Activating frequencies

Exploring non-linearities in the Fourier domain

W.R. Uijens

Master thesis
Activating frequencies
Exploring non-linearities in the Fourier domain

by

W.R. Uijens

to obtain the degree of Master of Science in Computer Engineering at the Delft University of Technology, to be defended publicly on Friday April 13, 2018 at 10:00.

Student number: 4356799
Project duration: January 1, 2017 – January 31, 2018
Thesis committee: Prof. Dr. Marcel Reinders, TU Delft
Dr. Jan van Gemert, TU Delft
Dr. Klaus Hildebrandt, TU Delft
Dr. Gert-Jan Burghouts, TNO

An electronic version of this thesis is available at http://repository.tudelft.nl/.
After an internship at the intelligent imaging department of TNO, I knew I wanted to continue in the world of machine learning. My supervisor there Dr. Gertjan Burghouts invited me to also do my thesis at TNO. Before I could start, I needed a supervisor from the TUDelft. Luckily Dr. Jan van Gemert, head of the computer vision lab at the TUDelft, was willing to supervise me. It was Jan who came up with the challenging idea of trying to find an activation function in the Fourier domain. This was an scary but exciting subject, since we truly couldn’t even guess what we would find.

During this project I was given the trust to independently come up with ways to approach this fundamental research question. However to paraphrase the famous saying: it takes a village to write a thesis. All throughout this project Jan and Gertjan have supported me, advised me and guided me. On top of that the colleagues during my stay at TNO helped me tremendously. Wouter Oosterheert, who coordinated all the interns at the department and organized a weekly deep learning stand-up, and Dr. Maarten Kruithof with his extensive deep learning knowledge deserve special mention.

At the TUDelft side, I am also very grateful for the help of Dr. Seyran Khademi. In the final stages of the project our early morning meetings at Utrecht central station were invaluable.

Finally, the loving support of my girlfriend Rosalie made me smile after even the worst set-backs. I could not have done it without you.

W.R. Uijens
Delft, April 2018
**Contents**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>iii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Theoretical background</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Object recognition</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Machine learning</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Deep neural networks</td>
<td>9</td>
</tr>
<tr>
<td>2.4 Convolutional Neural networks</td>
<td>13</td>
</tr>
<tr>
<td>3 Problem statement</td>
<td>15</td>
</tr>
<tr>
<td>3.1 Reducing computational load</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Pointwise activation functions</td>
<td>16</td>
</tr>
<tr>
<td>4 Method</td>
<td>19</td>
</tr>
<tr>
<td>4.1 Considerations in experimental set-up</td>
<td>19</td>
</tr>
<tr>
<td>4.2 Experimental architecture</td>
<td>21</td>
</tr>
<tr>
<td>4.3 Training procedure</td>
<td>23</td>
</tr>
<tr>
<td>5 Results</td>
<td>25</td>
</tr>
<tr>
<td>5.1 Spatial activation functions in the Fourier domain</td>
<td>25</td>
</tr>
<tr>
<td>5.2 Learned activation functions</td>
<td>27</td>
</tr>
<tr>
<td>5.3 necessary conditions</td>
<td>29</td>
</tr>
<tr>
<td>6 Conclusion</td>
<td>33</td>
</tr>
<tr>
<td>Bibliography</td>
<td>37</td>
</tr>
</tbody>
</table>
Introduction

One of the major research interests in artificial intelligence (AI) community over the past few decades is teaching a computer to 'see'. Or more specifically: to teach a computer to recognize classes of objects in images. Does a given image contain a cat, a dog, a train or a fork? Algorithms that can achieve this can be applied in a wide range of applications. Examples of possible applications include detecting road signs for self-driving cars, detecting animals for wild-life monitoring projects and detecting faulty products in the manufacturing industry. Specifically, the field of solving these kinds of problems is called computer vision.

The set of algorithms that are currently achieving state of the art performance in computer vision are called Convolutional Neural Nets (CNNs). These algorithms are a subset of so-called deep learning algorithms. The algorithms under consideration in this thesis are decades old. Work on CNNs started in the late 1980s/early 1990s\(^1\). Yet looking back, the origin might be tracked all the way to the early 1960s investigations into the visual cortex of a cat\(^2\). However the accuracy of and interest in CNNs exploded around 2011. First of all this was the time that a seminal dataset was published: ImageNet\(^3\). This dataset consists of a million images labeled with 1 of 1000 potential classes each. It is in contexts like this, massive amounts of data that CNNs can truly shine.

However tuning (or: training) these algorithms to work on these large datasets takes large amounts of computation. This means that exploiting these large amounts of data is infeasible on CPUs. However at the same time, people started to use GPU and started harnessing their massive parallel computation abilities (originally developed for video games). This all came together in one of the seminal papers of the field\(^4\). In the classification challenge of ImageNet, this papered scored 15.3% error rate\(^1\), where the next competitor

\(^1\)The error-rate in this case is the top-5 error-rate. At times the images can be ambiguous and a single image might fit multiple categories. To account for this, the algorithms get to guess 5 times for each images which of the 1000 classes it fits in. If the right class is one of the 5, that counts as having it right.
scored 26.2%. The following years the field developed quickly and now the current state of the art is an error rate of 2.2%.

Since the error-rate of these CNNs drop rapidly, with each new variant introduced, other concerns come to the forefront. As stated, these CNNs can take incredible computational power use to their full potential. For example, researcher at Facebook were able to train a CNN on ImageNet in a single hour, however they needed 352 GPUs to do that[5]. This shows that the amount of computational power the implementation of a CNN algorithm (commonly called its 'architecture') needs is an important factor to consider.

One way to speed up CNNs is to speed up their convolutions, which account for approximately 90% of the computational cost in practice[6]. This can be done by replacing the convolutions by the equivalent multiplications in the frequency domain[7]. Depending on the exact architecture of the CNN, this can give major gains in computational efficiency.

However the result is that now most computations are spent going back and forth to the frequency domain, using Fast Fourier Transforms (FFTs)[8]. Now the question arises whether all parts of the CNN can be implemented in the frequency domain, making the FFTs obsolete and gaining even more computational efficiency. The answer is that most components of the CNN algorithm have counterparts implemented directly in the frequency domain. Only 1 component still needs a frequency domain counterpart, before CNNs working in the frequency domain can be designed. Several papers already noticed this and mentioned finding such a counterpart as future work[7][9][10]. The component in question is the so-called activation function or non-linearity.

This thesis is as far as this author is aware the first published investigation into an alternative for the activation function in the frequency domain.

The main contributions of this thesis are:

1. experimental results on a variety of activation functions tested in the frequency domain and compared against both a spatial counterpart and having no activation function at all.

2. a fundamental argument addressing the limitations of classic pointwise activation functions in the frequency domain.

3. a set of requirements which must be met for an activation function in the frequency to be competitive against their spatial counterparts.

The rest of this thesis is structured as followed: first in chapter 2 an in depth overview is given of CNNs. Then in chapter 3 the problem of finding an activation function is described in depth. Next in chapter 4 the experimental set-up is described. After that the results of the experiments on various activation functions and some theoretical contributions are described in chapter 5. Finally chapter 6 contains the conclusion.
2

Theoretical background

In this thesis the problem under consideration is Convolutional neural networks (CNNs)[1] and an investigation into an approach to make them less computationally intensive. Before this approach is discussed, it is good to first discuss what CNNs are, what kind of problems they solve and why CNNs in general and faster CNNs in particular are useful.

CNNs in different configurations can be used to solve a wide variety of problems (e.g. natural language processing[11]). However in this thesis only CNNs that categorize image data are considered. The idea is that the input of the algorithm is some kind of image data and the desired output is some kind of labeling of that image data.

Usually that general problem of categorizing image data is sub-divided depending on the kind of output that is desired. The desired output of a given problem usually falls into one of 3 categories: (image) classification, object detection and object segmentation. These are problems where humans until recently far outperform computers, and still human performance is a major benchmark in this area of research[12]. Therefor the difficulty in solving these problems algorithmically might be unintuitive. That is why this chapter is set-up as followed.

The first paragraph is going to explain the definitions of image classification, object detection and object segmentation. Then the next paragraph is going to give an intuitive understanding of the difficulties of solving these problems. The final 2 paragraphs are then going to be about the algorithms that are currently used to solve these problems. Those are respectively going to be about the general class of machine learning algorithms and about the convolutional neural networks specifically.

Suppose one has an image of interest. Now image classification is the problem of given a set of labels, to assign the correct label to that image. Object detection is the problem of given a set of objects of interests, to find each instance of the objects of interests in the image, together with the location of each object of interest in the image. Finally object segmentation is the problem of given a set of labels, to classify each pixel of the image with the correct
To illustrate these definitions, it is helpful to consider an example. Suppose the images of interests are pictures of cats and dogs made by an animal shelter specializing in these 2 species. A possible image classification problem would be to label the images containing a cat as 'cat' and the images containing a dog as 'dog'. This might be useful if it is known that each image contains exactly one cat or one dog and to automatically label which species each image contains.

