Receptive Fields Neural Networks using the Gabor Kernel Family

Govert Verkes

January 20, 2017
18 ECTS

Supervisor: Rein van den Boomgaard
Signed:
Abstract

Image classification is an increasingly important field within machine learning. Recently, convolutional neural networks (CNN’s) have been proven to be one of the most successful approaches. CNN’s perform outstanding when ample training data is available. However, because CNN’s have a large number of parameters they are prone to overfitting, meaning it will work well on training data, but not on unseen data. Moreover, there are no specific mechanisms in a CNN to take variances, like scale and rotation, into account. Jacobsen et al. [1] proposed a model to overcome these problems, called the receptive field neural network (RFNN). We extend upon the results of Jacobsen et al. to use a weighted combinations of fixed receptive fields in a convolutional neural network. The key difference is that we are using the Gabor family basis for the fixed receptive fields, instead of the Gaussian derivative basis. The use of the Gabor family is inspired by their ability to model receptive fields in the visual system in the mammal cortex and their use in the field of computer vision.

We performed an exploratory study on the Gabor family basis in the RFNN model using three well established datasets of images, the Handwritten Digit dataset (MNIST), the MNIST Rotated and the German Traffic Signs dataset (GTSRB). Our results show that for fewer training examples the Gabor RFNN performs better than the classical CNN. Moreover, the results compared with the Gaussian RFNN suggest that the Gabor family basis has a lot of potential in the RFNN model, performing close to state-of-the-art models. In the future, we should look for a method of learning the Gabor function parameters, making it less sensitive to parameters chosen a priori.
## Contents

1 Introduction 7

2 Related work 9
   2.1 Receptive fields 9
   2.2 Scale space 11
   2.3 Convolutional neural networks 12
       2.3.1 Scattering convolution networks 13

3 Theoretical background 15
   3.1 Artificial neural network 15
       3.1.1 Single layer network (Perceptron) 16
       3.1.2 Multilayer perceptron 17
       3.1.3 Activation function 18
       3.1.4 Backpropagation 18
       3.1.5 Backpropagation example 22
   3.2 Convolutional neural network 23
       3.2.1 Convolution operation 23
       3.2.2 Convolution layer 25
       3.2.3 Max pooling 26
       3.2.4 Backpropagation 27

4 Receptive fields neural network 29
   4.1 Theory 29
       4.1.1 Gaussian convolution kernels 29
       4.1.2 Gabor convolution kernels 32

5 Implementation 35
   5.1 Theano and Lasagne 35
   5.2 Classical convolutional neural network 37
   5.3 Receptive fields neural network 38

6 Experiments 39
   6.1 MNIST 39
       6.1.1 Setup 39
       6.1.2 Results 40
   6.2 MNIST Rotated 41
       6.2.1 Setup 41
       6.2.2 Results 42
   6.3 GTSRB 42
       6.3.1 Setup 43
       6.3.2 Results 43

7 Discussion & Conclusion 45
CHAPTER 1

Introduction

Image classification is a very fundamental and increasingly important field within machine learning. Moreover, its applications are still growing, e.g. self-driving cars that need to recognize their environment and security camera’s that are looking for specific behaviour. Recently, convolutional neural networks (CNN) have been proven to be one of the most successful approaches in image classification. In 2012 Krizhevsky et al. [2] have shown a revolutionary performance with CNN’s on the ImageNet classification contest [3], achieving an error rate of 16.4% compared to an error rate of 26.1% achieved by the second best (in 2011 the best scoring error rate was 25.77%) [4, 5]. In 1989, CNN’s had been introduced by Lecun [6] inspired by the neocognitron (Fukushima [7]). However in the beginning their capabilities were limited by the available computing power. Only recently, their performance and training ability greatly improved as a result of more powerful hardware (graphics cards) and by using powerful GPU implementations.

The ability of CNN’s to solve very complex problems is clearly established as shown by recent success. One of the reasons CNN’s are able to solve very complex problems, is their ability to learn a large number of parameters [2]. However, in order to learn this large number of parameters, a CNN generally needs a lot of training data to prevent it from overfitting.

Another issue with CNN’s is that little is understood about why they achieve such a good performance [5]. There is no proper explanation for many of the learned parameters, a CNN simply succeeds by trial and error. Without a proper understanding of the model, improving it is a very difficult task. Scale invariance is one of the concepts where it is not entirely clear if a CNN actually learns this and if so, how a CNN does this. But even if it does learn scale invariance, we must train the network with training data presented at different scales. CNN’s do not have a specific mechanism to take scale invariance into account [9].

Human beings are very good in handling different scales, for example if we look at a car from a 2 meter or a 200 meter distance, we have no difficulty perceiving both as a car. Therefore much research has been done on how the visual system of the (human) brain works, and understanding this might help us solving scale invariance for image classification problems. Scale-space theory showed that scale invariance can be achieved with a linear representation of an image convolved with the Gaussian kernel at different scales [10]. Moreover, the Gaussian derivative kernels up to 4-th order was shown to accurately model the receptive fields in the visual system [11, 12]. Jones and Palmer showed that the Gabor filter model gives an accurate description of receptive fields in the visual system as well [13, 14].

Recently, Jacobsen et al. proposed a model, called the receptive field neural network (RFNN) [1], to overcome the issues discussed in the previous paragraphs. It tries to solve the need for large amounts of training data to prevent overfitting, by reducing the number of parameters to learn, but still keeping the same expressive power as CNN’s. The RFNN accomplishes this by using Gaussian kernels, which also results in a more scale-invariant model.

In this paper we elaborate on the RFNN model and try to use the Gabor function as kernel basis instead of the Gaussian derivative kernel basis. Since the bases are quite similar, the expectation is that the performance of both bases will also be similar. However, the Gabor basis is more flexible in terms of free parameters. Hence, it could conceivably be hypothesised that it will learn complex problems faster if the the free parameters are tuned correctly.
2.1 Receptive fields

A common approach to solving problems in the field of artificial intelligence is to look at biological structures, i.e. how living organisms solve the problem. Examples of this are the perceptron and the artificial neural network discussed in section 3.1.1 which are inspired by neurons in the brain. Furthermore, in the field of computer vision inspiration is taken from the (human) brain. Several successful methods have been designed with varying degrees of correspondence with biological vision studies, and so is the method described in this paper. Therefore, first a brief description of the visual system will be provided in the next paragraph.

The components involved in the ability of living organisms to perceive and process visual information are referred to collectively as the visual system. The visual system starts with the eye, light entering the eye passes through the cornea, pupil and lens (see 2.1). The lens then refracts the light and projects it onto the retina. The retina consists of photoreceptor cells, divided into two types, rods and cones. Rods and cones fulfill a different purpose, rods are more sensitive to light but not sensitive to color, on the other hand cones are less sensitive to light intensity but are sensitive to color \[15\]. The rods and cones in the retina are connected to ganglion cells via other cells. Each of these ganglion cells bears a region on the retina where the action of light alters the firing of the cell. These regions are called receptive fields \[16\].

![Autonomy of the eye, including the retina on which light gets projected](Source: National Eye Institute, NEI).

The ganglion cells’ receptive fields are organized in a disk, having a “center” and a “surround”. The “center” and the “surround” respond oppositely to light. The workings of these ganglion cells’ receptive fields are illustrated in Figure 2.2.

Ganglion cells are connected to the lateral geniculate nucleus (LGN), which is a relay center in the brain. The LGN relays the signals to simple cells and complex cells, this connection can be seen in Figure 2.3. In the 1950’s, [Hubel and Wiesel] did a Nobel prize winning discovery on
Figure 2.2: Workings of the on and off center receptive fields of the retinal ganglion cells

these simple and complex cells [17]. As a result of this connection, the simple and complex cells bear a combination of multiple ganglion cells' receptive fields. This results in a more complex receptive field, which responds primarily to oriented edges and gratings. An example of the firing of a simple cell after applying a certain stimulus can be seen in Figure 2.4.

Figure 2.3: Simple and complex cells in the virtual cortex as a combination of ganglion cells

It has been shown that these receptive fields in the visual system can be accurately modelled in terms of Gabor functions [13-14] or Gaussian derivatives up to 3rd-4th order derivative [11-12]. Consequently, this is a strong motivation for the use of these exact function families in the image classification models discussed in this paper.
2.2 Scale space

In the field of computer vision, the scale-space theory tries to solve the problem of scale invariance. Scale invariance is very important in many real world vision problems, for example in an image objects can appear at different scales, but we would like the computer to still recognize these objects as if they are the same. Because no way exists to know a priori what scales are relevant, the scale-space theory tries to solve this problem by creating a linear (Gaussian) scale space, converting a single image to the same image at many different scales \[10\]. The scale-space theory, as with many concepts explained in this paper, is inspired by the workings of (human) visual perception \[18\].

Scale-space is defined by multiple feature maps \(L\) created by convolution of the image \(I\) with two dimensional Gaussian kernels \(G\) at different scales \(\sigma\):

\[
L = I \ast G_{\sigma}
\]  
(2.1)

Where,

\[
G_{\sigma}(x, y) = \frac{1}{2\pi\sigma} e^{-\frac{(x^2 + y^2)}{2\sigma^2}}
\]  
(2.2)

Convolution will be explained in section \[3.2.1\] but for now one can see this convolution as a smoothening of the image. A scale-space representation at 6 different scales can be seen in Figure \[2.5\]. After constructing a scale-space representation of an image, we can use this basis for further visual processing. For example, image classification or feature extraction \[18, 19, 10\].

Scale-space theory exclusively use Gaussian kernels as a basis, because convolution with the Gaussian kernel has been proven to be unique in the fact that it does not introduce new structures, which are not present in the original image.
2.3 Convolutional neural networks

Over the past few decades, many approaches to image classification have been proposed. Notable are the support vector machine (SVM) [20], a well-known linear classifier used for many classification problems, decision trees and artificial neural networks (ANN’s). Recently in the late 2000’s, a very successful and popular approach became the convolutional neural network (CNN), a type of ANN that works especially well on sensory tasks, such as images, video, and audio. Despite the success of CNN’s, little is understood about why CNN’s work well and finding optimal configurations can be a challenging task [21]. Therefore, other approaches have been proposed that take a more proof based approach, where it is explanatory why they work well, an example of this is the scattering convolution network.
2.3.1 Scattering convolution networks

As explained in the introduction, it is clear that a major difficulty of image classification comes from the considerable amount of rotational or scale variability in images. Bruna and Mallat, 2013 [21] came with invariant scattering networks, a different type of network in which this variability is eliminated. The elimination of these variabilities is done by scattering transformations, using wavelets. Without going into much depth on how wavelets are mathematically substantiated, the basic idea of the scattering network is to have wavelet transform kernels to make the input representation that is invariant to translations, rotations, or scaling. The actual classification is done using a SVM or PCA classifier.

Even though Bruna and Mallat have shown state-of-the-art classification performances on handwritten digit recognition and texture discrimination, scattering networks need carefully chosen wavelet kernels based on mathematical models. The optimal kernels are different for different image classification problems. Although, this results in very good performance for specific smaller datasets, the scattering network’s performance is not very good on more complex datasets with a lot of variability. In this paper we hope to combine the strengths of both the scattering convolution network, by using mathematical and biological apriori determined models, and at the same time the CNN, for its ability to learn.
CNN’s are very much based on the classical artificial neural network (ANN), therefore it is important to understand the concept of an ANN before explaining the concept of a CNN.

