Bachelor Informatica

University of Amsterdam

Informatica — Universiteit van Amsterdam

Investigating scale in Receptive Fields Neural Networks

Robert Jan Schlimbach

June 8, 2018

Supervisor(s): Rein van den Boomgaard
Signed:

Abstract

Convolutional Neural Networks (CNNs) are currently the de-facto standard of image classification ever since Krizhevsky beat all competition using these in the ImageNet classification challenge of 2012 [12]. Since then multiple improvements have been made in the design of CNNs, manifesting itself in the creation of the notable Network-in-Network [15] and Residual Networks [6] architectures which are practically the most performant CNN architectures at this time. These networks are shown to be scale resilient, e.g. they manage to detect objects in images on different scales without too much loss of performance. This scale resilience is however not particularly grounded in theory, and a more theoretically founded scale-invariant network is to be desired.

Jacobsen et al. [8] introduced the notion of a Receptive Fields Neural Network (RFNN). The RFNN replaces the convolution kernels of a regular CNN with a weighted sum of convolutions with a set of analytically defined convolution kernels. What is then learned are the weights in the linear combinations of the fixed convolutions. This network design might be a promising start to introduce scale-space theory into deep learning, since this fixed basis might be crafted to support the central pillars of scale-space theory. One of these central pillars is the σ or 'scale' parameter of the Gaussian function [9]. We follow [24] in taking a basis of Gaussian derivatives for the fixed kernels of a RFNN. In the following paper we investigate the impact of the scale parameter in these Gaussian derivatives on the classification performance in a simple two-layer RFNN setting, using a relatively new framework for machine learning, called PyTorch.

Lastly, we correct an error present in Verkes [24].

Contents

CHAPTER 1

Introduction

Scale invariance has long been a main goal of image processing. Lowe's paper [16] on Scale Invariant Feature Transform is notable in being one of the most cited and recognisable papers of the last two decades. The body of Lowe's algorithm is composed of constructing a Laplacian of Gaussians, albeit implemented as a Difference of Gaussians (DoG) to enable faster processing. This Laplacian of Gaussians if thoroughly grounded in Koenderink's theory of scale-space [9], with blob detection being done on the input image blurred with Gaussian kernels each with a different σ parameter, to determine the ideal local scale of the input image to do further processing on.

Convolution Neural Networks, created in 1989 and formalised in 1995 by Yann LeCun [13][14], have become very popular since Krizhevsky used them to great effect in the ImageNet classification challenge of 2012 [12]. One of the main advantages of CNNs over other feed-forward Artificial Neural Networks (ANNs), which consist mainly of fully-connected layers, is the ability to preserve spatial information between layers. One of the other advantages of CNNs over ANNs is the ability to fully use GPU accelerated computing. Convolution in the context of discrete signals is practically nothing else than repeated shifted matrix multiplication, something GPU's are particularly good at in comparison to regular CPU parallelised computing. This last point is the reason Krizhevsky was able to take LeCuns network design to such great new lengths where LeCun was not. He was able to train his model with a number of parameters and for a number of training cycles unheard of in the time of LeCuns 1995 paper. CNNs have the property of being somewhat scale-resilient, as can be seen in 1.1. This scale-resiliency has to be learnt however. This is done by using both a large amount of input images, and by modifying the input in various ways, like up-scaling the to be detected object in the input image. In an ideal world the network would learn to recognise to be detected objects at any scale, without any input modification.

Jacobsen et al. [8] proposed the Receptive Fields Neural Network in 2016, as a modification of the Convolution Neural Network design. This design replaces the convolution kernals of the CNN design with a weighted sum of convolutions with some fixed basis. Both Jacobsen et al. [8] and Verkes [24] showed the advantages of the RFNN design in their respective papers. Jacobsen using a Network-in-Network design using a basis of Gaussian derivatives and Verkes showing the potential of using a basis of Gabor filters. Both Jacobsen and Verkes have in their papers confirmed the similar performance of the RFNN network compared to a regular CNN.

What is still remains uninvestigated, is the impact of the scale parameter of the Gaussian derivatives which fill the fixed kernels on the performance of the RFNN network design. Verkes [24] simply chose a scale of 1.5 and 1 for the first and second layer of the RFNN in his experiments, but this choice remains unfounded. It is the ultimate hope that the scale parameter might be learnt by the network resulting in the network selecting the best 'scale' to examine an input image at, in a similar vein as [16]. However, first a quantitative analysis should be performed on the general impact of the scale parameter on the performance of a simple RFNN, which is what this paper will try to do.

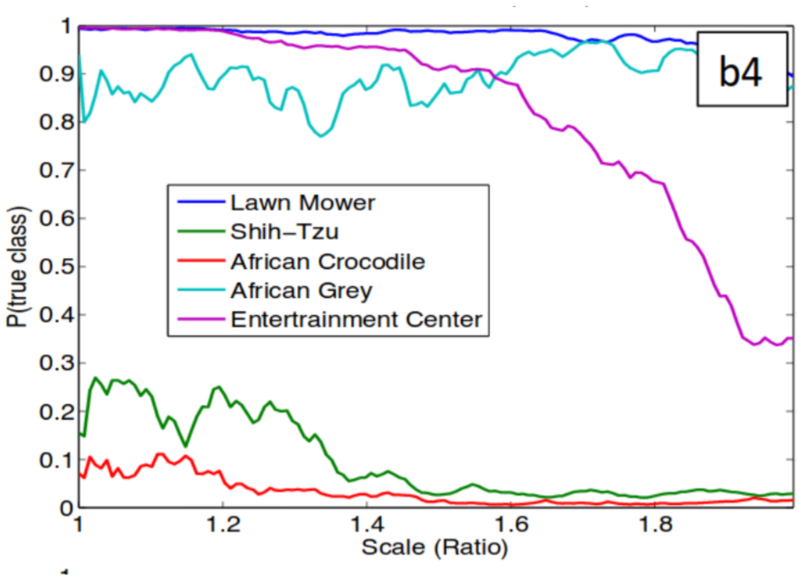


Figure 1.1: Example of scale resilience of CNNs as displayed by the performance of the network correctly classifying the respective classes versus the ratio of how much the image is scaled in relation to the original image. Source: Zeiler and Fergus [26]

CHAPTER 2

Network Architectures

2.1 Basic Neural Networks

A basic or 'vanilla' Artificial Neural Network is usually defined as a Multi-Layer Perceptron [4]. This means a network containing an input layer, an output layer, and at least one 'hidden' layer, with each hidden layer at least followed by a non-linear activation function.
Most of the layers in an ANN are usually fully-connected layers, meaning all inputs of a previous layer map to all nodes of the next layer. However, there are also a plethora of additional layers to be added, each with their own functionality and purpose. Examples of these are input normalisation, pooling and dropout layers. Each type and sub-type of these additional layers can be seen as additional hyper-parameter to the network. It is therefore important to have knowledge of as many types as possible, because a decision has to be made whether, why and when to include them into the design of a network. We discuss some of the many types and sub-types of layers as possible, outlining the pros and cons of each.