However there are cases in which image classification is not the right approach. Suppose pictures are taken of large groups of cats and dogs. Here a single label might not be appropriate. More useful might be to identify each cat and dog in the picture individually. In this case cats and dogs would be the objects of interest, all objects of interest would need to be found in the picture and then correctly labeled cat or dog. Here found usually means that the smallest rectangle is given that fully encompasses the object of interest. This would be an object detection problem.¹

At times, an even more granular output is required than the bounding boxes given by an object detection algorithm. Suppose that the problem if interest is to remove the background in an image of cats and dogs. That means it is necessary to determine at a pixel level which pixel is part of an object of interest (in our example a cat or a dog), and which pixels are not part of an object of interest and therefor background. This labeling of each pixel as either a specific object of interest (in our example: as part of a cat or a dog) or as part of the background is called object segmentation.

¹Note that image classification can be seen as a simplified version of object detection. The difference is that in image classification there is the constraint of there only being 1 object of interest in the input image and no bounding box required to be given in the output.
2.1. Object recognition

Initially, it might not seem obvious why a task that comes so easy to most humans would be so hard to teach computers. To gain some intuition in the difficulties faced when trying to solve such a problem, consider the problem of recognizing whether an image contains a cat. One might describe a cat as having: fur, 2 eyes, a mouth, a tail, 4 legs etc. However this description immediately leads to problems. First of all a cat might not have some of these features (there are cats without fur, without a tail, missing an ear or a leg, etc.). On top of that features that the cat does have might not be visible on the image. Features might be occluded or simply outside the field of view when the image was made (see figure 2.1 for examples). And even if they are not occluded, there is the problem of describing these features of a cat: how would one, for example, give a general description of a cat's ear?

When thinking about this problem of describing a cat, it is good to keep in mind what an image is. The images under consideration in this thesis consist of a grid of pixels. Each of these pixels consist of either 1 (gray-scale) or 3 (red, green and blue) color channels. In this thesis these images are represented as 3-dimensional tensors\(^2\), where the dimensions of the tensors represent the width, height and color channels of the image. So in short, the input of any process that needs to recognize a cat in an image in this context is a 3-dimensional tensor.

Now consider what the output of a program that recognizes cats in pictures would need to be. One option is a binary output which is 1 if the program detected a cat and 0 otherwise. However this might be a bit ambitious. At times images can be ambiguous and even humans might disagree whether a given image (without any other context) contains a cat. A safer bet would be to output a value between 0 and 1. Then the program could not only output whether it ‘thinks’ there is a cat in the image, but also signal how certain it is of its judgment. Values close to 1 would signal high confidence that there is indeed a cat, and conversely low values close to 0 would signal high confidence that there isn’t a cat. Values in between would signal various degrees of doubt.

The choice for an interval between 0 and 1 as output also helps to distinguish between multiple classes. Suppose it is known that there is an animal in a given image, but it is necessary to classify which of 10 given species it belongs to. This can be solved by building a detector that gives its confidence of the presence of each of the 10 species individually. Then the most confident detector would be the output (e.g. if the highest output came from the dog-detector, the animal would be classified as a dog).

In summary, the problem of object recognition is to write a function that takes as input an image in the form of a tensor. This function needs to output for each class in a given list of possible classes how confident it is that the

---

\(^2\)In this context, tensors can be understood as a generalization of 2-dimensional matrices to higher dimensions.
image belongs to that class. This allows the computer to make a final class-
sification of the image by selecting the class with the highest corresponding
confidence score.
However the challenge remains to design such a function.

2.2. Machine learning
The central question in object classification is now how to design a function
that takes in an image in the form of a tensor and output a value close 1 if
the image is of a given class and close to 0 otherwise. At the heart of almost
every approach to solve that question is machine learning, and in most cases
specifically: supervised learning.
Supervised learning[13] can be summarized as □each by example(s).□ To use
supervised learning, a large set of images which are labeled with their correct
category is necessary. So in the example of wanting to distinguish between
cats and dogs, a large set of cat images labeled as cat and a large dataset of dog
images labeled as dog is necessary. Next this dataset is used as a training-set
to train our algorithm in distinguishing between cats and dogs.
In general a machine learning algorithm consists of a set of parameters \( \theta \) and
a function \( f_{\theta} : I \rightarrow O \). Here \( I \) and \( O \) are the input- and output-domain for
the specific problem at hand. In our example of classifying images as either
containing a cat or a dog, the input is \( I = \mathbb{R}^H \times \mathbb{R}^W \times \mathbb{R}^C \), where \( H \) and \( W \) are
the height and width of the images. Then \( C \) is the number of color-channels:
3 for standard rgb-images and 1 for gray-scale images. The output is \( O = \mathbb{R} \times \mathbb{R} \).
Here each real number gives a confidence score for each class. The
first number represents the confidence that the image is a cat, the second
number that the image is a dog.\(^3\)
The challenge is now to pick an optimal set of parameters \( \theta \) that makes \( f_{\theta} \) the
best possible classifier. To do this, a set of \( N \) labeled images is used. Lets call
this ordered set of images of cats and dogs \( I \). Now construct a ordered set of
labels for these images as followed: if an image contains a cat label it \((1,0)\)
and if the image contains a dog label it \((0,1)\). Call this set of label: \( L \)
Now the average error \( f_{\theta} \) makes for any given \( \theta \) can be quantified as followed:

\[
error(\theta) = \frac{1}{N} \sum_{i=1}^{N} \| f_{\theta}(I_i) - L_i \| \tag{2.1}
\]

The instances of \( f \) considered in this thesis have as extra property that they
are either differentiable or piece-wise differentiable. Specifically, this allows
us to take the partial derivatives of \( error(\theta) \) w.r.t. \( \theta \). One straightforward way
to use this property is gradient descent\(^4\). This is an iterative process where

\(^3\)Note that the ordering is an arbitrary choice.
\(^4\)The concept of using gradients in optimization problems has been around for centuries. This concept
was first made practically applicable in the general deep learning context in [14]. Then it was first used
in the specific CNN context in [15]. For a full history of this subject, please refer to professor Jürgen
Schmidhuber in [16].
for each step the following is done:

$$\theta \leftarrow \theta - \eta \frac{\partial \text{error}(\theta)}{\partial \theta}$$  \hspace{1cm} (2.2)

Here $\eta$ is the learning rate: a small number which determines the step-size of this iterative process. Now in practice it has been found that this optimization method works well\cite{17}. However many variations have popped-up on this approach that tried to improve on it. Examples are: ADAGRAD\cite{18}, RMSPROP\cite{19}, ADADELTA\cite{20} and ADAM\cite{21}. The main focus of these advancements is to automatically adjust the learning-rate $\eta$ for each parameter individually. There is no clear best optimizer which performs best in all circumstances, but ADAM usually works well\cite{22}.

After finding the optimal parameters $\theta_{opt}$, it is important to have a measure of how accurate the final function $f_{\theta_{opt}}$ is. One straightforward way to do this is to test the accuracy of $f_{\theta_{opt}}$ on the training set. What percentage of images in the training set does $f_{\theta_{opt}}$ classify correctly? However this is a very poor way of measuring the accuracy of $f_{\theta_{opt}}$. Since the classification of each image in the training set is already known, classifying them again is not very useful. The real goal of optimizing $f_{\theta}$ is to classify images that have not been previously classified. And there might be very large difference between the accuracy on the seen training-set and unseen new data, a phenomena called overfitting.

Overfitting is the process where during the training of $f^5$, spurious patterns are found in the training set. These spurious patterns can lead to great accuracy on the training set. However because they don’t (necessarily) hold in new data, can lead to drastically lower accuracy on new data.

Going back to the cats and dogs example: suppose pictures were taken of a large set of cats and dogs to create the training-set. First during the afternoon, the pictures of the cats were taken. Then later in the day, as the light was fading, the picture of the dogs were taken\textsuperscript{6}. This can have as a result that the pictures of the cats are much brighter than the pictures of the dogs. For such a training-set, a very simply function might be able to distinguish between cats and dogs. Simply adding all pixels and classifying all images above a threshold as cats and below threshold as dogs could lead to close to 100% accuracy. But for new data this pattern would not necessarily hold. In general pictures of cats and dogs can be taken in any lighting condition, so in new data this function might not even perform better than random guessing. The function has ‘overfit’ on the training-set.