3.1 Artificial neural network

An artificial neural network (ANN) is a model in machine learning used to approximate complex functions based on a certain input. The concept of an ANN, as with many developments in the artificial intelligence, is fundamentally based on how the human brain works and in particular how neurons in the brain interact. The first steps in the development of the ANN goes back to Warren S. McCulloch, in 1943 [22]. He developed a mere logical model of how neurons in the brain work which had no ability to learn. Since then the research on neural networks split into a biological approach and an artificial intelligence approach. The biological approach endeavored a model of the brain as accurate as possible, whereas the other approach focused more on applications in the artificial intelligence (AI). In 1958, Rosenblatt [23] came with the idea of a perceptron, a simplified mathematical model of a neuron in the brain. The perceptron was very much based on the earlier works of McCulloch and Pitts [22], however the perceptron had the ability to learn. Furthermore [Rosenblatt] implemented a perceptron on a custom hardware machine called the Mark I, and showed it was able to learn classification of simple shapes in a 20x20 input image [24]. This ability to learn was a huge step for AI, and raised many high expectations. The New York Times at that time, even reported on the Mark I that “The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.” [25].

The interest in perceptrons rapidly decreased with a book published by [Minsky and Papert] on the mathematical analysis of perceptrons. [Minsky and Papert] showed that the perceptron was not able to learn more complex functions, because it was limited by a single layer. An example is the exclusive-OR (XOR) logic gate, which is impossible to model with a perceptron. According to [Minsky and Papert] the solution was to stack multiple perceptrons, which is now known as the multilayer perceptron (MLP) or feed-forward neural network, types of an ANN. However, the learning method used by the perceptron did not work for such a model. Therefore the interest of the ANN’s declined for a while.

ANN’s regained interest in 1986 after a proposed backpropagation algorithm by Rumelhart et al. [27]. The backpropagation algorithm made it possible to actually let the network improve by learning from its output error. However the backpropagation algorithm is computationally expensive and back in 1986 there was far from enough computing power to effectively train the network. Therefore, it was still not a method that was actively used within AI, simply because other methods that required much less computing power were far more popular (e.g. SVM’s and decision trees).

Recently in the late 2000’s, ANN’s regained interest again, mainly because of faster GPU implementations which have only been possible with recent improvements [28]. Consequently,
ANN’s have been extremely successful since. An example of this success is the feed-forward neural network from the research group of Jürgen Schmidhuber at the Swiss AI Lab IDSIA, which has won multiple competitions on machine learning and pattern recognition [29].

3.1.1 Single layer network (Perceptron)

As discussed above the most simple form of an ANN is the perceptron, understanding of the perceptron provides a good basis for understanding the more complex networks. The perceptron developed by Rosenblatt [23] is a mathematical model of the biological neuron. Consequently the model of an perceptron is very much based on our understanding of neurons in the brain. Neurons in the brain (Figure 3.1) have axon terminals that are connected to dendrites of multiple other neurons. Through this connection it communicates using an chemical signal going from the axon terminals to the dendrites. In this way the neuron receives multiple chemical signals with different amplitudes. If the sum of all signals together achieve a certain amplitude, from the cell body a signal will be transmitted to all its axon terminals.

![Figure 3.1: Simple representation of a neuron in the brain with its axon terminals and dendrites. The arrows indicate the flow of a signal in the neuron, the signal will only trough the axon to the axon terminals if a certain threshold is reached in the cell body. (Source: "Anatomy and Physiology" by the US National Cancer Institute's Surveillance, Epidemiology and End Results (SEER) Program.)](image)

The perceptron is based on the neuron and works similar, it calculates a weighted sum of all its inputs and based on an activation function the perceptron outputs a signal itself. The activation function can be seen as a threshold function, that ultimately defines the output. An example of a perceptron with three inputs can be seen in Figure 3.2.

![Figure 3.2: Perceptron with three input units, the weighted sum of the input units is forwarded to an activation function](image)

The perceptron is a linear classifier, that means it is able to linearly separate classes in a classification problem. This is simple to see, when Figure 3.2 is written as a formula,

\[ K(w_1x_1 + w_2x_2 + w_3x_3) \]  

(3.1)
where \( x_i \)'s are the inputs and \( K \) is an activation function. Because the perceptron is a linear classifier, it is not able to classify everything correctly if the data is not linearly separable.

### 3.1.2 Multilayer perceptron

The multilayer perceptron (MLP), also called feed-forward neural network, is a type of ANN which is combination of multiple subsequent interconnected perceptrons, i.e. multiple layers of perceptrons. The goal of an ANN is to approximate some complex function \( f^* \), where the function \( f^* \) gives the correct output (this can also be a vector in the case of multiple outputs) for a given input. For example in a classification problem, \( f^* \) outputs the class of a given input \( x \).

A simple fully-connected ANN consisting of four layers is shown in Figure 3.3, this network is called fully-connected because all nodes in subsequent layers are interconnected. The first layer in the network is called the input layer, each node in this layer receives a single value and will pass its value multiplied by a weight to every node in the next layer.

The next two layers in the network are called hidden layers, these layers will receive a weighted sum from all outputs of the previous layer. The hidden layer then feeds this sum into an activation function, the result of this activation function will be forwarded to every node in the next layer. Note that a network may consist of any number of hidden layers, but generally at least one hidden layer otherwise the network is simply a perceptron. Hence, if our input vector is \( x = [x_1 \ x_2 \ \cdots \ x_N]^T \) and the weight vector to the first node in layer \( l \) is \( w^{(l)} = [w^{(l)}_{11} \ w^{(l)}_{21} \ \cdots \ w^{(l)}_{N1}]^T \). Then, the weighted input for node \( j \) in hidden layer \( l \), denoted as \( net^{(l)}_j \), becomes:

\[
net^{(l)}_j = (w^{(l)}_j)^T o^{(l-1)} = \sum_{i=0}^{N} w^{(l)}_{ij} o^{(l-1)}_i \tag{3.2}
\]

\[
o^{(l)}_i = \begin{cases} x_i & \text{if } l = 0 \ (1 \text{ is the input layer}) \\ K(net^{(l)}_i) & \text{else} \end{cases} \tag{3.3}
\]

where \( K \) is the activation function.

The final layer in the network is called the output layer, this layer will output a value that says something about the input. For example, in the case of a classification problem the number of nodes in the output layer corresponds with the number of classes and each node will give a positive or negative indication of whether the input belongs to that class.

![Figure 3.3: ANN with an input layer, two hidden layers and one output layer](image)

During training of the network, we are trying to approximate the function \( y = f^*(x) \). The training instances will give us examples of what \( y \) should be at certain \( x \). So for every training instance the input layer and output layer are known, namely the input layer is \( x \) and the output layer is \( y \). However, the weights in the network are not specified by training instances, and the network will have to decide how to adjust them in order to better approximate \( f^*(x) \).

In order to say something about how well the network function, \( f(x) \), approximates \( f^*(x) \), we need to decide on something called an error function or cost function. An error function is
essentially a measure of how far a certain solution is from the optimal solution. An example of a commonly used error function is the mean squared error (MSE):

\[
MSE = \frac{1}{n} \sum_{i=1}^{n} (o_i - y_i)^2
\]  

(3.4)

Where \(o_i = f(x_i)\) and \(y_i = f^*(x_i) \forall i \in (1, ..., n)\) are predictions and actual values corresponding with \(n\) training examples, respectively. Another commonly used error for a classification problem function is the categorical cross entropy:

\[
H(o_i, y_i) = -\sum_{i=1}^{n} y_i \log(o_i)
\]  

(3.5)

Simply put, the categorical cross entropy originating from the information theory, is a method for measuring the difference between two distributions (in this case, the real labels and predicted labels), which is exactly what we want from a cost function.

3.1.3 Activation function

A weighted sum of all inputs to a node is passed into an activation function. There are several activation functions that can be used, one of the most basic ones is a simple step function, which given a weighted sum, outputs 0 or 1 depending on whether the sum is lower or higher than a certain threshold. A more conventional activation function is the logistic function or sigmoid. The logistic function is a “S” curved function (see Figure 3.4a), with the following equation:

\[
K(x) = \frac{1}{1 + e^{-x}}
\]  

(3.6)

Another commonly used activation function is the rectified linear unit (ReLU) [30]. The ReLU function (see Figure 3.4b) is defined as,

\[
K(x) = \max(0, x)
\]  

(3.7)

While the logistic activation function is probably more conforming to the biological neuron [31], the ReLU has been shown to perform better on deep neural networks [30], i.e. networks with 3 or more layers. The main advantage of the ReLU function over the logistic function is the fact that the gradient will remain significant even for high values. Whereas the logistic function’s gradient will approach zero for high values, the ReLU’s gradient will remain 1. Since the gradient is used whilst training the network, using the ReLU was shown to significantly decrease training time [30]. In this paper’s experiments, we will use the ReLU as the activation function, unless otherwise specified.

3.1.4 Backpropagation

When setting up an ANN the weights are initialized randomly, (albeit in a certain interval), therefore these weights most likely need to be adjusted in order for the network to perform better. In 1986, Rumelhart et al. [27] introduced the backpropagation algorithm for adjusting these weights. The backpropagation algorithm adjusts the weights with respect to a certain cost function by calculating the gradient of the cost function with respect to all the weights in the network. In other words, it computes how much each individual weight in the network contributes to the cost function (i.e. output error) and in which direction (i.e. positive or negative) it needs to be changed in order to decrease the cost function. Essentially this method searches for a (local) minimum of the cost function. The process of finding a minimum is also called gradient descent.

In order to get a understanding of how the backpropagation algorithm works, we will go through the mathematical steps first. Assume we are using the MSE as error function (note that in the experiments we will use the cross entropy, however MSE is easier for this explanation),

\[
E = \frac{1}{2} (o - y)^2
\]  

(3.8)
Figure 3.4: Two examples of commonly used activation functions. a) Logistic function (or sigmoid function), \( y = \frac{1}{1+e^{-x}} \). b) Rectified linear unit (ReLU), \( y = \max(0, x) \)

Where \( E \) is the error, \( o \) is the output of the network and \( y \) is the actual output corresponding with the training instance. In order to adjust the weights with respect to the output error, we need to find the derivative of the error with respect to each weight in the network:

\[
\frac{\partial E}{\partial w_{ij}^{(l)}} = \frac{\partial}{\partial w_{ij}^{(l)}} \frac{1}{2} (o - y)^2
\]  

The above equation can be solved by using the product rule and the fact that the output node is a weighted sum of all output nodes of the previous layer passed into an activation function:

\[
\frac{\partial E}{\partial w_{ij}^{(l)}} = \frac{\partial E}{\partial o_j^{(l)}} \frac{\partial o_j^{(l)}}{\partial net_j^{(l)}} \frac{\partial net_j^{(l)}}{\partial w_{ij}^{(l)}}
\]  

Where \( o_j^{(l)} \) is the output of node \( j \) in layer \( l \), and \( net_j^{(l)} \) is the sum of all inputs to node \( j \) in layer \( l \), i.e. \( o_j^{(l)} = K(net_j^{(l)}) \) with \( K \) the activation function. Now computing each term is fairly straightforward:

\[
\frac{\partial net_j^{(l)}}{\partial w_{ij}^{(l)}} = \frac{\partial}{\partial w_{ij}^{(l)}} \sum_{k=1}^{n} w_{kj}^{(l-1)} o_k^{(l-1)} = o_i^{(l-1)}
\]  

because only one term in this sum depends on \( w_{ij} \), namely when \( k = i \).

