JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Peham Laura

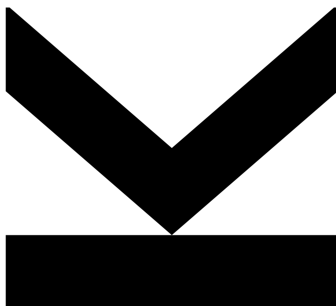
Submitted at
Department of Knowledge-Based Mathematical Systems

Supervisor
Assoc. Prof. Mag. Dr. S. Saminger-Platz

Co-Supervisor
DI Robert Pollak, JKU
Dr. Bettina Heise, RECENDT

August 2019

Deep Learning Approaches to OCT-Image Classification for Technical Materials



Master Thesis
to obtain the academic degree of
Diplom-Ingenieurin
in the Master's Program
Computermathematik

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Statutory Declaration

I hereby declare under oath that the submitted Master's Thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited. The submitted document here present is identical to the electronically submitted text document.

Laura Peham
Linz, August 2019

Acknowledgment

Throughout the writing of this thesis I have received a lot of support and assistance and I am grateful to all of those with whom I have had the pleasure to work during this project.

I would first like to thank my thesis supervisor Assoc. Prof. Mag. Dr. S. Saminger-Platz. This thesis would not have been possible without your commitment to enable it. Your advice and expertise over the past months has been invaluable and you supported me greatly, especially in finishing this work in time. At this point I would also like to thank you for welcoming me at the institute several times throughout my years of study, it has been great working there.

I would also like to thank my most important advisor Robert Pollak. Your door was always open whenever I ran into trouble, got stuck or had a question. With your input, guidance and expertise you not only made this thesis possible but also taught me a lot more than theoretical studies could ever have.

I am indebted to many of my colleagues to support me throughout my work: Paul Wiesinger, thank you for your technical support in all things. Your skillfulness and rapidity in solving technical problems impressed me many times. And Werner Zellinger, thank you for the numerous thought-provocating discussions and valuable comments. Without your profound knowledge in machine learning and willingness to share it I would have gotten stuck at times. And I would like to thank the entire team of the department of knowledge-based mathematical systems for welcoming me in the team over the past few years.

This work would not have been possible without the Research Center for Non-Destructive Testing GmbH (RECENDT). Thank you for providing the topic of this thesis and for the great opportunity to gain insight in the topics of non-destructive testing and OCT-imaging. I am grateful for everyone who helped me with understanding and using the OCT-systems, but especially I want to thank Dr. Bettina Heise. Your supervision, guidance and ideas haven been fundamental for my work.

Finally, I owe my deepest gratitude to my family and my partner for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without you. Thank you for always being there for me.

Abstract

Optical coherence tomography (OCT) is a non-destructive and non-contacting imaging technology and can visualize the internal structures of various materials. OCT-imaging has been used primarily for biomedical issues, e.g. , in ophthalmology to visualize retina layers or in dermatology for the detection of skin diseases. In recent years OCT imaging has also become popular for industrial applications for non-destructive testing of materials which is needed in quality assurance and for the development of new materials. **Deep Learning** models are able to learn patterns from complex datasets and to make predictions for new data based on this knowledge. Medical specialist literature has already covered the successful application of deep learning on OCT-images for biomedical issues but the application of deep learning for OCT-image classification in industrial contexts has to the best of our knowledge not been covered by specialist literature so far. As the analysis and categorization of different materials is an important issue in many industrial processes the use of deep learning on OCT-images for material classification in two different tasks will be examined: The first task is to categorize 3D-printed objects with respect to their material (expressed by different color pigments in the dough) with only their (greyscale) OCT-image as input. Given are objects in green, grey, red and transparent colors and two different shapes. In most of the cases the OCT-recordings of the four colors are well distinguishable by humans. Therefore 12800 images (3200 per color) from different areas on the objects were recorded also including surface defects, interior defects, plain surfaces and different slopes. Evaluating different model architectures obtained at most 99,47% accuracy on an independent test set. Additionally features from the OCT images were extracted to compare the performance of the deep learning model with two other machine learning methods namely random forest and support vector machine which obtained 95,29% resp. 95,19% accuracy on the test set. The second task is the categorization of OCT-images of different materials with respect to their coating. Given are five different paper-like materials with coated and uncoated areas on it. 4500 OCT-images of coated and 4500 OCT-images of uncoated areas were recorded and classified with a final test accuracy of 98,58%. Again this result was compared with the results obtained by the other machine learning methods: Random forest reaches a test accuracy of 95,11% and support vector machine reaches a test accuracy of 94,04%.

