261k	43	96.85(±0.1)	56.54(±0.3)
	✓	-	-	✓	43k	41	97.22(±0.1)	57.78(±1.5)
	✓	-	-	-	43k	6	73.60(±9.5)	55.59(±1.8)
EwaldLeNet	✓	✓	3	✓	91k	16	98.51(±0.2)	73.37(±0.9)
	✓	✓	4	✓	152k	17	98.69(±0.1)	74.08(±1.6)
	-	✓	3	✓	140k	16	93.95(±0.2)	53.34(±0.6)
	-	✓	4	✓	261k	17	96.56(±0.2)	57.31(±0.4)
	✓	-	-	✓	43k	12	98.99(±0.0)	69.67(±0.5)
	✓	-	-	-	43k	4	97.37(±0.2)	63.03(±1.4)

Table 1: EwaldNet3 and EwaldLeNet results on CIFAR10 and MNIST. Reported value is the mean best accuracy on the test set of 5 models. The number of parameters and time per epoch is for the models on MNIST, and time per epoch is on a Nvidia Tesla P100 GPU as provided by Kaggle

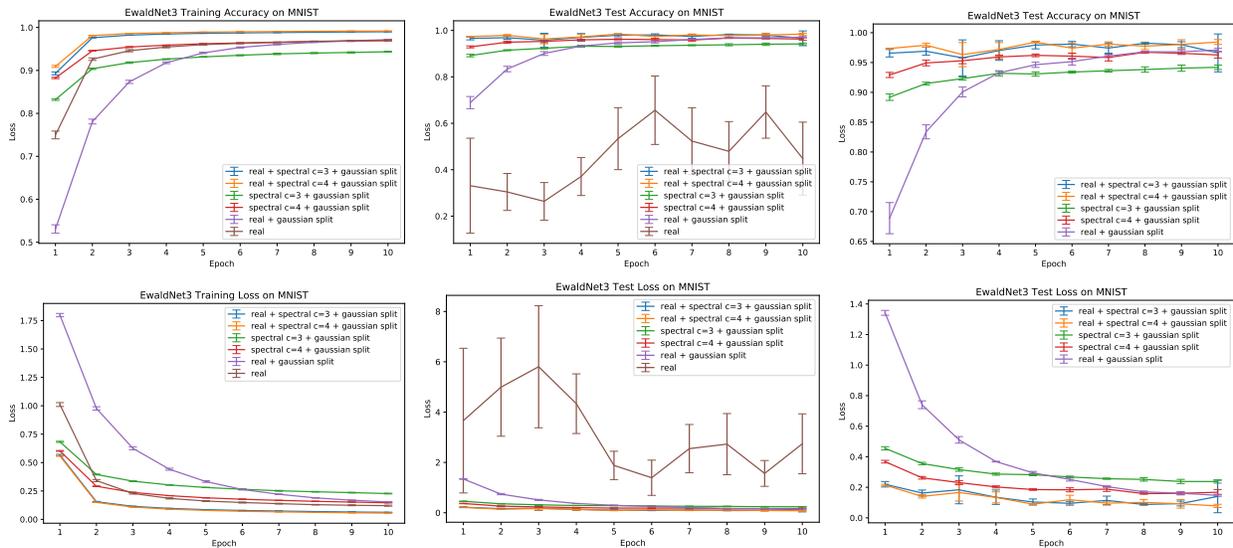


Figure 11: EwaldNet3: Training progression on MNIST. For clarity, metrics on the test data are shown twice in the middle and last column, with the later skipping the results of poor performing 'real' EwaldNet3