2.1.1 Activation Functions

When modelling biological networks, an activation functions is used to mimic the firing of a singular neuron [7]. In learning an Artificial Neural Network, activation functions are used to introduce non-linearity into the network.
All other layers in a regular Neural Network contain only linear operations, and since the combination of linear functions is itself a linear function it can easily be recognised that without the activation function only linear relations can be learned.

sigmoid & tanh

The activation of a neuron in nature can be seen as a step function, namely whether the neuron is firing or not. For learning in Artificial Neural Networks this is function is problematic, since it is non-differentiable and thus not usable in backpropagation.
The sigmoid activation function is defined as follows:

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}} \tag{2.1}$$

It has the nice property that it it mimics the step function in its output domain of $[0, 1]$, while it is in fact differentiable. Because of this the sigmoid function has historically been popular as an activation function [23].
There are however some mayor issues with using the sigmoid function as activation function:

1. It saturates. Since the output value of an activation layer is used as input for the next layer, the output of the function can only be positive, when using a sigmoid function. The individual values in the later layers of a network will then trend towards only having high positive values.

2. It has vanishing gradients. When a significantly low or high value enters a sigmoid function, the gradient associated with this value will be very low. This results in very slow learning rates when using gradient descent.

The tanh activation function is defined as follows:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-1}} \tag{2.2}$$

The tanh function retains the nice biological interpretation of a continuous step function, but since the it is centered around zero, it does not saturate like the sigmoid function. It does however still have the problem of vanishing gradients when the input is significantly small or large.

ReLU

The ReLU or Rectified Linear Unit is defined as follows:

$$\text{ReLU}(x) = \max(0, x) \tag{2.3}$$

Alex Krizhevsky used the ReLU activation function to great success in his landmark paper of 2012 [12]. Since then the ReLU function has been the mainstay of machine learning [23].
The most distinct advantage of the ReLU over the previously mentioned activation functions is that it does not suffer from vanishing gradients; since the output of the ReLU function is simply the identity when the input is positive, the gradient will never vanish.
The most distinct disadvantage of the ReLU is that it is possible for a particular ReLU node to 'die', e.g. when all inputs of a specific ReLU node are zero, all derivatives at the backpropagation stage will also be zero. This results in the respective learning node associated to the specific ReLU node not getting any gradients anymore to edit its weights, resulting in the node no longer being usable in the Network. It is shown that in some circumstances up to 40% of a networks nodes may be 'dead' due to this behaviour [23].
There are various alterations of the basic ReLU activation function available that attempt to fix this behaviour.

Sidenote:
It might be fun to note that the ReLU function is in fact the integral of the step function, resulting in even the ReLU function having some (very loose) connection to the biological interpretation of a neuron.

Solving the dead ReLU problem

The leaky ReLU, introduced in [17], is defined as follows:

$$\text{Leaky ReLU}(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \tag{2.4}$$

As shown, the Leaky ReLU features allows for a very small gradient whenever an input is below zero. This gives 'dead' nodes the possibility of recovering to a non 'dead' state over the span of many backpropagation steps. Whether or not this intuition holds true has however not conclusively been confirmed.
The PReLU or Parametric ReLU is a generalization of the Leaky ReLU and is defined as follows:

$$\text{PReLU}(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases} \tag{2.5}$$

The PReLU promises to generalize better than the Leaky ReLU, and may use a learnable α parameter.
The ELU or Exponential LU is defined as follows:

$$\text{ELU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ a(e^x - 1) & \text{otherwise} \end{cases} \tag{2.6}$$

Some experiments indicate that the ELU activation function converges faster than all previous functions [2].
These three activation functions are quite similar in performance to each other and even to the classic ReLU in most cases. Since the dead ReLU problem is particularly prevalent in deep networks and not so much wide networks, these functions primary use case is in these very deep networks.

Other noteworthy activation functions

Other activation functions which will not be explained but are useful to note are:

1. Maxout, an activation function inspired by the dropout algorithm.
2. Swish, an alteration of the Sigmoid activation function.

2.1.2 Loss Functions

Mean Squared Error

The Mean Squared Error is defined as follows:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{2.7}$$

Where Y_i is the expected value and $\hat{Y_i}$ is the associated measured value. The Mean Squared Error is primarily useful for linear regression. For classification it is not particularly useful because it assumes the underlying distribution of the error is a normal distribution, which is simply not the case; a classification is either correct or false.

Softmax

The Softmax function is defined as follows:

$$\text{Softmax}(x)_i = \frac{e^{x_i}}{\sum e^x} \tag{2.8}$$

Where x is a set of values.
The Softmax function is particularly nice for classification because it relates to Bayesian theory, in that each output of the Softmax function is the equivalent of the probability of that output given the input data. This relation to Bayesian theory is also nice in that a summing all output values after applying a Softmax function to them results in the value 1.

2.1.3 Backpropagation

Backpropagation is a method of automatically calculating gradients needed to edit the learnable parameters of Artificial Neural Networks such that the loss function of the ANN decreases. Backpropagation is a special case of automatic differentiation, and is usually implemented by recursively applying the chain rule for each node and calculating the derivative of the error with respect to some learnable parameter $w_{i,j}^{(l)}$, usually expressed in the form of:

$$\frac{\partial E}{\partial w_{i,j}^{(l)}} \tag{2.9}$$

Where l indicates the layer and i, j express a node within the layer and an index within the node respectively, of which the learnable parameter belongs to.
Solving this with the chain rule for some learnable parameter in layer l at node i at index j, expands to the following expression:

$$\frac{\partial E}{\partial w_{i,j}^{(l)}} = \frac{\partial E}{\partial o_i^{(l)}}\frac{\partial o_i^{(l)}}{\partial w_{i,j}^{(l)}} \tag{2.10}$$

Where $o_i^{(l)}$ is the output of the node to which $w_{i,j}^{(l)}$ belongs.

For a simple fully-connected layer the following holds:

$$o^{(l)} = (o^{(l-1)})^T \cdot w^{(l)} \tag{2.11}$$

Which results in:

$$\frac{\partial o_i^{(l)}}{\partial w_{i,j}^{(l)}} = o_j^{(l-1)} \tag{2.12}$$

2.1.4 Gradient methods

Non-Adaptive methods

Gradient Descent

Gradient descent, also known as Batch Gradient Descent, is the classic way of iteratively optimising a function towards some minimum. The function is used by calculating the derivative of the error over all possible input values, and modifying the learnable parameters by subtracting from it its gradient multiplied with the 'learning rate' parameter η. This parameter is allowed to change every iterations, and can be calculated for some problems.

Stochastic Gradient Descent

Stochastic Gradient Descent or SGD is largely equivalent to Gradient Descent, except at every step it does not take the optimal step towards some desired minimum but a step defined by the gradient calculated over a random subset of the available data.
For training an Artificial Neural Network, using SGD is most often desirable over regular Gradient Descent. Since modern datasets can sometimes have more than a few million training datapoints, calculating the gradient over all of these can be computationally very expensive. Iteratively adjusting the parameters of a network from the gradient over a *mini-batch* of these examples if most often more than enough to converge to a minimum equivalent in performance to Gradient Descent in a fraction of the time.
The optimal size of the batch of SGD is 1 [25]. However it is often more time efficient to use an increased batch size to enable data-level parallelism.

SGD with momentum

One optimisation of standard SGD is adding a momentum term to the standard SGD algorithm. This means that in one iteration the taken step is not simply determined by the size of the gradient and the learning rate, but rather the to be taken step is added to a 'velocity', which then in turn is used to update the learnable parameters.

Nesterov momentum

Nesterov momentum or Nesterov Accelerated Momentum (NAG) is a slight alteration of the regular SGD with momentum scheme. Instead of the current gradient being added directly to the velocity of the particle, first a 'dummy' step is taken in the direction of the current velocity and the gradient is calculated at this 'dummy' position. This gradient is then added to the original momentum and the actual iteration step is taken with this updated momentum from the original location.
Using this momentum update scheme results in a more stable and responsive behaviour of the gradient momentum descent [21].