The first defense against overfitting is making sure the training-set is large and equally diverse as the data the function is going to be applied on after training. However even on a large and diverse training-set, overfitting can still

\textsuperscript{5}The process of finding the optimal parameters $\theta$ for $f$ is usually referred to as ‘training’

\textsuperscript{6}Taking pictures of cats and dogs under different lightning conditions might seem like an obvious error in hindsight. However a persistent anecdote in the machine learning community tells a story exactly like this. However instead of categorizing cats and dogs, the story is about categorizing bushes with or without a camouflaged tank. For an overview of the anecdote and an inquiry into its accuracy, see [23]
occur. To get an accurate indication of the accuracy of the training classification function, it is important to test this accuracy on a separate a test set. Before training begins, the labeled dataset needs to be split in a training-set and a test-set. The test-set is then put aside and during training only the training-set is used. Then after the training is completely finished, the test-set is used to evaluate the final accuracy of the trained function[24].

The downside of this approach is that only 1 set-up can be tested. Only one choice for a function $f$ can be made and only one choice for optimizer, learning-rate etc. If we try multiple variations here and keep what works best on the test-set, we might still overfit on the test-set. That is why usually the labeled dataset is usually split into 3 parts: a training-set and test-set as before and then a validation set. This validation set is then used to validate whether choice we make in designing our function $f$ and in setting up the training procedure have a beneficial effect. Then at the end, when we trained the final network that performed best on the validation set, only then do we test our final accuracy on the test-set.

Fundamentally the problem of overfitting is part of the more general question of how much freedom we should give to the function $f$ in finding the optimal classification scheme. Each instantiation of the parameters $\theta$ corresponds to a possible classification function $f_\theta$. More parameters means that the search space of possible functions $f_\theta$ gets larger. This in turn increases the chances that the search space contains functions which abuse spurious patterns in the training-set. If the optimization algorithms subsequently identify such a function as the function which performs optimally on the training-set, overfitting occurs.

Conversely, decreasing the number of parameters might lead to underfitting. The lower number of parameters means that the space of possible functions $f_\theta$ decreases. This in turn increases the risk that the function(s) $f_\theta$ which describes the actual pattern in the data (e.g. the actual differences between cats and dogs) is no longer in the search space. If all functions that describe the actual pattern in the data (very well) are removed, the best functions left to find for the optimization algorithm will necessarily perform worse. The corresponding drop in accuracy is called underfitting.

The trade-off between overfitting and underfitting is a central problem in machine learning. Finding that Goldilocks zone where the search space of possible function $f_\theta$ is large enough to include functions that capture every aspect of the patterns that reside in the data. However a search space that is at the same time small enough to not include any functions that abuse any spurious patterns that only reside in the training-set. Later in this chapter convolutional neural networks will be discussed. These networks were specifically designed to tailor the search spaces of deep neural networks (DNNs) to the Goldilocks zone of images. The rest of this chapter will be focused on first explaining DNNs and then on their variant CNNs specifically.
2.3. Deep neural networks

Deep Neural Networks (DNNs) are a subset of machine learning algorithms. From an historical perspective, they can be viewed as multi-layer perceptron. To introduce DNNs, first an introduction of the perceptron is therefor in order. The perceptron\cite{25} is a binary classifier. Its input $I \in \mathbb{R}^N$ is a set of real numbers, represented as a vector. Then its output is:

$$ f_{(w,b)}(I) = \begin{cases} 1 & \text{if } w^T \cdot I + b > 0 \\ 0 & \text{otherwise} \end{cases} $$

Here $w \in \mathbb{R}^N$ and $b \in \mathbb{R}$. Note that in this case the parameter set $\theta$ as referred to in the previous section is the pair $(W, B)$. Initially this approach seemed to have great promise. However later work\cite{26} showed that a single perceptron was unable to learn the XOR-function. That is: if the input consisted of only of 2 binary digits and the desired output was simple the XOR of those 2 digits, a single perceptron unit could not replicate this output. This is a fundamental issue and independent of the size of the training-set or the optimization method.

This might have spelled the end of the perceptron, weren’t it for the key observation that this only holds for a single perceptron considered in isolation. When stacking multiple layers of perceptrons (feeding the output of one (or several) perceptron(s) into a new perceptron), such multi-layered perceptrons are able to learn functions like XOR. This layering of simple classifier proved crucial and is what the word 'deep' in deep neural networks refers to.

A natural follow-up question is whether these deeper networks do have the capacity to learn enough functions. If after studying these multilayer perceptrons for a while we find a different basic/simple functions that these layered networks can’t approximate, these deeper networks wouldn’t help much. An ambitious goal might be to be able to approximate every continuous function. Fortunately, this goal is attainable if we are willing to modify the perceptron somewhat.

To show DNNs can approximate every continuous function on the input data, the goal is to apply the 'universal approximation theorem' (UAT)\cite{27}. However for this theorem is not applicable to the perceptron in the form as defined in equation 2.3. The classic perceptron is a binary classifier it can only output 0 or 1. Such a function can’t even approximate the constant function $g(x) = 0.5$. However there are variants of the perceptron for which the UAT is applicable. The first step in introducing those variants is rewriting equation 2.3 to:

$$ T(I) = \begin{cases} 1 & \text{if } I > 0 \\ 0 & \text{otherwise} \end{cases} $$

$$ f_{(w,b)}(I) = T(w^T \cdot I + b) $$

Here it is made explicit that the perceptron consists of 2 parts: first a simple linear classifier. Then on top of that linear classifier a non-linear (thresholding) function $T$ is applied. This non-linear function is usually called an
'activation function' in the literature. The UAT attaches some conditions to this activation function before the UAT can be applied. Most notably this activation function needs to be continuous and monotonically increasing. This rules out the threshold function $T$ as an activation function.

However several alternatives for the threshold function have been proposed over the years. Examples include the sigmoid[28], tanh, rectified linear units (ReLU)[29] and variants on the ReLU like the elu[30] and the selu[31]. Or less used functions like the softplus[32] or the softsign[33]. Each of these fulfills the requirements posed by the UAT.

To be able to write about DNNs more clearly, a notational convention. The activation functions discussed in the previous paragraph are all function of the form: $\phi : \mathbb{R} \to \mathbb{R}$. To write the following equations more compactly, read $\phi(X)$ with $x \in \mathbb{R}^N$ as $\phi$ applied element-wise to the $N$ components of $X$. This allows the perceptron equation to be rewritten as followed:

$$f_{(W,B)}(I) = \phi(W \cdot I + B)$$

(2.6)

Here $I \in \mathbb{R}^N$ as previous. However the change is that $W \in \mathbb{R}^{M \times N}$ and $B \in \mathbb{R}^M$. Finally $\phi$ is an activation function (e.g. tanh, sigmoid, ReLU, etc.) and applied element-wise to the $M$-dimensional vector it receives as input. Functions like $f$ in equation 2.6 are described as 'neurons' and are the neurons referred to by the term 'deep neural networks'.

One last remark needs to be made before the UAT can be applied. Suppose the pattern/function we want to closely approximate is $G : I \to O$. Here $I$ is the input domain (e.g. images) and $O$ the output domain (e.g. the probabilities that the image contains respectively a dog or a cat). The first assumption necessary is that $G$ is continuous, i.e. that arbitrary small changes to the input always imply arbitrary small changes to the output. The second assumption is that the input is bounded, i.e. the numeric values of $I$ are bounded by some upper constant. This assumption is reasonable in the case that $I$ is an image, since most commonly used image formats don’t allow arbitrary high values. Finally using equation 2.6 and these assumptions, the UAT can be introduced in its full form. Consider any continuous function $G : I \to O$ where $I \in [-k,k]^N$ for some value $k \in \mathbb{R}_+$ and $O \in \mathbb{R}^p$. Now consider approximating $g$.

\footnote{The reason these functions are referred to as 'neurons' is because they are inspired by (but certainly not exactly the same as) the neurons that compose the human brain.}[34]
with the following function:

\[ g'(l) = W_2 \cdot f_{W_1,B_1} + B_2 \]  

(2.7)

Here \( f \) is defined as in equation 2.6, \( W_1 \in \mathbb{R}^{M \times N}, B_1 \in \mathbb{R}^M, W_2 \in \mathbb{R}^{P \times M} \) and \( B_2 \in \mathbb{R}^P \). So the approximating function \( g' \) is a linear combination of the output of the 'neurons function' \( f \). Now the UAT states that if the earlier conditions discussed are satisfied\(^8\) and \( M \) is large enough, the following holds true:

\[ \forall \epsilon \exists W_1,W_2,B_1,B_2 \|g'(l) - g(l)\| < \epsilon \]  

(2.8)

Or less formally: if there are enough parameters (i.e. \( M \) is large enough) there exist an instantiation of those parameters that make \( g' \) an arbitrary good approximation of \( g \)\(^9\).