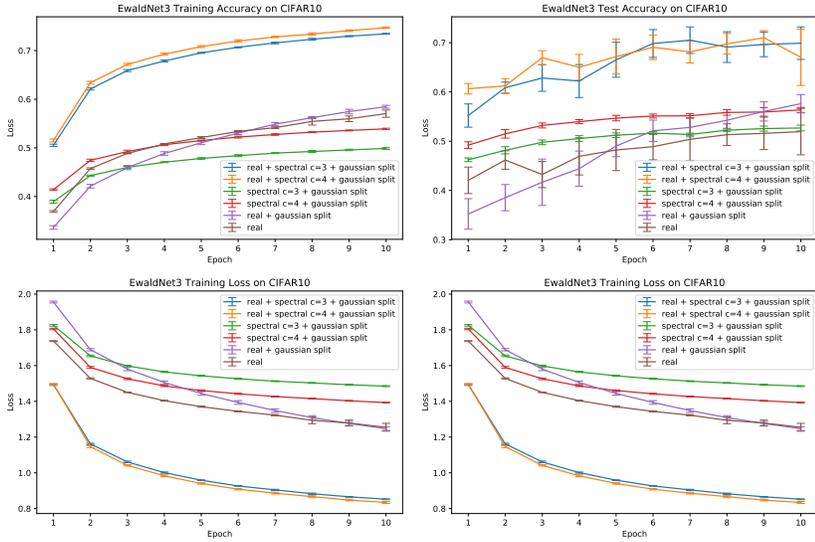


Figure 12: EwaldNet3: Training progression on CIFAR10

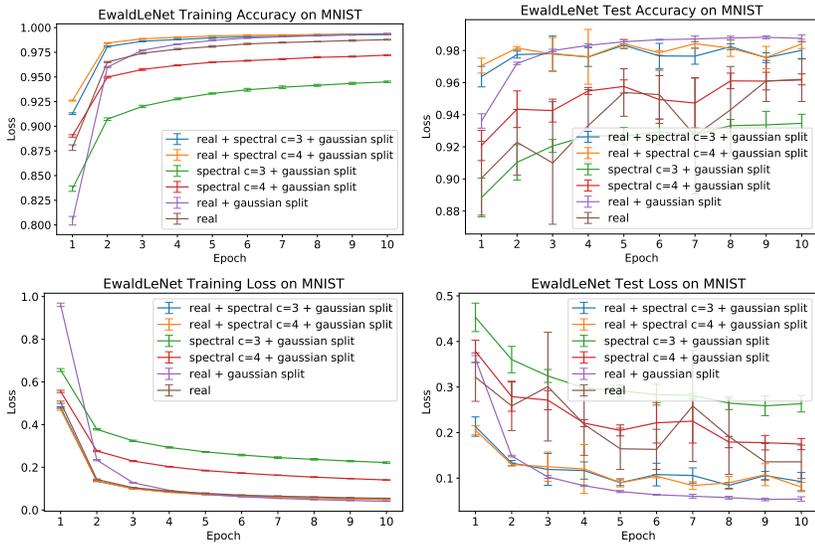


Figure 13: EwaldLeNet: Training progression on MNIST

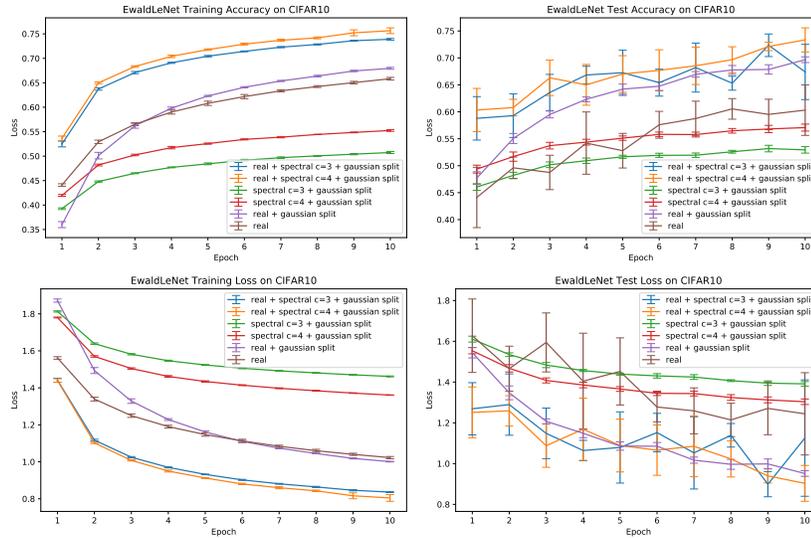


Figure 14: EwaldLeNet: Training progression on CIFAR10

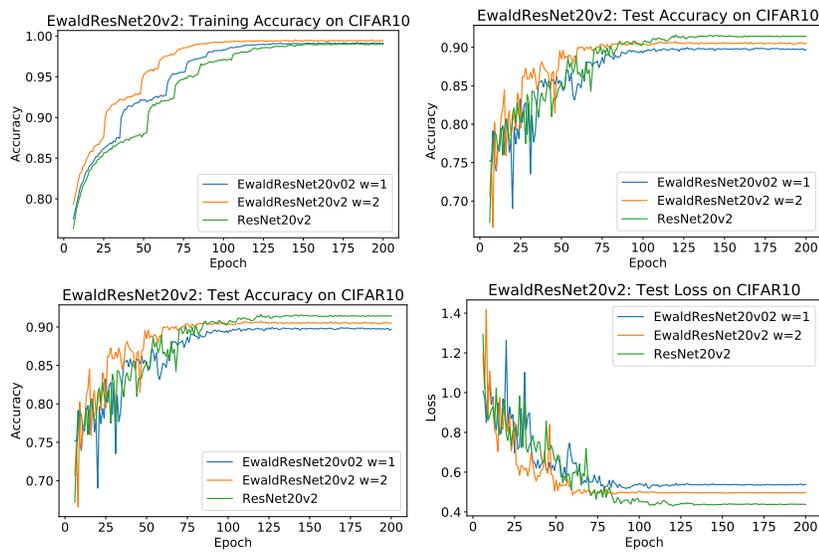


Figure 15: Training progression of the EwaldResNet ($n=2$) with width multipliers 1 and 2 vs the standard ResNet ($n=2$) on CIFAR10. Only a single run is reported