The second term in eq. 3.10 only contains the activation function, if we are using the logistic function as activation function (were are simply differentiating eq 3.6), we get:

\[
\frac{\partial o_j^{(l)}}{\partial net_j^{(l)}} = \frac{\partial}{\partial net_j^{(l)}} K(net_j^{(l)}) = K(net_j^{(l)})(1 - K(net_j^{(l)}))
\]  

or if we are using the ReLU instead:

\[
\frac{\partial o_j^{(l)}}{\partial net_j^{(l)}} = \frac{\partial}{\partial net_j^{(l)}} \max(0, net_j^{(l)}) = \begin{cases} 1 & \text{if } o_j^{(l)} > 0 \\ 0 & \text{if } o_j^{(l)} \leq 0 \end{cases}
\]  

The first term in eq. 3.10 depends on the output node’s layer. If the output node belongs to the last layer, i.e. the output layer, the derivative becomes:

\[
\frac{\partial E}{\partial o_j} = \frac{\partial}{\partial o_j} \frac{1}{2} (o_j - y_j)^2 = o_j - y_j
\]
When the output node belongs to a hidden layer, we have to recursively take the derivative in the following way:

$$\frac{\partial E}{\partial w_{ij}} = \delta_j^{(l)} o_j^{(l-1)}$$  \hspace{1cm} (3.15)

Where $\delta_j$ is defined as follows for the sigmoid activation function:

$$\delta_j^{(l)} = \frac{\partial E}{\partial o_j^{(l)}} \frac{\partial o_j^{(l)}}{\partial \text{net}_j^{(l)}} = \begin{cases} (o_j^{(l)} - y_j^{(l)}) o_j^{(l)} (1 - o_j^{(l)}) & \text{if } j \text{ is an output neuron,} \\ \sum_{k \in K} \delta_k^{(l+1)} w_{jk}^{(l+1)} o_j^{(l)} (1 - o_j^{(l)}) & \text{if } j \text{ is an inner neuron.} \end{cases}$$  \hspace{1cm} (3.16)

Where $K$ are all the nodes receiving input from $j$. In the case of the ReLU activation function we get:

$$\delta_j^{(l)} = \frac{\partial E}{\partial o_j^{(l)}} \frac{\partial o_j^{(l)}}{\partial \text{net}_j^{(l)}} = \begin{cases} (o_j^{(l)} - y_j^{(l)}) & \text{if } j \text{ is an output neuron and } o_j^{(l)} > 0, \\ \sum_{k \in K} \delta_k^{(l+1)} w_{jk}^{(l+1)} & \text{if } j \text{ is an inner neuron and } o_j^{(l)} > 0, \\ 0 & \text{if } o_j^{(l)} \leq 0, \end{cases}$$  \hspace{1cm} (3.17)

Now that we have a way of computing the gradient with respect to each weight, we need a method for adjusting the weights. Generally during training we compute the output for a batch of training instances, then we compute the gradient of the error for this batch, i.e. eq. 3.8 becomes:

$$E = \sum_{i} \frac{1}{2} (o_i - y_i)^2$$  \hspace{1cm} (3.18)

Where $o = \{o_1, o_2, \ldots, o_n\}$ is on batch of training instances. The weights are then updated based only on this batch of training instances. This process is repeated until all training instances have been processed. The whole process of processing all weights is repeated for a chosen number of iterations, also called epochs. For example, if you have 1000 training examples, and your batch size is 200, then it will take 5 iterations to complete 1 epoch.

Formally during each weight update a change, $\Delta w$, is added to the weights. Denoting the weights at the $t$-th training iteration as $w_{ij,t}$ we are updating in the direction of the negative gradient, weighted by a learning rate:

$$w_{ij,t+1} = w_{ij,t} - \eta \frac{\partial E}{\partial w_{ij,t}}$$  \hspace{1cm} (3.19)

The second term is negative because the gradient of the cost function gives the direction of steepest ascent. The learning rate, $\eta$, is a parameter that determines how large of a step to take in the direction of the negative gradient. There are multiple approaches for setting the learning rate. One could simply decide the value of a fixed learning rate at the beginning of training. However there are disadvantages of a fixed learning rate, if the learning rate is too small, the training will take many more training iterations, and thus a lot more time. On the other hand, if the learning rate is too large it might never converge to a minimum, i.e. it will step over the minimum. However, one can also argue that as the cost function gets closer to the minimum, its gradient will be lower and therefore it will converge even with bigger learning rates.

Multiple different approaches have been proposed, in order to achieve a faster convergence of the cost function and to deal with the limitations of a fixed learning rate. In this paper we will discuss AdaGrad and AdaDelta, which were used in the experiments.

**AdaGrad**

In 2011, Duchi et al. published a method called AdaGrad [32]. In this method each weight has a separate learning rate dependent on previous gradients. The update rule for AdaGrad using is defined as follows:

$$\Delta w_{ij,t} = - \frac{\eta}{\sqrt{\sum_{\tau=1}^{t} g_{ij,\tau}}} g_{ij,t}$$  \hspace{1cm} (3.20)
Where $\eta$ is a global learning rate chosen at the beginning of training and $g_t$ is the gradient at the $t$-th iteration:

$$g_{ij,t} = \frac{\partial E}{\partial w_{ij,t}} \quad (3.21)$$

Simply put, AdaGrad increases the learning rate for smaller gradients and decreases the learning rate for greater ones. The idea behind AdaGrad is to even out the magnitude of change for all weights. Additionally, it decreases the learning rate over time. However there are two important drawbacks with AdaGrad, first of all if the initial gradients are very large the learning rate will be small for the remainder of the training. This can be solved by increasing the global learning rate, $\eta$, but that means AdaGrad is very dependant on an initial value for $\eta$. Furthermore, as training progresses the learning rate will continue decreasing, up to the point that it is essentially zero, and thus stopping training completely.

AdaDelta

AdaDelta, published by [Zeiler] in 2012, is a modified version of AdaGrad that tries to solve the limitations of AdaGrad [33]. Instead of progressively decreasing the learning rate, AdaDelta decays the influence of gradients at previous training iterations the older they are. Since storing previous gradients is inefficient, this is implemented by exponentially decaying and saving a running average, $E[g^2]_t$:

$$E[g^2]_{ij,0} = 0 \quad (3.22)$$
$$E[g^2]_{ij,t} = \rho E[g^2]_{ij,t-1} + (1 - \rho)g_{ij,t}^2 \quad (3.23)$$

The resulting weights update is then:

$$\Delta w_{ij,t} = -\frac{\eta}{\text{RMS}[g]_{ij,t}}g_{ij,t} \quad (3.24)$$

where:

$$\text{RMS}[g]_{ij,t} = \sqrt{E[g^2]_{ij,t} + \epsilon} \quad (3.25)$$

where a constant $\epsilon$ is added to better condition the denominator as in [34]. Additionally, it takes previous values of $\Delta w_t$ into account in a similar way. So instead of having a fixed $\eta$ in eq. 3.24 the numerator is dependent on $\Delta w_t$:

$$E[\Delta w^2]_{ij,t} = \rho E[\Delta w^2]_{ij,t-1} + (1 - \rho)\Delta w_{ij,t}^2 \quad (3.26)$$

The resulting weights update is then:

$$\Delta w_{ij,t} = -\frac{\text{RMS}[\Delta w]_{ij,t-1}}{\text{RMS}[g]_{ij,t}}g_{ij,t} \quad (3.27)$$

where:

$$\text{RMS}[\Delta w]_{ij,t} = \sqrt{E[\Delta w^2]_{ij,t} + \epsilon} \quad (3.28)$$

The experiments show that AdaDelta is a lot less sensitive to the initial choice of parameters (i.e. $\eta$) compared to AdaGrad [33]. Moreover, AdaDelta shows faster convergence of the test error than AdaGrad [33].

21
Summarizing the equations in a MLP:

**Feed-forward:**

\[ \text{net}^{(l)}_j = (w^{(l)}_j)^T o^{(l-1)} = \sum_{i=0}^{N} w^{(l)}_{ij} o^{(l-1)} \]  
\[ o^{(l)}_i = \begin{cases}  
  x_i & \text{if } l = 0 \text{ (1 is the input layer)} \\
  \text{K}(\text{net}^{(l)}_i) & \text{else} 
\end{cases} \]

**Backpropagation:**

\[ \frac{\partial E}{\partial w^{(l)}_{ij}} = \delta^{(l)}_j o^{(l-1)} \]  
\[ \delta^{(l)}_j = \begin{cases}  
  \frac{\partial E}{\partial o^{(l)}_j} \text{K}'(\text{net}^{(l)}_j) & \text{if } l \text{ is the output layer,} \\
  \sum_k \delta^{(l+1)}_k w^{(l+1)}_{jk} \text{K}'(\text{net}^{(l)}_j) & \text{if } l \text{ is an inner layer} 
\end{cases} \]  
\[ w^{(l)}_{ij,t+1} = w^{(l)}_{ij,t} - \eta \frac{\partial E}{\partial w^{(l)}_{ij,t}} \]

### 3.1.5 Backpropagation example

In order to get a better understanding of how the backpropagation algorithm works this section briefly illustrates an example of a single iteration of the algorithm using only one training instance. Assume we start with the network shown here in Figure 3.5:

![Figure 3.5: Simple MLP consisting of an input layer with 2 nodes, hidden layer with 2 nodes and an output layer with 1 node, the weights and inputs are both set.](image)

And suppose we are using the ReLU activation function and the following training instance: \( \{x_1 = 0.55, x_2 = 0.9; y = 0.5\} \). For this example we use the node name as notation for the output of that node, i.e. \( H_1 \) is the output of \( H_1 \). Feed-forwarding this training instance into the network gives us the following result for all nodes:

\[ H_1 = \max(0, 0.4 \times 0.55 + 0.8 \times 0.9) = 0.94 \]  
\[ H_2 = \max(0, 0.2 \times 0.55 + 0.5 \times 0.9) = 0.56 \]  
\[ O_1 = \max(0, 0.6 \times 0.94 + 0.3 \times 0.56) = 0.732 \]

and therefore the output of the cost function is:

\[ \text{cost} = \frac{1}{2} (0.732 - 0.5)^2 = 0.027 \]

Note that we use the MSE cost function instead of the entropy loss function (eq. 3.5), we do this to simplify the example. Using eq. 3.15, 3.17 and a simple learning rule (eq. 3.19) with
The output of the cost function did decrease compared to the initial network:  
$$w_{11,1}^{(2)} = w_{11,0}^{(2)} - 1.0((O_1 - y) \times H_1) = 0.6 - 1.0(0.232 \times 0.94) = 0.382$$  
$$w_{21,1}^{(2)} = w_{21,0}^{(2)} - 1.0((O_1 - y) \times H_2) = 0.3 - 1.0(0.232 \times 0.56) = 0.170$$  
(3.38)  
(3.39)

