Kölbl, Laura:

Deep Convolution Neural Networks for Image Analysis

Masterarbeit, Sommersemester 2017

Fakultät für Mathematik, Informatik und Statistik

Ludwig-Maximilians-Universität München

https://doi.org/10.5282/ubm/epub.41012

Deep Convolution Neural Networks for Image Analysis

Master's thesis

Ludwig-Maximilians-University Munich

Institute of Statistics

Author: Laura Kölbl

Supervisor: Prof. Dr. Volker Schmid

Submission date: 19.06.2017

Table of contents

Deep Convolution Neural Networks for Image Analysis **1**

1 Understanding Deep Convolutional Neural Networks **3**

1.1 Architecture of Convolutional Neural Networks 3

1.1.1 Convolutional Layers 4

1.1.2 Activation Function 6

1.1.3 Pooling Layer 8

1.1.4 Fully Connected Layer 9

1.1.5 Linear Classifier 9

1.2 Understanding the Training of Convolutional Neural Networks . 11

1.2.1 Weights and Biases 11

1.2.2 Objective Function 12

1.2.3 The Gradient Descend Algorithm 14

1.2.4 The Backpropagation Algorithm 16

1.2.5 The Vanishing Gradient Problem 19

1.2.6 Batch Normalization 20

2 Training Deep Convolutional Neural Networks **22**

2.1 Training, Test and Validation set 22

2.2 Preprocessing the Data 23

2.3 Initialization Strategies 24

2.4 Gradient Descent Optimization Algorithms 26

2.4.1 Momentum 27

2.4.2 Adagrad 28

2.4.3 RMSProp 29

2.4.4 Adadelta 30

2.4.5 Adam 30

2.5 Batch Size 31

2.6 Overfitting and Regularization 31
2.6.1 Early Stopping 32
2.6.2 Data Set Augmentation 32
2.6.3 L1 and L2 Regularization 32
2.6.4 Dropout 33

3 DCNNs: The VGG, Inception and ResNet Architecture 35
3.1 The VGG Network 35
3.2 Inception Network: Inception-v3 38
3.3 Deep Residual Network: ResNet 44

4 Implementation in R Using the MXNet Deep Learning Library 48
4.1 The MXNet Deep Learning Library 48
4.2 Parallelization and Hardware 48
4.3 The Pretrained Models: VGG-19, Inception-v3 and ResNet-152 . 49
4.4 Finetuning the VGG, the Inception and the ResNet on Varying Amounts of Places2 Image Data 49
4.4.1 The Places2 Data Set 50
4.4.2 Training Settings and Results for the VGG, Inception and ResNet on the Places Data 51
4.4.3 Summary of the Test Results 55
4.5 Training the VGG, the Inception and the ResNet on Medical Data 58
4.5.1 The Breast Cancer Data Set 59
4.5.2 Results for the Medical Data 60

5 Conclusion 62

6 Electronic Appendix 65

Abstract

This master's thesis is about deep convolutional neural networks (DCNNs) and their application on classification tasks. It is particularly concerned with the fields of supervised learning and feedforward networks.

At first, the architecture and the mechanisms behind DCNNs are explained from scratch. Next, the work gives an overview over different methodological approaches that are crucial for the training of neural networks in practice and also a description of the VGG, Inception and ResNet architecture.

Further, results from the finetuning of pretrained VGG-19, Inception-v3 and ResNet-152 networks on small data sets are presented.

With three classes of the Places2 data set, it was investigated how the test accuracy changes with respect to the three models and different data sizes (20 to 1000 images per class). The results showed the best test accuracies and smallest training times for the Inception-v3 network. They also indicate that 250 images per class can be enough to acquire a very competent model (e.g. 97.8 percent test accuracy for the Inception-v3).

Applying finetuning to a medical data set on the other hand did not yield a network that can discriminate between benign and malignant breast cancer cases. The data set encompassed 29 benign and 30 malignant mammography images.

Deep Convolution Neural Networks for Image Analysis

Deep convolutional neural networks (DCNNs) are a part of machine learning, where machines learn to detect patterns solely from data.
They can handle a wide range of applications. With regard to the classification of images, DCNNs are utilized for instance for facial recognition tasks [Sun2014], the determination of art epochs [Hentschel2016], and also medical applications like breast cancer classification [Lévy2016].
Challenges for the image recognition tasks are different sizes and viewpoints, but also background clutter and varying illumination. Nevertheless have state-of-the-art networks already reached and even surpassend human performance [He2015b].
These results are possible also because there are large-scale data sets like the ImageNet [Russakovsky2015] and Places [Zhou2016], which provide millions of labeled images. But not for all tasks are large amounts of labeled data at hand. This is why the application of convolutional neural networks on small data sets will be investigated in this work.

This master's thesis is about deep convolution neural networks, especially about their application on image analysis. It is confined to supervised learning and feedforward neural networks.
The aim of the first part of this work is to enable the reader to understand how neural networks work in theory and give an understanding of current methods for their use in practice.

The first chapter will cover the architecture of convolutional neural networks and how they work. In chapter two it is explained how to train a CNN and which parameters can be set. Chapter three contains the ideas and the architectures behind the VGG, the Inception and the ResNet.
In the second part, deep neural networks are implemented in R using the MXNet deep learning library (chapter four). Pretrained VGG-19, Inception-v3 and ResNet-152 networks are used to finetune small subsets of the Places2 data and a medical data set.
The three networks were applied at first to the Places2 data set. They were finetuned using different amouts of training data to investigate the impact on the test accuracies and the training time.
The same pretrained networks were then used to finetune a medical data set, trying to obtain a network that is able to distinguish between benign and malignant breast cancer cases.
In the last chapter, the results of the implementation are summarized and possible future extensions of the present work are proposed.

1 Understanding Deep Convolutional Neural Networks

1.1 Architecture of Convolutional Neural Networks

For a deeper understanding of the ideas and workings behind convolutional neural networks, one must first comprehend their architecture.
A neural network consists of different layers. A convolutional neural network is a network that consists of at least one convolutional layer. Such a very simple CNN can be seen in figure 1.1.

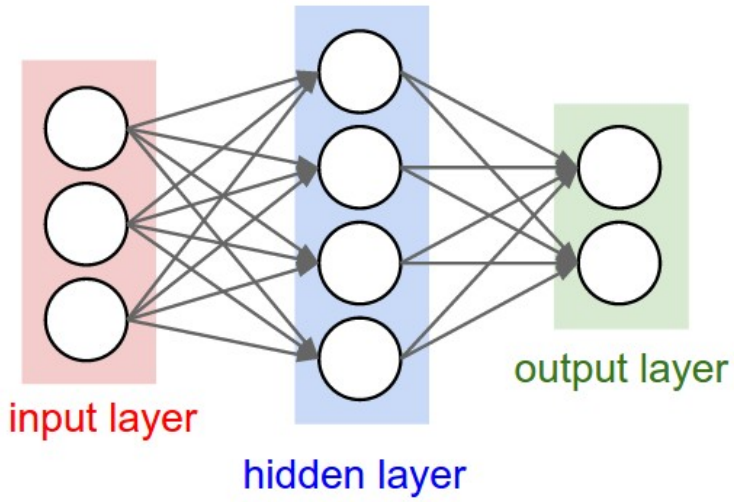


Fig. 1.1: Convolutional neural network with one convolutional layer, source: Karpathy 2017

The more of those convolutional layers, the deeper is the network.

Next to convolutional layers there are also pooling layers and fully connected layers. Additionally, after every convolutional layer there is an activation function. And at the very end of the network a linear classifier computes the network output [Guo2015].

1.1.1 Convolutional Layers

The idea behind convolutional neural networks is partly lent from biology, from the way neurons work in living creatures: our neurons are firing when the input of their receptive field exceeds a certain threshold. While there are many input signals, there is only one output signal.
In CNNs that are used for image classification, the initial input is an image, represented by pixels. Behind the input layer there is a hidden layer called convolutional layer, consisting of so-called neurons. They are represented by the circles in figure 1.1. The neurons, like their biological analogy, can maintain many inputs and transform them into one output.
Figure 1.2 shows this for an input layer of 8x8 and a kernel of size 3x3: the activations of the neurons are computed with a kernel. The kernel (in this case a 3x3 matrix) is also called convolution matrix, filter or feature detector. This matrix glides over the input matrix, calculating the dot product for each pixel value of the input. This computation involves the surrounding pixels, an area called the local receptive field (here 3x3). By including nearby pixels convolutional neural networks can also use information about proximity.
The output matrix is called convolved feature, activation map or feature map (s. figure 1.3).
The width of the feature map is the amount of kernels that have been used. Every kernel identifies specific structures, e.g. edges. The more kernels you have, the more structures can be discovered.

The stride determines the “jumps“ the kernel makes while sliding over the input matrix. If the stride is one, the filter is only moved one pixel.
But every picture also has borders. To make it possible to apply the kernel

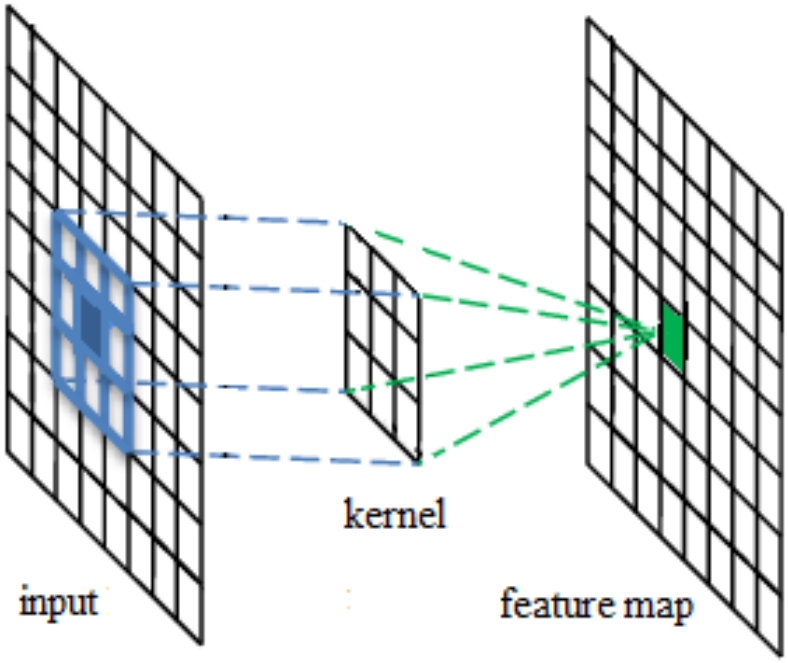


Fig. 1.2: Visualisation of a convolutional layer; the kernel matrix glides over the input matrix, computing the values of the feature map, source: adapted from Guo et al. 2015

to the border pixels one can add padding, for example zero-padding. When using zero-padding the input matrix is filled with zeros where the image would have ended. If there was no padding the output matrix would be smaller than the input matrix after the convolution [Goodfellow2016] [Guo2015] [Karpathy2017][Neubert2016].

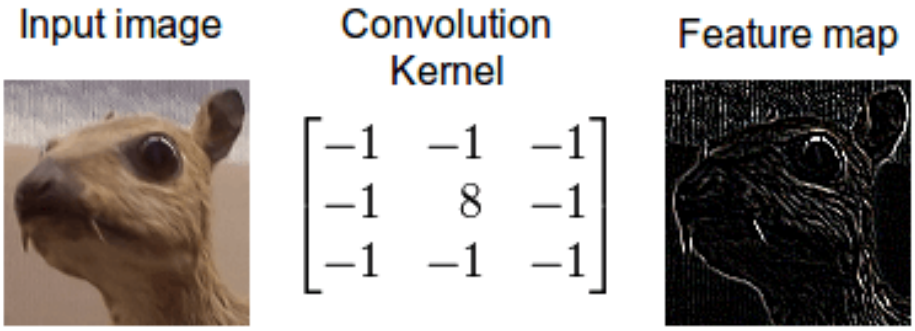


Fig. 1.3: Visualisation of a convolved feature, source: Dettmers 2015

1.1.2 Activation Function

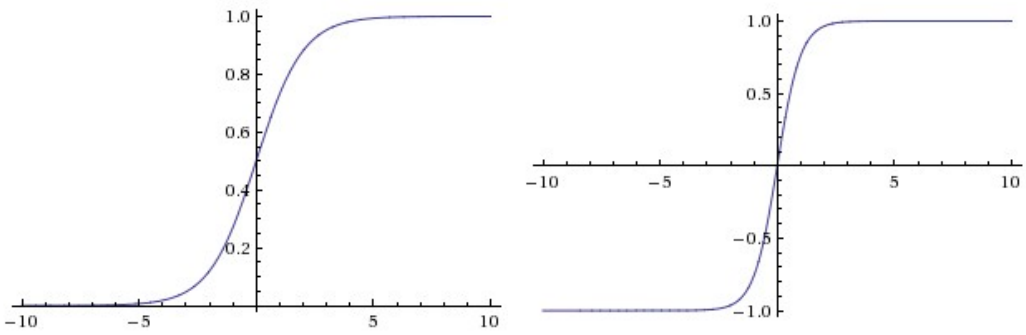
The output of the convolutional layer is the input to the activation function. This input is normally linear, because it so far has been only being exposed to linear transformations. The activation function adds some non-linearity to it. This happens at every neuron in the layer.
Common activation functions are the sigmoid function, the tanh function and the ReLU function (s. table 1.1).

sigmoid function	$\frac{1}{1+e^{-x}}$
tanh function	$\frac{2}{1+e^{-2x}-1}$
ReLU function	$f(x)=\begin{cases}0, & x<0\\ x, & x\geq 0\end{cases}$

Table 1.1: examples of common activation functions

The sigmoid function has range [0, 1] (s. figure 1.4a). Values that are fed to the sigmoid function are “squeezed“ to fit in this range. For very large values this means that they take the value one, and for very small values it means that they take the value zero. The interpretation is pretty intuitive since we can read the zeros as a neuron not firing, and the ones as a neuron firing with maximum frequency.