	#params	train acc	test acc	train loss	test loss
EwaldResNet20v2 w=1	1M	99.19	89.96	0.1330	0.5175
EwaldResNet20v2 w=2	1.9M	99.52	90.77	0.1319	0.4844
ResNet20v2	574k	99.08	91.62	0.1357	0.4292

Table 2: Number of parameters and best encountered loss and accuracy on the test and training sets during training of the EwaldResNets

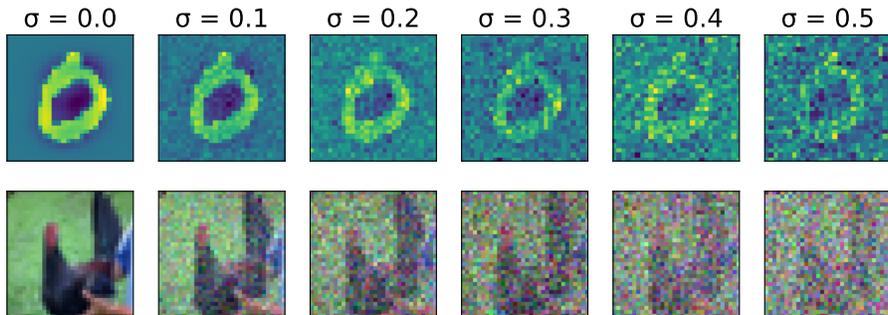


Figure 16: Example of images from MNIST (*top*) and CIFAR10 (*bottom*) with additional mean zero gaussian noise with increasing standard deviations

from 6.4 are evaluated on their respective test sets with artificial mean zero gaussian noise. The model performances were measured at standard deviations ranging from 0 to 0.5.

Examples of noisy images as used in this experiment are shown in Figure 16 and results are reported in Figure 17 (EwaldNet3) and Figure 18 (EwaldLeNet).

The models that solely use the spectral path can still perform well with noisy inputs. This is a promising result and shows that computation in the spectral space could lead to very robust features. However, this performance does not carry over when both spectral and real path is used. This indicates that the real path is dominating in these networks and some capacity in the spectral path might remain unused. An outlier is the “*real + gaussian split*” model on MNIST that is significantly more robust. The robustness seems to stem from the fact that the gaussian split removes the channel-wise mean (a model that explicitly removes the channel-wise mean while using both paths also show this robustness). However, this robustness does not emerge when training on CIFAR10 and seems to be dataset specific.

6.7 Sensitivity Towards High and Low Frequencies

To test if the EwaldBlocks are more sensitive towards global shapes than normal CNNs we evaluate model performance on high- and low-passed inputs at different cutoff frequencies. Examples of images at different cutoffs as used

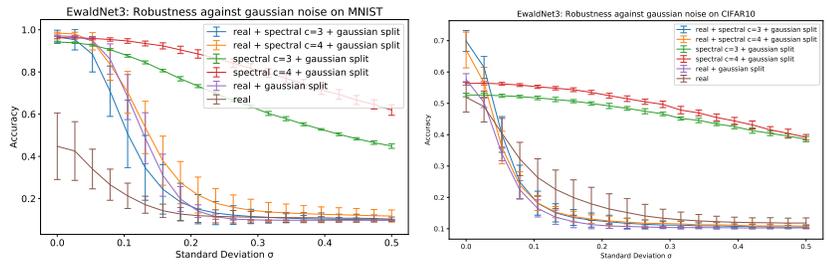


Figure 17: EwaldNet3: Robustness against zero mean gaussian noise with increasing variance on MNIST. Results were averaged over 5 models each trained for 10 epochs

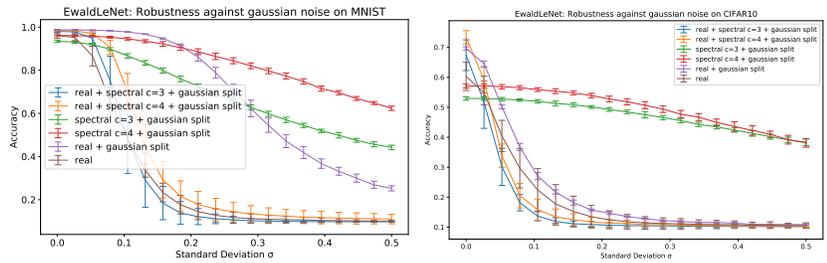


Figure 18: EwaldLeNet: Robustness against zero mean gaussian noise with increasing variance on MNIST. Results averaged over 5 models each trained for 10 epochs