Note that this gives to guarantees on how many parameters are necessary or how to find a good instantiation of those parameters.

The UAT means that underfitting is no fundamental problem: with enough parameters any continuous function can be approximated arbitrarily well. However that still leaves to problem of finding an instantiation of those parameter that doesn’t capture any spurious patterns in the data (i.e. doesn’t overfit). One central approach to generalize better and overfit less is to create DNNs.

DNNs can be iteratively defined based on the 'neuron function' of equation 2.6, by repeatedly applying this function\([28]\). Informally written out, the equation for such a network looks like:

\[
R^k_{(W,B)}(l) = \phi(W_k \cdot \phi(W_{k-1} \cdot ... (W_1 \cdot l + B_1) ... + B_{k-1}) + B_k) 
\]  

(2.9)

Or more formally using a recursive definition:

\[
R^0_{(W,B)}(l) = l 
\]

(2.10)

\[
R^i_{(W,B)}(l) = \phi(W_i \cdot R^{i-1}_{(W,B)}(l) + B_i) 
\]  

(2.11)

Since the network is deeper, \( W \) is no longer a single matrix as in equation 2.6, but a set of \( k \) matrices. Similarly \( B \) is now a set of \( k \) vectors. The sizes of these matrices \( W \) and vectors \( B \) are based on the sizes of the output-vectors of each \( R^i_{(W,B)} \) with the following relations:

\[
R^i_{(W,B)} \in \mathbb{R}^{N_i} \rightarrow W_i \in \mathbb{R}^{N_i \times N_{i-1}}, B_i \in \mathbb{R}^{N_i}, l \in \mathbb{R}^{N_0} 
\]  

(2.12)

However usually it is not a good idea to use \( R^k_{(W,B)} \) directly as the output. The output of \( R^k_{(W,B)} \) is constrained to be in the domain of the activation function \( \phi \) (see equation 2.11). For example if \( \phi \) is the tanh function, this gives the constraint \( R^k_{(W,B)} \in [-1,1]^N \). To tailor the output of the whole network to the

---

\(^8\)Recall: \( g \) needs to be continuous, \( l \) needs to be bounded and \( \phi \) needs to be continuous and monotonically increasing.

\(^9\)For the proof, refer to the original UAT paper\([27]\)
target domain, usually one more output layer is added. This can be done in the form of another linear combination layer. However usually the desired output is some (pseudo-)probability distribution over classes. In that case a softmax-layer [35] is added to ensure the sum of the output-vector is 1. This leads to the following definition:

\[
softmax(X)_i = \frac{\exp x_i}{\sum_i \exp x_i}
\] (2.13)

\[
R^{k+1}_{(W,B)} = softmax(W_{k+1} \cdot R^k_{(W,B)} + B_{k+1})
\] (2.14)

Finally \(R^{k+1}_{(W,B)}\) is a definition of a DNN that is used in practice.

As discussed in section 2.2, there are various algorithms to find good/optimal parameter instantiation for \(W\) and \(B\), if we know the partial derivatives for these parameter w.r.t. the error the current instantiation makes. Luckily, smart application of the chain-rule for derivatives gives us these partial derivatives. The algorithm to calculate these partial derivatives efficiently is called 'backpropagation' [14]. Please refer to the reference for the details behind deriving those derivatives. In this thesis an automatic differentiation framework will be used in the experiments that automatically calculates these gradients.

One final question to be asked is why deep neural networks are necessary when shallow networks are all that is necessary to avoid underfitting according to the UAT? The answer is that in practice deeper networks need fewer parameters to learn, causing it to overfit less [36]. In images this may partially be explained by the properties found in natural images [37]. Whatever the reason, in practice it is found that even in extreme cases deeper networks outperform shallower competitors [38]. However DNN still overfit on images, due to the sheer size of the number of pixels in images. Suppose the images that need to be classified are 256x256 pixels. Next suppose that we fix \(N_1\) to be 1000 (\(N_1\) as in equation 2.12). This would mean that the parameter matrix \(W_1\) alone would have 65.5 million entries.

To make matters worse, such a weight matrix \(W_1\) could find patterns in any subset of pixels. However there are several properties that are known of natural images. First of all a property that holds in natural images is locality: the underlying phenomena captured by an image manifests itself in adjacent pixels. A second property would be translation invariance: shifting an image doesn’t change what it represents. Neither of these properties is enforced by DNNs, while the large number of parameters leave plenty of opportunity to learn spurious patterns. Decreasing the number of parameters while at the same time enforcing a degree of locality and translation invariance would help with decreasing overfitting. It is with this thought in mind that convolutional neural nets were designed.

\[\text{For comparison: in the image search of www.google.com, this qualifies as an 'icon' sized images. It would be too small to fall into either the 'medium' or 'large' image category.}\]
2.4. Convolutional Neural networks

Convolutional neural networks (CNNs)\footnote{2} are a subset of DNNs which are tailor-made to suit the problem domain of images. They contain 2 major changes to the design of regular DNNs as described in section 2.3, and formalized in equation 2.14. First some of the matrix multiplications of equation 2.11 are replaced by convolutions. Then some form of pooling-layers are added to decrease the size of the input of each layer. The rest of this section consists of introducing these concepts formally.

The classical CNN\footnote{2} was originally used to classify handwritten digits. A leftover from that time is the MNIST\footnote{39} of handwritten digits, a common benchmark to this day\footnote{40}[\footnote{41}]. Whereas in the previous section in equation 2.11 the input was (re)interpreted as a vector, here the interpretation of images as a tensor is retained. This means that the input image is viewed as a tensor of the shape: \( I \in \mathbb{R}^{H \times W \times C} \) \footnote{11}. This is what is taken as input. This 3-dimensional tensor input can then be convolved by a 3-dimensional tensor. This tensor is of the shape \( F \in \mathbb{R}^{N \times N \times K} \). This tensor \( F \) is interpreted as a stack of \( K \) filters, each of height \( N \) and width \( N \) (analogous to the tensor representing the input image). Note that the same variable is used to represent both the width and the height of the filters. This is because in practice almost always square filters are used. This leaves 2 stacks of 2-dimensional matrices, one representing each color-channel of the image and one representing each filter. Each filter of the filter stack can then be convolved over each color-channel of the image. Convolution in this case can be intuitively understood as 'moving a filter over a matrix'. On each position, multiply the overlapping elements and then sum the products. The end result is a new matrix. To ensure the final matrix is equal in size as the original image of size \( W \times H \), the image is zero-padded. This leaves a 4-dimensional tensor of size: \( (H, W, C, K) \). Finally a sum is taken over the third dimension, to leave us with a tensor of size \( (H, W, K) \). This is the way two 3-dimensional tensors are convolved in this context, which is denoted by: \( I \ast F \). In CNNs, the elements of \( F \) would be the trainable parameters. These convolution operators are designed as to leave the height and width of the input intact, only modifying the number of channels. However it is desirable to decrease the height and width as well during the processing of the input. Smaller inputs lead to less computation necessary and larger receptive field for the filters in later layers. For this end, pooling-layers were designed. The most well known examples of pooling are average-pooling\footnote{1} and max-pooling\footnote{42}. Both approaches take as input a stack of matrices. Then each matrix is split into \( 2 \times 2 \) sub-matrices. Finally in the case of average-pooling each \( 2 \times 2 \) sub-matrix is replaced by the average of its 4 elements. For max-pooling, the \( 2 \times 2 \) sub-matrices are replaced by the maximum-values of each of their 4 elements. This turns a tensor of size \( (H, W, C) \) into a tensor of size \( (\frac{H}{2}, \frac{W}{2}, C) \).

\footnote{11}{Here \( H \) is interpreted as the height of the image, \( W \) as the width and \( C \) as the number of color channels. For RGB-images, \( C \) would be 3 and for gray-scale image \( C \) would be 1.}
Combining these elements gives the classic CNN[15]. Step by step, this networks is defined as followed:

\[ H_1(I) = \text{average\_pool}(\phi(F_1 \ast I + B_1)) \]  
(2.15)

The input is convolved with a stack of filters, and then an activation function is applied pointwise (the sigmoid function was used in [1]). Average pooling (or localized averaging) is applied to the result. Then:

\[ H_2(I) = \text{average\_pool}(\phi(F_2 \ast H_1(I) + B_2)) \]  
(2.16)

The previous process is applied again, making the network deeper. Do note that the output of \( H_1 \) and \( H_2 \) were 3-dimensional tensors. However before the next step a shift of perspective is in order. Any tensor can be reinterpreted as a vector, by simply rearranging its elements. While a mathematician may frown at this, for a computer scientist it is no problem giving the same data a new indexing. Doing this allows the DNN equations from section 2.3 to be applied here, recall specifically equation 2.11 and equation 2.14. Applying those equations leads to the following finalization of the CNN:

\[ H_3(I) = \phi(W_3 \cdot H_2(I) + B_3) \]  
(2.17)

\[ H_4(I) = \text{softmax}(W_4 \cdot H_3(I) + B_4) \]  
(2.18)

Note that \( W_3 \) and \( W_4 \) are matrices, and are multiplied (instead of convolved) by the output of the previous function. It is this function \( H_4 \) that was the first working example of a CNN, with many to follow.
Convolutional neural networks (CNNs)[1] have been successfully used as a function to solve image classification[43] problems and related problems. Examples of related problems include object classification in videos[44], object segmentation[45] (where is the object in the image) and single image super-resolution[46] (increase the resolution of an image).