The biggest problem about the sigmoid function is that the gradients can be miniaturized. Especially neurons saturating at zero or one cause very small gradients, which leads to a damped or dead signal. This is also called the



(a) sigmoid function (b) tanh function

Fig. 1.4: The sigmoid and the tanh activation function, source: Karpathy 2017

"vanishing gradient problem" and is explained more precisely in section 1.2.5.

The tanh function results into values in the interval [-1,1]. It behaves very similar to the sigmoid function, thus being also subjected to the possibility of vanishing gradients. But the advantage of the tanh function over the sigmoid function is that it is zero-centered (s. figure 1.4b).

Lately, the most commonly used activation function is the ReLU function:

$$f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{1.1}$$

Compared to the sigmoid or tanh function the ReLU function was found to be learning faster, and reaching better training accuracies earlier in training [Krizhevsky2012]. Moreover, ReLU functions are not saturating like the sigmoid- and tanh function and they do not involve as much mathematic operations as f.e. exponentials .
A visualization of the ReLU can be found in figure 1.5.

On the downside, ReLU units can "die". If a large gradient passes a ReLU neuron, the weights may change in a manner that causes that neuron to never activate again. But this problem can be addressed by using adequately small learning rates.

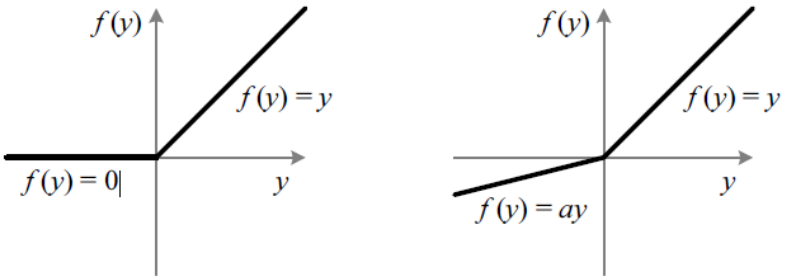


Fig. 1.5: The ReLU (l.) and the PReLU activation function, source: Karpathy 2017

Another option to address the dying of ReLU units are variations of the ReLU function like Leaky ReLUs and PReLUs (s. figure 1.5). With these altered functions negative values are not turned to zero, but turned into a small negative slope:

$$f(x) = \begin{cases} ax, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{1.2}$$

In case of the Leaky ReLU, a is some small fixed constant. In Parametric ReLUs (PReLU) a is a parameter learned from the data. The thereby added computational cost is negligible.

The advancement from Leaky ReLUs is still ambiguous. In the case of PReLUs, He and al. [2015b] claim to have achieved a 1.2 percent improvement in the error rate with PReLUs compared to ReLUs [He2015b] [VanDoorn2014] [Karpathy2017].

1.1.3 Pooling Layer

With a pooling layer the dimensions of the feature maps and the number of parameters can be reduced. The aim is to keep important information, while less crucial information shall be tossed. Pooling therefore reduces the data and speeds up the computation.

In max pooling layers for example only the neurons with the highest activation within the filter area are retained. In figure 1.6 max pooling is visualized for a 2x2 pooling filter and stride 2. The former [4x4] matrix is here reduced to a

smaller [2x2] matrix.

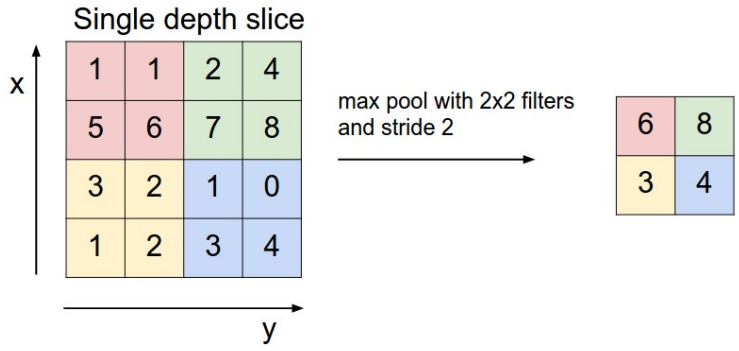


Fig. 1.6: Max pooling, source: Karpathy 2017

There is another pooling operation called average pooling, where the pooling filter computes the average of the filter area instead of the maximum.

When the pooling filter has the same size as the input layer, the pooling operation is called global pooling. If for example global max pooling would be applied to the matrix in figure 1.6, the output would be 8 [Goodfellow2016] [Guo2015].

1.1.4 Fully Connected Layer

In a fully connected layer each neuron is connected to every neuron in the previous layer, and each connection has its own weight. Therefore using these layers entails a lot of parameters. Fully connected layers are found at the very end of the network and enable high-level reasoning [Guo2015] [Neubert2016].

1.1.5 Linear Classifier

After the last fully connected layer, there is a classifier that computes the output values for each class. In case of image classification, these values are the probabilities for each class. The predicted image class is usually simply the one

with the highest computed probability.

A standard classifier is the softmax classifier. The softmax models a joint probability distribution, therefore the probabilities computed by the softmax add up to one. This means if the probability for one class goes up, the probability for another has to go down [Tang2013].

1.2 Understanding the Training of Convolutional Neural Networks

With the knowledge about the architecture of neural networks in mind, in this section it will be explained how the training of CNNs for image analysis works.

As mentioned, the focus of this thesis lies on feedforward networks and supervised learning. Here is a short description.
Supervised learning means the network is provided with the true labels for the images during learning. In contrast, unsupervised learning would mean the network does not know the true class of an image whilst learning and is trained to disclose a hidden pattern [Ghahramani2004].
Feedforward networks are networks where the information is only passed forward through the network. It is never passed back to a previous layer. Network that allow the back passing of information in the network, causing cycles, are called recurrent neural networks (RNNs) and are very helpful for example when it comes to tasks that involve spoken or written language [Lipton2015].

When our task is image classification, we want a network that, when we provide it with image data, can predict the correct label for these images. For a network to be able to do so it first needs to be trained using already labeled data.
The aim of the training is to find the best values for the network parameters, which are called weights and biases [LeCun2015].

1.2.1 Weights and Biases

Weights and biases are the parameters that are modified during the training process.
In order to teach the network the right values for the parameters, we feed pictures, for example of birds and other animals, to the network, as well as the

correct labels. The machine computes the scores for each image based on its current parameters. These scores represent the likelihood for each class. They might actually indicate a picture is an image of a snake: due to the provided label the network knows its error, and can modify the parameters into ones that would rather lead to an output score that would indicate "bird". So in order to reduce the error, the weights and biases get adjusted.
The error between the scores that are computed based on the current parameter values and the "true" scores can be measured by the objective function [LeCun2015].

1.2.2 Objective Function

The objective function measures the distance between the predicted value and the true value. Objective function is the more general term, objective functions that are minimized are called cost functions.
An example for a cost function is the quadratic cost function, mean squared error (MSE):

$$C(w,b) \equiv \frac{1}{2n} \sum_{x} ||y - a||^2 \tag{1.3}$$

where w stands for the weights, b for the bias and n is the total number of training inputs. With y being the desired output, this cost function becomes small when the computed output of the network (a) is nearly equal to it. Therefore the aim is to minimize the MSE function.
The MSE function is used more commonly in context of regression tasks rather than classification tasks, but it is mentioned because it is very intuitive. For classification it is more usual to use the cross entropy cost function.

The definition of the cross-entropy between a true distribution p and an estimated distribution q is:

$$H(p,q) = -\sum_x p(x) \cdot log\, q(x) \tag{1.4}$$

In the classification context, the "true"probability distribution assigns full probability on the true class, and zero probability on the false classes. Therefore, for our classification problem the mean cross-entropy cost can be written as

$$C = -\frac{1}{n}\sum_x [y \cdot ln(a) + (1-y) \cdot ln(1-a)], \tag{1.5}$$

with n being the total number of training inputs and the sum being over all training inputs x. Again the output of the network is denoted by a and the respective desired output by y. The cross-entropy only outputs positive values since a only takes values between [0, 1]. The output will get closer to zero the better the prediction is.

For classification tasks, the cross-entropy further reduces to

$$C = -ln(a_y^L) \tag{1.6}$$

because the desired output for the false classes is zero. In this equation a_y^L is the probability the network predicts for the true class. For high probabilities the cost will be small, whereas for small probabilities the cost will be larger. This is more or less the same as minimizing the negative log likelihood of the true class (log loss).

For a more thorough understanding, here a small example.

Table 1.2 shows probabilities that have been computed by a network and the respective desired outputs (the ground truth).

At a first glance, you can see that both networks have the same classification error (0.5). Both networks misclassify in the second line. Still, the first network calculates a smaller probability (0.1) for the true class than the second network (0.3).

For the cross-entropy function we would compute

network	network output	desired output	true or false
network 1	0.2 0.2 0.6	0 0 1	true
	0.1 0.1 0.8	0 1 0	false
network 2	0.2 0.2 0.6	0 0 1	true
	0.3 0.3 0.4	0 1 0	false

Table 1.2: exemplary network output probabilities and their respective desired outputs for two images fed to two different neural networks

$$-(ln(0.6) \cdot 1 + ln(0.1) \cdot 1) \approx 2.81 \text{ for the first network and}$$

$$-(ln(0.6) \cdot 1 + ln(0.3) \cdot 1)) \approx 1.71 \text{ for the second network.}$$

Therefore in this example the first network is worse at predicting than the second one, because it has a higher cost value than the second network.
The cross-entropy is used primarily for training. The performance of the network is still measured by the classification accuracy on the test data set [Nielsen2015] [Karpathy2017].

1.2.3 The Gradient Descend Algorithm

The minimization of the objective function is performed using an algorithm called gradient descent. The negative gradient can show the direction of the steepest descend (and the steepness) to minimize the objective function.
For illustration purposes the objective function is often described as a mountain scenery: "the objective function, averaged over all the training examples, can be seen as a kind of hilly landscape in the high-dimensional space of weight values. The negative gradient vector indicates the direction of steepest descent in this landscape, taking it closer to a minimum, where the output error is low on average"[Lecun2015, pp.436f.].
Ideally you want to reach the global minimum of the objective function, and not fall into a local minimum. The weights and biases are variables of the objective function. The gradient vector is a vector which has the partial deri-

vatives to a function for each variable of the function as entries. The gradient vector for a function with m weights therefore would be denoted by

$$g = \bigtriangledown C \equiv (\frac{\delta C}{\delta w_1}, .., \frac{\delta C}{\delta w_m})^T \tag{1.7}$$

This gradient vector is then used for the weight updates. The new weight value θ_{t+1} is calculated by:

$$\theta_{t+1} = \theta_t - \eta g_t \tag{1.8}$$

The weight change $\triangle g_{t+1} = \eta \cdot g_t$ is subtracted from the old weight value θ_t. This minus sign is due to the fact that we need the negative gradient to "descend"(as opposed to ascend which would require the positive gradient). The learning rate η determines the length of the steps that are made with each parameter update. A learning rate that is too small can really slow down learning, because you only take small steps down the hill. But with a learning rate that is too high it is possible that you take to large steps and kind of jump over the valleys.

In practice it is very common to use stochastic gradient descent (SGD). When using SGD, the gradients are not calculated for the whole training set before updating the weights. Instead the average gradient is calculated for a small "mini-batch"(a sample) of the input data. The weights are then updated based on this average gradient, which serves as a noisy estimate of the average gradient over the full training set. When the parameters have been updated based on a mini-batch, an iteration has been completed. A round (or epoch) is completed when all the training inputs have been used for the SGD. So the number of rounds indicates how often the whole training set has been fed to a network [Nielsen2015] [LeCun2015].

1.2.4 The Backpropagation Algorithm

Backpropagation is an algorithm that can be used to compute the gradients of the objective function with respect to the weights and biases. The backpropagation algorithm is composed of the propagation forward through the network, the computation of the network error and then afterwards, the backwards propagation.

Propagation forward through the network

At first the input that is fed to the network is propagated forward through the network, layer for layer. The input z to each layer l is computed by:

$$z_j^l = \sum_k w_{jk}^l \cdot a_k^{l-1} + b_j^l \tag{1.9}$$

where a_k^{l-1} is the activation of the prior layer, w_{jk}^l are the weights for the current layer and b_j^l is the bias of the neuron. The index j stands for the j-th neuron in the current layer and k for the k-th neuron in the previous layer (s. figure 1.7).

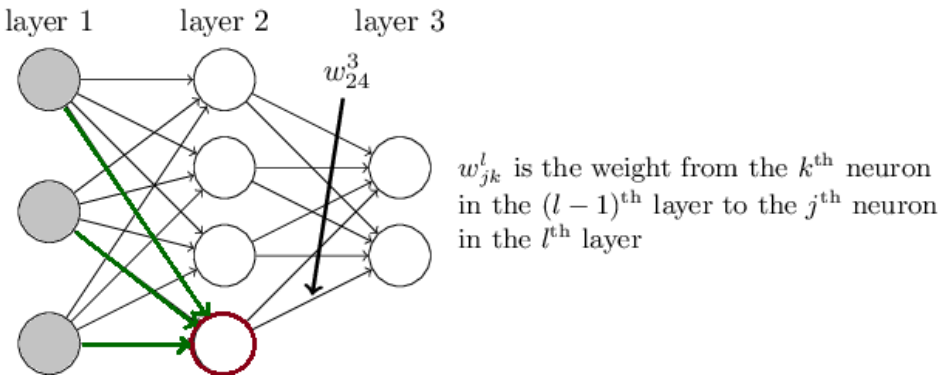


Fig. 1.7: Forward propagation, source: adapted from Nielsen 2015

As an example, for the computation of the input of the red coloured neuron (layer 2) you would multiplicate the green weights with the respective grey ac-

tivations, take the sum and then add the neuron-specific bias to it (the figure does not show the bias).

The output activation for a neuron is computed by applying the activation function to the input. Consequently, for the output activations of a layer we apply this function to all the inputs to the layer:

$$a^l = \sigma(w^l \cdot a^{l-1} + b^l) \tag{1.10}$$

where w is the weight vector of layer l, a^{l-1} is a vector containing the activations of the prior layer and b is the bias vector for the layer. The activation function is represented by the sigma.
The initial weights for a network that is trained from scratch are randomly chosen. For the first layer the activations are the input vector composed of the pixel values of the training images. For later layers the activations are the output of the preceding layer.