2.2 Convolutional Neural Network

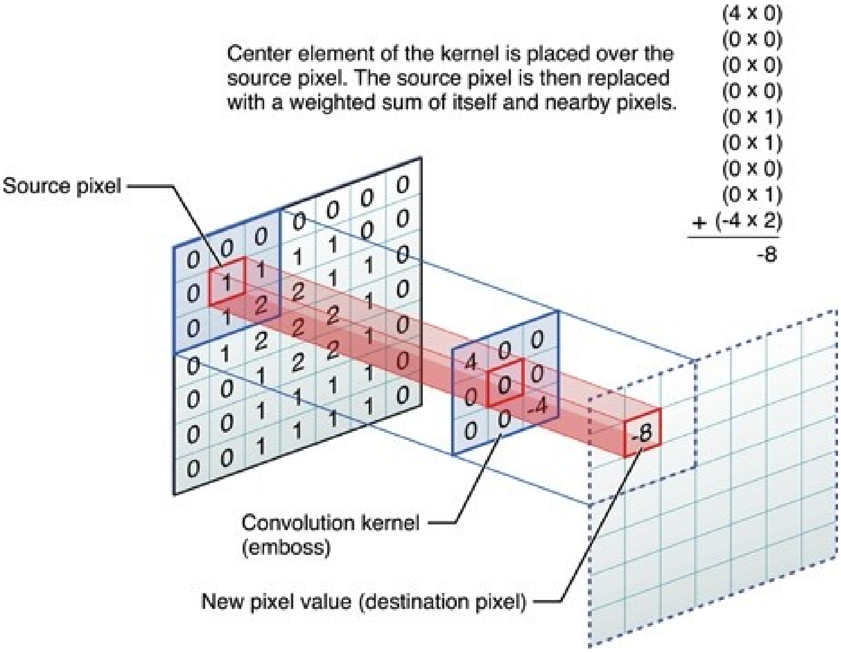
2.2.1 CNNs vs regular ANN

A Convolutional Neural Network is a special type of Artificial Neural Network. Where ANNs mostly feature fully-connected layers, CNNs replace some or many of these layers with convolution layers.
A convolution layer is a layer where the network convolves the input image with some filter, resulting in an output image usually of the same dimensions as the input image.
A convolution layer has two distinct advantages over a regular fully-connected layer:

1. A convolutional layer is able to preserve the spatial information of the image.
2. A convolutional layer has vastly less parameters than a fully-connected layer of the same input size.

2.2.2 Convolution operation

The convolution is defined as taking some input image, and sliding a kernel or 'filter' of the image, and for each sliding step taking the weighted sum of all image values overlapping with the filter. See 2.1 for a visual explanation of the convolution operation.



Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

Figure 2.1: Convolution operation. Source: Abhishek Shivkumar from Quora

2.2.3 Pooling Layers

Pooling layers reduce the size of an input image by sliding a window over the image and letting the output image be some operation performed over all values inside the sliding window. Usually this is done by letting the sliding window skip every other pixel, called a 'stride' of 2, and take the maximum value of all pixels within the window, called 'max pooling'.

2.3 Receptive Fields Neural Network

2.3.1 Definition

As previously stated, Receptive Fields Neural Networks differ from Convolutional Neural Networks in fixating the kernels with which the input image is convolved. Instead the design of a

RFNN lets the network learn a weighted sum of these fixed covolutions.
This can be easily understood when looking at the formula of a single output channel of a regular Convolutional Neural Network layer:

$$g_j = \sum_i f_i * w_{j,i} \tag{2.13}$$

Where g_j is a particular channel in the output, f_i is a channel in the input, and $w_{j,i}$ is the i-th filter associated with g_j.

Jacobsen et al. [8] realised that this $w_{j,i}$ term in the formula could be rewritten to be some weighted overcomplete basis:

$$w_{j,i} = \sum_k \alpha_{j,i,k} \cdot \phi_k \tag{2.14}$$

Where every $\alpha_{j,i,k}$ is a learnable parameter, and every ϕ_k is some fixed filter.

Substituting this in 2.13 results in the basic formula for one output channel in a RFNN layer:

$$\begin{aligned} g_j &= \sum_i f_i * (\sum_k \alpha_{j,i,k} \cdot \phi_k) \\ &= \sum_i \sum_k \alpha_{j,i,k} \cdot (f_i * \phi_k) \end{aligned} \tag{2.15}$$

2.3.2 Using a Gaussian basis of derivatives

Formally, the definition of the RFNN does not specify a basis to chose as fixed kernels. There is however a prime candidate for this basis: a basis of Gaussian derivatives. Verkes [24] shows that a weighted combination of Gaussian Derivatives has in theory the same expressive power as an arbitrary kernel in a CNN.

Gaussian kernels also have the benefit of being separable, meaning:

$$G^\sigma(x,y) = G^\sigma(x)G^\sigma(y) \tag{2.16}$$

This also means that calculating the derivative of a 2D Gaussian is quite easy:

$$\frac{\partial^{n+m}}{\partial x^n \partial y^m} G^\sigma(x,y) = \frac{\partial^n}{\partial x^n} G^\sigma(x) \, \frac{\partial^m}{\partial y^m} G^\sigma(y) \tag{2.17}$$

It is also shown that Gaussian derivatives up to arbitrary order are simply a 0-th order Gaussian multiplied by scaled Hermite polynomials [19]:

$$\frac{\partial^n G^\sigma(x)}{\partial x^n} = (\frac{-1}{\sigma\sqrt{2}})^n H_n(\frac{x}{\sigma\sqrt{2}}) G^\sigma(x) \tag{2.18}$$

Where H_n has the following recurrence relation:

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x) \tag{2.19}$$

Resulting in the cases:

$$H_n(x) = \begin{cases} 1, & : \ n = 0 \\ 2x, & : \ n = 1 \\ 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x), & : \ n > 1 \end{cases} \tag{2.20}$$

2.3.3 Correcting Verkes [24]

Verkes has the following error in his paper; the expansion of the recurrence relation of the Hermite polynomials is correctly reported to be:

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x) \tag{2.21}$$

Noteworthy is that this formulation uses the H_{n+1} formulation, instead of the shifted H_n, which will be important shortly.
If we then inspect Verkes' code, we find the following:

```python
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)
```

We can see the first term of $n >= 2$ being correctly implemented as `2*x*hermite(x, n-1)`, but the second term still uses the H_{n+1} formulation.

In implementation, this error causes cascading failure:

1. kernels filled with Gaussian derivatives now have a non-zero sum
2. each Gaussian convolution gives a bias to the output image
3. a large number of convolutions are performed leading to bias accumulation
4. bias accumulation leads to exploding gradients [1]

Using separated Gaussians

Oddly enough, although both Jacobsen and Verkes acknowledge the fact that Gaussian kernels are separable, neither of them makes use of this fact in their implementation. As we will show in 3, we do in fact make use of the separability of Gaussian kernels.

CHAPTER 3

Implementation

3.1 Choosing PyTorch as machine-learning framework

The largest two Python-integrated machine learning frameworks at this time are PyTorch and TensorFlow.
PyTorch is a framework deeply embedded in the Python ecosystem, both in syntax and in going as far as using the `ndarray` of the very popular scientific computing package NumPy as basis for the basic building block of the framework, the `Tensor`. PyTorch's main focus is on research, and is primarily developed by the Facebook AI Research group [3].
TensorFlow also has a large Python interface, although its interface is more a shell over its back-end C++ code than is the case with PyTorch. Its syntax is deeply rooted in symbolic math. TensorFlow's main focus is on usage in production environments, and is therefore the most used machine learning framework by large businesses. It is mainly developed by Google, and originated from Google's Brain team [22].