Where $w_{ij,t}^{(l)}$ is the weight at layer $l$, at training iteration $t$ and between node $i$ of layer $l$ and node $j$ of layer $(l)$. Doing this for the weights for the input layer gives us:

$$w_{11,1}^{(1)} = w_{11,0}^{(1)} - 0.1((O_1 - y) \times H_1 \times I_1) = 0.4 - 1.0(0.232 \times 0.94 \times 0.55) = 0.280$$  
$$w_{12,1}^{(1)} = w_{12,0}^{(1)} - 0.1((O_1 - y) \times H_2 \times I_1) = 0.2 - 1.0(0.232 \times 0.56 \times 0.55) = 0.129$$  
$$w_{21,1}^{(1)} = w_{21,0}^{(1)} - 0.1((O_1 - y) \times H_1 \times I_2) = 0.8 - 1.0(0.232 \times 0.94 \times 0.9) = 0.604$$  
$$w_{22,1}^{(1)} = w_{22,0}^{(1)} - 0.1((O_1 - y) \times H_2 \times I_2) = 0.5 - 1.0(0.232 \times 0.56 \times 0.9) = 0.383$$  
(3.40)  
(3.41)  
(3.42)  
(3.43)

This finishes up one iteration of the backpropagation algorithm. Feed-forwarding the training instance with the new weights will show us the improvement:

$$H_1 = \max(0, 0.280 \times 0.55 + 0.604 \times 0.9) = 0.698$$  
$$H_2 = \max(0, 0.129 \times 0.55 + 0.383 \times 0.9) = 0.416$$  
$$O_1 = \max(0, 0.382 \times 0.698 + 0.170 \times 0.416) = 0.337$$  
(3.44)  
(3.45)  
(3.46)

The output of the cost function did decrease compared to the initial network:

$$\frac{1}{2}(0.337 - 0.5)^2 = 0.013 < \frac{1}{2}(0.732 - 0.5)^2 = 0.027$$  
(3.47)

However, we did overshoot the correct value, which could indicate our learning rate is too large.

### 3.2 Convolutional neural network

A convolutional neural network (CNN) is a type of ANN, where the inter-layer connectivity of nodes is directly inspired by the visual mechanisms of mammals. As was mentioned in section 2.1, we know from Hubel and Wiesel’s early work on the cat’s visual cortex [17], that the visual cortex contains an arrangement of interconnected simple and complex cells. The simple and complex cells fire on stimuli in small regions of the visual field, called receptive fields.

The sub-regions are tiled with overlap to cover the entire visual field. These cells act as local filters over the input space and are well-suited to exploit the strong spatially local coherence present in natural images. Since some mammals have a very developed visual cortex and have one of the best visual systems in existence, it seems obvious to create models inspired by the mechanisms of such visual systems.

There are three very important properties that characterize a CNN. 1) CNN’s have sparse inter-connectivity between layers, so instead of having a fully connected network, there are only a number of nodes connected to a node in the next layer. 2) CNN’s have shared weights, so rather than having an unique weight for each interconnected pair of nodes, CNN’s share their weights between multiple connections. These two characteristics are a result of taking a convolution. 3) CNN’s perform sub-sampling, which reduces spatial resolution in order to make it more spatially invariant. We will elaborate more on these properties in section 3.2.2 and 3.2.3.

#### 3.2.1 Convolution operation

In mathematics convolution is an operation between two functions, that produces a new function. The convolution is defined for both continuous functions and discrete functions. The convolution
operation for two continuous functions \( f \) and \( g \) is defined as follows:

\[
(f * g)(x) \overset{\text{def}}{=} \int_{-\infty}^{\infty} f(a) g(x-a) \, da
\]  

(3.48)

\[
= \int_{-\infty}^{\infty} f(x-a) g(a) \, da.
\]  

(3.49)

Where \( * \) is the mathematical symbol for the convolution operation. What convolution intuitively does is for each point, \( a \), along the real line multiply the value \( f(a) \) with the function \( g(x) \) centered around \( a \), and add all these functions together.

Since data in machine learning is most often discrete and represented as a multidimensional matrix, the discrete convolution operation is more useful, which is similarly defined:

\[
s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)
\]  

(3.50)

In CNN’s, the functions \( x, w \) and \( s \) are often referred to as respectively input, kernel and feature map. For a two-dimensional input image \( I \) and kernel \( W \), we get a two-dimensional feature map:

\[
s(i,j) = (I * W)(i,j) = \sum_{m} \sum_{n} I(m,n)W(i-m,j-n)
\]  

(3.51)

The convolution operation on an input image can also be explained as sliding a flipped two dimensional kernel over the image. The reason for flipping the kernel is to keep the convolution’s commutative property. While this commutative property is useful for mathematical proofs, it is not important in CNN’s. Instead, most CNN implementations use another operation closely related to convolution called cross-correlation. Cross-correlation is similar to convolution and defined as follows:

\[
s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t+a)
\]  

(3.52)

The cross-correlation operation on an image can be explained as sliding a two dimensional kernel over the image, an example of this for a 4x3 image and 2x2 kernel can be seen in Figure 3.6.

![Figure 3.6: Example of a two-dimensional cross-correlation on a 2x3 input image and an 2x2 kernel. (Source: Ian Goodfellow and Courville [35])](image-url)
3.2.2 Convolution layer

The convolution layer is what makes a CNN distinct from other ANN’s, the layer performs convolution on its input with multiple different kernels. As with a multilayer perceptron (MLP), the output values of convolution are passed into a non-linear activation function. Hence, for a single HxH kernel, \( w \), in a convolution layer with input, \( net \), and activation function, \( K \), a forward pass is formulated as follows:

\[
o^{(l+1)}(i,j) = K((w^{(l+1)} \ast o^{(l)}))(i,j)) = \sum_a \sum_b K(w(a,b) \cdot o^{(l)}(i - a, j - b))
\]  

(3.53)

where \( o^{(l)}(i,j) = K(net^{(l)}(i,j)) \) is the output of the convolution layer and \( i, j \) are nodes indices denoted with two numbers since we are dealing with input images. If we calculate this for every \( i \) and \( j \), i.e. all pixels in the image, we get a new output feature map. Furthermore, doing this for multiple kernels gives us multiple output feature maps.

As mentioned above, there are two important properties of a CNN, which are the result of taking a convolution, namely, sparse connectivity and shared weights.

Sparse connectivity

Generally ANN’s are fully-connected, i.e. all nodes in subsequent layers are interconnected. One of the reasons for using fully-connected layers is to simplify the design. Another reason can be, that by using more connections a network is able to represent more complex functions. However, it usually also introduces redundancy and will most likely increase the number of computations as a result of more parameters [36]. Moreover it might also be prone to overfitting, meaning it will perform a lot better on the training instances compared to unseen instances (test set).

As a result of taking a convolution, CNN’s are not fully-connected, but instead have a sparse connectivity. Two effects caused by sparse connectivity are a smaller amount of computations, and less redundancy between parameters. However as mentioned before, the first and foremost motivation for CNN’s design is the resemblance with the mammal visual cortex. The sparse connectivity is the direct result of taking a convolution, namely the number of inputs for a node is equal to the size of the convolution kernel. This is illustrated in Figure 3.7 where each square is a node (for example a pixel in an image), the size of the convolution kernel (blue) is 9 and thus there are only 9 input nodes to a single node in the subsequent layer (red). The difference this makes compared to a MLP (fully-connected) is illustrated in the first transition of Figure 3.8.

![Figure 3.7: Illustration of sparse connectivity as a result of the convolution operations.](image-url)
Sparse connectivity

Weight sharing

Figure 3.8: Illustration of the difference in connections between a MLP (left) and a CNN (right). In the last network the connections with the same color share the same weight.

Shared weights

Another result of taking a convolution is shared weights. Since we take the same convolution across the whole image, the weights are the same for every input. One can understand this by sliding the kernel in Figure 3.7 across the whole image, the weights stay the same only the input changes. This effect of weight sharing is also illustrated in Figure 3.8.

3.2.3 Max pooling

Max pooling is a very simple operation somewhat similar to convolution, but instead of picking a weighted average for all values within a region, max pooling takes the maximum value within a region as output. However, in contrast with the convolution operation where the kernel regions overlap, with max pooling we will use a tiled method, i.e. the max pooling regions never overlap.

There are several motivations for why max pooling is preferred. First of all, by eliminating non-maximal values, the amount of data in next layers is reduced and therefore reduces the number of computations. Secondly, it provides a form of translation invariance [37]. In order to understand why max pooling may provide translation invariance, assume we have a max pooling region of 2x2. There are 8 directions in which one can shift the image by a single pixel, i.e. horizontal, vertical and diagonal. Now, for 3 out of the 8 possible shifts the region’s maximum stays within the 2x2 region. Consequently, there is good chance the output of the max pooling region is exactly the same as before the shift.

To give a illustrative example of this, assume we have the 2x2 pooling region delineated in Figure 3.9. Before and after shifting the max value in the region is 8, therefore the shifting did not change anything for region’s output. One can easily see that this is the case in 3 of the 8 possible shifts. Of course this situation illustrated here will not always occur, but it does give some feeling on why max pooling provides a form of translation invariance.

![Max Pooling Illustration](image)

Figure 3.9: Illustrative example of translation invariance due to max pooling. After the shift to the right, the max pooling region will still pass the same value to the next layer. This happens in 3 of the 8 possible shift directions.

The whole convolution layer including max pooling is shown in Figure 3.10. In this example random convolution kernels were used and a max pooling region of 2x2.
3.2.4 Backpropagation

As for the MLP, we will use backpropagation in order to train the CNN. Although the concept of backpropagation is the same as for the MLP, there are a couple of important differences. As explained above, the convolutional layers consists of a convolution and a max-pooling operation.

Backpropagation for the max-pooling operation is done by backpropagation only on the max value, because only this max-value contributes to the next layer and thus to the error. It is important to note that, in order to backpropagate only the max value, we need to keep track of the position of the max value. In the case of a tie, i.e. two values in the max-pooling region are the max value, we need to decide on how to backpropagate. We have multiple solutions: 1) We could take one of the values at random. 2) We could always take a certain one, for example always take the value closest to the top-left of the max-pooling region. 3) We could backpropagate all max values sharing the error equally. The latter is used for the experiments in this paper.

Backpropagation for the convolution operation is similar to that of the MLP. After all, the convolution layer is a MLP layer with sparse connectivity and shared weights. Therefore, we can use the equations from section 3.1.4.