The accuracy that can be reached with CNNs seems as of yet to scale unbounded in the (log of) the size of the training set[47]. This would suggest that we could solve any image classification problem with arbitrary precision as long as we have a large enough training set.

In practice the precision we reach is always limited, because the available (labeled) training data is limited. However we still work with the largest possible datasets available for a problem to get the best accuracy. These datasets can be so big in fact, that training a single networks on these datasets can take dozens of GPUs months. Even on the commonly used ImageNet[3] dataset it takes hundreds of GPU-hours to train a single network[5]. That makes finding the optimal architecture and hyper-parameters slow, expensive or both.

On top of that smartphone manufacturers have started using CNNs on their devices[48]. In the context of low-power devices like smartphones, the kind of CNNs that work on dedicated servers require too much computational power. This means that although CNNs can achieve state of the art accuracy, their computational requirements are cause for concern.

### 3.1. Reducing computational load

One direct response to reduce the computational cost of the convolutions is to replace them by a multiplication (e.g. a Hadamard product) in the frequency domain[7]. For an input image of $N$ pixels and a filter of $n$ pixels, this changes the time-complexity from $O(N \cdot n)$ to $O(N \log N)$. With an optimization that can be reduced to $O(N \log n)$ time[10].

The underlying theory is the convolution theorem which says that convolution in the spatial domain is equivalent to multiplication in the frequency domain. The basis for the speed-up is the observation that going back and forth to the
frequency domain and doing one multiplication there is asymptotically and usually also in practice less computationally intensive then doing the convolution in the spatial domain directly. Important to note here is that the convolution itself is now a simple pointwise product. The computational bottleneck that causes the $O(N \log n)$ time-complexity is the Fast-Fourier Transforms (FFTs) that are needed to go back and forth between the spatial- and frequency domain. Already in the future work section of [7] where this technique was first successfully applied to CNNs, it is asked whether we could do away with the FFTs entirely. If not only convolutional layers, but also pooling-layers and activation layers could be implemented in the frequency domain, CNNs could be built that stay in the frequency domain. Going back and forth using FFT would no longer be necessary. Luckily there is already a pooling-layer that works in the frequency domain and is competitive w.r.t. accuracy compared to its spatial competitors like average-pooling and max-pooling. This frequency domain pooling layer is called spectral-pooling [9]. It works by simply dropping the higher frequencies and keeping only the lower frequencies. Because there is a candidate pooling-layer in the frequency domain, the only thing stopping a CNN that works fully in the frequency domain is an activation function that works in the frequency domain. Investigating such a activation function is the goal of this thesis. The rest of this chapter goes into detail why, although in principle an activation layer could be any function of all its inputs, a pointwise applied function on each input is the logical choice to investigate.

3.2. Pointwise activation functions

When designing an activation layer that works in the Fourier domain, we might look at the activation layers that work well in the spatial domain for inspiration. One thing those layers have in common is that all commonly used activation layers consist of a single function that is applied to every pixel of the input independently. This is such a common practice that activation layers are commonly referred to by the name of this function (e.g. ReLU, sigmoid, tanh).

There is good theoretical basis for this common practice. The theoretical basis for the expectation that neural nets can work well is the universal approximation theorem[27] (UAT). One version of this theorem (theorem 2 of [27]) says that a neural net with a single hidden layer can approximate any continuous function arbitrarily well. The only condition is that the hidden layer contains an ‘activation layer’ which consists of a single continuous, bounded and non-constant function which is applied to the output of every neuron in the hidden layer.

In the CNN context, the UAT can also be applied if we use conventional activation layers (e.g. ReLU, sigmoid, tanh). The functions used are continuous, bounded and non-constant and the same function is applied to every pixel of the output of the convolution. The UAT also might require arbitrary many
neurons in the hidden layer to reach arbitrary precision in approximating any continuous function. We achieve this in the CNN context by using multiple filters in each convolution layers. Therefor because we use a pointwise applied function as activation function, we can conclude that if we have a sufficient number of filters, then by the UAT we are guaranteed to be capable of representing any continuous function with sufficient precision. This a strong motivation of writing our activation functions this was. Given the effectiveness of using pointwise applied function as activation layers in the spatial domain, we might consider using a pointwise function in the Fourier domain as an activation function.

Another reason to prefer a pointwise function is computational cost concerns. One activation function that is known to work in the Fourier domain is to simply:

1. apply an inverse Fourier transform (i.e. go back to the spatial domain),
2. apply a conventional spatial activation layer (e.g. the ReLU) and
3. apply a regular Fourier transform to go back to the Fourier domain.

For an activation layer in the Fourier domain to be relevant it either needs to provide better accuracy than the option above. That is: it needs to out perform all spatial activation layers w.r.t. accuracy. Alternatively it could still be interesting if it was only comparable in accuracy w.r.t. conventional spatial activation layers, but offer a (significant) speed-up over going to the spatial domain and back to apply a conventional activation layer.

In this authors opinion, the second approach has more potential than the first. The first approach has a very large search space. Any activation functions regardless of its computational complexity is a potential candidate. The first approach also has a high bar of success. A successful activation function would need to offer better accuracy on benchmark tests than all other options for activation functions.

In contrast, the second approach first of all has a smaller search space. Only functions that are no more computationally expensive than 2 Fourier transforms and 1 conventional activation function are potential candidates. This second approach also has a lower bar of success. Simply being comparable to conventional activation functions still makes it competitive in some use-cases if it is computational cheaper.

This is why in this thesis we focus on the second approach and investigate approaches computationally cheaper than 2 Fourier transforms and a conventional activation layer. Considering that in the asymptotic case, the Fourier transform can be done in $O(N \log N)$ time, this is the computational upper bound for our search space. Investigating this search space is the goal of this thesis.
The approach chosen for the experiments in this thesis consisted of 3 parts. First a dataset was chosen as a benchmark to test on. Then a network was chosen that gave good results using conventional activation-functions. Finally the bulk of the time was spent investigating alternative approaches to conventional activation functions.

A big methodological hurdle was coming-up and creating a set-up in which competitors to conventional activation functions could be easily tested. This chapter focuses not only on which set-up was finally used, but also on the lessons learned in creating this set-up.

4.1. Considerations in experimental set-up

One problem with trying to find an activation function is that research into deep learning architectures is a very empirical science. Most papers on this subject prove their worth by showing their new architecture achieve good (or preferably: state of the art) performance on one of the major datasets (see [49]). This means that to test any new activation function, it needs to be compared with the current state of the art in activation functions on a benchmark dataset. However this poses a challenge since there is no current research on the topic of activation functions in the Fourier domain.

One solution would be to try to set-up a network that works in the Fourier-domain a try to achieve good/state of the art performance on a benchmark dataset directly. However this would lead to 2 problems. First of all a focus purely on accuracy would not do the approach of this thesis justice. The goal of implementing a CNN in the Fourier domain is a decrease in computational load while sacrificing as little accuracy as possible. A better approach would be to see if this approach can achieve a Pareto-point on an accuracy/speed plot of architectures working on a given benchmark dataset. This still would lead to the difficulty that there is less benchmark data on this front, especially for classification problems. However more importantly is the second problem. A second problem is that the focus in this thesis is specifically on the activation functions and how they perform in the Fourier domain. The complexity
of the learning process of neural networks ensures that at times it can learn 'to cope' with poorly performing parts of the network. This can make it very difficult to draw deeper conclusions from a network that performs reasonably well. In this thesis it is important to test proposed activation functions in a set-up where those activation functions can’t 'hide' it if they perform poorly.