We chose PyTorch for all implementation for a couple of reasons:

- PyTorch is deeply embedded in the Python ecosystem. This means that practically any Python library can be used in conjunction with PyTorch. Its syntax syntax should be very intuitive for anyone with any Python experience.
- Memory usage is hugely optimised. This is especially important when using very large models or when on a memory constrained platform, which is something we will discuss shortly, in section 3.2.
- Debugging is much easier in PyTorch compared to frameworks like TensorFlow. Since PyTorch has a deeply integrated Python front-end, and is not a superficial shell over a C++ back-end, default stack traces simply work. This means that putting in breakpoints like those of the default Python library `pdb` stop the runtime at the exact spot of the breakpoint, and allow all default memory inspection operations, which is simply not possible in TensorFlow.

3.1.1 Dynamic vs. Static graphs

One important distinction point of PyTorch versus other machine-learning frameworks is the usage of dynamicly generated graphs. Every forward pass PyTorch generates a new graph, meaning operations like `if/else` statements are no more difficult to implement than including an `if/else` statement in the forward pass function within a defined network model. This might seem like a trivial feature, but this is in fact one of the reasons why PyTorch is such a powerful framework for research and rapid-prototyping. In frameworks like TensorFlow or Caffe the entire graph needs to be defined before-hand, including but not limited to all branching paths. This makes development cumbersome, losing the ability to quickly discover better network designs [18].

PyTorch's usage of dynamic graphs is also of particular interest to us, might we want to explore the usage of learnable scale in the future. If learnable scales are integrated into the RFNN design, we would want to re-calculate each Gaussian filled kernel every forward pass. Following 3.4, it might be so that the size of the re-calculated Gaussian kernel is smaller or larger than the current kernel size. This means we would want to hot-swap the respective kernel for a completely new one. Implementing this behaviour in a static-graph framework such as TensorFlow would be very difficult, if not impossible.

3.1.2 Shape of images and weights within PyTorch

One area where PyTorch might not be immediately intuitive is the shape of its `Tensors`. For example, for regular Python users, when applying a convolution to some image, one might expect the image to be of shape $(Width \times Height \times Channels)$ and the convolution filter to also be of comparable shape $(Width \times Height \times Channels)$.
This is however not the case in PyTorch. In PyTorch the input shape of an operation is usually of the shape:

$$\text{input\_shape} = (Batch\_Size \times In\_channels \times Img\_Shape) \tag{3.1}$$

And the operator shape to be:

$$\text{operator\_shape} = (Out\_Channels \times In\_Channels \times Img\_Shape) \tag{3.2}$$

Where Img_Shape is the dimensionality of the image, e.g. for a 2D image:

$$\text{Img\_Shape} = (Width \times Height) \tag{3.3}$$

3.2 Google Collabs

All experiments were done in Google Collabs. Collabs is a relatively new service from Google, specifically targeted at collaborative machine learning research [5].
Its provided service is an online jupyter notebook environment, allowing users to upload and run ipython notebooks, free of charge. Collabs offers user the possibility to utilize GPU accerated computing, by providing virtualized nVidia K80 GPUs, which are able to interface directly with popular machine learning frameworks like PyTorch and TensorFlow.

Usability of Collabs

Although Collabs promises collaborative development and the availability of powerful GPUs for running experiments on, some drawbacks were found.
The first drawback is not having persistent storage available. Storage is routinely flushes, meaning results of training a network need to be downloaded immediately, or risk being lost. A found solution is letting the notebook upload results to Google Drive, however a more permanent solution is desired.
The second drawback is the virtualized GPUs not have nearly the amount of memory nor computing resources available as a dedicated per-user GPU. During experiments `CUDAOutOfMemory` errors were encountered quite frequently, while PyTorch memory inspection tools revealed only 0.3GB of memory being allocated for the model; a nVidia K80 GPU should have 20GB of memory on-board.
The third and most major drawback is Collabs revoking access to a GPU when too much GPU-time was being used. It is common practice in machine learning, letting models train overnight when running experiments. However, when trying to reconnect to the runtime the next day to run more experiments, Collabs reported no GPU accelerated runtime was available. After some time, it was found out that any new Google account did immediately receive access to a GPU accelerated runtime when requested. This resulted in having to create a new dummy Google accounts every time an experiment was run overnight.

3.3 RFNN

3.3.1 Layout of a GaussianConv layer

When generating a GaussianConv class, three distinct functions are defined:

1. Initialization; generating Gaussian convolution kernels, setting up weighted sum convolution layer.
2. Forward pass; this function defines all computations to be done in the forward pass.
3. Backward pass; this functionality does not actually need defining by the user, and is defined by letting the GaussianConvLayer class inherit from `torch.nn.Module`. This allows PyTorch's `autograd` package to automatically calculate gradients when performing the backward pass.

3.3.2 Dimensionality of GaussianConv layer

Dimensionality of the Gaussian derivative-based kernels

Dimensionality of the kernels used within a Gaussian layer is determined by the selected scale. The size of these is determined by the formula:

$$\text{kernel\_size} = 2 * ceil(\sigma * scale) + 1 \tag{3.4}$$

Where σ chosen to be 3, such that 99.7% of the Gaussian curve is represented within the kernel, since 99.7% of the energy of a Gaussian falls within the domain $[-3\sigma, 3\sigma]$.
Next, since the number of derivatives in a Gaussian pyramid is a triangular number, the number of to be constructed kernels is calculated as follows:

$$\text{num\_derivatives} = \binom{order + 2}{2} \tag{3.5}$$

Since we split the 2D Gaussian convolution into two kernels, it follows that the eventual shape of the kernel stack is ($num\_derivatives \times 2 \times kernel\_size$). This is then split and reshapen into its dx and dy parts, which result in the following kernel shapes:

$$\begin{aligned} \text{dx} &= (num\_derivatives \times 1 \times 1 \times kernel\_size) \\ \text{dy} &= (num\_derivatives \times 1 \times kernel\_size \times 1) \end{aligned} \tag{3.6}$$

Dimensionality of the weights

The weighted sum part of the GaussianConvLayer is implemented as a (1x1) convolution over all channels of the output of the Gaussian convolutions. The number of outputs is selectable, resulting in a kernel size of:

$$weights = (out\_channels \times in\_channels * num\_derivatives \times 1 \times 1) \tag{3.7}$$

Usage of *groups*

When convolving an input with a certain number of output channels, the size of the output generally is equal to dimensions of the input, except that the output has only a single channel. Thus for example if one has 3 input channels and wants 9 output channels, 9 filters are used, each 3 channels deep. However, if we set $groups = 3$, each kernel is duplicated over the channels, resulting in far fewer weights being learned, namely the kernel depth is no longer 3 but is set to 1. This can be seen if we plug these values into PyTorch:

```
>>> img = torch.ones(1, 3, 32, 32)  # Batch-size * Input-channels * Img-shape
>>> default = nn.Conv2d(in_channels=3, out_channels=9, kernel_size=1, groups=1)
>>> with_groups = nn.Conv2d(in_channels=3, out_channels=9, kernel_size=1, groups=3)

>>> list(default.parameters())[0].shape
```

```
torch.Size([9, 3, 1, 1])
>>> list(with_groups.parameters())[0].shape
torch.Size([9, 1, 1, 1])

>>> default(img).shape
torch.Size([1, 9, 32, 32])
>>> with_groups(img).shape
torch.Size([1, 9, 32, 32])
```