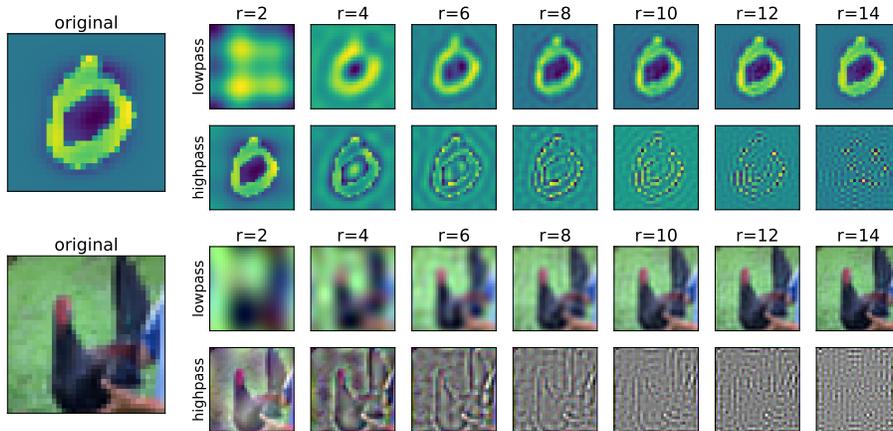


Figure 19: Example of images from MNIST (*top*) and CIFAR10 (*bottom*) low- and high-passed at different cutoff radii

in this experiment are shown in Figure 19. Results are shown in Figure 20 for EwaldNet3 and in Figure 21 for EwaldLeNet. Models only using the spectral path show much better performance at smaller cutoffs on the lowpass inputs, but equal to the noise robustness this property does not transfer to models using both paths.

All models using the gaussian split and the real path show better performance on high-pass inputs. This is expected, because the real path only has access to the high frequencies of the input when the split is used. But it also shows, that low frequencies are not necessary for shallow CNNs to perform well.

6.8 Network Analysis

In this section, the learned weights and the resulting activation maps of the '*real + spectral (c=4) + gaussian split*' EwaldLeNet are visualized. The model was trained on CIFAR10 for 10 epochs and achieves a test accuracy of 73.94%.

The learned spectral kernel weights of the pointwise multiplication are shown in Figure 22. For the visualization, the kernels were transformed back to the real space. The convolutional kernels of the real paths are shown in Figure 23. Further, the activation maps of the network is shown in Figure 24. For a comparison, the activation maps of the '*real*' EwaldLeNet - which is just a normal CNN - are shown in Figure 25. In Table 3, the learned widths of the gaussian kernel convolution for signal splitting are shown.

Activation maps after the first EwaldBlock from the real path are sharper when the spectral path is activated, and activations from the spectral path are very soft. Further, the activation maps from the last layer are more noisy than their counterparts of the CNN.

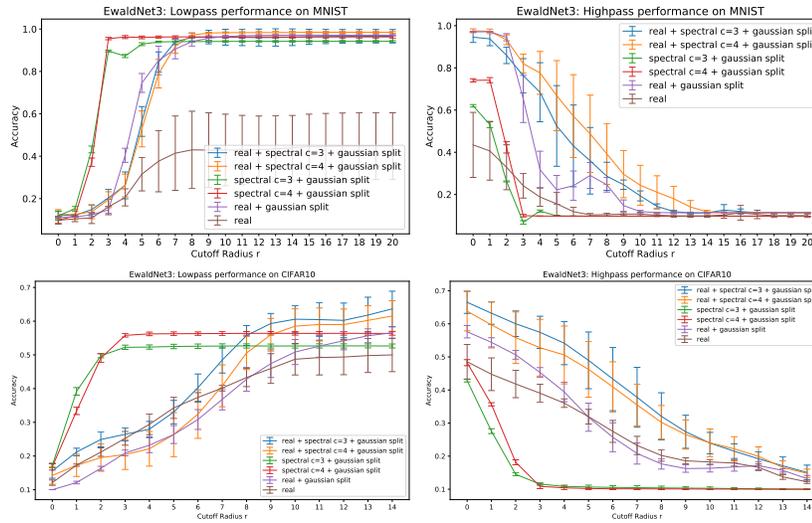


Figure 20: EwaldNet3: performance on low and high-passed input images of MNIST (*top*) and CIFAR10 (*bottom*). Results averaged over 5 models each trained for 10 epochs