Computation of the network error

When finally the output of the last layer has been computed, it is compared with the desired output using the cost function. Thereby the cost function quantifies the error. This error is now used as the starting value for the backpropagation back through the network.

Propagation backwards through the network

With the propagation through the network we got error values that tell us how far off our network is from the desired output. But our interest lies in knowing for each weight how much a change can affect the total error.

For this purpose partial derivatives $\frac{\delta C}{\delta w}$ and $\frac{\delta C}{\delta b}$ are computed that quantify for each weight and bias how much the error increases or decreases if the re-

spective weight slightly changes.

In order to explain the reasoning behind the backpropagation algorithm, we look at the last layer of the network in figure 1.8.

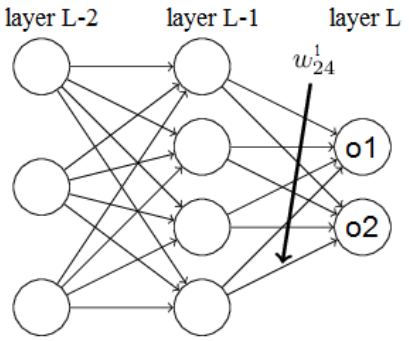


Fig. 1.8: The last few layers of a CNN; o1 and o2 are the output neurons of the last layer, source: adapted from Nielsen 2015

During the propagation forwards through the network, we calculated the input to the last layer L and then applied the activation function σ to compute the output to the layer. This output was then used to calculate the error using the cost function C.
We can understand this propagation as a nested function $E(\sigma(Z(w_x)))$ where Z is the function used for the calculation of the input. Therefore we can apply the chain rule and calculate $\frac{\delta C}{\delta w_2^L 4}$ by:

$$\frac{\delta C}{\delta w_{24}^L} = \frac{\delta C}{\delta out_2^L} \cdot \frac{\delta out_2^L}{\delta in_2^L} \cdot \frac{\delta in^L}{\delta w_{24}^L} \tag{1.11}$$

where in_2^L is the input and out_2^L is the output of the second neuron (o2) in the last layer.
When we want to calculate this derivative for weights earlier in the network, naturally one has also to take into account that each neuron alters the output of many output neurons. But for a general understanding this should be sufficient.

Further, the error δ_j^l of neuron j in layer l is defined as:

$$\delta_j^l \equiv \frac{\delta C}{\delta z_j^l} \tag{1.12}$$

Therefore for the error for the output neuron o2, we would calculate:

$$\delta_{o2} = \frac{\delta C}{\delta out_2^L} \cdot \frac{\delta out_2^L}{\delta in_2^L} \tag{1.13}$$

Now when we compare equation 1.11 and 1.13, we can see that:

$$\frac{\delta C}{\delta w_{24}^L} = \delta_{o2} \cdot out_4^{L-1} \tag{1.14}$$

because $\frac{\delta in^L}{\delta w_{24}^L} = a_4^{L-1} = out_4^{L-1}$.

Equation 2.15 also holds true in general:

$$\frac{\delta C}{\delta w_{jk}^l} = \delta_j^l \cdot out_k^{l-1} \tag{1.15}$$

In equation 1.15, you can nicely see that a small activation in a neuron (the output of a neuron equivalents its activation) leads to small gradients for the respective weights [Mazur2015] [Nielsen2015].

1.2.5 The Vanishing Gradient Problem

A problem with the gradient descent algorithm in combination with backpropagation are vanishing gradients.

The name vanishing gradients describes the phenomenon that during backpropagation, the gradients are decreasing, causing small gradients for early layers. The result is that the neurons in these layers are learning more slowly than the ones in later layers of the network.

The vanishing gradient problem is very common with the sigmoid function, so we'll take it as an example here. The backpropagation algorithm uses derivatives to find the proper weights for the network. The derivative of the sigmoid function takes a value between (0, 1/4] (s. figure 1.9). The more layers a net-

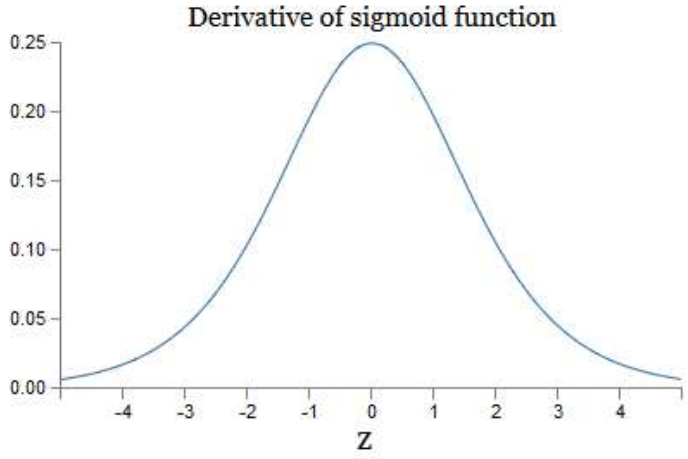


Fig. 1.9: The derivative of the sigmoid function, source: Nielsen 2015

work has, the more often this derivative is multiplied during the computation of the gradient $\frac{\delta C}{\delta w}$ (because of the chain rule).

High weights can somewhat lessen the vanishing of the gradients, but only to a small extent. When the number of layers is high, at some point even large weights ultimately will fail to make up for the many multiplications of the derivatives across the layers.

The problem can be addressed by using the ReLU function, because the gradient is zero for inputs that are ≤ 0, and 1 for positive input values.

A similar problem that can occur are exploding gradients, which is the opposite of the vanishing gradient problem. Too large gradients in earlier layers can be caused by high weight values. Exploding gradients are less of an issue for feedforward neural networks than for recurrent DNNs, but they can happen if the learning rate is set too high [Nielsen2015].

1.2.6 Batch Normalization

Batch Normalization (BN) is a method that aims at accelerating the training of deep networks by reducing internal covariate shift.

Internal covariate shift describes the phenomenon that the distributions of the inputs to the layers continuously change during training. When the network adapts the weights and biases for a layer, this changes the input for the next

layer, and so forth.

This property of neural networks is also the reason why a small change in an early layer can result in a large change in later layers.

Batch normalization tries to diminish internal covariate shift with a normalization step. This step ensures that each layer input has zero mean and variance one.

The normalization equation for the input x of layer l can be depicted as:

$$\hat{x}^{(l)} = \frac{x^{(l)} - E[x^{(l)}]}{\sqrt{Var[x^{(l)}]}} \tag{1.16}$$

where the mean $E[x^{(l)}]$ and the variance $Var[x^{(l)}]$ are computed from the mini-batch. The normalized value $\hat{x}$ is then scaled and shifted by two learnable parameters γ and β:

$$y^{(l)} = \gamma^{(l)}\hat{x}^{(l)} + \beta^{(l)} \tag{1.17}$$

With Batch Normalization, the gradients are less sensitive to the scale of the parameters and their initialization. It is possible to use higher learning rates and still reach convergence. It also helps with vanishing and exploding gradients and escaping poor local minimas. Batch Normalization even has a regularization effect [Ioffe2015].

2 Training Deep Convolutional Neural Networks

2.1 Training, Test and Validation set

The data is usually split into a training, validation and test data set.
The training data set is the data that the network is being trained on. The network will change weights and biases to best fit this training data set. During training, the network will regularly output the training accuracy. This is the classification accuracy[1] you obtain if you would apply the model on the training data set.
This training data set is fed to the network for a certain number of rounds. In general the training accuracy will tend to grow with each round until it stagnates at some value.
But our actual aim is not to have a really good accuracy on our training set but for our network to have the best possible accuracy on the test set (test accuracy).

The test accuracy is a measure of the performance of a network. The test set consists of images that have neither been used for the training nor for the validation of the network. Therefore it indicates how well the network can generalize to unknown data. The ultimate aim is to receive a network with a high ability to generalize over data.
But the model with the best training accuracy is not consequently the one with

[1] The classification accuracy is easily computed by just dividing the number of incorrectly classified pictures by the total number of pictures.

the best test accuracy. At some point in the training the network can adjust too much to the training data set. This is called overfitting. To prevent it from happening, one can provide the network with a validation data set during the training.

The validation data set can be used for the model selection. It enables to check the capability of the network to generalize to new image data with every round. Thus a high validation accuracy is more important than a high training accuracy [Witten2000] [Buduma2017] [Goodfellow2016].

2.2 Preprocessing the Data

Before being fed to the CNN, the images are usually preprocessed.
The most common form of preprocession is the subtraction of the mean. This means subtracting the mean image from each image. This can be done by calculating the arithmetic mean of each pixel and subtracting this mean value from the respective pixels. For example, the mean across the first pixels of every picture in the training data set is subtracted from each first pixel. Subtracting the mean centers the input to zero.
The mean image of the training data set is also used for preprocessing the validation and test data set. This means that the mean image of the training data set is subtracted from the validation and test images, not the mean image of the validation or test data.
Another form of preprocessing is normalization. In addition to subtracting the mean image and thereby centering the images to zero, you also divide each pixel by the standard deviation. For image input data this division is not absolutely necessary, because pixel scales are usually roughly equal from the start [Karpathy2017].

2.3 Initialization Strategies

The initialization of the weights is pretty important for the training of the network. Decent initial weights can make the network converge faster, bad initialization can even cause the network not to converge at all.
A really bad idea would be to initialize the weight to be all zero, because when all the weights take the same value, every neuron will compute the same output and consequently also the same gradients later during backpropagation. This leads to the same parameter updates for all the weights. Therefore some sort of asymmetry is key for the initialization of the weights.
If the initial weights in a network are too small, then the signal shrinks as it passes through each layer and the network learns really slowly. But the weights should also not be too large, because this can cause large gradients which can prevent the network from converging.
A pretty basic strategy for initialization is to simply draw the weights from gaussian or uniform distributions using small values, e.g. N(0, 0.01) or U(-0.07, 0.07). Because a good initialization is key for fast learning and the convergence of very deep models, scientists have been working out somewhat more sophisticated ways of initializing weights.

We will discuss the Xavier Initializer, and the MSRA[2] PReLU. Both of these initializers use the fan-in and the fan-out as parameters.
The fan-in of a neuron is the number of inputs to a neuron. The fan-in matters insofar that the more inputs a neuron has, the more variance will its output have. So the fan-in is a reasonable choice to scale the weights of a neuron by.
The fan-out is the number of outputs of a neuron. It is used to scale the variance during backpropagation. When the fan-in is the number of input units of the current layer, the fan-out can also be described as the number of input units to the following layer. This is why in the following the fan-in is denoted by n_l and the fan-out by n_{l+1}, where l stands for the regarded layer.

[2]Microsoft Research Asia

The Xavier initializer is named after Xavier Glorot and was proposed in the paper "Understanding the difficulty of training deep feedforward neural networks"[Glorot2010]. The aim was to keep the variance of the gradients the same through the layers and during backpropagation. This leads us to two conditions:

$$\forall l, n_l Var(w_j) = 1 \text{ and} \tag{2.1}$$

$$\forall l, n_{l+1} Var(w_j) = 1 \tag{2.2}$$

These conditions can not always be met at the same time so the compromise would be to choose:

$$Var(W_i) = \frac{2}{n_l + n_{l+1}} \tag{2.3}$$

as the variance of the weights. It is the mean between the fan-in and the fan-out.

Still a normalization factor has to be introduced because else the variance of the back-propagated gradient is decreasing with each layer. The normalization factor is 3. After multiplicating this factor with equation 2.3 and taking the square root to get the standard deviation we get the Xavier initializer:

$$W \sim U[-\frac{\sqrt{6}}{\sqrt{n_l + n_{l+1}}}, \frac{\sqrt{6}}{\sqrt{n_l + n_{l+1}}}] \tag{2.4}$$

The weight values can also be sampled from a gaussian distribution:

$$N(0, \frac{2}{n_l + n_{l+1}}) \tag{2.5}$$

It is also possible to just choose to adjust for the fan-in. Then you solely make sure the variance is the same when you are forwarding through the network (but not during backpropagation). The variance would be $Var(W_i) = \frac{1}{n_j}$ in that case.

The Xavier initialization was invented for linear activation functions like tanh and sigmoid.

Since then the ReLU activation function has come up and proved to be a preferable option. But the ReLU activation function is not a linear function. Still the Xavier initializer inspired an initialization method especially designed for ReLU/PReLU activations.
It is called MSRA PReLU and was proposed by He et al. [He2015b].
The adapted condition that fits also to ReLU and PReLU activations can be formulated as:

$$\frac{1}{2}(1+a^2)\, n_l Var(w_j) = 1, \forall l \tag{2.6}$$

where a stands for the small negative slope from the PReLU activation function. The PReLU function is depicted as:

$$f(x) = \begin{cases} ax, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{2.7}$$

If you set the parameter a to zero in equation 2.6, you get the equation for the ReLU activation function (this holds also true for equation 2.7). Setting a=1 yields the equation for the linear case, s. equation 2.1.
He et al. also showed that it is not necessary to fulfil both conditions in equation 2.1 and 2.2. A decent scaling of either the backward or the forward signal suffices to also fit the other criteria.
The bottom line is, for ReLU they advise using a gaussian distribution with zero-mean and a standard deviation of $\sqrt{\frac{2}{n_j}}$ [He2015b].

2.4 Gradient Descent Optimization Algorithms

Optimization in deep learning is somewhat different to pure optimization, because it is an indirect optimization. In particular, we optimize the loss function though we are actually interested in optimizing f.e. the validation accuracy

[Nielsen2015].

Next to a pure SGD optimization there is also the possibility to use SGD with Momentum, Adagrad, RMSProp, Adadelta or the Adam optimization algorithm. With exception of the Momentum, these algorithms provide adaptive learning rates.