In order to simplify the derivation we disregard max pooling in this explanation, but as mentioned above, backpropagating the max pooling layer is easy and does not involve any equations. We start with formulating the feed-forward equation (eq. 3.53) recursively for layer \( l + 1 \) and using the definition of a convolution (eq. 3.51):

\[
net^{(l+1)}(i, j) = (w^{(l+1)} * K)(net^{(l)})(i, j) = \sum_{a} \sum_{b} (w^{(l+1)}(a, b) K(net^{(l)}(i-a, j-b)))
\] (3.54)

Here \( net^{(l)} \) is the input of layer \( l \) and \( K \) is the activation function. In order to backpropagate using gradient descent, we need the gradient of the error with respect to the layer’s weights:

\[
\frac{\partial E}{\partial w^{(l)}(a, b)} = \sum_{i} \sum_{j} \delta^{(l)}(i, j) \frac{\partial net^{(l)}(i, j)}{\partial w^{(l)}(a, b)}
\] (3.56)

![Figure 3.10: Example of the complete convolution layer including max-pooling, using random convolution kernels and a max pooling region of 2x2](image)
where $\delta^{(l)}(i,j)$ is defined as:

$$
\delta^{(l)}(i,j) = \frac{\partial E}{\partial \text{net}^{(l)}(i,j)} = \sum_{i',j'} \frac{\partial E}{\partial \text{net}^{(l+1)}(i',j')} \frac{\partial \text{net}^{(l+1)}(i',j')}{\partial \text{net}^{(l)}(i,j)}
$$

The double sum over all the values in $\text{net}^{(l+1)}$ arises from the fact that, when backpropagating, $\text{net}^{(l)}(i,j)$ is only dependent on the values in $\text{net}^{(l+1)}(i,j)$. Further solving eq. 3.57 we get:

$$
\delta^{(l)}(i,j) = \sum_{i',j'} \frac{\partial E}{\partial \text{net}^{(l+1)}(i',j')} \frac{\partial \text{net}^{(l+1)}(i',j')}{\partial \text{net}^{(l)}(i,j)}
$$

Now using the fact that $\delta^{(l)} = \delta^{(l+1)}(k, l)$, we get:

$$
\delta^{(l)}(i,j) = \sum_{i',j'} \delta^{(l+1)}(i',j')(w^{(l+1)}(a,b)K'(\text{net}^{(l)}(i,j)))
$$

Since the last term is non-zero only when $\text{net}^{(l)}(i' - a, j' - b) = \text{net}^{(l)}(i,j)$, we can write:

$$
\delta^{(l)}(i,j) = \sum_{i',j'} \delta^{(l)}(i',j')w^{(l+1)}(a,b)K'(\text{net}^{(l)}(i,j))
$$

Using the definition of a convolution, we get:

$$
\delta^{(l)}(i,j) = \sum_{i',j'} \delta^{(l+1)}(i',j')w^{(l+1)}(a,b)
$$

Where $W$ is the flipped kernel, since $w(-x,-y)$ is the flipped kernel of $w(x,y)$.

Now going back to eq. 3.56, we only need to solve the right part of the derivative:

$$
\frac{\partial E}{\partial w^{(l)}(a,b)} = \sum_{i} \sum_{j} \delta^{(l)}(i,j) \frac{\partial \text{net}^{(l)}(i,j)}{\partial w^{(l)}(a,b)}
$$

In the first two steps, we use the definition of $\text{net}^{(l)}(i,j)$ (eq. 3.54) and the fact that everything in the derivative sum becomes 0 if not dependent on $w^{(l)}(a,b)$. In the last step we use the same trick as in eq. 3.63.

Summarizing the equations in a CNN:

**Feed-forward (without max-pooling):**

$$
\text{net}^{(l+1)}(i,j) = (w^{(l+1)} \ast K(\text{net}^{(l)}))(i,j)
$$

$$
\phi^{(l)}(i,j) = K(\text{net}^{(l)}(i,j))
$$

**Backpropagation:**

$$
\frac{\partial E}{\partial w^{(l)}(a,b)} = (\delta^{(l)} \ast K(\text{net}^{(l)}))(a,b)
$$

$$
\delta^{(l)}(i,j) = (\delta^{(l+1)} \ast (w^{(l+1)}))(i,j)K'(\text{net}^{(l)}(i,j))
$$
The concept of receptive fields neural networks (RFNN) combines both the idea of a scattering network to have fixed filters and on the other hand a CNN for its ability to learn the most effective combination of filters [1]. As mentioned before, CNN’s, using many layers, are very successful in the field of image classification and are able to solve very complex problems. However, in order for a CNN to perform well and prevent it from overfitting a lot of training data is needed [38]. On the other hand scattering networks are very good at learning more complex datasets with a lot of variability, but excel at datasets with less variability [21].

4.1 Theory

Recently, Jacobsen et al. [1] proposed the RFNN model that tries to combine the strengths of CNN’s and scattering networks, by replacing the convolution layers in a CNN with layers that perform convolution of the image with the Gaussian derivative kernels at different scales. This creates a number of feature maps equal to the amount of kernels. Then weighted combinations of these feature maps are passed to a pooling layer. An example of what the Gaussian RFNN model looks like is seen in Figure 4.1.

4.1.1 Gaussian convolution kernels

The motivation behind Gaussian convolutional kernels is: i) As mentioned in section 2.1, it has been proven that Gaussian kernels, up to 3rd and 4th order derivatives, are sufficient to capture all local image features perceivable by humans [11, 12]. ii) Scale-space theory has shown that the Gaussian basis is complete, and is therefore able to construct the Taylor expansion of local structure in an image [18]. This completeness implies that an arbitrary learned weighted combination of Gaussian derivative kernels has, in principle, the same expressive power as a learned kernel in a CNN.

The Gaussian kernels are fundamentally constructed using the Gaussian formula in one dimension:

$$G_{\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp \left( -\frac{1}{2} \left( \frac{x}{\sigma} \right)^2 \right)$$  \hspace{1cm} (4.1)

where $\sigma$ is called the scale of the Gaussian formula. The Gaussian formula for two dimensions is the product of two 1D Gaussians:

$$G_{\sigma}(x, y) = \frac{1}{\sigma^2 \sqrt{2\pi}} \exp \left( -\left( \frac{x^2 + y^2}{2\sigma^2} \right) \right)$$  \hspace{1cm} (4.2)

$$= \frac{1}{\sigma\sqrt{2\pi}} \exp \left( -\frac{1}{2} \left( \frac{x}{\sigma} \right)^2 \right) \cdot \frac{1}{\sigma\sqrt{2\pi}} \exp \left( -\frac{1}{2} \left( \frac{y}{\sigma} \right)^2 \right)$$  \hspace{1cm} (4.3)
As a result the kernel for two dimensions is separable, and thus a convolution between two 1D Gaussian kernels:

$$G_\sigma(x, y) = G_\sigma^x(x) * G_\sigma(y) \tag{4.4}$$

where $G_\sigma(x, y)$ is the 2D kernel, $G_\sigma^x(x)$ is the “horizontal” Gaussian kernel and $G_\sigma(y)$ is the “vertical” Gaussian kernel. The 2D zero order Gaussian kernel is shown in Figure 4.2a. Thanks to the separability, instead of convolution with the 2D Gaussian kernel, we can convolve the image, $I$, with two 1D kernels:

$$I * G_\sigma(x, y) = I * G_\sigma^x(x) * G_\sigma(y) \tag{4.5}$$

This separability “trick” reduces the complexity of a convolution from $O(MNkk)$ to $O(MN2^*k)$ for $M \times N$ image and a kernel size of $k$.

Furthermore, because of separability we can easily compute the higher order Gaussian derivative kernels. For example, for the first order derivative with respect to $x$, we compute the first order derivative of the 1D Gaussian kernel of $x$ and convolve this kernel with the zero order derivative kernel of $y$:

$$G_\sigma^x(x) = \frac{\partial G_\sigma(x)}{\partial x} \tag{4.6}$$

$$G_\sigma^z(x, y) = G_\sigma^x(x) * G_\sigma(y) \tag{4.7}$$

Additionally, it has been shown that Gaussian derivatives of arbitrary order can be expressed using Hermite polynomials [39]:

$$\frac{G_\sigma^z(x)}{\partial x^n} = (-1)^n \frac{1}{(\sigma\sqrt{2})^n} H_n\left(\frac{x}{\sigma\sqrt{2}}\right)G_\sigma(x), \tag{4.8}$$
where the Hermite polynomials satisfy the following recurrence relation:

\begin{align}
    H_0(x) &= 1 \\
    H_1(x) &= 2x \\
    H_{n+1}(x) &= 2xH_n(x) - 2nH_{n-1}(x)
\end{align}

A plot of the 2D first order Gaussian derivative kernel with respect to \( x \) is shown in Figure 4.2b.

A single layer in the RFNN convolves its input with the Gaussian derivative basis kernels, where derivatives are used up to a certain order, but not higher than 4th order. An example of this Gaussian derivative basis at scale 1.5 is shown in Figure 4.3. The feature maps created by convolution with each of these kernels are then added in multiple weighted combinations. So, if we denote the Gaussian basis functions by \( \psi \), one weighted combination, \( j \), is formulated as follows:

\[
    net^{(l+1)}_j = \alpha^{(l+1)}_{ij,1}(o_i * \psi_1) + \alpha^{(l+1)}_{ij,2}(o_i * \psi_2) + \cdots + \alpha^{(l+1)}_{ij,n}(net^{(l)}_j * \psi_n)
\]

where, \( o_i = K(net^{(l)}) \), is the \( i \)-th feature map of the previous layer passed into an activation function, for the first layer this is the input image \( (o_0 = I) \), and \( \alpha_{ij,k} \) is the weight parameter for the feature map \( i \) convolved with \( k \)-th Gaussian basis function. The \( \alpha \) are the parameters that are being learned using backpropagation.

In order to see why this weighted combination has the same expressive power as a learned kernel in a CNN, we start by noting that this equation is arithmetically equivalent to a \( net^{(l)}_j \)
convolved with a weighted combination of basis functions:

\[ \text{net}_{l+1}^{i} = \text{net}_{l}^{j} \ast (\alpha_{1}\psi_{1} + \alpha_{2}\psi_{2} + \cdots + \alpha_{n}\psi_{n}) \] (4.13)

Furthermore, following scale-space theory we know that a scaled kernel in a CNN, can be approximated using the Taylor expansion:

\[ G_{\sigma}(x)F(x) = \sum_{m} G_{\sigma}^m(x)F(x)(x-a)^m \] (4.14)

In other words, eq. 4.14 shows that the scaled kernel can be approximated by a combination of weighted Gaussians derivatives, just like eq. 4.13.

Learning a RFNN model using backpropagation is fairly simple, if we know the gradient of the error, \( \delta^{(l+1)} \), with respect to nodes in layer \( l+1 \), then the gradient of the error w.r.t to \( \alpha_{1}, \ldots, \alpha_{n} \) in layer \( l+1 \) is:

\[ \frac{\partial E}{\partial \alpha_{ij}^{(l+1)}} = \delta_{i}^{(l+1)}(\alpha_{i}^{(l)} \ast \psi_{i}) \] (4.15)

4.1.2 Gabor convolution kernels

The RFNN model is based on the Gaussian derivative basis, and as pointed out in the previous section there is good motivation for using the Gaussian kernel. However, as was mentioned in the introduction of this paper, another family of functions which is argued to correspond closely to the receptive fields in the visual cortex is the Gabor family \[13, 14\], and it has been shown that receptive fields of simple cells can be modelled by the Gabor functions \[40\]. The hypothesised advantage of the Gabor family compared to the Gaussian family is its ability to learn faster due to its flexibility. Therefore, it seems an interesting approach to explore the characteristics or even the benefits of the use of the Gabor function family in the RFNN model.