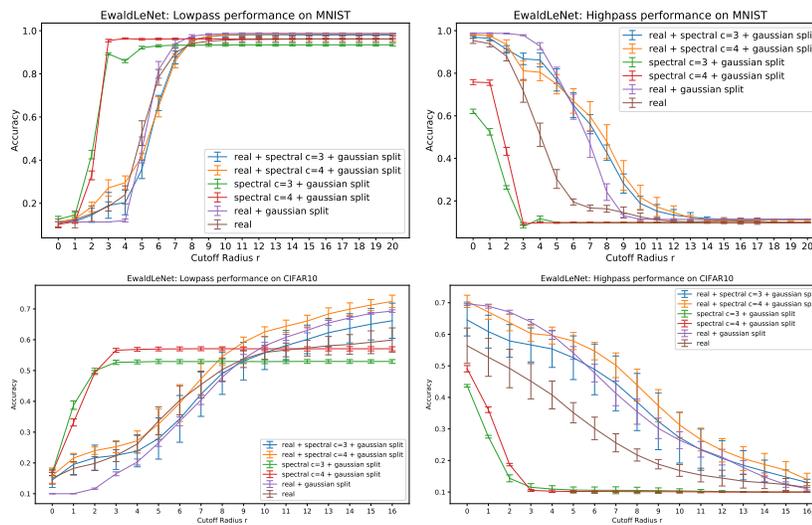


Figure 21: EwaldLeNet: performance on low and high-passed input images of MNIST (*top*) and CIFAR10 (*bottom*). Results averaged over 5 models each trained for 10 epochs

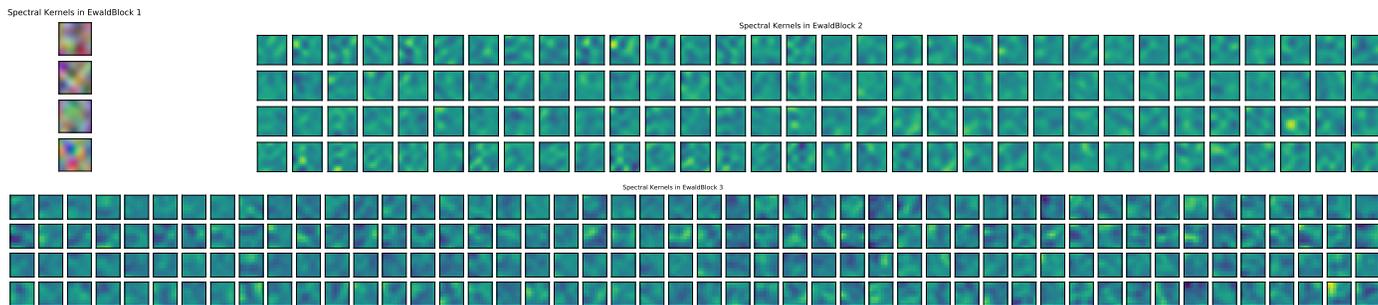


Figure 22: 'real + spectral ($c=4$) + gaussian split' EwaldLeNet: Learned Filters of the first 4 pointwise multiplication kernels in the spectral path

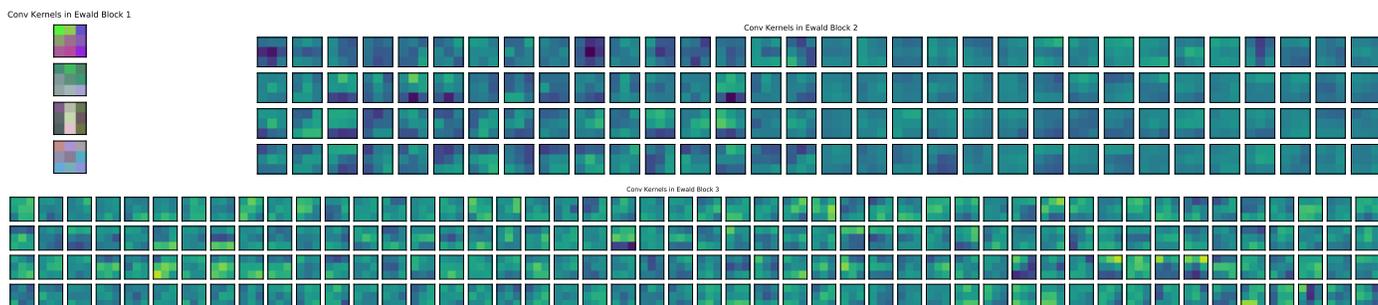


Figure 23: 'real + spectral ($c=4$) + gaussian split' EwaldLeNet: Learned Filters of the first 4 convolutional kernels in the real path