In a regular CNN this would be undesirable behaviour, since this means that the network has no way of learning the difference between the channels, since the learnable weights are shared over all channels. In our case this is no problems however, since we only learn the weights of the channels *after* they have been convolved with the Gaussian kernels. This means we only need to generate the stack of Gaussian kernels and duplicate these over the number of input channels, which is implemented in the following way:

```
in_channels, order, scale = 3, 4, 1

# shape=(15,2,7)
kernels = fill_Gaussian_kernels(order, scale)

# shape=(15,1,2,7)
kernels = kernels.reshape(kernels.shape[0], 1, *kernels.shape[1:])

# shape=(45,1,2,7)
kernels = kernels.repeat(in_channels, axis=0)
```

Example

As input for the layer we have a single image that has 3 channels, and this image is 32x32 pixels. As order we have selected to generate all derivatives up to the 4rd order, which is $\binom{4+2}{2} = 15$ pairs of derivatives in the x and y direction. As scale we select the default $scale = 1$, thus each of these kernels has a total number of elements of $2 * ceil(3 * 1) + 1 = 7$, where the shape of the kernels in the x direction is $(7, 1)$ and the shape of the kernels in the y direction is $(1, 7)$.

Thus we have a layer containing:

- A kernel for the derivatives in the x direction of shape (45x1x1x7)
- A kernel for the derivatives in the y direction of shape (45x1x7x1)
- A 2D convolution layer with 45 in-channels and 10 out-channels thus of shape (10x45x1x1)

And in the forward pass:

1. The input is a matrix of shape (1x3x32x32)
2. A convolution takes place with the kernels containing the derivatives in the x direction where $groups = in\_channels = 3$
3. A convolution takes place with the kernels containing the derivatives in the y direction where $groups = in\_channels * num\_derivatives = 3 * 15 = 45$
4. a convolution takes place with a (10x45x1x1) kernel to generate a weighted sum over all convolved channels.

The shape of the output is:

1. After input: (1x3x32x32)
2. After convDx: (1x45x32x32)
3. After convDy: (1x45x32x32)
4. After weighted sum: (1x10x32x32)

CHAPTER 4

Experiments

4.1 MNIST

The MNIST dataset is a database of handwritten digits containing 60000 training and 10000 testing examples. Every image is 28×28 pixels and is 1 channel deep.

Figure 4.1: Sample taken from the MNIST dataset, Source: Josef Steppan from Wikimedia Commons

State-of-the-art classification currently sits at 99.7% accuracy. We will primarily investigate the performance of our network in the context of the impact of different scales in the two Gaussian-Conv2d layers. It is not necessarily our goal nor our expectation to beat the state-of-the-art performance, but to merely compare the RFNNs performance to the control CNN to ground our results.

4.2 Hyper-parameters

Since there is a vast scala of hyper-parameters in training an ANN, the chosen values and algorithms are founded by the pros and cons as shown in chapter 2.
A good learning rate was determined by examining the convergence rates of the networks. The number of epochs was set at 30 for the control CNN because the network had not yet converged at 20 epochs, which is the case using the RFNN.
For all following experiments, the hyper-parameters were set as follows:

- Number of epochs: (RFNN: 20), (control CNN: 30)
- Learning rate: 0.001
- Learning rate decay: $learning\_rate = learning\_rate * 0.1$, every 10 epochs
- Gradient method: SGD with Nesterov momentum
- Momentum: $1 - (10 * learning\_rate)$
- Batch size: 500

4.3 RFNN setup

As test setup a Two-Layer Receptive Fields Neural Network was constructed using the following layer layout:

1. Batch Normalisation
2. GaussianConvLayer2d(in_channels=1, out_channels=64, order=4, scale=size_one, stride=1)
   - MaxPool(kernel_size=(2,2), stride=2)
   - ELU
3. GaussianConvLayer2d(in_channels=64, out_channels=128, order=4, scale=size_two, stride=1)
   - MaxPool(kernel_size=(2,2), stride=2)
   - ELU
4. LinearLayer(in_channels=7*7*128, out_channel=128)
   - Dropout(p=0.5)
   - ELU
5. LinearLayer(in_channels=128, out_channels=10)
6. LogSoftmax

Figure 4.2: Layout of the 2-layer RFNN

The choice to let the first layer be 64 outputs wide was taken following [24], who also chose a width of 64 for the first layer of his RFNN in his experiments. Verkes [24] selected his second layer to also have a width of 64, however, we selected the width of the second layer to be 128 following [20]'s rule of thumb, in that it is preferred to double the width of a layer whenever the size of the input is halved.

For scales an interval over the range (0, 2) was selected with in total 5 values to be able to examine the impact of different scales in a fine-grained manner. Following scale-space theory [19], the scale series is determined in the following way:

1. Define some start scale s_1 and some stop scale s_2.
2. Define some number of steps $n >= 1$ from the start scale to the stop scale.
3. The series is defined as:
$$[\alpha^0 s_1, \alpha^1 s_1, \ldots, \alpha^n s_1] \tag{4.1}$$
4. $\alpha^n s_1$ is now redefined as:
$$\alpha^n s_1 = \mathrm{s}_2 \tag{4.2}$$
5. It follows that:
$$\alpha = \sqrt[n]{\frac{s_2}{s_1}} \tag{4.3}$$
6. Since the value of α is now known, all values in the series can be easily calculated.

Figure 4.3: Determining all values in a scale series

In our case with $s_1 = 0.5$, $s_2 = 2.0$ and $n = 5$ this results in the series:

$$\text{scale series} = [0.500,\ \sim 0.707,\ 1.000,\ \sim 1.414,\ 2.000] \tag{4.4}$$

The product was taken of this series with itself, resulting in 25 2-tuples of scale values, which each represent the scale parameter in the first and in the second layer respectively.

4.4 Control CNN setup

Since it is difficult to directly compare the previously described RFNN to more (spatially) complex architectures like VGG-net [20] or ResNet [6], we choose to construct a regular ConvNet with a design as close as possible to the design of the RFNN layout.
Where representation power might be a concern when using the Gaussian basis and thus it is sensible to make sure the entire energy of the curve is represented by selecting a kernel size large enough to contain $2 * 3\sigma$ of the Gaussian curve, this is not so much a concern when using a regular CNN kernel.
Because we want to in some way enforce the CNN layer to have a roughly equivalent amount representation power as a GaussianConv layer, all kernel sizes of the constructed CNN are contained in the set {(3, 3), (5, 5), (7, 7), (9,9)}. These values follow loosely from 3.4 when setting σ to 2, and selecting the scales to be the same range as the values selected for the RFNN setup.
Thus, the layout of the CNN is as follows:

1. Batch Normalisation
2. Conv2d(in_channels=1, out_channels=64, kernel_size=size_one, padding=size_one//2, stride=1)
    - MaxPool(kernel_size=(2,2), stride=2)
    - ELU
3. Conv2d(in_channels=64, out_channels=128, kernel_size=size_two, padding=size_two//2, stride=1)
    - MaxPool(kernel_size=(2,2), stride=2)
    - ELU
4. LinearLayer(in_channels=7*7*128, out_channel=128)
    - Dropout(p=0.5)
    - ELU
5. LinearLayer(in_channels=128, out_channels=10)
6. LogSoftmax

Figure 4.4: Layout of the control ConvNet

4.5 CNN baseline

The baseline is obtained from training the control CNN layout described in 4.4 with the MNIST dataset.