The approach taken in this thesis to solve the problems mentioned in the previous paragraph is to implement a set-up where we directly compare potential activation functions in the Fourier domain with a conventional activation function like the ReLU in the spatial domain. This is done by starting with a simple conventional network that uses the ReLU in its architecture. Then to compare the ReLU with an activation function $\text{Test}_\text{Act}$ in the Fourier domain directly, the ReLU is replaced by the following function:

$$\text{IFFT}(\text{Test}_\text{Act}(\text{FFT}(X)))$$  \hspace{1cm} (4.1)

For a schematic representation of what this means for the overall architecture, see figure 4.1. Instead of applying the ReLU to the output of the convolution, a Fast Fourier Transform (FFT) is applied. This moves the output to the Fourier domain, which means that now the candidate activation function that needs to work in the Fourier domain can be applied. Finally an inverse Fourier transform is applied to move the output back into the spatial domain for the next layer in the network. This then allows a direct comparison between the ReLU and the candidate activation function, in the same context.

One might think that this is all that is necessary to start experimenting (at least this author initially thought so), however one last check is necessary as was found during initial experiments for this thesis. Initially MNIST\cite{40} was chosen as the benchmark dataset. This is one of the most commonly used datasets in the field and is a relatively easy problem\footnote{MNIST is a dataset of 28x28 greyscale images of human drawn digits. The state of the art on the problem of classifying these digits is 99.87\%, or 23 out of the 10.000 test digits misclassified\cite{50}. Especially since human-written digits can be ambiguous at times, this is more or less a perfect score on this problem.}.

This makes MNIST usually a good starting point for experiments. The conventional network that used ReLUs had an classification accuracy of 99.6\% after
training. Then several Fourier activation functions were tested against the ReLU using the set-up as described in figure 4.1. One experiment involved raising the Fourier experiments to a given power $c$. This gave promising results, so various values were tried and it was found that setting $c = 1$ gave very good results. However raising a (complex) number to the power 1 is simply the identity function. This gave the surprising result that while the network had an accuracy of 99.6% when trained, the accuracy only dropped to 99.4% if the ReLU was dropped entirely from the convolutional part of the network. This could be explained by the fact that in the conventional CNN set-up used, the convolutional layers are followed by fully-connected layers. A problem like MNIST was easy enough to be almost entirely solved by those layers. This example taught the lesson that it is important to not only compare to the current state of the art, but also to doing nothing. The set-up used for the experiments published in the results section was written with this lesson explicitly in mind.

4.2. Experimental architecture

The dataset used in the experiments is CIFAR-10\[51]. This dataset was chosen due to the large number of papers using this as a benchmark\[49] and because it is a more difficult classification challenge than MNIST. It contains 32 by 32 RGB-images, each of which fits 1 of 10 possible categories. See figure 4.2 for examples.

As a basis, a simple tensorflow model from the official Tensorflow Github was used\[52]. For an overview of the architecture of this network, see table 4.1 This network suffers from the same problem as the network initially used for MNIST. When removing the ReLUs in the convolutional layers (i.e. layers 2 and 6 in table 4.1), the accuracy remains the same, 87%. Now the challenge

<table>
<thead>
<tr>
<th>layer name</th>
<th>number of filters/hidden units</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Convolutional layer</td>
<td>64</td>
</tr>
<tr>
<td>2. ReLU</td>
<td></td>
</tr>
<tr>
<td>3. max-pooling</td>
<td></td>
</tr>
<tr>
<td>4. batch-normalization</td>
<td></td>
</tr>
<tr>
<td>5. Convolutional layer</td>
<td>64</td>
</tr>
<tr>
<td>6. ReLU</td>
<td></td>
</tr>
<tr>
<td>7. max-pooling</td>
<td></td>
</tr>
<tr>
<td>8. batch-normalization</td>
<td></td>
</tr>
<tr>
<td>9. Fully-connected layer</td>
<td>392</td>
</tr>
<tr>
<td>10. ReLU</td>
<td></td>
</tr>
<tr>
<td>11. Fully connected layer</td>
<td>192</td>
</tr>
<tr>
<td>12. ReLU</td>
<td></td>
</tr>
<tr>
<td>13. cross-entropy output layer</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: The original architecture as found in \[52], this architecture was adapted to form the basis for the experiments in this thesis.
Figure 4.2: The 10 classes of CIFAR-10 with 10 example images of each class.
is to alter this architecture in such a way that removing the ReLU does make a difference.

The theory behind the ReLU is that it makes the network non-linear, giving the network the opportunity to use the linear convolutional- and fully connected layers the ’opportunity’ to learn non-linear functions. However here max-pooling and the ReLUs after the fully-connected layers are also non-linear. Thus the hypothesis is formed that removing the ReLUs after the fully-connected layers and replacing max-pooling by average-pooling should make the network rely more on the ReLUs after the convolutional layers. 

<table>
<thead>
<tr>
<th>pooling layer</th>
<th>FC ReLU</th>
<th>ReLU after conv</th>
<th>no ReLU after conv</th>
<th>difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>max-pooling</td>
<td>yes</td>
<td>87%</td>
<td>87.3%</td>
<td>0.3%</td>
</tr>
<tr>
<td>max-pooling</td>
<td>no</td>
<td>82.8%</td>
<td>81.1%</td>
<td>1.7%</td>
</tr>
<tr>
<td>average-pooling</td>
<td>yes</td>
<td>87.3%</td>
<td>68.6%</td>
<td>18.7%</td>
</tr>
<tr>
<td>average-pooling</td>
<td>no</td>
<td>83.4%</td>
<td>37.1%</td>
<td>46.3%</td>
</tr>
</tbody>
</table>

Table 4.2: This table shows the accuracy of variations on the network of table 4.1. The first column shows whether the 2 max-pooling layers are replaced by average-pooling layers. The second column shows whether the 2 ReLU layers after the fully connected layers are removed (i.e. removing layers 10 and 12 in table 4.1. Columns 3 and 4 show the results after respectively keeping and removing the ReLUs after the convolutional layers (layers 2 and 6 in table 4.1. The final column shows the difference in accuracy between column 3 and 4. The percentages in this table are all classification accuracy of the architectures on the dataset CIFAR-10.[51]

This hypothesis is tested in table 4.2. This table shows that removing other non-linear layers (removing the ReLU after the fully-connected layers, replacing max-pooling by average-pooling) has only a modest impact on the accuracy. However the ReLUs after the convolutional becoming increasingly crucial. By switching to average-pooling and removing the ReLUs after the convolutional layers, removing the ReLU after the convolutional layers drops the accuracy by 46.3 percentage points, instead 0.3. It is clear that with these changes it is indeed the ReLU that is crucial for the accuracy of the network. It is this set-up that is used for testing alternatives for the ReLU in the Fourier domain. For completeness the architecture used in the experiments is fully described in table 4.3.

### 4.3. Training procedure

The time needed to train a network and the accuracy reach at the end of training can differ significantly based on the parameters used for training (see for example the results in [21]). Although in this work little difference in accuracy was found by varying the hyper-parameters, it is still important to mention those parameters used.

As the optimizer ADAM is used with a learning-rate set to 0.1. This learning rate did matter in practice as setting it too low caused slow convergence and setting it too high caused divergence in some of the experiments. Setting the learning rate at 0.1 proved to work good in all experiments. Another option
Table 4.3: An adaption of the architecture described in table 4.1. It shows the architectures used to compare a potential activation function in the Fourier domain Test_Act with the ReLU in the spatial domain. The first column shows the layers both architectures have in common, the second and third column show where they differ. Note that 'identity' is a filler layer, a layer that doesn’t change its input.

would have been a dynamic learning-rate that decreases every epoch. However in practice it was found that this required a lot of tuning to get right for little benefit. Especially concerning was that decreasing the learning-rate too quickly could cause seemingly perfect convergence to a non-optimal solution. Decreasing the learning-rate too slow could caused slower convergence. Therefore the learning-rate was kept fixed in the final experiments.

In the search through the hyper-parameter, weight decay was also considered. However setting it too high quickly caused instability in training and lower accuracy. The benefit of optimal weight decay was marginal (<1%). Due to the large time investment needed to find optimal weight decay parameters and little value it brought, weight decay was left out of consideration for the final experiments. Finally during training, mini-batch learning was used with a batch-size of 128.
In this chapter several potential activation function are tested, in the set-up described in chapter 4. This chapter is first going to describe several potential activation function that were tested. The results of these alternatives do not reach the accuracy of the ReLU. In the second part of this chapter, some theoretical considerations are given why pointwise activation function might give poor results. When looking at the results in this chapter, keep in mind that in the same experimental context the ReLU scored 83.4%, no activation function scores 37.1% and since there are 10 classes, random guessing would score around 10% classification accuracy.

5.1. Spatial activation functions in the Fourier domain

One major hurdle in designing activation functions for the Fourier domain, is that Fourier coefficients are complex numbers. This while the vast majority of literature on activation functions focuses on real-valued input. One function already published for complex networks in the spatial domain is the modReLU\[53\], which will be discussed in the next section.