It is very important that the learning rate is appropriately chosen. It is usually sensible to choose a higher learning rate at the beginning of the training than at later iterations. With adaptive learning rates it is no longer necessary to manually adapt the learning rate during training, and the learning rates are set individually for each parameter [Ruder2016].

But at first, the momentum will be explained.

2.4.1 Momentum

Momentum is a method whose aim is to accelerate SGD learning by choosing the step size depending on the previous gradients' information.

When the previous gradients were consistently pointing into the same direction, the steps are chosen bigger, thus increasing the velocity. On the other hand, when the gradients' signs frequently change along a dimension, the steps are chosen smaller. The Momentum is especially advantageous in long narrow valleys (figure 2.1).

A parameter update with Momentum is defined as:

$$\triangle \theta_t = \rho \triangle \theta_{t-1} - \eta g_t \tag{2.8}$$

where ρ is the Momentum parameter, a constant that induces a slow decay of the previous parameter updates $\triangle\theta_{t-1}$. The second term $-\eta g_t$ is the usual SGD parameter update with learning rate η and the gradient at time step t g_t.

The momentum hyperparameter is often chosen as 0.5, 0.9 or 0.99. A higher momentum value means to allow more influence of the previous gradients on the current weight update. The momentum value can be raised with the pro-

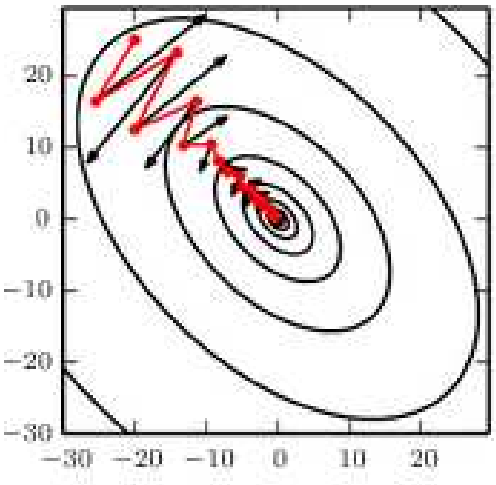


Fig. 2.1: SGD with Momentum, the red lines show the effect of Momentum, opposed to pure SGD optimization (black), source: Goodfellow et al. 2016

gression of training [Zeiler2012] [Goodfellow2016].

2.4.2 Adagrad

The Adagrad method was introduced by Duchi and Hazan [2011].

When using an Adagrad algorithm, a different learning rate is set for each parameter at every step of the neural net. For more frequent parameters, the learning rate is set lower than for those which are less frequent. The idea behind this is that the less frequent parameters are changed less often and the higher learning rate would balance this.

For a regular SGD optimizer the update of the parameter θ at time step t is:

$$\theta_{t+1} = \theta_t - \eta \cdot g_t \tag{2.9}$$

where the learning rate η is the same for every parameter.

But with Adagrad we have a different learning rate for every parameter.

There is an initial vector θ with entries for each parameter. This initial vector is adapted by:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{2.10}$$

with η being the learning rate and G_t containing the accumulation of the past squared gradients with respect to all parameters θ; the current gradient is denoted by g_t and ϵ is a small value introduced to avoid divisions through zero. At the start, G_t is zero, because there is not any history of previous gradients. With progress in training this value is steadily increasing, for this value encompasses the sum of squares of all the past gradients. This causes the denominator $\sqrt{G_t + \epsilon}$ to steadily increase with each iteration, therefore the new value for θ is decreasing with each weight update.

With the use of Adagrad it is no longer required to adapt the learning rate manually. Still a global learning rate needs to be set. But the downside of the Adagrad algorithm is that it, with progressing training, reduces the learning rate more and more, because all past gradients are used and the sum of squared gradients is therefore growing steadily. At some point the network then is incapable of further learning [Duchi2011] [Ruder2016].

2.4.3 RMSProp

The RMSProp algorithm is an adaption of the Adagrad algorithm that aspires to solve the problem of the ever-decreasing learning rate.

Instead of accumulating the previous squared gradients, the method uses the exponentially decaying average of the squared gradients (s. equation 2.11). With the decay, they achieve to diminish the influence of less recent gradients [Ruder2016].

$$E(g^2)_t = \rho E(g^2)_{t-1} + (1 - \rho) g_t^2 \tag{2.11}$$

ρ is the decay rate.

The formula for the parameter updates is defined as:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E(g^2) + \epsilon}} \odot g_t \tag{2.12}$$

2.4.4 Adadelta

Adadelta is also an adjusted version of the Adagrad optimizer. Next to solving the problem with the decreasing learning rate, this method as well obviates the need to set a global learning rate. It was presented by Matthew Zeiler [Zeiler2012].

Similar to the RMSProp the Adadelta algorithm uses the exponentially decaying average of the squared gradients, $E(g^2)_t$ (equation 2.11). But it also uses the exponentially decaying average of the squared parameter updates:

$$E(\triangle\theta^2)_t = \gamma E(\triangle\theta^2)_{t-1} + (1-\gamma) \triangle \theta_t^2 \tag{2.13}$$

Since $E(\triangle\theta^2)_t$ is unknown, it is approximated by $E(\triangle\theta^2)_{t-1}$, making the Adadelta update rule:

$$\theta_{t+1} = \theta_t - \frac{\sqrt{E(\triangle\theta^2)_{t-1} + \epsilon}}{\sqrt{E(g^2)_t + \epsilon}} \tag{2.14}$$

2.4.5 Adam

The Adam optimizer (Adaptive Moment Estimation) made its first appearance in a paper submitted by Kingma and Ba [Kingma2015]. It uses the exponentially decaying average of the previous gradients m_t and the exponentially decaying average of the previous squared gradients v_t:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{2.15}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{2.16}$$

The β values determine the decay rate.

The computed values for m_t and v_t serve as estimates for the first (the mean) and the second moment (the uncentered variance) of the gradients. Since m_t and v_t are initialized with zeros, they show a bias towards zero, particularly at the beginning of training.

To compensate for this bias, they introduce bias-corrected values $\hat{m}_t$ and $\hat{v}_t$:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.17}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.18}$$

These are used for the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{2.19}$$

As default values the authors recommend using 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ [Ruder2016].

2.5 Batch Size

The batch size determines with how many images the average gradient for the SGD is calculated. With a higher batch size the training is faster, as long as you have the computational capacity for the parallel computation of all these inputs. A higher batch size gives a better estimate of the gradient of the training set. But with small batch sizes one can add some noise to the network, which can be helpful to escape local minima [Goodfellow2016].

2.6 Overfitting and Regularization

Overfitting means that the network is adjusting too much to the training data set. At some point the network does not learn features anymore that are generally distinctive for the classes, but that are specific for the training data set. Such overfitting of the network is a true hazard, because it hurts the ability of the network to generalize to another data set.

The problem often arises when there is only a small amount of training samples compared with a very deep network which has a lot of parameters. Thus increasing the amount of training data or reducing the size of the network are two possible ways of reducing overfitting in such cases.

But there are further approaches that can help to prevent overfitting without having to resort to one of these. They are called regularization strategies.
Regularization strategies strive for the improvement of the performance of a network on new data like the test data set, even if this means increased training errors. Four common strategies are early stopping, data set augmentation, L1/L2 regularization and dropout [Goodfellow2016] [Nielsen2015].

2.6.1 Early Stopping

Stopping the training early when the validation accuracy does not rise anymore, even if the train accuracy is still increasing, can prevent overfitting. It is advisable though to still watch the validation accuracy for several rounds before stopping the training, because the validation accuracy may not increase for a couple of rounds but then rise again [Goodfellow2016].

2.6.2 Data Set Augmentation

Another strategy is the augmentation of the data set. If the training data is scarce, one can expand their training data set by adding some altered images. This means for example mirroring or rotating some of the input images, or changing the size. The aim is to artificially create varieties that are found in reality but may not be represented by the data set [Goodfellow2016].

2.6.3 L1 and L2 Regularization

L1 and L2 regularization target overfitting by penalizing high weight values, which are typical for overfitted models.
When applying L1 or L2 regularization, an extra term is added to the cost function, which is called regularization term. This regularization term includes the weight values, and therefore the weight values are included in the calculation

of the cost function. This means smaller weight values will lead to smaller error values et vice versa.
The regularization term for the L1 regularization is defined as:

$$\frac{\lambda}{2n} \sum_{w} |w| \tag{2.20}$$

It is composed of the absolute values of all the network weights, and a scaling factor $\frac{\lambda}{2n}$. λ is known as the regularization parameter ($\lambda > 0$) and n is the size of the training set.
This regularization strategy can be interpreted as a compromise between small weights and the minimization of the cost function. A small λ means to favor the minimization of the cost function, and a large λ means to favor finding small weights.
The regularization term only includes the weights, not the biases.
The L2 regularization is also called weight decay. It is very similar to the L1, but it uses the sum of squares of the network weights:

$$\frac{\lambda}{2n} \sum_{w} w^2 \tag{2.21}$$

The difference between L1 and L2 regularization is that the L1 shrinks the weights by a constant amount whereas with L2 regularization the weights shrink by an amount proportional to w. The benefit of the L1 regularization is that it pushes weights for obsolete features to zero, performing a feature selection that makes the training less vulnerable to noisy inputs. But in practice, L2 regularization usually promises better performance [McCaffrey2015] [Buduma2017] [Goodfellow2016].

2.6.4 Dropout

Dropout is also a regularization strategy. The general idea for dropout is deduced partly from genes, because genes try to improve the fitness of an individual as independently of other genes as possible. Genetics are doing a good job ma-

king genes robust to any combination with other genes by mixing them up at random with each new generation. Similar to that, with dropout one can add noise to a network to make it more robust. In a network with dropout, a specified amount of neurons will die at random at each layer which has dropout added to it [Srivastava2014].

3 DCNNs: The VGG, Inception and ResNet Architecture

3.1 The VGG Network

The VGG is a deep neural network Karen Simonyan and Andrew Zisserman introduced in their paper "Very deep convolutional networks for large-scale image recognition"[Simonyan2015]. They wanted to investigate the impact of the network depth on the accuracy on large-scale image datasets and found that increasing the depth of a network would significantly enhance the accuracy. They used more, but very small (3x3) convolution filters.
So why does the network use stacks of 3x3 convolutional layers? The argumentation is that a stack of two 3x3 convolutional layers would have an effective receptive field of 5x5, and a stack of three 3x3 convolutional layers would have a receptive field of 7x7, because there is no pooling layer in between (s. figure 3.1). This reduces the number of parameters. A 7x7 convolutional layer f.e. results into $7^2 \cdot C^2 = 49f^2$ parameters while a stack of three 3x3 convolutional layers produces $3(3^2 \cdot C^2) = 27f^2$ parameters (f represents the number of filters). Also, three ReLU activations can be used instead of just one when using three 3x3 convolutional layers, adding non-linearity to the network.

The network architecture of the deepest VGG of the paper, VGG-19 is visualized in figure 3.2.
The input data is fed to two 3x3 convolutional layers. These first two convolu-

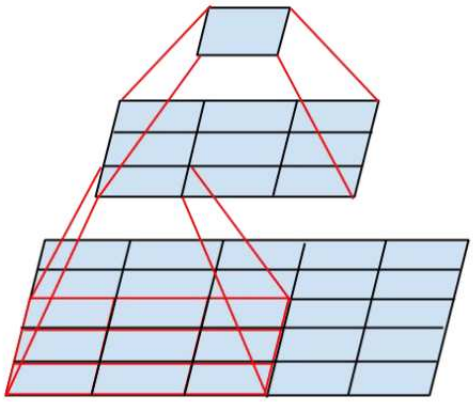


Fig. 3.1: Stacks of 3x3 convolutional layers, source: Szegedy et al. 2015b

tional layers consist of 64 filters[1]. Subsequently there is a max-pooling layer. After this pooling operation there is another stack of two 3x3 convolutional layers, this time 128 filters deep, again followed by a max-pooling layer. Hereafter come three stacks which each consist of four 3x3 convolutional layers and end with a max-pooling operation. The first stack is composed of convolutional layers with each 256 filters, the second and third stack of convolutional layers with each 512 filters.

After the stacks of convolutional layers there are three fully connected layers. The first and second layer both have a filter size of 4096, and additionally they are also equipped with dropout (dropout ratio 0.5). For the last fully connected layer the number of filters is the number of classes (1000 in figure 3.2).

The spatial padding of the convolutional layers is set to one pixel because the size of the convolutional layers is 3x3. Each hidden layer is followed by a ReLU activation function, and the max-pooling layers used for spacial pooling are each of size 2x2, with stride 2.

The spatial size of the input gets smaller throughout the network because of the convolutional and pooling layers, but the number of filters is increased, increasing the depth of the network.

The VGG is basically a very deep network while keeping convolutional and

[1]The number of filters is called channels in the VGG paper. The term channel is also used for the colourization, with 1 channel being greyscale and 3 channels being a RGB picture. So different to the paper it is called number of filters here.