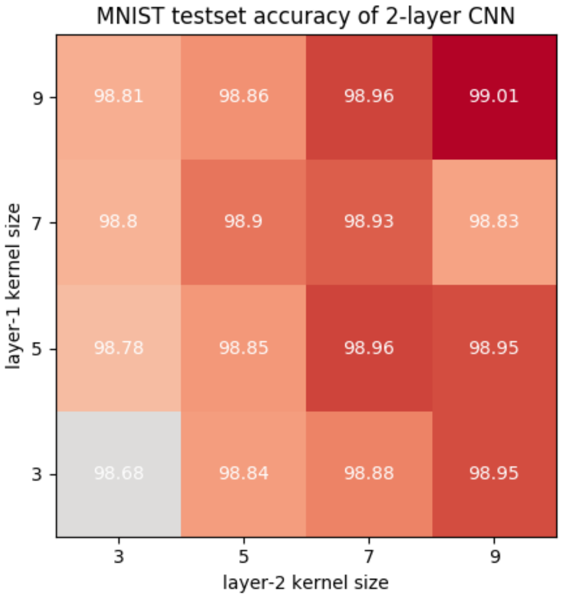


Figure 4.5: Classification results on the MNIST dataset from the control CNN layout

It can be seen that maximum performance lies at 99.01%. However, this maximum performance lies at the upper bounds of our selected kernel sizes at a size of 9×9, which is simply unrealistic for performant modern-day network designs; there are simply too many parameters to be learned. Krizhevsky, Sutskever, and Hinton [12] used a maximum kernel size of 11×11, but only when using a stride of 4, making the effective kernel size more along the lines of 5×5 − 7×7. A kernel size of 7×7 in the first layer is however still seen in modern network designs [6], so taking the maximum accuracy score out of this row, which is 98.93%, seems like a good baseline to compare the results of the rest of our experiments to.

4.6 Experiment 1

The first experiment testing the influence of the different scales in the layers our RFNN was done with the network layout exactly as described in 4.2.

The results were as follows:

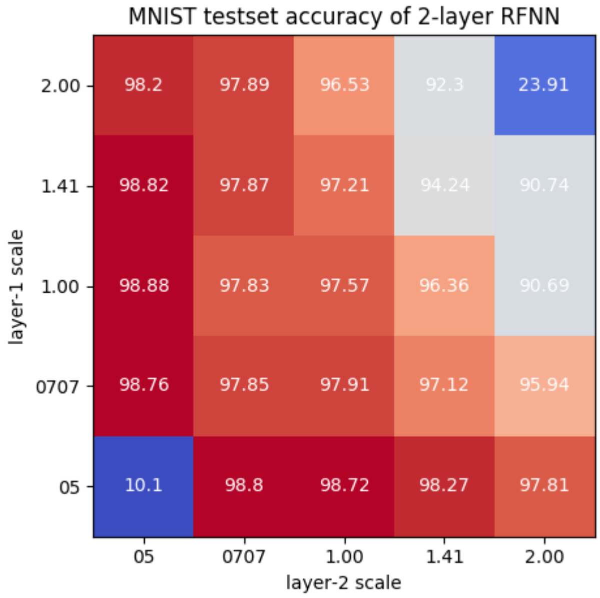


Figure 4.6: Classification results on the MNIST dataset from the unmodified RFNN layout

The first thing to note is that the maximum performance is not significantly different than the baseline performance. Secondly, there is a clear tapering present of performance towards layers having higher scales, and a clear increase in performance towards layers having smaller scales. All the more odd it is then, that having the scale of the first and second layer match, having scale = 0.5 results in an accuracy as random guessing.
One explanation might be the instability of having a Gaussian kernel using a scale of only 0.5. The highest value of the 0th order Gaussian is 0.8, which is almost the identity. The minimum and maximum values of the derivatives are also of quite high magnitude, which might result in bogus iteration steps in gradient descent.

4.7 Experiment 2

Following experiment 1, an investigatory path we explored is what would happen to the if the input image was up-scaled. Our intuition was that when the image was up-scaled, the respective receptive fields were also up-scaled with the same factor, and thus that the scale associated with detecting these would also have to be multiplied with the same factor.
Bilinear interpolation was used when upscaling the input data with a factor of 2.
We replaced the second pooling layer with a pooling layer having $kernel\_size = (4, 4)$ and $stride = 4$. This to ensure that the latter two fully-connected layer would be identical to the original setup, meaning the representation strength of these layers would remain the same.

This gave the following result:

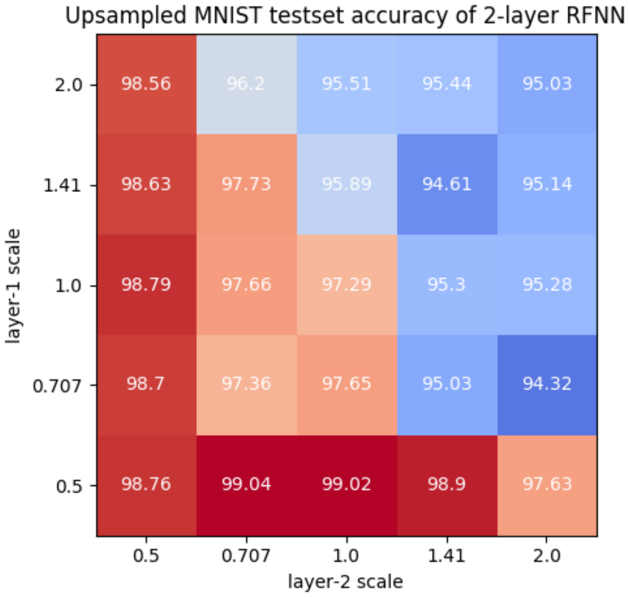


Figure 4.7: Classification results on the upscaled MNIST dataset

It is clear that even with an up-scaled dataset the best scales are still largely the same. There is still a large tapering of performance present towards layers with higher scales. It is however nice to note that both the scale combinations of (0.5, 0.5) and (2.0, 2.0) now have comparable results to the rest of this set of results.
Another result is that the performance has increased by around 0.1% to 99.04%. This might however not be that remarkable since the maximum performance obtained by the control CNN was also at this level.

4.8 Experiment 3

Since following experiment 2 it was still not entirely clear why the layers with the lower scales had better results, another idea was tried.
In the default definition of the Receptive Fields Neural Network no biases are taken into account. However, since the default implementation of the Convolution layer in PyTorch was used within the GaussianConv2d layer, biases were enabled by default in the weighted sum operation. One intuition we had was that the network might be somewhat 'cheating' by learning these biases, and that we should try to force the network to only use the weighted sum of the actual convolutions.