The first approach originated from the idea to apply the well-tested spatial activation function directly in the Fourier domain. One straightforward way to do this is to apply the spatial activation functions to either the real or imaginary parts of the complex Fourier coefficients (or both parts simultaneous). Confidence in this approach was boosted when after finishing these experiments other researchers showed that using the ReLU on both real and imaginary parts can be a competitive choice in complex spatial CNNs\[54\].

7 different activation functions were tested. Each was applied on the real part, imaginary part and both parts of the complex Fourier coefficients fed into the non-linearity. The results are shown in table 5.1. Based on these results, the activation functions can be grouped into 3 groups according to

\[1\] The authors of this paper call the activation function which consists of a ReLU applied to the real and imaginary parts of a complex input a complex ReLU or \(CReLU\). In this thesis that terminology is not followed for 2 reasons. First of all variants where the ReLU is applied to one component are also tested, which makes it confusing to call the variant where the ReLU is applied to both components \(the\) complex ReLU. Also confusion might arise with the concatenated ReLU, the \(CReLU\).
the range of their outputs.
The first group are the left 4 activation functions in the table (ReLU, ELU, softplus and the absolute value function). These have in common that each of their ranges is bounded from below, but unbounded for positive values. It is interesting to note that although these functions differ in properties such as differentiability, injectivity and the exact domain, each gives remarkably similar results. However the following columns show that the unboundedness of the range in the positive direction is crucial.

<table>
<thead>
<tr>
<th></th>
<th>ReLU</th>
<th>ELU</th>
<th>softplus</th>
<th>abs</th>
<th>tanh</th>
<th>soft-sign</th>
<th>sigmoid</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>60.0%</td>
<td>64.1%</td>
<td>63.6%</td>
<td>64.7%</td>
<td>39.8%</td>
<td>37.9%</td>
<td>36.6%</td>
</tr>
<tr>
<td>imaginary</td>
<td>57.0%</td>
<td>62.7%</td>
<td>62.5%</td>
<td>64.2%</td>
<td>38.5%</td>
<td>38.4%</td>
<td>36.5%</td>
</tr>
<tr>
<td>real &amp; imaginary</td>
<td>63.5%</td>
<td>65.2%</td>
<td>64.7%</td>
<td>63.6%</td>
<td>31.7%</td>
<td>29.5%</td>
<td>10.0%</td>
</tr>
</tbody>
</table>

Table 5.1: The results for several activation functions tested as in the set-up described in chapter 4. The input Fourier coefficients are interpreted in Cartesian coordinates. Then each row represents a spatial activation function that is either applied to all real parts of the Fourier coefficients, the imaginary parts or both. The percentages are classification accuracy on CIFAR-10.

The next grouping that can be identified are the next 2 activation functions: tanh and soft-sign. Their ranges are bounded from below and above. The consequence in the results is that when they are applied to one component of the Fourier coefficients (either real or imaginary), then the results are very close to having no activation function at all. On top of that: applying them to both components lowers the accuracy below the score of having no activation function. This suggests that an unbounded range really is necessary to achieve any results at all. This is confirmed in the last column.

In the last column the sigmoid function performs worst of all. The range of the sigmoid function is not only bounded, but also only positive. This means when applied to both the real and imaginary parts of a complex number, the output is bounded and contained in the first quadrant. This in practice is too much for the network, and learning can no longer take place. The accuracy is 10%, which corresponds to random guessing.

One more option is to use the absolute value function. However instead of applying the absolute value to the real and/or imaginary components of the Fourier coefficients, the absolute value (or Euclidean norm) of the full complex number can also be taken. This was also tested and resulted in an accuracy of 64.3%. This is in line with the accuracy obtained by applying the absolute value component-wise and thus gives no extra benefit in accuracy. It is however interesting to note that the absolute value (or Euclidean norm) of a complex number is the magnitude of the complex number in polar notation. One might consider applying the same function tested in table 5.1, but apply them to the magnitude and phase of the complex numbers instead. However both magnitude and phase are positive values. Each of the functions under consideration in table 5.1 are either bounded or approximate the identity function for positive values. This means that they didn’t an achieve
accuracy over that of the identity function (37.1%).

Summarizing, the best result in table 5.1 and this section is an accuracy of 65.2%. This is much better than the 37.1% accuracy of having no activation function, but still far removed from the 83.4% of the ReLU. All these activation functions tried so far have in common that they are fixed functions. The activation functions themselves have no learn-able weights/parameters. In the next section, several activation functions are considered that do have such learn-able weights built in.

5.2. Learned activation functions

When considering activation functions on complex input with a learned parameter, the natural first choice is to the example of such a function found in the literature: the modular ReLU or modReLU[53]. The equation for that function is (when viewing the Fourier coefficients in polar notation):

\[ \text{modReLU} : m \cdot e^{i\theta} \rightarrow \text{ReLU}(m - b) \cdot e^{i\theta} \]  (5.1)

Here \( m \) and \( \theta \) are respectively the magnitude and phase of the input Fourier coefficients. The function works by training a parameter \( b \) with the rest of the weights of the filters.\(^2\) For each complex value with magnitude greater than \( b \), the magnitude gets decreased by \( b \). All complex input inside the circle with magnitude \( b \) get mapped to 0.

Training networks that use the modReLU was difficult. The curious phenomenon occurred where regardless of initialization of \( b \), it always slowly decreased during training. However decreasing the initialization of \( b \) below the empirically found optimal value of 15, lowered the final accuracy found after training had converged. It is still an open question as to why this is the case. However since even the optimal value of 15 still only converged to an accuracy of 59.9%, experimentation eventually moved on to other candidate activation functions.

The highest accuracy was obtained in the experiments where activation functions were tried that consisted of adding, multiplying or raising to a power of a learn-able parameter. For each (of the 2) layers a single constant is learned. This constant is applied to either: the real part, imaginary part, both real and imaginary part, magnitude, phase or both the magnitude and phase of the complex Fourier coefficients. Applying can mean here: adding, multiplying or raising the input to this learned constant. The results are shown in table 5.2.

The results of table 5.2 show that adding or multiplying the Cartesian components of the complex input does not help learning nor does raising them to a constant power. However 2 experiments in table 5.2 do show some clear promise. Learning a constant to multiply to the phase of the complex input

\(^2\)Here \( b \) is shared within each layer.
Table 5.2: This table summarizes the results of different activation functions, based on a single shared learn-able constant per layer. The columns show whether that constant was added or multiplied to the input or the input raised to the power of the constant. The rows show on what part of the complex input the constant was applied. Here in the first 3 rows the complex input is viewed in Cartesian representation and the last 3 rows in polar-coordinates.

or raising the magnitude to a learn-able power does give significant results. When multiplying the phase of the complex input with a learn-able parameter, it is interesting to see the convergence of that parameter. During training it slowly converges to 0. This corresponds to the absolute value (or Euclidean norm) of the complex number, a function already tested in the previous section to give similar results.

In the experiments where the magnitude is raised to a learn-able power, the parameter converges to 1 of 2 local optima in each layer. In the various experiments, 3 local optima were found. See table 5.3 for the details of each optimum. These results show a local optimum for each layer in the range (0, 1) and in the range (1, ∞). This is caused by the poor performance at the values 0 and 1, as well as at very large values.

Raising the magnitude of the input to the power 0 projects the complex input on the unit-circle. This loses all magnitude information and experimentally it can be seen that this prohibits all learning. The resulting accuracy was 9.9% for a run where the magnitude power was fixed at 0. Fixing the parameter at 1 on the other hand leaves the identity function and the corresponding accuracy of 37.1%. Finally very high values for this parameter cause the values and gradients to explode and training to diverge. Experimentally this happens above a parameter value of 2.4.3

From the results of table 5.2, 2 ideas emerge as follow-up experiments. The 2 set-ups that worked best in table 5.2 were raising the magnitude to a learn-able parameter or multiplying the phase with a trainable parameter. The first idea is to join both approaches and do both in a single set-up. This means having 2 learn-able parameters per activation layer, $b_1$ and $b_2$, applied as followed:

$$m \cdot e^{i\theta} \rightarrow m^{b_1} \cdot e^{i\theta \cdot b_2}$$

(5.2)

However the highest accuracy reached by trying various initializations was 67.8%. The extra parameter space must make the optimization problem harder, because the optimizer can’t find the local optima of table 5.3.

3One of the early experiments consisted of raising the whole complex input to the third power. This experiment gives context to the fact why that approach failed to converge during training.
5.3. necessary conditions

In the previous section various pointwise activation functions were tried, but none of them really approached the ReLU in accuracy. On top of that, simply squaring each spatial input value also gives 78.3% accuracy. This leads to the question: why is it apparently easy to find a spatial activation function that easily outperforms any activation function that was tried in the Fourier domain? What would be required to find an activation function that does work?