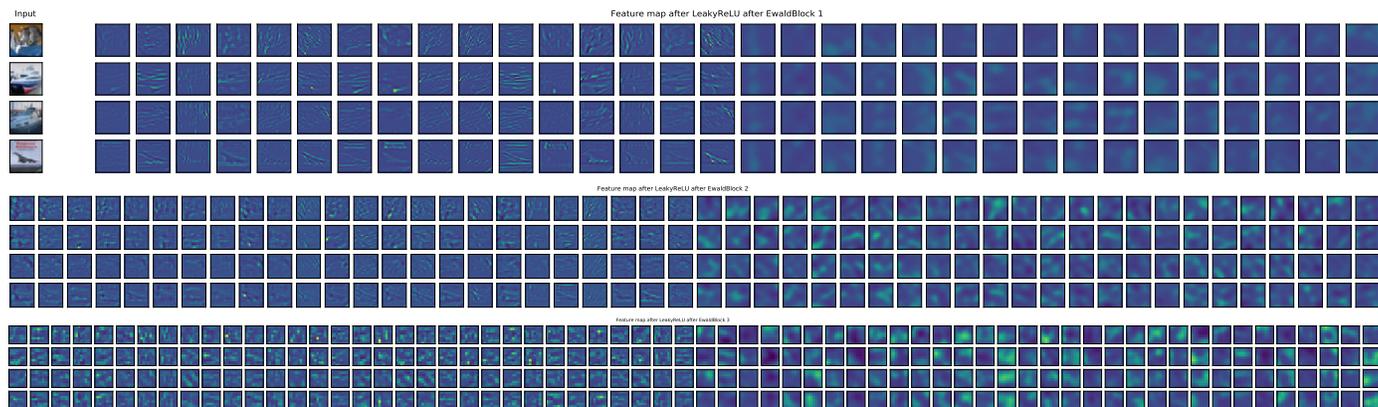


Figure 24: 'real + spectral ($c=4$) + gaussian split' EwaldLeNet: Activation maps after the LeakyReLU activation function was applied on the concatenated output of the real and spectral path. The left half of the shown kernel plots contains the outputs of the real path (the convolutions) and the right half the outputs of the spectral path

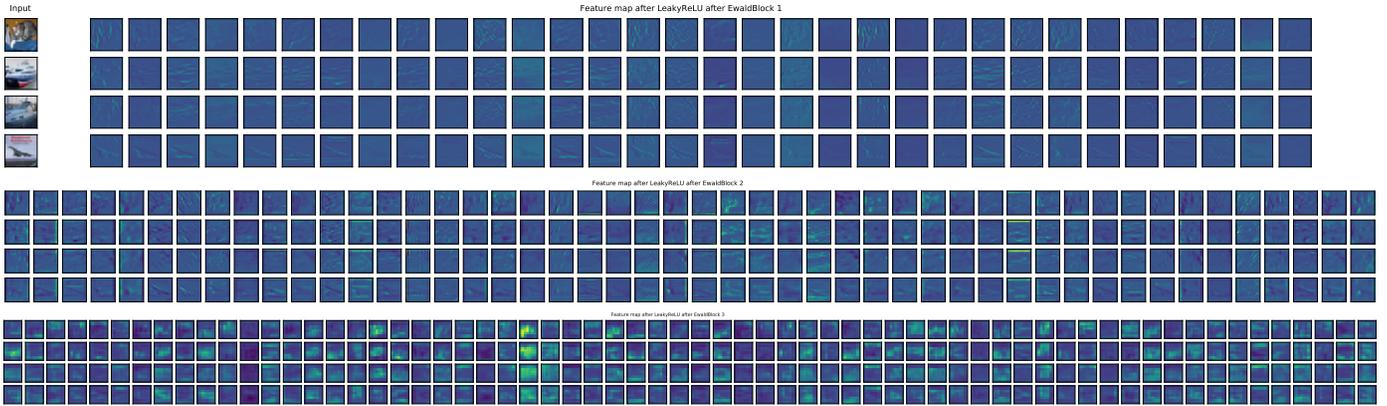


Figure 25: 'real' EwaldLeNet: Activation maps after the LeakyReLU activation function

	<i>real + spectral (c=4) + gaussian split</i>	<i>spectral (c=4) + gaussian split</i>	<i>real + gaussian split</i>
EwaldBlock 1	1.035	0.165	1.112
EwaldBlock 2	1.088	0.170	4.521
EwaldBlock 3	0.899	1.603	4.521

Table 3: Learned σ of the gaussian kernels in the splits from the best performing model of each variant of 5 runs

In Table 3 the learned widths σ of the gaussian kernels in each EwaldBlock of the best performing EwaldLeNet on CIFAR10 are shown.

7 Discussion and Outlook

In this thesis, I have shown that that the proposed EwaldBlock can make shallow CNNs perform better and that global features from the spectral path could lead to very robust models. However, there remains room for improvement and several open questions remain.

For one, the constant back and forth between real and spectral domain in the spectral path of the EwaldBlock gives significant computational overhead. Finding a way to do its whole computation in the spectral path would open many possibilities. One obstacle in that regard is, that it is unclear what a sensible activation function in the spectral space could be.

Another thing to look into, is the merging of the computational paths. I believe that the current approach is not well suited, as the signal split returns the global features back to the spectral path, as they are relatively smooth and similarly local features get returned back to the real path due to them being mostly high frequency. This means that only little information can actually be

shared between the paths.

On an other note, EwaldBlocks still have to be evaluated on datasets where global information is of high importance for solving the task. The benchmark datasets in this thesis have small input images where even shallow CNNs can reach sufficient receptive fields to grasp global features. This leaves little room for improvement with our approach on those datasets. However, datasets such as ImageNet or Places2 that would be a good benchmark are very large and their training exceeded the scope of this thesis.