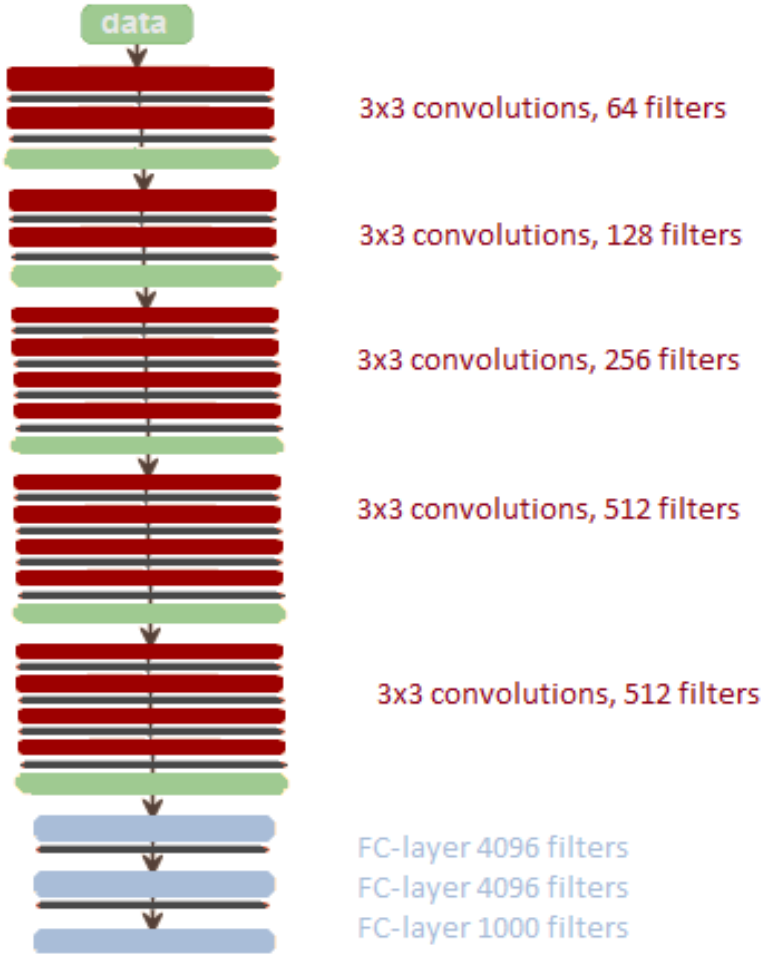


Fig. 3.2: The VGG-19 architecture; the convolutional layers are coloured red, the ReLU-activations are tinted grey, the pooling layers green and the fully-connected layers blue; source: own illustration

pooling layers small. But still, a lot of parameters need to be computed: the parameters of the VGG-19 add up to 144 million parameters.

As input Simonyan and Zisserman used 224x224 RGB images. For preprocessing, they subtracted the mean RGB value from each pixel. The authors achieved a 7.0 percent top-5 error rate with one VGG-19 network and an error rate of 6.8 with two combined networks on the ILSVRC-2012 data [Simonyan2015].

3.2 Inception Network: Inception-v3

While increasing the depth of a network by simply adding layers and using high filter numbers is a pretty forthright way of trying to increase the performance of a network, the downside is a larger number of parameters and an increased risk of overfitting, as well as a higher computational cost.
With the introduction of the Inception Module, Szegedy et al. had a somewhat different approach. Instead of a throughoutly sequential structure, part of the network works in parallel. The Inception Module was introduced alongside the GoogLeNet architecture by Szegedy et al. in their paper "Going deeper with convolutions"[Szegedy2015a].
The authors have put much thought into the computational part of deep learning. The basic idea is to combine advantages of both sparsity and dense computation.
The sparsity is connected to the idea of analyzing the correlations between units and to cluster units with high correlation into groups. Sparse data structures[2] can "break the symmetry and improve learning"[Szegedy et al. 2015, p.3], they are however prone to inefficient computing. Full connections on the other hand allow more efficient dense computation.
These considerations lead to the invention of the Inception Module.

The Inception Module

A graphic representation of the Inception Module can be found in figure 3.3.
Instead of having either one convolutional layer or a pooling layer, convolutional and pooling layers are working parallely. Since this would lead to too many outputs, by adding 1x1 convolutional layers before the 3x3 and 5x5 layers the authors achieved a dimensionality reduction regarding the feature maps (not spatially!). These 1x1 layers are also followed by a ReLU unit.
The optimal layered network topology can be found out by using correlation statistics. Neurons are clustered according to the correlation statistics of pre-

[2]sparse matrices are matrices with most of its elements being zero. A dense matrix is a matrix where most of the elements are not zero

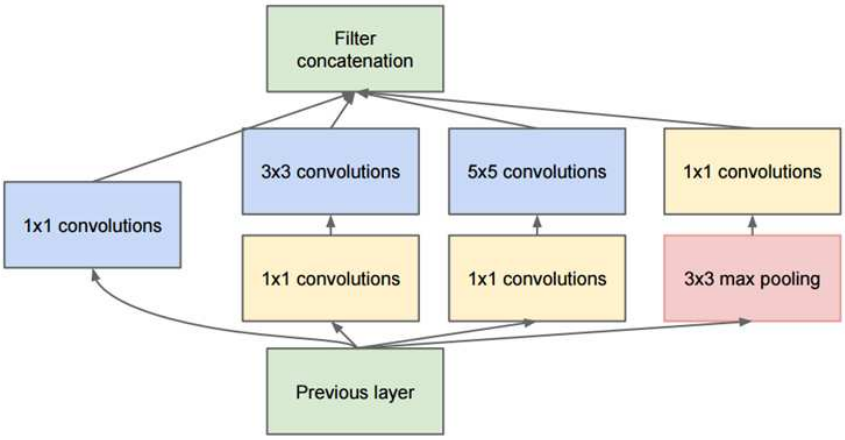


Fig. 3.3: Inception Module, source: Szegedy et al. 2015a

ceding layer activations. Neurons with highly correlated outputs are clustered. 1x1 convolutions cover very local clusters, whereas 3x3 and 5x5 convolutions can cover more spread-out clusters. This is why in the early layers of the CNN smaller convolutions are used, whilst in the later layers, where the high-level reasoning takes place, 3x3 and 5x5 are more common. The results are concatenated, including pooling.

This original Inception Module was used for the GoogLeNet [Szegedy2015a]. In the paper "Rethinking the Inception Architecture for Computer Vision" the authors introduced two new networks, the Inception-v2 and v3 [Szegedy2015b]. The Inception-v2 is just a stripped down version of the v3, so the remarks will be limited to the v3 here.
The Inception-v3 network adds some new ideas to the original GoogLeNet. A main element is the factorization of convolutions with large filter size.
For one thing, the authors factorized large convolutions into smaller convolutions. They replaced the 5x5 convolutional kernels of the Inception module with two 3x3 kernels to reduce the parameters, like in the VGG-network (figure 3.4).

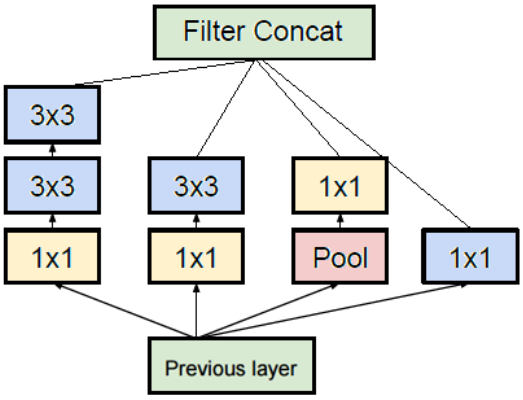


Fig. 3.4: Inception Module: factorization into smaller convolutions. This Inception module has two 3x3 convolutions instead of a 5x5 convolution (compared to the original Inception module). Source: adapted from Szegedy et al. 2015b

The authors also experimented with other types of spatial factorization. They found that a factorization of a nxn layer in a nx1 followed by a 1xn layer can also have a remarkable beneficial effect on the computational cost (s. figure 3.5).

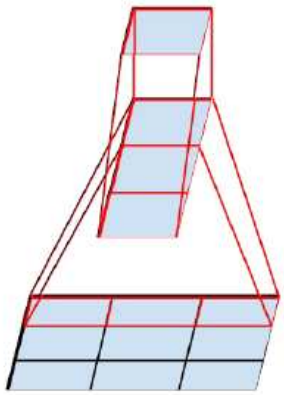


Fig. 3.5: Visualisation of the effect of successive asymmetric convolutions: 3x1 and 1x3, source: Szegedy et al. 2015b

So they exchanged each of the 3x3 layers in the Inception Module in figure 3.4, with a 3x1 and a 1x3 layer. The resulting Inception Module can be viewed in figure 3.6. But these asymmetric convolutions did not work too well in early layers, so they proceeded to use them on medium grid-sizes.

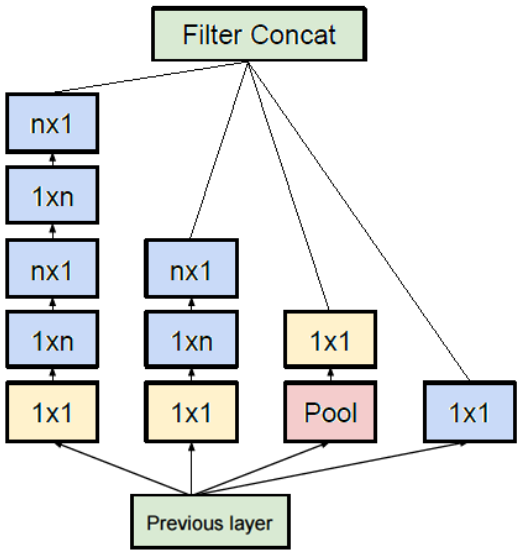


Fig. 3.6: Inception Module: factorization into asymmetric convolutions, source: adapted from Szegedy et al. 2015b

Another version of the Inception Module is one with expanded filter bank outputs (figure 3.7).

This Inception Module is used late in the network, when the feature map is already very small (8x8). The aim is to increase the number of filters. It is designed to obtain high-dimensional representations.

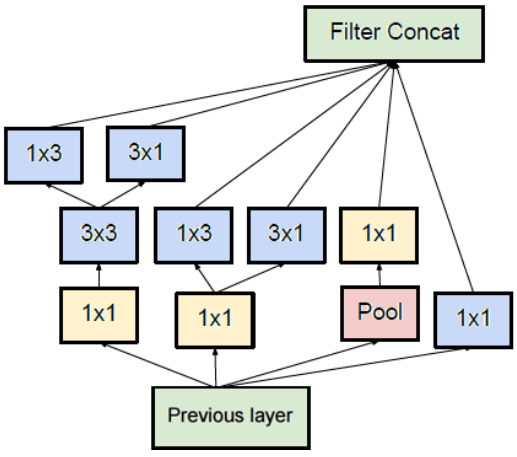


Fig. 3.7: Inception Module with expanded filter bank outputs, source: adapted from Szegedy et al. 2015b

All three of these Inception modules are used in the Inception network archi-

tecture, shown in table 3.1. The lower layers of the Inception network are kept traditionally, sequences of convolutional layers followed by max pooling (like in the VGG architecture).

type	patch size/stride or remarks	input size
conv	3x3/2	299x299x3
conv	3x3/1	149x149x32
conv padded	3x3/1	147x147x64
pool	3x3/2	147x147x64
conv	3x3/1	73x73x64
conv	3x3/2	71x71x80
conv	3x3/1	35x35x192
3x Inception	as in figure 3.4	35x35x288
5x Inception	as in figure 3.6	17x17x768
2x Inception	as in figure 3.7	8x8x1280
pool	8x8	8x8x2048
linear	logits	1x1x2048
softmax	classifier	1x1x1000

Table 3.1: The Inception network architecture, source: adapted from Szegedy 2015b

Additionally, Szegedy et al. equipped their Inception-v3 network with Batch Normalization, batch-normalized auxiliary classifiers and label smoothing regularization.

Auxiliary classifiers

The Inception networks are really deep. To nevertheless embrace the advantages of more shallow networks like larger gradient signals and regularization, auxiliary classifiers are added to intermediate layers (s. figure 3.8).

The loss outputs of these classifiers are weighted by a factor of 0.3 and then added to the total loss. The auxiliary classifier outputs are only used for training, not during inference to new data.

The auxiliary classifiers were already a part of the GoogLeNet architecture, but in the Inception-v3 they also added Batch Normalization to the auxiliary softmax functions.

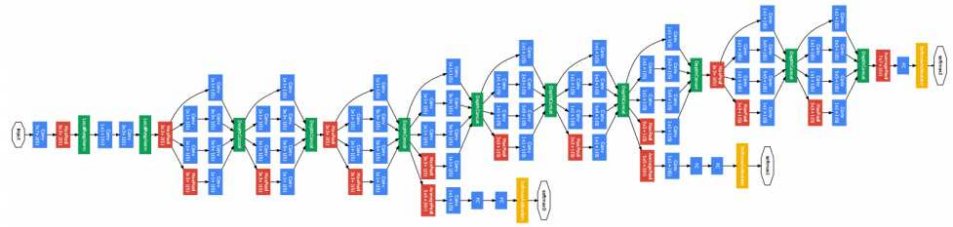


Fig. 3.8: Auxiliary classifiers as a part of the GoogLeNet architecture; the softmax layers are coloured yellow. (the convolutional layers are blue, pooling layers red, local response normalization and concat layers green, and softmax layers yellow); source: Szegedy et al. 2015a

Label smoothing regularization

Also, the Inception-v3 is using label smoothing regularization (LSR). When training a model with the cross-entropy, it is trying to maximize the log-likelihood of the true training labels. In other words, it tries to push the predicted probability of the true label closer to one. Szegedy et al. argue that this is not ideal when the aim is to obtain a model that is generalizable to new data. Still of course the predicted probability of the true label should be higher than the probabilities of the false labels, but the intention is to lessen the difference to make the model more adaptable.

The authors report a 4.2 top-5 error rate with a single model and 3.58 with a multi-model approach using the ILSVRC 2012 classification benchmark with the Inception-v3 architecture [Szegedy2015b].

3.3 Deep Residual Network: ResNet

The ResNet is a really deep neural network, presented by Microsoft Research Asia [He2015a]. They acknowledged the fact that vivid stacking of more and more convolutional layers does not continuously lead to better performance. At first the accuracy increases with the depth of the network, but at some point it even starts to degrade. So they invented a residual learning framework to tackle this degradation problem.