One thing to consider is the pointwise nature of a CNN in the frequency domain. Spatial convolutions are pointwise multiplications from a frequency perspective. Spatial average pooling approximates spectral pooling in the frequency domain, which again is a pointwise operation in the frequency domain. And now in the previous 2 sections proposed activation functions were tested, each of which is pointwise in the frequency domain. After all the whole premise of this thesis was the idea that it might be possible to create a CNN which has a constant number of operations for each frequency, independent of the input size. But this does mean that combined this proposed approach is from a frequency perspective a pointwise function of the Fourier transformed input-image, up until the fully connected layers.4

In this perspective, the spectral-pooling gives cause for concern. Since each layer up to and including the last convolutional layer works pointwise on each frequency, the output of that last convolution layer at each frequency only depends on each corresponding input frequency. E.g. the value of the DC-term of the output of the last convolution layer only depends on the DC-

---

4This means that in the architecture as described in table 4.3, layers 1 to 12 are a pointwise function of each frequency of the Fourier transform of the input image.
term of the input-image. Now Recall that spectral-pooling simply drops the higher frequencies to achieve pooling. This means that if after several layers of convolutional-, activation- and spectral-pooling layers, only the 25 lowest frequencies are left, then only the 25 lowest frequencies of the input-image have any effect on the output. This corresponds to doing all pooling at the very start of the network.

This is a major issue with the proposed approach investigated in this thesis. Implementing a CNN-architecture in the Fourier domain based on a pointwise activation function in the Fourier domain and spectral pooling is not really a valid option for larger images. Since all pooling is equivalent to compressing the input images right at the start, the accuracy will be severely hindered compared to architected in the spatial domain that use all information in the image. Conversely processing the whole image without pooling is infeasible for larger images. This means that the approach first suggested in [10] cannot work as envisioned there with a pointwise activation function and spectral pooling.

One natural question to be asked is if there can be found non-pointwise activation functions in the Fourier domain that do give accuracy comparable to the ReLU. For that purpose, consider the spatial activation function of squaring each entry. Recall from the beginning of this section that this activation function reached an accuracy of 78.3% when tested. Now consider the convolution theorem, which states that multiplication in the spatial domain equivalent is to convolution in the frequency domain (and vice versa). This means that an auto-convolution in the frequency domain (i.e. the input convolved over itself) is equivalent to taking the square in the spatial domain. This gives an activation function which outperforms the best pointwise activation function found in this thesis by 9 percentage-points.

More generally, consider a spatial pointwise activation function $f$. This function can be approached arbitrarily well by enough terms of its Taylor series. Suppose the first $n$-terms of the Taylor series of $f$ are recursively defined as:

$$f_0(x) = a_0$$

$$f_n(x) = x \cdot f_{n-1}(x) + a_n$$

Then the equivalent function in the frequency domain is, by the convolution theorem:

$$f_0(x) = a_0$$

$$f_n(x) = x \ast f_{n-1}(x) + a_n$$

Simply replacing multiplication with convolution allows the Taylor series to be moved to the frequency domain. This means that with enough convolutions, any pointwise spatial activation function can be approximated arbitrarily well in the frequency domain.

However the reason the frequency domain is used in the first place is to avoid convolutions because they are computationally expensive. In fact, because in practice in CNNs the filters are always smaller than the input on which they
are applied, auto-convolutions are more computationally expensive than the original filter-convolutions in conventional CNNs. This means that activation functions do exist in the frequency domain that can compete with the ReLU in the spatial domain on accuracy, only these options then can’t compete on computational expense.

Summarizing, the necessary conditions for an activation function that can compete both on computational expense and on accuracy are as followed. First of all such an activation function can’t be pointwise. As shown in this chapter, pointwise activation functions fundamentally can’t compete in the frequency domain with their spatial counter-parts. However the computational budget to retain an edge in speed over a spatial ReLU is slim. 2 FFTs and a ReLU cost only $O(N\log N)$ time\textsuperscript{5}. To compete with that, any activation function in the frequency domain that wants to compete with the ReLU needs to use less computational power. This leaves room for an activation function that computes more than a pointwise function, but use less than $O(N\log N)$ operations.

\textsuperscript{5}Here $N$ is the number of input frequencies.
Conclusion

This thesis-project started with the future work section of [7]. In that paper convolutions were first sped up using Fourier transforms. In its future work section the idea was mentioned that it might be possible to drop the Fourier transforms by finding activation functions that can be applied in the frequency domain. This idea was reiterated in [9] and [10]. However as far as the author of this thesis knows, as of yet nobody has published any work on this subject. Therefor this thesis is a first investigation into this new subject.

The idea behind [7] is that convolutions are faster computed in the frequency domain, because they reduce to multiplications there. So much faster that it is less computationally intensive to go back and forth to the frequency domain using FFTs and doing the convolution there, then it is to do the convolution in the spatial domain directly. This meant that where previously most computation of CNNs was done in the convolutions, now the computational bottle-neck was in the FFTs.

This naturally led to the question: is it necessary to go back to the spatial domain at all? The only thing that made that necessary was the activation function. No frequency alternative exists for spatial activation functions like the ReLU. This was the basis for the search in this thesis into candidates for such a frequency activation function.

In the result section a variety of candidates are investigated. These candidates were tested in a set-up where they were directly compared against the spatial ReLU and against having no activation function at all. In this set-up, the ReLU achieved a classification accuracy of 83.4% on the benchmark dataset CIFAR-10. At the same time the set-up was specifically designed such that removing the activation function entirely gave much worse performance: 37.1% classification accuracy.

Both unparameterized and parameterized activation functions were then tested. The parameterized activation functions consisted of 1 or 2 parameters in each activation layer that during training are optimized with all the weights in the filters of the convolutional layers. The best performing unparameterized activation function consisted of applying the ELU spatial activation function on
both the real and imaginary parts of each Fourier coefficient. This resulted in an accuracy of 65.2% after training. The best performing parameterized activation function consisted of raising the magnitude of the Fourier coefficients to the power of the parameter, while leaving the phase of the Fourier coefficients untouched. When those parameters are initialized at 1.5 to ensure training converges to the right local optimum, the resulting classification accuracy is 69.3%.

Although a variety of options were tested, none came close to the accuracy of the ReLU. This led to the question of whether a pointwise activation function (as is used in the spatial domain) could ever lead to a real competitor of the ReLU. The answer is that the set-up proposed is problematically pointwise in the frequency domain.

A fundamental property of CNNs is that they reduce the size of the input image while retaining as much of the relevant information as possible. A CNN architecture that works pointwise in the frequency domain cannot retain any information on the frequencies which it drops along the way. This means that dropping any frequencies in the pooling-layers is equivalent to compressing the input-image by dropping those frequencies right at the start. This means that a CNN architecture which works pointwise in the frequency domain needs to work on the full size of the input image in every layer. This either restricts it to very small images or requires prohibitively much computation to work with larger images.

A possible solution for this problem would be an activation function which isn’t pointwise. This author isn’t aware of any previous work into non-pointwise activation functions (neither in the frequency domain, nor in the spatial domain). Finding non-pointwise activation functions that work well isn’t difficult. The auto-convolution in the frequency domain (equivalent to squaring each spatial entry) is such an example with an accuracy of 78.3%. More generally: using n auto-convolutions the first n-terms of the Taylor series of any pointwise spatial function can be implemented in the frequency domain. The issue is then however back at computational complexity. The competitor is the spatial ReLU which can be implemented in the frequency domain by adding 2 (Fast) Fourier Transforms. This gives a computational budget of $O(N\log N)$ for an input of N frequencies. So the activation function sought in the future works section of [7] needs to:

1. do more than simply apply a pointwise function.
2. be equally or preferably less computational expensive than going to the spatial domain and back for a ReLU, which costs $O(N\log N)$ computations.
3. Compete with the ReLU on accuracy.

This is a big ask for an activation function, but it is not impossible that such a function exists. However no candidate meeting these requirements has been
found in the research for this thesis.

Besides a continued search for an activation function which fits the 3 requirements set out in the previous paragraph, another avenue for future work could lie in the pooling-function. The 2 most used pooling functions in the spatial domain are average-pooling and max-pooling, neither of which is a pointwise function. A pointwise activation function in the frequency domain might not be a problem if it is combined with a non-pointwise pooling function.

It is however not obvious what a natural pooling function would be that combines different frequencies instead of just dropping frequencies. Also the constraints on computational budget still stand. If an alternative pooling function has a higher computational cost than simply going to the spatial domain and using average- or max-pooling, nothing is gained. Still, it is another avenue that can be explored in the search for a CNN architecture that works fully in the frequency domain.
Bibliography