Using cosine transformations instead of Fourier transformations could also be beneficial, as pointwise multiplication in the cosine domain is equal to a convolution with symmetric padding. This would be more sensible for images (and other non periodic data) compared to the assumed periodicity of the Cyclic Convolution Theorem and may make it possible to use less padding in the spectral path.

Further, the EwaldBlock could easily be adapted to use efficient convolution variants such as group [39] or depth-wise [18] convolutions that aim to reduce redundancy in convolutional kernels and have been shown to increase computational efficiency in CNNs. By design of the EwaldBlock, the depth-wise separation can also be used in the spectral path.

List of Figures

1	Drawing of a dense Neural Network with one hidden layer . . .	7
2	A filter (<i>blue</i>) is convolved over the 3D input feature map (<i>left</i>) computing its 2D activation map (<i>right</i>)	11
3	Example of how an input signal can be decomposed into its underlying frequencies with the Fourier transformation: (<i>left</i>) the input signal; (<i>middle</i>) the magnitudes of the Fourier transformed signal, note the symmetry of the spectral components; (<i>right</i>) the decomposed signal received applying the inverse Fourier transform on single magnitudes and their complex conjugate (the symmetric counterpart)	13
4	(<i>first column</i>) A two dimensional DFT is applied channel-wise to a RGB image of size 320x320 pixels. In the spectral representation is shifted s.t. the 0^{th} frequency component $X_{0,0}$ is in the center of the image. (<i>subsequent columns</i>) the spectral coefficient matrix is truncated to $\frac{1}{4}$ the size of the previous column by removing the highest frequency components. The resulting matrix transformed back to the real space yielding a compressed (but blurred) versions of the original image (shown in the last row)	16
5	High level concept of Ewald Summation and the proposed Ewald-Block	21
6	Architecture of the proposed EwaldBlock	22
7	Computation flow for the spectral path with output shapes of each component	24

8	Schematic of spectral path computation	24
9	Architecture of the EwaldNet3 network	26
10	Architecture of the EwaldLeNet network	27
11	EwaldNet3: Training progression on MNIST. For clarity, metrics on the test data are shown twice in the middle and last column, with the later skipping the results of poor performing 'real' EwaldNet3	29
12	EwaldNet3: Training progression on CIFAR10	30
13	EwaldLeNet: Training progression on MNIST	30
14	EwaldLeNet: Training progression on CIFAR10	31
15	Training progression of the EwaldResNet (n=2) with width multipliers 1 and 2 vs the standard ResNet (n=2) on CIFAR10. Only a single run is reported	31
16	Example of images from MNIST (<i>top</i>) and CIFAR10 (<i>bottom</i>) with additional mean zero gaussian noise with increasing standard deviations	32
17	EwaldNet3: Robustness against zero mean gaussian noise with increasing variance on MNIST. Results were averaged over 5 models each trained for 10 epochs	33
18	EwaldLeNet: Robustness against zero mean gaussian noise with increasing variance on MNIST. Results averaged over 5 models each trained for 10 epochs	33
19	Example of images from MNIST (<i>top</i>) and CIFAR10 (<i>bottom</i>) low- and high-passed at different cutoff radii	34
20	EwaldNet3: performance on low and high-passed input images of MNIST (<i>top</i>) and CIFAR10 (<i>bottom</i>). Results averaged over 5 models each trained for 10 epochs	35
21	EwaldLeNet: performance on low and high-passed input images of MNIST (<i>top</i>) and CIFAR10 (<i>bottom</i>). Results averaged over 5 models each trained for 10 epochs	35
22	'real + spectral (c=4) + gaussian split' EwaldLeNet: Learned Filters of the first 4 pointwise multiplication kernels in the spectral path	36
23	'real + spectral (c=4) + gaussian split' EwaldLeNet: Learned Filters of the first 4 convolutional kernels in the real path	36
24	'real + spectral (c=4) + gaussian split' EwaldLeNet: Activation maps after the LeakyReLU activation function was applied on the concatenated output of the real and spectral path. The left half of the shown kernel plots contains the outputs of the real path (the convolutions) and the right half the outputs of the spectral path	36
25	'real' EwaldLeNet: Activation maps after the LeakyReLU activation function	37

List of Tables

1	EwaldNet3 and EwaldLeNet results on CIFAR10 and MNIST. Reported value is the mean best accuracy on the test set of 5 models. The number of parameters and time per epoch is for the models on MNIST, and time per epoch is on a Nvidia Tesla P100 GPU as provided by Kaggle	29
2	Number of parameters and best encountered loss and accuracy on the test and training sets during training of the EwaldResNets	32
3	Learned σ of the gaussian kernels in the splits from the best performing model of each variant of 5 runs	37