A residual network is made of building blocks, visualized in figure 3.9:

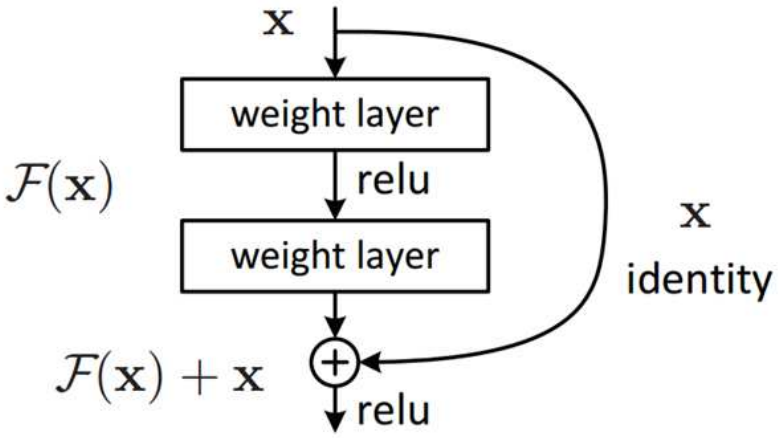


Fig. 3.9: ResNet building block, source: He et al. 2015a

The input x is fed to a stack of convolutional layers, as typical. But in addition to this, a shortcut is implemented. Shortcut connections are connections between layers that skip one layer or more. In residual networks, this shortcut connection is an identity function. Therefore the output of this shortcut is equal to its input.

The output of the identity function (x) is then added to F(x), which is the output of the weight layers: $F(x) + x$. When we define H(x) as the desired output, traditionally we would fit F(x):= H(x). But with the residual building block, we perform residual mapping F(x):= H(x) - x. This induces a reference to the layer inputs.

In a regular CNN, later layers are just fed abstract information broken down from the previous layers. In a ResNet the input information is preserved and fed over and over again into the numerous layers of the ResNet (s. figure 3.10).

With the use of residual building blocks, one can address the degradation

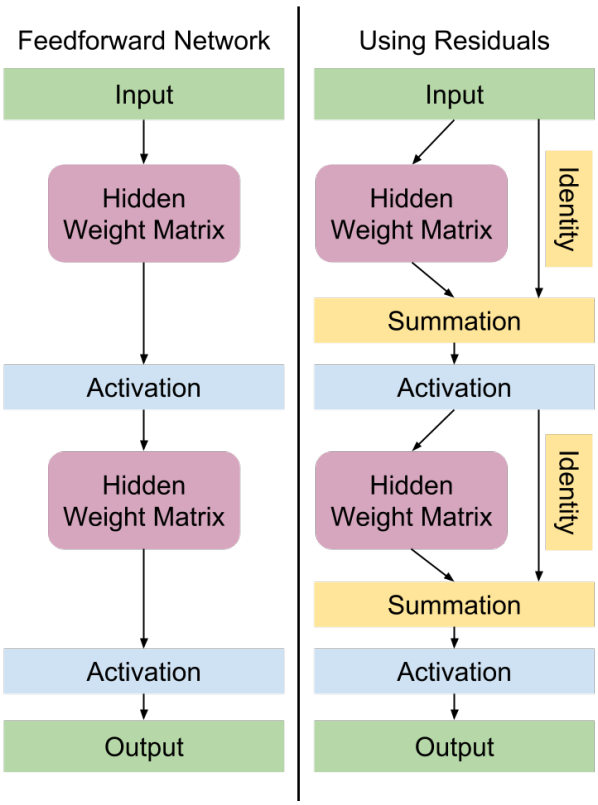


Fig. 3.10: Comparison between a regular CNN and ResNet, source: Bolte 2016

problem. He et al. (2015a) have tested a plain 18-layer and a 34-layer convolutional neural network against two residual networks with the same depths. The architectures can be viewed in figure 3.11.

Table 3.2 shows the Top-1 error for the two networks.

You can clearly see that the 34-layer plain CNN performs worse than the 18-

	plain	ResNet
18 layers	27.94	27.88
34 layers	28.54	25.03

Table 3.2: Top-1 error on the ImageNet validation set for the ResNet and the plain network architecture, source: He et al. 2015a

layer plain CNN. The ResNet architecture on the other hand achieved better results for the 34-layer network than for the 18-layer network, and altogether a better result than the plain network.

Beyond that, using the residual building blocks does not even cause any additional computational complexity or parameters.

The deepest ResNet presented in the paper has 152 layers. With the ResNet the authors won the ILSVRC 2015 with an error rate of 3.58 percent. The single-model results in the paper show a 4.49 top-5 error on the ImageNet validation set [He2015a].

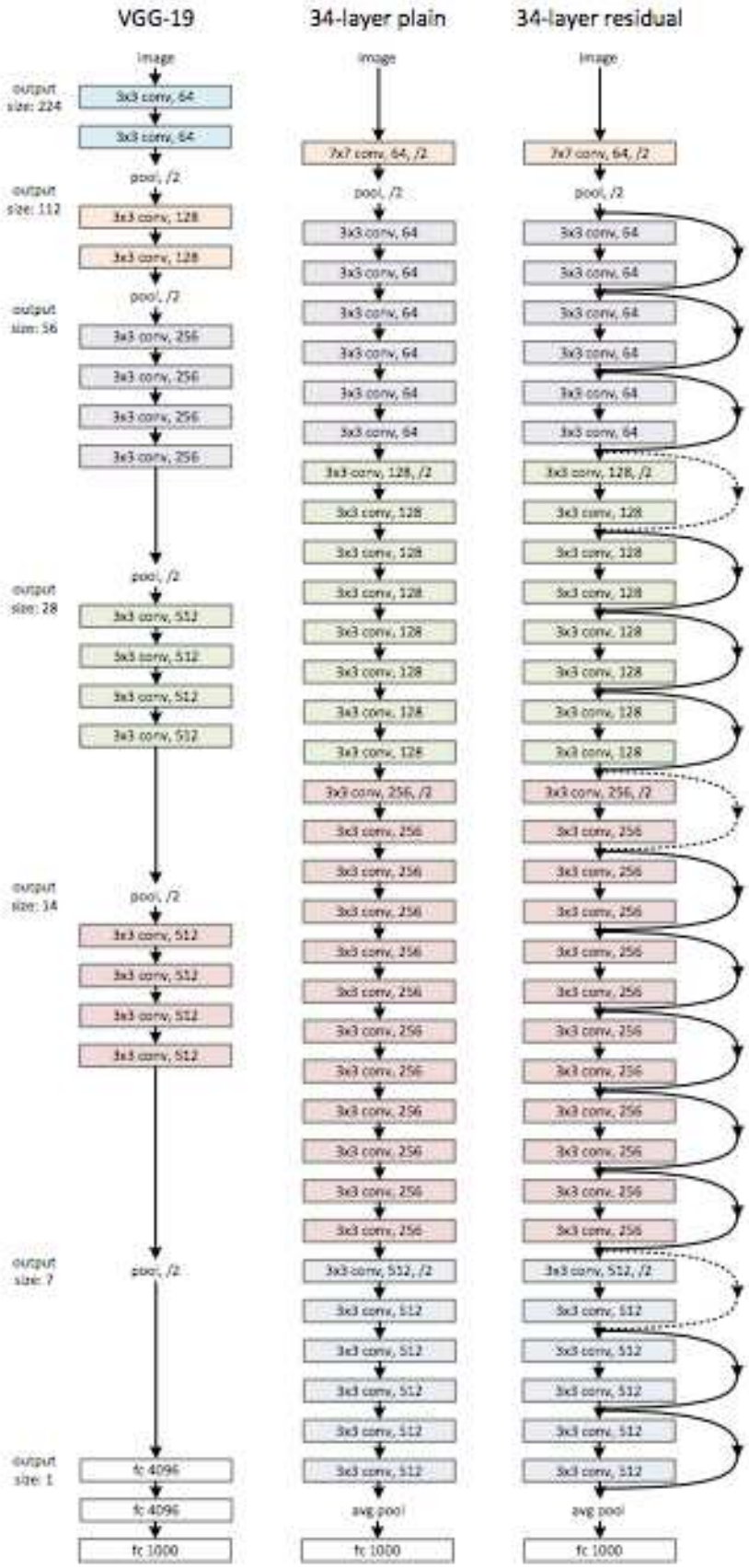


Fig. 3.11: ResNet vs. plain architecture, source: He et al. 2015a

4 Implementation in R Using the MXNet Deep Learning Library

4.1 The MXNet Deep Learning Library

When you delve into deep learning, there are a lot of deep learning libraries you can choose from. Apart from MXNet there are also for example the Theano, Torch, and Caffe library.
The choice of the right deep learning library depends on your needs and what programming language you would like to use. Since these libraries are continuously developing, the features they provide advance with time.
MXNet currently supports not only Python but also the programming languages R, Scala, Julia, C++, Matlab, Javascript and Perl. Also it is considered a very fast deep learning library which supports distributed computing. The open-source library MXNet was and is developed by the DMLC group [DMLC2016] [Chen2015a].

4.2 Parallelization and Hardware

MXNet was compiled with OpenBLAS [Xianyi2016] for parallelization in this project. Even faster learning can be achieved with the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN), but it did not work for Inception and ResNet architectures at the time. For the training a server with 2 Intel Xeon CPUs was used, each with 14 cores and hyperthreaded.

4.3 The Pretrained Models: VGG-19, Inception-v3 and ResNet-152

Training DCNNs is costly both in terms of time and computational resources. Luckily enough one does not always have to train his network from scratch. Several deep learning libraries have their own model zoos, where there are already pretrained models available.

Even if you have a dataset that is somewhat different from the dataset the pretrained network was trained on, using pretrained weights can save a lot of training time. In order to finetune a network that has been trained on other data, one has to remove the last layer of the network and replace it with a new one that has the number of classes of the new data as the number of hidden layers. In some way, using pretrained weights is a very smart form of initialization.

Finetuning shows a great improvement over training from scratch for scarce data [Hentschel2016] [Oquab2014].

For the following researches three pretrained models from the MXNet model zoo were used [DMLC2016]. All of them were trained on the ImageNet [Russakovsky2015] data set and achieve the same classification accuracies as in their respective papers. The networks are the VGG-19 network [Simonyan2015], the Inception-v3 network with Batch Normalization [Szegedy2015b] and the ResNet-152 [He2015a].

4.4 Finetuning the VGG, the Inception and the ResNet on Varying Amounts of Places2 Image Data

Training deep networks on a larger scale is not a sprint, it is a long run and can take up to weeks. But also for smaller data sizes the question is not only

how good a DCNN can classify images, but also how fast it can get that good. In addition, it requires a big enough data set to successfully train a network. Since labeled images in vast amounts are not always at hand, it is also reasonable to ask, how much data is really necessary.
Therefore the aim is to:
A) Compare the speed and the classification accuracy on the test set of the VGG-19, Inception-v3 and ResNet-152
B) Compare the classification accuracy between different data amounts (varying between 20 and 1000 pictures per class)
The basic idea is to finetune the three chosen DCNNs on data sets of varying sizes, and then to compare the test accuracy and the training times.
Since the pretrained models were trained using the ImageNet data set, different data is needed for the finetuning.

4.4.1 The Places2 Data Set

The Places2 Places365-Standart data set encompasses 1.6 million RGB train images from 365 scene categories [Zhou2016].
For the task the small (256x256) train images were downloaded and resized to a size of 224x224. The training was done only with a part of the data set. For the research three categories were chosen: cockpit, waterfall and staircase (s. figure 4.1).
It was at first planned to use the category sauna instead of waterfall, but the sauna image folder turned out to be infiltrated by a lot of images that showed something else (f.e. a church).
The categories themselves were chosen because they are pretty clearly defined.

(a) cockpit (b) waterfall

(c) staircase (d) staircase

Fig. 4.1: Images of the three classes: cockpit, waterfall and staircase, source: Zhou et al. 2016

4.4.2 Training Settings and Results for the VGG, Inception and ResNet on the Places Data

The three DCNNs VGG-19, Inception v-3 and ResNet-152 were trained on 1000, 500, 250, 100, 50 and 20 images per class. Since there are three classes (cockpit, waterfall and staircase) this means a network trained with 1000 images per class was trained on 3000 images total.

The training was conducted with the same sample of the training data for every network. As an example, for 20 images per class this means that the same 60 images were used to train the VGG, the ResNet and the Inception network. The evaluation and test set was exactly the same for every network and with any of the data sizes for comparability. The batch sizes were customized to the

respective size of the training data set (50, 50, 30, 30, 15 and 10).
For the validation and hence the model choice an evaluation set with 100 pictures per class was used. Especially with the lower training data sets (f.e. 20 or 50 images per class) it would even be unrealistic to have such a high amount of validation data. This is the reason why the time for the calculation of the validation accuracy is excluded in the training times.
The test data set encompasses 300 images per class, adding up to 900 images to test the networks on.
As a preprocessing step, the subtraction of the mean image was performed.

During the training of the Inception, the Adam optimization algorithm was used to set the learning rate. Initially the plan had been to train all three networks with this optimizer, but as it did not work very well with the ResNet and the VGG network, the learning rate was set manually for the training of these networks.
The learning rate value for VGG and ResNet was set to 0.001[1] and the momentum to 0.5. Later in training the learning rate was adjusted to 0.0007.

Results for the VGG-19 Network

At first the results for the VGG-19 network will be presented. They are summarized in table 4.1.
For both 1000 and 500 images per class, the VGG-19 network acquired a test accuracy of about 96 percent. The training with 1000 images lasted nearly 37 hours (11 rounds). With 500 images per class, the training took roughly 18 hours training time and 11 rounds as well. Training with 250 images per class resulted in a 93 percent test accuracy in nearly 16 hours.
With 100 and 50 images per class, the training accuracy obtained about 89 percent. The training times for 50 and 100 images per class are very similar

[1] If your network outputs NA values instead of training accuracies, it is likely that your learning rate is set too high. Even a value of 0.07 or 0.01 might already cause this to happen.

images per class	test accuracy	training time	rounds
1000	96.1	36.7h	11
500	95.8	17.7h	11
250	92.9	15.8h	19
100	89.3	4.3h	15
50	89	4.9h	34
20	83.3	1.1h	21

Table 4.1: The test accuracy, training times and number of rounds for the VGG-19 network, listed after the training images per class (3 classes); the times equate the training times until the best validation accuracy was achieved (the time needed for the computation of the evaluation accuracies is excluded)

(4.3 and 4.9 hours) while the training with 50 images per class took about twice as many rounds (34) as the training with 100 images.

For 20 images per class, the VGG network accomplished a test accuracy of 83 percent in a good hour training time.