The following results were observed:

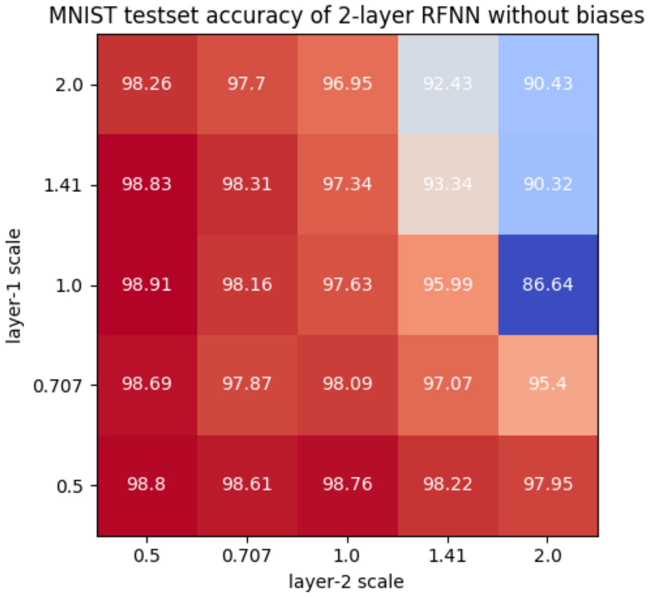


Figure 4.8: Classification results of a RFNN without biases on the MNIST dataset

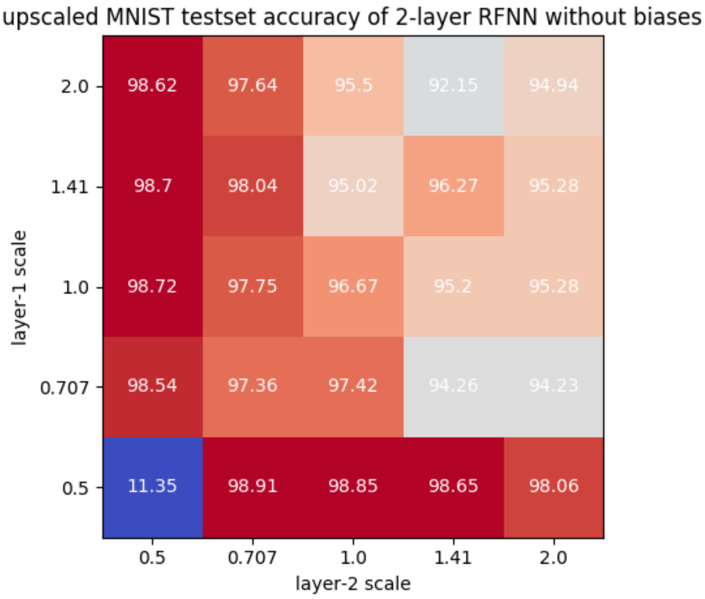
Still, the best performances were present in the layer with the lower scales, but this is largely what we expected after experiment 1.
What *is* however a remarkable result is that this is the first set of results for which out intuition holds true, which is:

Taken some s_1, s_2, where layer 1 uses s_1 and layer 2 uses s_2, the performance of the network is better if $s_1 > s_2$, than when $s_1 < s_2$.

4.9 Experiment 4

Since the performance of the network had improved across the board after the up-sampling of the input in experiment 2, a last experiment was performed where the network was augmented in by both using up-sampling of the input and disabling the bias of the weighted sums.
The expected result was a more stable distribution of accuracy over the different scale combinations, while also satisfying our intuition described in experiment 3.

The following results were observed:



	0.5	0.707	1.0	1.41	2.0
2.0	98.62	97.64	95.5	92.15	94.94
1.41	98.7	98.04	95.02	96.27	95.28
1.0	98.72	97.75	96.67	95.2	95.28
0.707	98.54	97.36	97.42	94.26	94.23
0.5	11.35	98.91	98.85	98.65	98.06

Figure 4.9: Classification results of a RFNN without biases on the upscaled MNIST dataset

Neither our intuition of experiment 3, nor the stable results of experiment 2 are present in this new set of results.

CHAPTER 5

Discussion & Conclusion

The main result to take away from all ran experiments is that selecting the correct scale parameter is important when training a RFNN. It is still quite odd that the best performing scales are so relatively low, with the best performing scale 2-tuple containing the value 0.5 at least once in all experiments.
An explanation for this might be that the MNIST dataset consists only of 'edge-like' classes, i.e. all classes consist of an object represented only by line segments. There are no classes present in the dataset which represent more 'blob-like' objects, i.e. regular objects like humans, vehicles, etc., which might require larger scales to be accurately classified.

As future work, it would be worthwhile to re-run the experiments on a dataset like CIFAR-10 [11]. This dataset contains images of the same object photographed at smaller and larges distances, and it would be interesting to analyse which scales give the best results in this context.

Also interesting to investigate is convergence rates of the RFNN design versus a comparable Convolutional Neural Network. Although no quantitative analysis has been performed in this paper, monitoring the number of epochs it took a network to converge to its final state while the experiments were running seems to suggest much faster convergence rates of RFNN designs than its CNN counterparts. Intuitively this would make sense; examining first-layer converged convolution kernels in Zeiler and Fergus [26] reveals the kernels to look eerily close to steered Gaussian kernels. Since a RFNN already has these Gaussian kernels built in, the only thing the network would have to learn is the 'steering'. Only learning the 'steering' of the Gaussian kernels should be much faster to converge due to the reduced parameter space of the RFNN design compared to learning an entire kernel in a regular CNN.

Ultimately, the most logical step extending this research would be incorporating a learnable scale parameter into the RFNN design. Due to the choice of using PyTorch with its dynamic graphs as a framework, instead of competitors like TensorFlow with their static graphs, this should not be too large a step to implement. As expressed in the introduction 1, the ultimate hope would be that this learnable scale parameter results in the network learning the best scale to examine an input image at, which in the ultimate case would result in a scale-invariant network.

And then we could finally stop pixel-fucking, and with that make Koenderink proud [10].

Bibliography

[1] Jason Brownlee. *Exploding gradients in neural networks.* 2018. URL: https://machinelearningmastery.com/exploding-gradients-in-neural-networks/ (visited on 06/08/2018).

[2] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (2015).

[3] Facebook. *Announcing PyTorch 1.0 for both research andproduction.* 2018. URL: https://code.facebook.com/posts/172423326753505/announcing-pytorch-1-0-for-both-research-and-production/ (visited on 06/08/2018).

[4] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning.* Vol. 1. Springer series in statistics New York, 2001.

[5] Google. *Welcome to Colaboratory!* 2018. URL: https://colab.research.google.com/notebooks/welcome.ipynb (visited on 06/08/2018).

[6] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778.

[7] Alan L Hodgkin and Andrew F Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve". In: *The Journal of physiology* 117.4 (1952), pp. 500–544.

[8] Jorn-Henrik Jacobsen et al. "Structured receptive fields in cnns". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2016, pp. 2610–2619.

[9] Jan J Koenderink. "The structure of images". In: *Biological cybernetics* 50.5 (1984), pp. 363–370.

[10] Jan J Koenderink and Andrea J van Doorn. "Image processing done right". In: *European Conference on Computer Vision.* Springer. 2002, pp. 158–172.

[11] Alex Krizhevsky. *CIFAR-10 and CIFAR-100 datasets.* 2018. URL: https://www.cs.toronto.edu/~kriz/cifar.html (visited on 06/08/2018).

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25.* Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[13] Yann LeCun et al. "Generalization and network design strategies". In: *Connectionism in perspective* (1989), pp. 143–155.

[14] Yann LeCun, Yoshua Bengio, et al. "Convolutional networks for images, speech, and time series". In: *The handbook of brain theory and neural networks* 3361.10 (1995), p. 1995.

[15] Min Lin, Qiang Chen, and Shuicheng Yan. "Network in network". In: *arXiv preprint arXiv:1312.4400* (2013).

[16] David G Lowe. "Object recognition from local scale-invariant features". In: *Computer vision, 1999. The proceedings of the seventh IEEE international conference on.* Vol. 2. Ieee. 1999, pp. 1150–1157.

[17] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *Proc. icml*. Vol. 30. 1. 2013, p. 3.

[18] Scott P Overmyer. "Revolutionary vs. evolutionary rapid prototyping: balancing software productivity and HCI design concerns". In: *Center of Excellence in Command, Control, Communications and Intelligence (C3I), George Mason University* 4400 (1991).