The Gabor function is defined by a sinusoidal wave multiplied with the zero order Gaussian function:

\[ \text{gabor}(x; f, \lambda, \sigma) = \exp \left( \frac{2\pi x'}{\lambda} \right) G_{\sigma}(x') \] (4.16)

\[ \text{gabor}(x, y; f, \lambda, \sigma) = \exp \left( \frac{2\pi x'}{\lambda} \right) G_{\sigma}(x', y') \] (4.17)

where

\[ \exp(i \frac{2\pi x'}{\lambda}) = \cos\left(\frac{2\pi x'}{\lambda}\right) + i \sin\left(\frac{2\pi x'}{\lambda}\right), \] (4.18)

and

\[ x' = x \cos \theta + y \sin \theta \] (4.19)

\[ y' = -x \sin \theta + y \cos \theta \] (4.20)

Here, \( \lambda \) is the wavelength of the sinusoidal wave, \( \theta \) is the rotation of the kernel and \( \sigma \) is the scale of the Gaussian. Gabor functions can be seen as a sinusoidal wave function under a Gaussian window with scale \( \sigma \). The Euler formula (eq 4.18) makes it that the “real” part of the Gabor function is the cosine multiplied with the Gaussian formula and the “imaginary” part of the Gabor function is the sine multiplied with the Gaussian formula.

Previous research has shown, that in many cases the Gabor family has similar performance to that of the Gaussian derivative family \[41, 42\]. This does not come as a surprise, since the two function families are very similar. Moreover, by using an appropriate choice of parameters, the Gabor functions can be made to look very similar to Gaussian derivatives \[39\], see also Figure [4.4a]

As was explained in the previous section, an important property of the Gaussian derivative family is its completeness and thus having the same expressive power as learned kernels in a
Gaussian

Gabor

(a)

(b)

Figure 4.4: **Left:** Gabor functions can be made very similar to the Gaussian derivatives using the right parameters. Continuous line: first order Gaussian derivative. Dotted line: a parametrized Gabor function, $1.3 \text{gabor}(x; f = 1, \theta = 0, \sigma = 1.2)$. **Right:** This plot shows the similarity between the 2nd order Gaussian and the negative Gabor function, where the wavelength is 3 standard deviations (i.e. 3 times the scale). This plot also shows how the Gabor function is constructed of a multiplication between a cosine and the Gaussian window.

CNN. Although there is no direct proof found for the completeness of the Gabor family and proving completeness goes well beyond the scope of this paper, its similarity with Gaussian derivative family suggests completeness. Moreover, completeness has been proven for the Gabor wave trains [43], which is similar to the Gabor family.

In order to use the Gabor family in the RFNN model, we need to construct a complete basis for the Gabor functions similar to the Gaussian derivative basis. The Gaussian derivative basis can be compared with waves, the higher order derivative the more waves, where the waves’ amplitude decreases farther from the center. In order to create a similar basis using the Gabor function we have to make the wavelengths in the Gabor function dependant on the scale. The spread of the Gaussian window is specified by the scale parameter, where the scale parameter is equivalent to length of one standard deviation. Furthermore, three standard deviations, i.e. three times the scale, span up exactly 99.7% of the function’s area, e.g. for a Gaussian function with scale $\sigma = 1.0$, the interval $[-3, 3]$ will span up 99.7% of the area (see Figure 4.5).

Using this fact, we can specify how many waves, which is determined by the wavelengths, will fall under 99.7% of the Gaussian window. That is, if we want to fit two full waves under 99.7% of the Gaussian window with scale $\sigma = 1$, the wave length has to be 3 times the scale (the complete window of 99.7% is 3 times the scale on both sides, so a wavelength of 3 will fit two waves). An example of this is shown in Figure 4.4b, this plot shows the similarity between the 2nd order
Gaussian derivative and a Gabor function generated with the method described above. Using this method we can construct a Gabor kernel basis similar to the Gaussian derivative kernel basis. Additionally, we can specify an angle for the kernel using eq. (4.19). An example of a Gabor basis which is made to look similar to the Gaussian basis is shown in Figure 4.6a and 4.6b.

For the experiments we used a rather rudimentary method for the instantiation of the Gabor basis. We started with a Gabor basis similar to the Gaussian derivative basis, we then reviewed the relevance of each kernel in the basis by looking at the mean weights for each kernel after training. Furthermore, we removed certain kernels and reviewed whether the results got better or worse and based on those results we concluded whether they were relevant. In Figure 4.7, a basis is shown similar to the one used in the experiments.

Figure 4.6: **Left:** The Gaussian derivative basis with \( \sigma = 2 \) and partial derivatives only with respect to one variable, i.e. from left to right and top to bottom: \( \{G_x, G_x^2, G_x^{xxx}, G_y, G_y^2, G_y^{yyy}, G_{xx}, G_{xx}^2, G_{xx}^{xxx}, G_{yy}, G_{yy}^2, G_{yy}^{yyy}\} \). **Right:** Gabor basis, all with \( \sigma = 2 \), and with \( \{\text{wavelengths; real (RE) or imaginary (IM) part}\} \) from left to right and top to bottom: \( \{12\sigma; \text{RE}, 6\sigma; \text{IM}, 3\sigma; \text{RE}, 3\sigma; \text{IM}, 2.4\sigma; \text{RE}, 6\sigma; \text{IM}, 3\sigma; \text{RE}, 3\sigma; \text{IM}, \ldots, 2.4\sigma\} \), where the last 4 are rotated by \( \frac{1}{4}\pi \).

Figure 4.7: Gabor basis similar to the one used in the experiments, with \( \sigma = 2.0 \), and with \( \{\text{wavelengths; real (RE) or imaginary (IM) part}\} \) from top to bottom: \( \{6\sigma; \text{IM}, 3\sigma; \text{RE}, 3\sigma; \text{IM}\} \), where each column is rotated by \( \frac{1}{4}\pi \).
The implementation for all networks is done using Theano, Lasange and Numpy\(^1\). The implementation is based on the Theano tutorial/documentation and on an example network from Jörn-Henrik Jacobsen [44].

The massive popularity of CNN’s (and ANN’s in general), is very much a result of the increasing computational power of GPU’s. For big networks, training on GPU’s instead of CPU’s can bring training time down from months to days [45]. It has been shown that GPU’s perform significantly better on simple tasks with many calculations, e.g. matrix multiplication [46]. This is a result of GPU’s having many cores, able to perform simple tasks, whereas CPU’s generally have a couple of cores, which are able to perform more complex tasks.

5.1 Theano and Lasagne

In this section we will give a very brief introduction on Theano and Lasagne. There are several libraries available, which are optimized for training neural networks on GPU’s. Theano is such a library for Python, and one of the first deep-learning frameworks. Lasagne is a library built on Theano, it provides functionality to build and train neural networks in Theano.

Most of this section is based on the Theano article [47], the Theano documentation [48] and the Lasagne documentation [49]. Theano describes itself as a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multidimensional arrays efficiently [47]. Theano uses directed acyclic graphs (i.e. directed graphs without cycles) in order to represent the mathematical expressions internally. These graphs contain two kinds of nodes:

- Variable nodes, representing data, for example matrices or tensors.
- Apply nodes, representing the application of mathematical operations.

Theano has so called *shared variables*, these contain constant values and can be shared between multiple Theano functions. The key value of shared variables show when using them in conjunction with the GPU, because shared variables will be created on the GPU by default, making it faster to access them when doing computations.

---

\(^{1}\) Numpy is a well-known library for scientific use, we will use it in our implementation and expect the reader to be familiar with this library.
In order to understand how the syntax of Theano works, consider the following simple example that computes a multiplication between a 2x2 matrix and 2-d vector:

```
import theano
import theano.tensor as T
import numpy

x = T.fvector('x')
A = theano.shared(numpy.asarray([[0, 1], [1, 0]]), 'A')
y = A * x
f = theano.function([x], y)
output = f([2.0, 1.0])
print output
```

After importing Theano and Numpy, we define a float32 vector, and name it 'x':

```
x = theano.tensor.fvector('x')
```

Then we define a matrix 'A' and make it a shared variable (i.e. it can be shared between multiple functions):

```
A = theano.shared(numpy.asarray([[0, 1], [1, 0]]), 'A')
```

Next we create the mathematical expression y, which multiplies x with the matrix A:

```
y = A * x
```

Now in order to create a Theano function, we give it its input (x) and its output (y):

```
f = theano.function([x], y)
```

At last we evaluate the function, which is similar to evaluating a Python function:

```
output = f([2.0, 1.0])
```

This whole code will give us the result:

```
\[
\begin{bmatrix}
0 & 1 \\
1 & 0
\end{bmatrix}
\begin{bmatrix}
2.0 \\
1.0
\end{bmatrix}
= \begin{bmatrix}
1.0 \\
2.0
\end{bmatrix}
\]
```

An important feature of Theano functions is the `updates` parameter:

```
import theano
import theano.tensor as T

state = theano.shared(0)
inc = T.iscalar('inc')
accumulator = theano.function([inc], state, updates=[[state, state+inc]])
```

Here we create a function with an `updates` parameter, the `updates` parameter must be supplied with a list of pairs of the form (shared-variable, new expression). After each function call the shared variable is updated to the new expression, so for example:

```
>>> accumulator(2)
array(0)
>>> state.get_value()
2
```
After calling the accumulator(2) the shared variable state will hold 2. The updates parameter is very useful for training a neural network, that is, we can update a network’s weights by providing an update function. Lasange provides several update functions that are useful for training neural networks, which will be used in the implementation.

Another Theano feature which is very useful for neural networks, is the ability to carry out calculations on a graphics card. In order to make Theano use the graphics card, we need to install CUDA [50] and add a flag device=gpu to Theano’s configuration.

5.2 Classical convolutional neural network

Before moving on to the implementation of the RFNN model described in the previous section, we explain how a CNN model is implemented, because this enables us to compare our RFNN results with CNN’s and it forms a basis for the RFNN implementation.

For the classical CNN we start by defining kernels for each layer, initializing them randomly within an interval:

```python
1 w1 = init_basis_rnd((64, 1, 7, 7))
2 w2 = init_basis_rnd((64, 64, 7, 7))
3 w3 = init_basis_rnd((64, 64, 7, 7))
4 w_o = init_weights(3136, 10)
```

Here we defined three collections of kernels and one normal hidden layer. The first collection, \( w_1 \), has 1 input, the image, and 64 outputs resulting in \( 1 \times 64 \) kernels. The most effective number of outputs will differ per dataset, 64 showed to be a very good number of outputs for the relatively simple MNIST handwritten digit dataset (more on this dataset in the results section). Each of the 64 kernels have a size of \( 7 \times 7 \), this will also differ per dataset, again \( 7 \times 7 \) showed good performance for the MNIST dataset. The other two collections of kernels are similar, except for the number of inputs. Since layer 1 has 64 outputs layer 2 will have 64 inputs, the same applies to layer 3.