The results for the VGG-19 network show that the test accuracy is clearly deteriorating with sinking data input. But one can also see that for the training with 1000 and 500 images per class there is barely a difference in the resulting test accuracy. This was also observed for 50 and 100 images per class.

The difference in the test accuracy of the VGG network trained with 1000 as opposed to 20 images per class amounts to 12.8 percent.

Results for the Inception-v3 Network

Varying amounts of data were also fed to the Inception-v3 network. Table 4.2 shows the results.

When feeding 1000 images per class to the Inception network, a test accuracy of almost 99 percent could be achieved. The training took nearly 20 hours. For 500 images per class, the network still got a test accuracy of 98 percent in 2.8 hours of training time. And training with 250 images per class for two hours yielded yet 97.8 percent test accuracy.

The Inception network trained with 100 images obtained a test accuracy result of 96.2 in 2.4 hours training time. With 50 images the test accuracy fell

images per class	test accuracy	training time	rounds
1000	98.8	19.8 h	18
500	98	2.8h	5
250	97.8	2.0h	7
100	96.2	2.4 h	23
50	94.9	27.2 m	7
20	90.7	24.0min	20

Table 4.2: The test accuracy, training times and number of rounds for the Inception-v3 network, listed after the training images per class (3 classes); the times equate the training times until the best validation accuracy was achieved (the time needed for the computation of the evaluation accuracies is excluded)

to about 95 percent, with about 27 minutes of training.

Cutting the images per class down to 20 images caused a perceptible drop in the test accuracy. Barely 91 percent of the test data set were classified correctly.

Comparably to the VGG, the Inception network also shows that the test accuracy drops continuously with smaller data sets. For 1000 images per class, the network only has an error rate of 1.2 percent. The result for 500 images per class, a 2 percent error rate, is also very promising. Even for 250 images per class the error rate does not exceed 2.2 percent.

Results for the ResNet-152

The outcome for the ResNet-152 architecture is depicted in table 4.3. Again, data ranging from 20 to 1000 pictures per class was fed to the network. For the 1000 images per class, the ResNet obtained 97.4 percent test accuracy in nearly 26 hours of training time. With 500 images per class, it could still achieve a test accuracy of nearly 97 percent (in approx. 19 hours' time). Training with 250 images per class for 16 hours accomplished a test accuracy of 96.4 percent. With 100 pictures per class, the training accuracy dropped to about 92 percent. For 50 images per class, a test accuracy value of 91 percent could be obtained in approximately 3 hours of training time. With 20 images per class, the test accuracy dropped to barely 87 percent.

images per class	test accuracy	training time	rounds
1000	97.4	25.7h	6
500	96.9	18.8h	9
250	96.4	16.2h	15
100	91.9	11.2h	30
50	91.2	2.8h	14
20	86.7	5.9h	80

Table 4.3: The test accuracy, training times and number of rounds for the ResNet-152, listed after the training images per class (3 classes); the times equate the training times until the best validation accuracy was achieved (the time needed for the computation of the evaluation accuracies is excluded)

Similar to the other two networks the results for the ResNet exhibit that the test accuracy values are constantly sinking with the reduction of the training data set.

The test accuracies for 1000, 500 and 250 images per class only have a small difference in their accuracies (0.5 percent) considering the quite huge difference in the amount of images involved in the training. On the other hand, there is a perceptible drop in the test accuracy when the data drops from 250 to 100 and from 50 to 20 images per class.

4.4.3 Summary of the Test Results

The test accuracies for the three networks, the VGG-19, ResNet-152 and Inception-v3 architecture, are summarized in table 4.4.

The Inception network accomplished the best results for the test accuracies for every data amount. The outcome accuracies for the VGG-19 network were generally noticeably smaller compared to the other networks. The ResNet yielded better results than the VGG network, but was inferior to the Inception network.

With choosing a well-architectured network, one can make up for a smaller dataset. For example, the Inception network achieved better results for 100 images per class than the VGG network did for 500 images. Also with 50 images per class the Inception network succeeded in having a higher test ac-

images per class	VGG	ResNet	Inception
1000	96.1	97.4	98.8
500	95.8	96.9	98
250	92.9	96.4	97.8
100	89.3	91.9	96.2
50	89	91.2	94.9
20	83.3	86.7	90.7

Table 4.4: A summary of the test accuracies for the VGG-19, ResNet-152 and Inception-v3. They are listed after the training images per class (3 classes)

curacy than both VGG and ResNet had for 100 images per class.

The drop in the test accuracy between 1000 and 20 images per class is also smaller for the Inception than for the other two networks. When reducing the data amount to 20 images the test accuracy for the Inception-v3 decreases by 8.1 percent. The ResNet suffers a loss of 10.7 percent, and the VGG loses 12.8 percent.

What is also revealed is that a drop in the training data hurts the test accuracies more if the data is already scarce. A drop from 50 to 20 images per class has a considerably higher impact than a drop from 1000 to 500 images per class. This suggests that the marginal benefit of additional image data is diminishing with increased data sets.

In table 4.5, the training times for a single round are depicted for 1000, 500 and 100 images per class.

images per class	VGG	ResNet	Inception
1000	3.3h	4.3h	1.1h
500	1.6h	2.1h	33.1min
100	17.1min	22.4min	6min

Table 4.5: Training times per round for the VGG-19, Resnet-152 and Inception-v3. They are listed for 1000, 500 and 100 images per class (3 classes)

The training times demonstrate that the Inception module was the fastest of the three. The ResNet needed more time to complete a round than the VGG,

which is not really surprising since it has a depth of 152 layers compared to the 19 layers of the VGG-19. But a faster completion of a round does not necessarily mean the total training time until the network converges is also smaller.

This is why it is also interesting to look at the total training times. The training times until convergence for the three networks are shown in table 4.6.

images per class	VGG	ResNet	Inception
1000	36.7	25.7h	19.8h
500	17.7h	18.8h	2.8h
250	15.8	16.2h	2.0h
100	4.3h	11.2h	2.4h
50	4.9h	2.8h	27.2min
20	1.1h	5.9h	24.0min

Table 4.6: A summary of the training times for the VGG-19, ResNet-152 and Inception-v3. They are listed after the training images per class (3 classes). The times equate the training times until the best validation accuracy was achieved (the time needed for the computation of the evaluation accuracies is excluded).

The Inception is again in the first place, with smaller total training times than the other two networks. For the ResNet and VGG network the picture is less plain. Sometimes the ResNet converges faster, and sometimes the VGG.

Test accuracies and training times both indicate that the Inception-v3 network architecture is the overall best choice. Second place is the ResNet, which has better test accuracy results than the VGG. In some cases the VGG may converge faster, but because of the better test accuracy results of the ResNet it is reasonable to favor the residual network architecture.

4.5 Training the VGG, the Inception and the ResNet on Medical Data

Medical image data is theoretically very suitable for deep learning. Because medical images are made for the purpose of health care, the data quality is due to regulations in general very high. But since there are laws protecting the patients' privacy, the data is hard to access and therefore available data sets are scarce [Cho2016].
Using neural networks for medical imaging can reduce diagnosis times and prohibit relevant information from being overlooked. Its strength is also the possibility to process large data sets [Amato2013].

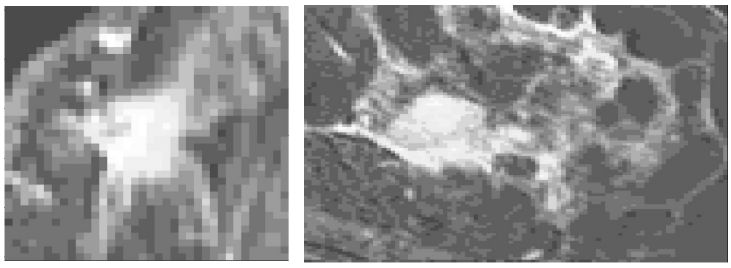
A publication of Cho et al. [Cho2016] engaged itself with the issue of the amount of data needed for the training of medical images. The task for the network was to distinguish between images of different body areas like the brain, the chest or the abdomen. The results presented in the publication suggest that a data amount of at least 50, but rather 100 to 200 images is needed, depending on the body part.
A possible strategy to train medical images even with small data sizes is finetuning. Though medical images are very different to natural images, finetuning medical images with a network that was trained on non-medical data was found to nevertheless increase the performance compared to training from scratch [Tajbakhsh2016].
A paper of Lévy and Jain [Lévy2016] relates to the classification of breast cancer. They finetuned an AlexNet and a GoogLeNet architecture with a data set consisting of 1820 mammography images from 997 patients to classify breast cancer masses. The authors reached a 93.4 percent sensitivity[2] with the GoogLeNet and claim to have outperformed radiologists with the network. With the AlexNet, they still reached an 89 percent sensitivity.
To investigate the performance of the VGG-19, Inception-v3 and the ResNet-

[2]The sensitivity is an important measure in medical applications because it is especially important to keep the number of false negatives small

152 on medical image data the networks were trained on breast cancer mammographies.

4.5.1 The Breast Cancer Data Set

The medical data set is composed of MRI-mammography images that were made for the detection of breast cancer [Hoffmann2013]. All of them are cut out so that the images show the tumor area only (s. figure 4.2). Therefore the image sizes are varying a lot.



(a) benign (b) malignant

Fig. 4.2: Mammography images of a benign and a malignant case

There are two classes, tumors that turned out malignant and tumors that turned out benign. The data set is balanced: it encompasses 29 patients with benign and 30 patients with malignant tumors. A non-balanced data set can really hurt the network performance [Mazurowski2008].
The images were weighted with the T1 signal. There are five time points. The first one is the base value, meaning that the contrast agent was not yet injected. During the time points two to five the concentration of the contrast agent is first sharply increasing and then decreasing linearly.
A linear regression model was fitted for each voxel, over the time points two to five.
Both the original and the regression images were used for the training. In the case of the regression data, the intercept and slope pixel values were added up. All the images were equipped with a frame of zero values to achieve an image size of 224x224 for every image.

For each case, there are a couple of image acquisitions. To augment the data, two mammography images were used from each case, but it was made sure that the same case does not appear in two different sets (f.e. both in training and test set). Because the pretrained models were trained on RGB images and the mammography images are greyscale, the greyscale channel was duplicated to get 3 channels. For preprocessing, the mean image was subtracted.

4.5.2 Results for the Medical Data

The neural networks were trained using learning rates between 0.0004 and 0.0007 and a momentum of 0.5. The Inception network was also trained with the Adam optimizer, but this did not enhance the outcome.
At first, the images that were received through the regression were used for the training of the neural networks VGG-19, Inception-v3 and ResNet. Table 4.7 shows the results:

network	test accuracy	training time	rounds
VGG-19	41.7	2.9h	26
Inception v3	58.3	1.2h	34
ResNet-152	50	1.7h	13

Table 4.7: Results of the VGG, Inception and ResNet on the Regression Data Set

For the VGG, the test accuracy reached a value of 41.7 percent, and for the Inception network 58.3 percent. The ResNet obtained a test accuracy of 50 percent. The training of the networks took between one to three hours. The outcomes indicate that none of the networks can truly distinguish malignant from benign tumors.

The training for the original breast cancer data was conducted with the images that were made at the third time point. In table 4.8, the results for these images are listed.

network	test accuracy	training time	iterations
VGG-19	54.2	2.8h	25
Inception v3	58.3	13min	6
ResNet-152	62.5	55min	7

Table 4.8: Results of the VGG, Inception and ResNet on the original breast cancer images

Also for this data the test accuracies are rather small. The VGG accomplished solely 54 percent, and the Inception and the ResNet got accuracies of 58 and 62.5 percent. The training lasted between 13 minutes and and three hours.

In summary: for both the original and the regression breast cancer data, neither network can differentiate between benign and malignant cases.

5 Conclusion

The results show that the data amount that is needed for the training of neural networks is dependent on the network architecture. The best classification accuracies could be achieved using the Inception-v3 with Batch Normalization. The ResNet architecture with 152 layers is the second best of the considered networks. The VGG-19 was last.
This is very similar to the single-model results on large data sets: in their respective papers the authors report top-5 error rates of 7.0 for the VGG , 4.49 for the ResNet and 4.2 for the Inception on ImageNet data [Simonyan2015] [He2015a] [Szegedy2015b].
There is already an approach by Szegedy et al. to combine the advantage of both the Inception Module and the residual building blocks, as well as a new Inception-v4 architecture which is also presented in the paper [Szegedy2016].

The main goal of this thesis was to find out how many images are needed to successfully train a CNN.
Surely the amount of data needed for the training is to an extent also dependent on the complexity of the classification task and also the quality of the images. But from the results on this data set the conclusion is that, with fine-tuning and a decent network, a data amount of 250 images per class already suffices to train a very able network. Anyway, a 2.2 percent error rate could be obtained with this size.
Even the training with smaller datasets of 50 to 100 images per class still obtained classification accuracies between 90 and 96 percent, depending on the network.

For Finetuning, using adaptive learning rate optimizers is not always advisable. It did enhance the performance of the Inception-v3. But for the VGG and ResNet on the other hand the test accuracies were clearly better for manually set learning rates than for the Adam optimizer.

The training with medical data did not yield a network that is able to distinguish between malignant and benign cases. The reason for this is inconclusive. A reasonable hypothesis is the small data size. The results for the non-medical data suggest that for a training data set consisting of 20 images per class a training accuracy of 87 to 90 percent is reachable. But for finetuning it does make a difference how similar the images the network was pretrained on are to the new dataset [Yosinski2014].
It is also possible that there is no regularly occurring difference between the malignant and benign tumors, or the neural networks are not able to detect the pattern. The second case can happen when the network requires more data, but also when the quality of the data does not suffice.