[19] Bart M Haar Romeny. *Front-end vision and multi-scale image analysis: multi-scale computer vision theory and applications, written in mathematica*. Vol. 27. Springer Science & Business Media, 2008.

[20] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[21] Ilya Sutskever. "Training recurrent neural networks". In: *University of Toronto, Toronto, Ont., Canada* (2013).

[22] Tensorflow. *TensorFlow about*. 2018. URL: `https://www.tensorflow.org/` (visited on 06/08/2018).

[23] Stanford University. *CS231n: Convolutional Neural Networks for Visual Recognition*. 2017. URL: `http://cs231n.github.io/neural-networks-1/` (visited on 05/30/2018).

[24] Govert Verkes. "Receptive Fields Neural Networks using the Gabor Kernel Family". PhD thesis. BS Thesis, University of Amsterdam, 2017.

[25] D Randall Wilson and Tony R Martinez. "The general inefficiency of batch training for gradient descent learning". In: *Neural Networks* 16.10 (2003), pp. 1429–1451.

[26] Matthew D Zeiler and Rob Fergus. "Visualizing and understanding convolutional networks". In: *European conference on computer vision*. Springer. 2014, pp. 818–833.

Appendices

APPENDIX A

Implementation in PyTorch

A.1 Implementation of GaussianConv2d layer

```
import numpy as np
import torch
from torch import nn
from torch.nn import functional as F
from torch.autograd import Variable
from scipy.misc import comb


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-1)*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 order_triangle(order):
    """returns a generator with the ordered sequence of orders
    of the derivative piramid up to a certain order
    e.g.: order_triangle(2)
         (0,0)          (0th-order)
      (1,0)(0,1)        (1st-order)
    (2,0)(1,1)(0,2)     (2nd-order)

    output: generator((0,0),(1,0),(0,1),(2,0),(1,1),(0,2))
    """
    for i in range(order+1):
        for j in range(i+1):
            yield (i-j, j)

def fill_gaussian_kernels(order, scale):
    filter_extent = np.ceil(3*scale).astype(int)
    num_derivatives = int(comb(order+2, 2)) # 0-th order should have 1 element
    x_values = np.arange(-filter_extent, filter_extent+1, dtype=np.float)

    kernels = np.empty((num_derivatives, 2, len(x_values)))
    for i, (n_dx, n_dy) in enumerate(order_triangle(order)):
        kernels[i, 0] = gaussian(x_values, scale, n_dx)
        kernels[i, 1] = gaussian(x_values, scale, n_dy)
    return kernels


class GaussianConv2d(nn.Module):
    def __init__(self, in_channels, out_channels, order=4, scale=1, stride=1):
        super(GaussianConv2d, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.num_derivatives = int(comb(order+2, 2))
        self.padding = int(np.ceil(3*scale))
        self.kernel_size = 2*self.padding + 1

        kernels = fill_gaussian_kernels(order, scale)
        kernels = kernels.reshape(kernels.shape[0], 1, *kernels.shape[1:])
        kernels = kernels.repeat(in_channels, axis=0)

        dx = kernels[...,np.newaxis,0,:]
        dy = np.swapaxes(kernels[...,np.newaxis,1,:], -1, -2)
        self.dx = torch.tensor(dx, requires_grad=False, dtype=torch.float).cuda()
        self.dy = torch.tensor(dy, requires_grad=False, dtype=torch.float).cuda()
        self.weights = nn.Conv2d(in_channels*self.num_derivatives, out_channels,
                                 kernel_size=1, stride=stride, bias=False)


    def forward(self, x):
        out = F.conv1d(x, self.dx, groups=self.in_channels,
                       padding=(0, self.padding))
        out = F.conv1d(out, self.dy, groups=self.num_derivatives*self.in_channels,
                       padding=(self.padding, 0))
        out = self.weights(out)
        return out
```

A.2 Implementation of RFNN training regime

```
from warnings import warn
from itertools import product

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.nn.parameter import Parameter
from torch.autograd import Variable
from torchvision import datasets, transforms


CUDA = torch.cuda.is_available()

if CUDA:
    DEVICE = torch.device('cuda')
else:
    warn("GPU acceleration not found, defaulting to CPU")
    DEVICE = torch.device('cpu')

class TwoLayerRFNNet(nn.Module):
    def __init__(self, size_one, size_two):
        super(TwoLayerRFNNet, self).__init__()
        self.batchnorm = nn.BatchNorm2d(1)
        self.l1 = nn.Sequential(
            GaussianConv2d(1, 64, scale=size_one),
            nn.MaxPool2d(2),
            nn.ELU()
        )
        self.l2 = nn.Sequential(
            GaussianConv2d(64, 128, scale=size_two),
            nn.MaxPool2d(2),
            nn.ELU()
        )
        self.l3 = nn.Sequential(
          nn.Linear(128*7*7, 128), # =(num_channels*shape)
          nn.ELU(),
          nn.Dropout(),
        )
        self.l4 = nn.Linear(128, 10)

    def forward(self, input):
        out = self.batchnorm(input)
        out = self.l1(out)
        out = self.l2(out)
        out = out.view(-1, np.prod(out.shape[1:]))
        out = self.l3(out)
        out = self.l4(out)
        return F.log_softmax(out, dim=1)


# Loss and Optimizer
```

```
log_interval = 10
batch_size = 500

kwargs = {'num_workers': 1, 'pin_memory': True} if CUDA else {}
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                   ])),
    batch_size=batch_size, shuffle=True, **kwargs
)
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=False, transform=transforms.Compose([
                       transforms.ToTensor(),
                   ])),
    batch_size=batch_size, shuffle=True, **kwargs
)

def train(model, learning_rate, epoch):

    optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                          momentum=1-(10*learning_rate), nesterov=True)

    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = Variable(data), Variable(target)
        if CUDA:
          data, target = data.cuda(), target.cuda()

        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % log_interval == 0:
            print('Train␣Epoch:␣{}␣[{}/{}␣({:.0f}%)]\tLoss:␣{:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))

def test(model):

    model.eval()
    test_loss = 0
    correct = 0
    for data, target in test_loader:
        data, target = Variable(data), Variable(target)
        if CUDA:
          data, target = data.cuda(), target.cuda()

        output = model(data)
        test_loss += F.nll_loss(output, target, size_average=False).data.item() #
        pred = output.data.max(1, keepdim=True)[1] # get the index of the max log-
        correct += pred.eq(target.data.view_as(pred)).long().cpu().sum()

    test_loss /= len(test_loader.dataset)
    print('␣␣Test␣set:␣Average␣loss:␣{:.4f},␣Accuracy:␣{}/{}␣({:.0f}%)'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))


def scale_series(in_scale, out_scale, num_steps=4):

    alpha = np.power(out_scale/in_scale, 1/num_steps)
    alphas = np.power(alpha, np.arange(num_steps+1))
    scales = in_scale * alphas

    return product(scales, repeat=2)


def search_parameter_space():
    scale_tuples = scale_series(0.5, 2)
    for (s1, s2) in scale_tuples:
        print("Training␣with␣scales:␣{},␣{}".format(s1, s2))

        model = TwoLayerRFNNet(s1, s2).cuda()

        train_accuracy = []
        test_accuracy = []

        num_epochs = 20
        learning_rate = 0.001
        for epoch in range(1, num_epochs + 1):
            if not epoch % 10:
              learning_rate /= 10
            train_accuracy.append(train(model, learning_rate, epoch))
            test_accuracy.append(test(model))

if __name__ == '__main__':
    search_parameter_space()
```