References

- [1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.
- [2] Johannes Ballé, Valero Laparra, and Eero P. Simoncelli. End-to-end optimized image compression. *CoRR*, abs/1611.01704, 2016.
- [3] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.
- [4] Wieland Brendel and Matthias Bethge. Approximating CNNs with bag-of-local-features models works surprisingly well on imagenet. In *International Conference on Learning Representations*, 2019.
- [5] Chun-Fu Chen, Quanfu Fan, Neil Mallinar, Tom Sercu, and Rogério Schmidt Feris. Big-little net: An efficient multi-scale feature representation for visual and speech recognition. *CoRR*, abs/1807.03848, 2018.
- [6] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen. Compressing convolutional neural networks in the frequency domain. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16*, pages 1475–1484, New York, NY, USA, 2016. ACM.
- [7] Yunpeng Chen, Haoqi Fan, Bing Xu, Zhicheng Yan, Yannis Kalantidis, Marcus Rohrbach, Shuicheng Yan, and Jiashi Feng. Drop an octave: Reducing spatial redundancy in convolutional neural networks with octave convolution. *CoRR*, abs/1904.05049, 2019.

- [8] François Chollet et al. Keras, 2015. Software available from keras.io.
- [9] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.
- [10] Tom Darden, Darrin York, and Lee Pedersen. Particle mesh ewald: An $n \log(n)$ method for ewald sums in large systems. *The Journal of chemical physics*, 98(12):10089–10092, 1993.
- [11] Daan Frenkel and Berend Smit. *Understanding molecular simulation: from algorithms to applications*, volume 1. Elsevier, 2001.
- [12] Robert Geirhos, Patricia Rubisch, Claudio Michaelis, Matthias Bethge, Felix A. Wichmann, and Wieland Brendel. Imagenet-trained cnns are biased towards texture; increasing shape bias improves accuracy and robustness. *CoRR*, abs/1811.12231, 2018.
- [13] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.
- [14] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
- [15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.
- [16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.
- [17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *CoRR*, abs/1603.05027, 2016.
- [18] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.
- [19] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Q. Weinberger. Multi-scale dense convolutional networks for efficient prediction. *CoRR*, abs/1703.09844, 2017.
- [20] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.
- [21] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

- [22] Tsung-Wei Ke, Michael Maire, and Stella X. Yu. Neural multigrid. *CoRR*, abs/1611.07661, 2016.
- [23] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *CoRR*, abs/1609.04836, 2016.
- [24] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [25] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.
- [26] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *CoRR*, abs/1312.4400, 2013.
- [27] Tsung-Yi Lin, Piotr Dollár, Ross B. Girshick, Kaiming He, Bharath Hariharan, and Serge J. Belongie. Feature pyramid networks for object detection. *CoRR*, abs/1612.03144, 2016.
- [28] Stephen A Martucci. Symmetric convolution and the discrete sine and cosine transforms. *IEEE Transactions on Signal Processing*, 42(5):1038–1051, 1994.
- [29] Michaël Mathieu, Mikael Henaff, and Yann Lecun. Fast training of convolutional networks through ffts. In *International Conference on Learning Representations (ICLR2014), CBLS, April 2014*, 2014.
- [30] Andrew Ng. Normalizing activations in a network. https://www.youtube.com/watch?v=tNIpEZLv_eg.
- [31] Frank Noe. Lecture notes for computational science ws 18/19. *Institute for Mathematics und Computer Science, Free University Berlin*, 2018.
- [32] B Osgood. Lecture notes for ee 261: the fourier transform and its applications. *Electrical Engineering Department, Stanford University*, 2013.
- [33] Oren Rippel, Jasper Snoek, and Ryan P Adams. Spectral representations for convolutional neural networks. In *Advances in neural information processing systems*, pages 2449–2457, 2015.
- [34] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *CoRR*, abs/1505.00387, 2015.
- [35] Antonio Torralba and Aude Oliva. Statistics of natural image categories. *Network: computation in neural systems*, 14(3):391–412, 2003.
- [36] Chiheb Trabelsi, Olexa Bilaniuk, Dmitriy Serdyuk, Sandeep Subramanian, João Felipe Santos, Soroush Mehri, Negar Rostamzadeh, Yoshua Bengio, and Christopher J. Pal. Deep complex networks. *CoRR*, abs/1705.09792, 2017.

- [37] Huiyu Wang, Aniruddha Kembhavi, Ali Farhadi, Alan L. Yuille, and Mohammad Rastegari. ELASTIC: improving cnns with instance specific scaling policies. *CoRR*, abs/1812.05262, 2018.
- [38] Yunhe Wang, Chang Xu, Shan You, Dacheng Tao, and Chao Xu. Cnnpack: Packing convolutional neural networks in the frequency domain. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 253–261. Curran Associates, Inc., 2016.
- [39] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. *CoRR*, abs/1611.05431, 2016.
- [40] Hao Zhang and Jianwei Ma. Hartley spectral pooling for deep learning. *CoRR*, abs/1810.04028, 2018.
- [41] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. Pyramid scene parsing network. *CoRR*, abs/1612.01105, 2016.