Future research on this topic could involve investigating the effects of data augmentation techniques like alternating image sizes, mirroring, and rotating on the test accuracy for different data sizes.
For deep learning, it is very important that the training data set covers the variety of the class. The benefit of more pictures is that the network learns more versions of the depicted. Thus it is very likely that the performance of the networks could be even further enhanced with additional use of data augmentation techniques.
For the medical images, the image sizes were rather small. An approach would be to use a model that is more fit to small input data sizes where it is not necessary to artificially increase their size.
With respect to the medical application in general, it would be interesting to finetune medical image data on a model pretrained with other medical data

to draw comparisons to results achieved with pretrained models trained on non-medical data. But certainly this would require the availability of suitable medical data sets.
Furthermore there are multi-model approaches where the predictions of several models are combined by following the majority "vote", or averaging over the network outputs [Zhou2002]. Anderson et al. [Anderson2016] used this idea for improving finetuning by using both a pretrained and a model trained on the target data in combination in their paper "Beyond fine tuning: A modular approach to learning on small data".
At last, it is worth mentioning the work of Yosinski [Yosinski2015] and Zeiler [Zeiler2013] regarding the visualization of DCNNs, which is letting us understand more thoroughly the way neural networks learn and how they perceive images.

Since Yann LeCun's paper about convolutional layers in 1998 [LeCun1998] and later the stunning results of their use with the AlexNet in the ILSVRC 2010 competition [Krizhevsky2012], a lot of researchers have engaged themselves in the enhancement of convolutional neural networks. The results so far are promising and one can expect some more development in the future of the scientific field of deep convolutional neural networks.

6 Electronic Appendix

This is a short description of the files enclosed with this thesis on two DVDs. One DVD contains the Places files, and the other the breast2 files.

6.1 RCode

There are two R-Code folders, R-Code breast and R-Code Places.
In the Places folder there are the four R-files. In the "Places_resize" file is the code to resize the 356x356 Places data to 224x224. With the code in the "Places_create_data" file these resized images can be turned into a data set and saved in a CSV file. The "finetune_places" and "test_places" files contain the code needed to finetune networks and test them on the test data set.

The breast2 folder contains four R-files. The code in the "breast2_images_data" and "breast2_regression_data" files transforms the breast2 data into 224x224 images by adding a frame of zeros to the images, turns them into a dataset and creates a CSV file. The "finetune_breast" and "test_breast" R-files are for the finetuning and testing of the breast2 data.
Furthermore, each R-Code folder includes a functions folder with functions and other code that is called by the mentioned files.
The R-Code depends on the packages mxnet [Chen2015b], EBImage [Pau2010], oro.nifti [Whitcher2011], data.table [Dowle2017] and gdata [Warnes2015].

6.2 Data

The data is in two folders called data Places and data breast2.
The data Places folder contains 5000 356x356 Places2 images for each of the three classes (staircase, waterfall and cockpit). The trainM files folder holds the mean pixels used for the preprocessing of the Places data sets.
In the breast2 data folder there are the original images as well as the regression images of the breast2 dataset, including a text file with the class categories for the files.
For the original images there are three folders. The breast2 images folder contains the full data, whereas in the benign-3 and malignant-3 folders there are the images for the third time point ordered according to the class.
In the breast2 regression folder, the regression data is ordered after class and regression output (intercept, slope).
The corresponding CSV files to the breast2 data can be found in the CSV files folder.

6.3 .params and .json Files

The .params and .json files for the pretrained models are saved in the pretrained models folder. The network files that achieved the accuracies presented in this work are in the folders params_files Places and params_files breast2.

References

[Amato2013] Amato, F.; López, A.; Peña-Méndez, E. M.; Vaňhara, P.; Hampl, A.; Havel, J. (2013): Artificial neural networks in medical diagnosis. J Appl Biomed Vol. 11, pp. 47–58.

[Anderson2016] Anderson, A.; Shaffer, K.; Yankov, A.; Corley, C.D.; Hodas, N.O. (2016): Beyond Fine Tuning: A Modular Approach to Learning on Small Data. arXiv:1611.01714v1.

[Bolte2016] Bolte, B. (2016): Looking at Residual Networks. http://ben.bolte.cc/blog/2016/resnet.html (accessed 05.03.17).

[Buduma2017] Buduma, N.; Locascio, N. (2017): Fundamentals of Deep Learning: Designing Next-generation Machine Intelligence Algorithms. O'Reilly Media.

[Chen2015a] Chen, T.; Li, M.; Li, Y.; Lin, M.; Wang, N.; Wang, M.; Xiao, T.; Xu, B.; Zhang, C.; Zhang, Z. (2015): MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems, arXiv:1512.01274v1.

[Cho2016] Cho, J.; Lee, K.; Shin, E.; Choy, G.; Do, S. (2016): How much data is needed to train a medical image deep learning system to achieve necessary high accuracy? arXiv:1511.06348v2.

[Dettmers2015] Dettmers, T. (2015): Understanding Convolution in Deep Learning. http://timdettmers.com/2015/03/26/convolution-deep-learning (accessed 13.03.17).

[DMLC2016] DMLC (2016): Flexible and Efficient Library for Deep Learning. http://mxnet.io, http://mxnet.io/model_zoo (accessed 02.06.17).

[Duchi2011] Duchi, J.; Hazan, E.; Singer, Y. (2011): Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research Vol. 12, pp. 2121-2159.

[Ghahramani2004] Ghahramani, Z. (2004): Unsupervised Learning, in: Bousquet, O. et al. (eds): Advanced Lectures on Machine Learning. LNAI 3176. Springer, pp. 72-112.

[Glorot2010] Glorot, X.; Bengio, Y. (2010): Understanding the difficulty of training deep feedforward neural networks. Proceedings of Machine Learning Research Vol. 9 (AISTATS 2010 Proceedings), pp. 249-256.

[Goodfellow2016] Goodfellow, I.; Bengio, Y.; Courville, A. (2016): Deep learning. MIT press.

[Guo2015] Guo, Y.; Liu, Y.; Oerlemans, A.; Songyang, L.; Wu, S.; Lew, M.S. (2015): Deep learning for visual understanding: A review. Neurocomputing Vol. 187, pp. 27-48.

[He2015a] He, K.; Zhang, X.; Ren, S.; Sun, J. (2015): Deep Residual Learning for Image Recognition. arXiv:1512.03385v1.

[He2015b] He, K.; Zhang, X.; Ren, S.; Sun, J. (2015): Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. arXiv:1502.01852v1.

[Hentschel2016] Hentschel, C.; Wiradarma, T.P.; Sack, H. (2016): Fine Tuning with scarce Training Data - Adapting ImageNet to Art Epoch Classification. IEEE International Conference on Image Processing (ICIP).

[Hoffmann2013] Hoffmann, S., Shutler, J.D., Lobbes, M.; Burgeth, B.; Meyer-Bäse, A. (2013): EURASIP J. Adv. Signal Process, 2013: 172.

[Ioffe2015] Ioffe, S.; Szegedy, C. (2015): Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv:1502.03167v3.

[Karpathy2017] Karpathy, A. (2017): CS231n Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io (accessed 20.03.17).

[Kingma2015] Kingma, D. P.; Ba, J. L. (2015): Adam: a Method for Stochastic Optimization. International Conference on Learning Representations, San Diego.

[Krizhevsky2012] Krizhevsky, A.; Sutskever, I.; Hinton, G. E. (2012): ImageNet Classification with Deep Convolutional Neural Networks. Advances in Neural Information Processing Systems 25 (NIPS 2012).

[LeCun1998] LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. (1998): Gradient-Based Learning Applied To Document Recognition. Proceedings of the IEEE Vol. 86, Issue 11, pp. 2278-2324.

[LeCun2015] LeCun, Y.; Bengio, Y.; Hinton, G. (2015): Deep learning. Nature Review Vol. 521.

[Lévy2016] Lévy, D.; Jain, A. (2016): Breast Mass Classification from Mammograms using Deep Convolutional Neural Networks. 30th Conference on Neural Information Processing Systems, Barcelona, Spain. arXiv:1612.00542v1.

[Lipton2015] Lipton, Z. C.; Berkowitz, J.; Elkan, C. (2015): A Critical Review of Recurrent Neural Networks for Sequence Learning. arXiv:1506.00019v4.

[Mazur2015] Mazur, M. (2015): A Step by Step Backpropagation Example. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example (accessed 21.03.17)

[Mazurowski2008] Mazurowski, M. A.; Habas, P. A.; Zurada, J. M.; Lo, J.Y.; Baker, J.A., Tourassi, G.D. (2008): Training neural network classifiers for

medical decision making: the effects of imbalanced datasets on classification performance. Neural networks Vol. 21, Issues 2-3: pp. 427–436.

[McCaffrey2015] McCaffrey, J. (2015): Test Run - L1 and L2 Regularization for Machine Learning. MSDN Magazine Blog Vol. 30, Nr.2.

[Neubert2016] Neubert, A.; Fripp, J.; Chandra, S.S.; Engstrom, C.; Crozier, S. (2016): Automated Intervertebral Disc Segmentation Using Probabilistic Shape Estimation and Active Shape Models, in: Vrtovec, T. et al. (eds): Computational Methods and Clinical Applications for Spine Imaging. CSI 2015. Lecture Notes in Computer Science Vol. 9402. Springer, Cham.

[Nielsen2015] Nielsen, M. A. (2015): Neural Networks and Deep Learning, Determination Press.

[Oquab2014] Oquab, M.; Bottou, L.; Laptev, I.; Sivic, J. (2014): Learning and Transferring Mid-Level Image Representations using Convolutional Neural Networks. IEEE Conference on Computer Vision and Pattern Recognition, Columbus, OH, United States.

[Ruder2016] Ruder, S. (2016): An overview of gradient descent optimization algorithms. arXiv:1609.04747v1.

[Russakovsky2015] Russakovsky, O.; Deng, J.; Su, H.; Krause, J.; Satheesh, S.; Ma, S.; Huang, Z.; Karpathy, A.; Khosla, A.; Bernstein, M.; Berg, A.C.; Fei-Fei, L. (2015): ImageNet Large Scale Visual Recognition Challenge. IJCV, 2015.

[Simonyan2015] Simonyan, K.; Zisserman, A. (2015): Very deep convolutional networks for large-scale image recognition. arXiv:1409.1556v6.

[Srivastava2014] Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R. (2014): Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research 15, pp. 1929-1958.

[Sun2014] Sun, Y.; Wang, X.; Tang, X. (2014): Deep Learning Face Representation by Joint Identification-Verification, in: Advances in neural information processing systems, arXiv:1406.4773v1.

[Szegedy2015a] Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; Rabinovich, A. (2015): Going deeper with convolutions. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 1-9.

[Szegedy2015b] Szegedy, C.; Vanhoucke, V.; Ioffe, S.; Shlens, J.; Wojna, Z. (2015): Rethinking the Inception Architecture for Computer Vision. arXiv:1512.00567.

[Szegedy2016] Szegedy, C.; Vanhoucke, V.; Ioffe, S.; Alemi, A. (2016): Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. arXiv:1602.07261v2.

[Tajbakhsh2016] Tajbakhsh, N.; Shin, J. Y.; Gurudu, S. R. (2016): Convolutional Neural Networks for Medical Image Analysis: Full Training or Fine Tuning? IEEE Transactions on Medical Imaging Vol. 35, Issue 5, pp.1299-1312.

[Tang2013] Tang, Y. (2013): Deep Learning using Linear Support Vector Machines. International Conference on Machine Learning 2013: Challenges in Representation Learning Workshop. Atlanta, Georgia, USA.

[VanDoorn2014] Van Doorn, J. (2014): Analysis of Deep Convolutional Neural Network Architectures. 21th Twente Student Conference on IT, Enschede, The Netherlands.

[Witten2000] Witten, I.H.; Eibe, F. (2000): Data Mining: Practical Machine Learing Tools and Techniques with Java Implementations. San Diego, CA: Morgan Kaufmann.

[Xianyi2016] Xianyi, Z.; Quian, W.; Saar, W. (dev.) (2016): OpenBLAS. An optimized BLAS library. http://www.openblas.net (accessed 13.06.17).

[Yosinski2014] Yosinski, J.; Clune, J.; Bengio, Y.; Lipson, H. (2014): How transferable are features in deep neural networks. Advances in Neural Information Processing Systems 27 (NIPS 14), pp. 3320-3328.

[Yosinski2015] Yosinski, J.; Clune, J.; Nguyen, A.; Fuchs, T.; Lipson, H. (2015): Understanding Neural Networks Through Deep Visualization. arXiv:1506.06579v1.

[Zeiler2012] Zeiler, M. D. (2012): Adadelta: an adaptive learning rate method. arXiv:1212.5701v1.

[Zeiler2013] Zeiler, M. D.; Fergus, R. (2013): Visualizing and Understanding Convolutional Networks. arXiv:1311.2901v3.

[Zhou2002] Zhou, Z.; Wu, J.; Tang, W. (2002): Ensembling Neural Networks: Many Could Be Better Than All. Artificial Intelligence Vol.137, No.1-2, pp. 239-263.

[Zhou2016] Zhou, B.; Khosla, A.; Lapedriza, A.; Torralba, A.; Oliva, A. (2016): Places: An Image Database for Deep Scene Understanding. arXiv:1610.02055.

R-Packages:

[Chen2015b] Chen, T.; Kou, Q.; He, T. (2015): mxnet: Mxnet. R package version 0.9.3.

[Dowle2017] Dowle, M.; Srinivasan, Arun (2017): data.table. Extension of "data.frame". R-package version 1.10.4.

[Pau2010] Pau, G.; Fuchs, F.; Sklyar, O.; Boutros M.; Huber, W. (2010). "EBImage - an R package for image processing with applications to cellular phenotypes" Bioinformatics, 26(7), pp. 979–981.

[Warnes2015] Warnes, G.R.; Bolker, B.; Gorjanc, G.; Grothendieck, G.; Korosec, Ales; Lumley, T.; MacQueen, D.; Magnusson, A.; Rogers, J. et al. (2015):

gdata: Various R Programming Tools for Data Manipulation. R package version 2.17.0.

[Whitcher2011] Whitcher, B.; Schmid, V.; Thornton, A. (2011): Working with the DICOM and NIfTI Data Standards in R. Journal of Statistical Software, 44(6), 1-28.

Declaration

I declare that I completed this work on my own and that information which has been directly or indirectly taken from other sources has been noted as such.

Munich, June 2017 _____