The initial values for the weights of the kernels should be sampled from a symmetric interval. Glorot and Bengio [51] has shown that for the a sigmoidal activation function the weights should be uniformly sampled from the interval \(-\sqrt{\frac{6}{\text{in} \times \text{out}}}, \sqrt{\frac{6}{\text{in} \times \text{out}}}\), where \text{in} and \text{out} are the number of inputs and outputs for that layer. However, there is no proof that this interval applies to the ReLU activation function and nothing was found on a interval for ReLU’s with proof. Nevertheless, for the experiments we did use this interval.

After initializing the weights we define the actual network as a function, where one convolution layer is constructed as follows:

```python
1 l1b = T.nnet.relu(dnn_conv(X, w1, border_mode=(5,5)))
2 l1 = dnn_pool(l1b, (3,3), stride=(2, 2))
3 l1 = dropout(l1, p_drop_conv)
```

Starting with the first layer, the input (i.e. the image) is convolved with the kernels for the first layer using dnn_conv. The result of this convolution is then fed into the ReLU activation function. Next up max pooling using dnn_pool and in this example a pooling size of (3,3). The last step involves dropout. What dropout essentially does is leaving out a percentage of random nodes in the network for one iteration. This has proven to be a very simply yet effective way to reduce overfitting [52]. The same steps are done for every convolution layer in the network. The output of the last convolution layer is flattened and fed into a regular hidden layer:

```python
1 l1b = T.nnet.relu(dnn_conv(X, w1, border_mode=(5,5)))
2 l1 = dnn_pool(l1b, (3,3), stride=(2, 2))
3 l1 = dropout(l1, p_drop_conv)
```

```python
1 l4 = T.flatten(l3, outdim=2)
2 pyx = dropout(l4, p_drop_hidden)
3 pyx = softmax(T.dot(l4, w_o))
```
In order to train the network we need a cost function and an update method. As was pointed out in the introduction, the cost function used in all implemented networks is the categorical cross entropy (eq. 3.5). The actual code for doing this is simple:

```python
cost = T.mean(T.nnet.categorical_crossentropy(py_x, Y))
```

Where \(py_x\) are the network’s predictions on the training set and \(Y\) are the actual output values from the training set. Using this cost function, we will define an update function to adjust the network’s weights. As mentioned earlier Lasagne provides several update functions, the function used for all implementations is the \texttt{adadelta} function. We need to pass both the cost function and the weight variables that need be updated into the update function, together with some hyper-parameters (explained in section 3.1.4).

```python
weights = [w, w2, w3, w_o]
updates = adadelta(cost, weights, learning_rate=lr, rho=0.95, epsilon=1e-6)
```

The last step is creating a training function, this training function needs the training instances, \(X\) and \(Y\) as input, the cost function as output, and the update function for updating the weights:

```python
train = theano.function(inputs=[X, Y, lr], outputs=cost, updates=updates)
```

Because \(py_x\) is the output of the whole network and \(py_x\) is an argument of the cost function, the cost function encapsulates the whole network. This concludes all functions necessary for training the network. The actual training is done by passing batches of training examples to the training function, when all training examples have been passed one training iteration is complete. This process is then repeated for a specified number of times, i.e. the number of training epochs.

The format for the input data, i.e. \(X\), is a 4D tensor, where the first index denotes the instance, the second index the channel (i.e. R,G and B for a color image or just 1 for a gray image) and the last two index the pixels. The format for the labels, i.e. \(Y\), is a 1xN matrix denoting the class labels corresponding with the indices in \(X\).

### 5.3 Receptive fields neural network

Having explained how to construct a CNN model, we move on to the RFNN model. Starting with the Gaussian RFNN model, we need to create the Gaussian derivative kernel basis. This means implementing eq. 4.8 and a recursive implementation for the Hermite polynomials. For the Gabor basis, we simply implemented eq. 4.17. See Appendix A.2 and A.1 for the code.

The rest of the implementation is the same for both the Gaussian and Gabor kernels. We have to initiate random alphas and multiply these with all the kernel bases to create weighted combinations:

```python
bases.append(init_gabor_basis(sigma))
alphas.append(init_alphas(no_of_kernels, no_of_prev_nodes, no_of_bases))
w.append(T.sum(alphas[-1][:, :, None, None, :], axis=2))
```

The rest of the implementation is very similar to the CNN implementation, for example, one layer in the RFNN looks like:

```python
l1b = T.nnet.relu(dnn_conv(X, w[0], border_mode=(5,5)))
l1 = dnn_pool(l1b, (3,3), stride=(2, 2))
l1 = dropout(l1, p_drop_conv)
```

There is one very important difference, which is the params in the update function:

```python
params = [alphas[0], alphas[1], alphas[2], w[4]]
updates = adadelta(cost, params, learning_rate=lr, rho=0.95, epsilon=1e-6)
```

The last parameter in \texttt{params} is for the hidden layer at the end. Training is the exact same as it was in the CNN implementation.
The choice of test datasets was mostly based on possibility of training a network on the available hardware. All experiments were run on GPU’s (NVidia Titan).

6.1 MNIST

The MNIST dataset \cite{7} consists of handwritten digits (see Figure 6.1), it is a well-known benchmark dataset in the field of machine learning. Therefore there is a great amount of methods to compare the results to. Moreover setting up a network for the MNIST dataset is relatively easy because of its low image complexity, and the ample amount of data available. The dataset consists of a training set with 60,000 instances and a test set of 10,000 instances.

![Example of handwritten digits in the MNIST dataset](image)

Figure 6.1: Example of handwritten digits in the MNIST dataset

Previous results show excellent performance of CNN’s on the MNIST dataset. Compared to other classification methods achieving the best scores on average, with state-of-the-art performance at 99.77% accuracy \cite{54,55}. Since the dataset is essentially one scale, i.e. all digits are approximately the same size, the dataset is already scale invariant. All images are 28x28 and only contain one digit, resulting in 10 different classes (i.e. digit 0-9).

6.1.1 Setup

In this section, we will explain the specific setups with the best performance, for all three models used: Classical CNN, Gaussian RFNN and Gabor RFNN.

For the classical CNN model, we used 3 convolution layers each consisting of 64 different kernels all with size 7x7, the model concluded with one hidden layer.

The Gaussian RFNN model also used 3 convolution layers and one hidden layer. The first convolution layer used all 15 Gaussian basis kernels (i.e. up to 4th order) at scale 1.5, and constructs 64 weighted combinations. The second and third layer used 15 Gaussian basis kernels (i.e. up to 4th order) at scale 1.0 and again constructs 64 weighted combinations. The model concluded with one hidden layer. This setup showed the highest performance.

For the Gabor RFNN we used multiple different models, however for all models we used 3 convolution layers and one hidden layer. Additionally, for all models we used scale 2.0 for the first layer and 1.5 for the second and third layer. We will show results for models while varying the wavelengths and the number of distinct wavelengths. We will be comparing the best setup of the Gabor RFNN with the the CNN and Gaussian RFNN models. We will also show the mean and standard deviation of the weights linked to one kernel after training.
6.1.2 Results

In table 6.1 we show the performance of three different Gabor RFNN models using different wavelengths for all layers, and using only 5000 training instances.

<table>
<thead>
<tr>
<th>Wavelengths</th>
<th>Top-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>{12\sigma, 6\sigma, 3\sigma}</td>
<td>98.05%</td>
</tr>
<tr>
<td>{6\sigma, 3\sigma}</td>
<td>98.04%</td>
</tr>
<tr>
<td>{4\sigma, 3\sigma}</td>
<td>98.18%</td>
</tr>
</tbody>
</table>

Table 6.1: Result of the RFNN model using Gabor kernels for the MINST dataset using 5000 training examples, varying the wavelengths for all layers. In all of these results we used 4 rotations (\{0, \frac{1}{4}\pi, \frac{1}{2}\pi, \frac{3}{4}\pi\}) and 3 or 2 different wavelengths. The scale for layer one was 2.0 and for the other two layers a scale of 1.5 was used.

Figure 6.2 shows the mean and standard deviation of all weights linked with a certain kernel in the basis.

Figure 6.2: The mean and standard deviation of all weights linked to a certain kernel in the basis, shown along the x-axis. The basis used consists of three different kernels at 4 different rotations.

Results for all three models, varying the amount of training instances used is shown in Figure 6.3.
The best results for all three models on the MNIST dataset using all training examples as well as the state-of-the-art performance is shown in table 6.2.

<table>
<thead>
<tr>
<th>Model</th>
<th>Top-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gauss RFNN</td>
<td>99.42%</td>
</tr>
<tr>
<td>Gabor RFNN</td>
<td>99.43%</td>
</tr>
<tr>
<td>CNN</td>
<td>99.57%</td>
</tr>
<tr>
<td>DropConnect NN [56]</td>
<td>99.79%</td>
</tr>
</tbody>
</table>

Table 6.2: Result of the Gabor RFNN, Gauss RFNN and CNN model and the state-of-the-art performance on the MNIST dataset using all 60,000 training instances.

6.2 MNIST Rotated

The MNIST Rotated dataset [57], is the MNIST dataset where the digits were rotated by an angle generated uniformly between 0 and $2\pi$ radians, see Figure 6.4. The dataset consists of 12000 training instance and 50000 test instances. State-of-the-art results on this dataset have an classification error around 3%, with the best result at 2.2% [56].

Figure 6.4: Example of handwritten digits in the MNIST dataset, where the digits were rotated by an angle generated uniformly between 0 and $2\pi$.

6.2.1 Setup

The model’s setup for this experiment is the exact same as the experiment for the standard MNIST dataset. For this model we will only show the results of the best model compared to the
Gauss RFNN and CNN.

6.2.2 Results

The classification error of the Gabor RFNN model compared to the Gaussian RFNN and CNN model is shown in Figure 6.5, varying the amount of training instances used.

Figure 6.5: Results for all three models on the MNIST rotated dataset, varying the number of training instances used for training.

The best results for all three models using all 12,000 training instances is shown in Table 6.3, where the results for the Gabor RFNN model are for two different bases using

<table>
<thead>
<tr>
<th>Model</th>
<th>Top-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gauss RFNN</td>
<td>95.04%</td>
</tr>
<tr>
<td>Gabor RFNN (4 orientations)</td>
<td>94.92%</td>
</tr>
<tr>
<td>Gabor RFNN (2 orientations)</td>
<td>88.96%</td>
</tr>
<tr>
<td>CNN</td>
<td>94.62%</td>
</tr>
<tr>
<td>TI-POOLING CNN [56]</td>
<td>97.80%</td>
</tr>
</tbody>
</table>

Table 6.3: Result of the Gabor RFNN, Gauss RFNN and CNN model and the state-of-the-art performance on the MNIST rotated dataset using all 12,000 training instances. For the Gabor RFNN we show the results for two different bases, one with the kernels in 4 different orientations (see Figure 4.6b), and one with the kernels in 2 different orientations (see Figure 4.7).

6.3 GTSRB

The German Traffic Sign Benchmark (GTSRB) [58] is a multi-class, single-image classification challenge held at the International Joint Conference on Neural Networks (IJCNN) 2011. The training data set consists of 39,209 training images and the test set consists of 12,630 test images and there are 43 different classes. Image sizes vary between 15x15 to 250x250 pixels, to simplify the implementation all images have been scaled to 42x42. Examples of images in the dataset are
shown in Figure 6.6. Best results on the GTSRB dataset achieves a successful classification rate of 99.46\% using a multi-column deep neural network \cite{59}, whereas the human performance on this dataset is 98.84\% \cite{60}.

![Figure 6.6: Examples of traffic signs in the GTSRB dataset.](image)

6.3.1 Setup
For all three models we used the same setups used in the previous experiments.

6.3.2 Results
We tested all three models on the GTSRB dataset, training the models with all training instances (table 6.4) as well as varying the amount of training instances (Figure 6.7).

<table>
<thead>
<tr>
<th>Model</th>
<th>Top-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gauss RFNN</td>
<td>97.85%</td>
</tr>
<tr>
<td>Gabor RFNN</td>
<td>98.72%</td>
</tr>
<tr>
<td>CNN</td>
<td>97.96%</td>
</tr>
<tr>
<td>MCDNN \cite{59}</td>
<td>99.46%</td>
</tr>
</tbody>
</table>

Table 6.4: Result of the Gabor RFNN, Gauss RFNN and CNN model and the state-of-the-art performance on the GTSRB dataset using all 39,209 training instances.

![Figure 6.7: Results of the Gauss RFNN, Gabor RFNN and CNN classification error for the GTSRB dataset, varying the amount of training instances used for training.](image)
Our results show that the Gabor basis achieves results similar to the Gaussian basis. These results provide further support that the Gabor basis is a viable basis and also support the hypothesis that the Gabor basis is very similar to the Gaussian basis. The performance of the Gabor basis also stays on par with the CNN’s performance, this suggests the hypothesized completeness of the Gabor basis.

For MNIST rotated and GTSRB, the Gabor basis showed slightly better results for fewer training instances compared to both the CNN and Gaussian RFNN. Especially for the rotated MNIST dataset, this slightly better performance for a small number of training examples might be caused by the usage of rotated kernels. Although for the Gaussian kernel basis it is shown that rotated kernels can be expressed as a weighted combination of multiple non-rotated Gaussian derivative kernels [61]. This weighted combination needs to be learned, explaining why the Gabor basis performs better for fewer training examples. However, the results, table 6.3, also show that the Gabor basis performs significantly worse when using only 2 orientations for the kernels. This indicates that the Gabor basis is not able to learn orientations as easily as the Gaussian kernel.

One interesting finding is that adding more kernels with different wavelengths does not always improve the performance, instead it can actually decrease the performance of the model (see table 6.1). The reason might be that adding more kernels results in a model with more parameters, which can often cause the model to overfit. Another reason why more kernels may decrease performance, might be that specific kernels detect features that are not relevant to classification, and may even have a negative effect on the classification output.

Another important finding is that certain kernels in the basis are more important than others, which can be seen in Figure 6.2. Although, the significance of certain kernels might differ per dataset, our results show that for two different datasets, MNIST and GTSRB, we achieve a high performance using the same kernels. Figure 6.2 also shows us that the standard deviation of some kernels is a lot higher, because we only used 5000 training instances this might indicate that this specific kernel is relevant only to specific subsets of the training instances. The scale of the kernels which performs best seems comparable to that of the Gaussian kernels, although the best results were found with a scale slightly higher than the Gaussian kernels’ scale.

Despite these promising results, questions remain. For example it is unknown if the performance of the Gabor RFNN on large datasets is as good as CNN’s performance. We were not able to test our results on large datasets due to time and hardware constraints. Therefore, the model presented in this paper should be tested on a bigger dataset, such as ImageNet [3]. The expectation is that the CNN will outperform the RFNN on larger dataset, something we can already see happening with the MNIST dataset when using all 60,000 training instances. We also did not test the scale invariance of the model, and its performance on a dataset with different scales compared to the CNN’s performance, future work on this are therefore recommended.

Another interesting question that remains, is the similarity of the kernels after training between the Gabor RFNN, the Gaussian RFNN and the CNN. Because the resulting kernels in a Gabor RFNN model are more explanatory, we might be able to better understand the model after training. Moreover, comparing the kernels of the Gabor RFNN with a CNN can give us insight in what a CNN learns, something which is still unclear [8].

Because the parameters are important for both the Gabor kernel basis as well as the Gaussian
kernel basis, further research should be undertaken to investigate the possibility of learning these parameters in the RFNN model. The Gabor kernels have more flexibility in terms of parameters to learn than the Gaussian kernel. This observation and the fact that both bases have similar results may support the hypothesis that, if we are able to learn these parameters, the Gabor kernel is able to adjust faster than the Gaussian kernel.

In conclusion, this study has shown that the Gabor basis is a viable kernel basis for the RFNN model and performs better than the CNN model when a lower amount of training instances is used. When the number of training instances is increased, the performance of the Gabor RFNN stays on par with CNN’s performance and close to state-of-the-art performance.
Acknowledgements

I would like to thank my supervisor Rein van den Boomgaard for all the very helpful discussions and e-mails. I would also like to thank Jörn-Henrik Jacobsen, as he helped me with the implementation as well as a deeper understanding of the RFNN model. At last, I would like to thank the people that took the time to read and review my thesis.


Appendices
A.1 Code for Gaussian basis

This is based on code from Jørn-Henrik Jacobsen: https://github.com/jhjacobsen/RFNN

```python
import numpy as np

def hermite(x, n):
    """ x: argument of the Hermite polynomial
         n: order of the Hermite polynomial
    ""
    if n == 0:
        return 1
    elif n == 1:
        return 2*x
    elif n >= 2:
        return 2*x*hermite(x, n-1) - 2*n*hermite(x, n-2)

def gaussian(x, sigma, n):
    """ x: argument of Gaussian function
         sigma: scale of Gaussian function
         n: derivative order of Gaussian function
    ""
    if n == 0:
        return 1.0/(sigma*np.sqrt(2.0*np.pi)) * np.exp((-1.0/2.0)*np.square(x/sigma))
    elif n > 0:
        return np.power(-1, n)*(1.0/np.power(sigma*np.sqrt(2), n))*hermite(x/(sigma*np.sqrt(2)), n)*gaussian(x, sigma, 0)

def init_gauss_basis(sigma, no_of_bases):
    """ sigma: scale of Gaussian basis
         no_of_bases: number of bases to return (i.e. 10 means up to 3rd order)
    ""
    filterExtent = 3*sigma
    x = np.arange(-filterExtent, filterExtent+1, dtype=np.float)
```

55
imSize = np.int(filterExtent*2+1)
impulse = np.zeros((imSize, imSize))
impulse[imSize/2,imSize/2] = 1.0
#
gaussBasis = np.empty((15, imSize, imSize))
g = []
for i in range(5):
    g.append(gaussian(x, sigma, i))

gauss0x = filters.convolve1d(impulse, g[0], axis=1)
gauss0y = filters.convolve1d(impulse, g[0], axis=0)
gauss1x = filters.convolve1d(impulse, g[1], axis=1)
gauss1y = filters.convolve1d(impulse, g[1], axis=0)
gauss2x = filters.convolve1d(impulse, g[2], axis=1)

gaussBasis[0,:, :,] = filters.convolve1d(gauss0x, g[0], axis=0) # g_0
gaussBasis[1,:, :,] = filters.convolve1d(gauss0y, g[1], axis=0) # g_x
gaussBasis[2,:, :,] = filters.convolve1d(gauss0x, g[1], axis=1) # g_y
gaussBasis[3,:, :,] = filters.convolve1d(gauss0y, g[2], axis=1) # g_xx
gaussBasis[4,:, :,] = filters.convolve1d(gauss0x, g[2], axis=0) # g_yy
gaussBasis[5,:, :,] = filters.convolve1d(gauss1x, g[1], axis=0) # g_xy
gaussBasis[6,:, :,] = filters.convolve1d(gauss0y, g[3], axis=1) # g_xxx
gaussBasis[7,:, :,] = filters.convolve1d(gauss0x, g[3], axis=0) # g_yyy
gaussBasis[8,:, :,] = filters.convolve1d(gauss1x, g[2], axis=1) # g_xxy
gaussBasis[9,:, :,] = filters.convolve1d(gauss1x, g[2], axis=0) # g_yyx
gaussBasis[10,:, :,] = filters.convolve1d(gauss1y, g[2], axis=1) # g_yyxx
gaussBasis[11,:, :,] = filters.convolve1d(gauss1y, g[2], axis=0) # g_yyyy
gaussBasis[12,:, :,] = filters.convolve1d(gauss1y, g[3], axis=1) # g_xyyy
gaussBasis[13,:, :,] = filters.convolve1d(gauss1x, g[3], axis=0) # g_yyxx
gaussBasis[14,:, :,] = filters.convolve1d(gauss2x, g[2], axis=0) # g_yyzz

return gaussBasis[0:no_of_bases,:, :]
A.2 Code for Gabor basis

def gabor_fn(sigma, theta, frequency, psi, gamma):
    sigma_x = sigma
    sigma_y = float(sigma) / gamma

    # Bounding box
    nsteds = 3
    xmax = max(abs(nsteds * sigma_x), abs(nsteds * sigma_y))
    xmax = np.ceil(max(1, xmax))
    ymax = max(abs(nsteds * sigma_x), abs(nsteds * sigma_y))
    ymax = np.ceil(max(1, ymax))
    y, x = np.mgrid[-ymax:ymax + 1, -xmax:xmax + 1]

    # Rotation
    rotx = x * np.cos(theta) + y * np.sin(theta)
    roty = -x * np.sin(theta) + y * np.cos(theta)

    g = np.zeros(y.shape, dtype=np.complex)

    # Standard 2d Gaussian
    g[:] = np.exp(-0.5 * (rotx ** 2 / sigma_x ** 2 + roty ** 2 / sigma_y ** 2))
    g /= np.sqrt(2 * np.pi * sigma_x * sigma_y)
    g *= np.exp(1j * (2 * np.pi * frequency * rotx + psi))

    return g

def init_gabor_basis(sigma):
    filter_size = 2*np.ceil(3*sigma)+1
    basis = np.empty((17, filter_size, filter_size))

    kernel = gabor_fn(sigma=sigma, theta=0, frequency=1/(12.*sigma), psi=0.0, gamma=1.0)
    basis[0, :, :] = np.real(kernel) / np.linalg.norm(np.real(kernel))

    i = 1
    # Create 4 kernels in each of 4 rotations
    for theta in (0, 2, 4, 6):
        theta = theta / 8. * np.pi

        kernel = gabor_fn(sigma=sigma, theta=theta, frequency=1/(6.*sigma), psi=0.0, gamma=1.0)
        basis[i, :, :] = -np.imag(kernel) / np.linalg.norm(np.real(kernel))
        basis[i+1, :, :] = -np.real(kernel) / np.linalg.norm(np.real(kernel))
        basis[i+2, :, :] = np.imag(kernel) / np.linalg.norm(np.real(kernel))
        basis[i+3, :, :] = np.real(kernel) / np.linalg.norm(np.real(kernel))

        i+=4

    return theano.shared(floatX(basis))