Contents

1 Introduction **1**

1.1 Problem Description . . . 2

1.2 Structure of this Work . . . 2

2 Optical Coherence Tomography **3**

2.1 Comparison of OCT with Other Imaging Modalities . . . 3

2.2 Technology . . . 4

2.3 Introduction and Applications . . . 5

3 Machine Learning **8**

3.1 Preliminaries . . . 8

3.2 Neural Networks — Deep Learning . . . 17

3.2.1 The Structure of Neural Networks . . . 18

3.2.2 Learning . . . 19

3.2.3 Regularization . . . 22

3.2.4 Hyper-Parameter Tuning . . . 28

3.2.5 Convolutional Neural Networks . . . 33

3.3 Selection of Feature-Based Machine Learning Methods . . . 37

3.3.1 Random Forests . . . 37

3.3.2 Support Vector Machines . . . 41

4 Deep Learning for OCT-Images **48**

4.1 State of the Art . . . 48

4.2 The First Task – 'Material' Classification . . . 48

4.2.1 Problem Description . . . 48

4.2.2 Deep Learning . . . 50

4.2.3 Feature-Based Methods – Random Forests and Support Vector Machines . . . 55

4.2.4 Comparison 59
4.2.5 Remarks 60
4.3 The Second Task – Inspection of Coatings 61
4.3.1 Problem Description 61
4.3.2 Deep Learning 62
4.3.3 Feature-Based Methods – Random Forests and Support Vector Machines 64
4.3.4 Comparison 69
4.3.5 Remarks 69
4.4 Comparison and Conclusions 71
4.5 Possible Extensions and Outlook 72

5 Implementation **74**

List of Figures

2.1 Comparison of OCT with other imaging modalities 4
2.2 Schematic OCT setting, from [63] 5
2.3 OCT-imges of medicine and industry 7

3.1 Polynomial fit 12
3.2 Under- and overfitting 13
3.3 The bias-variance trade off 14
3.4 Confusion matrix for a classification task with k classes $c_1, .., c_k$ 14
3.5 Confusion matrix for a binary classification task 15
3.6 Data splitting 17
3.7 k-fold cross-validation 18
3.8 ReLU activation function 19
3.9 Schematic representation of an artificial neural network 20
3.10 Detailed view of one neuron 21
3.11 Applying dropout to a neural network 24
3.12 Obtained networks after applying dropout 25
3.13 Application of batch normalization 27
3.14 Data augmentation. 28
3.15 A basic convolutional neural network. 33
3.16 An example for a 2-D convolution operation. 35
3.17 An example for a horizontal-edge detecting filter. 36
3.18 The max-pooling. 36
3.19 Max-pooling makes the network translation invariant. 37
3.20 The different kinds of splitting 40
3.21 The separating hyperplanes 42
3.22 The margin of a separating hyperplane 43
3.23 The different kinds of points 46

LIST OF FIGURES

4.1 3D-print objects 49
4.2 OCT-images of the 3D-printed objects 49
4.3 The recordings of different surfaces and defects. 50
4.4 The network architecture of the final model. 52
4.5 False predicted images. 53
4.6 Training and validation accuracy curves 53
4.7 Some filters of the convolutional layers for the first task. 54
4.8 Visualization of the output layer for the first task. 55
4.9 Visualization of one decision tree for the first task. 58
4.10 Detailed visualization of the first three depth-levels. 59
4.11 Boxplots of test accuracies for the first task. 61
4.12 OCT-images of the second task. 62
4.13 Wrongly predicted images of the coating-task. 64
4.14 Some filters of the convolutional layers for the second task. 65
4.15 Visualization of the output layer for the second task. 66
4.16 Visualization of one decision tree for the coating task. 68
4.17 Detailed visualization of the first three depth-levels. 69
4.18 Boxplots of test accuracies for the second task. 70

List of Tables

4.1 Number of recordings for the first task 50
4.2 Accuracies and average training times for the three different machine learning methods applied to the first task. 60
4.3 The validation and test accuracies for the five different data-splits obtained with deep learning. 63
4.4 The validation and test accuracies for the five different data-splits obtained with random forest classifiers. 67
4.5 The validation and test accuracies for the five different data-splits obtained with support vector machine classifiers. 68
4.6 Average accuracies and training times for the three different machine learning methods applied to the second task. 70
4.7 Accuracies for the three different machine learning methods. 71

1 Introduction

In recent years terms like 'Artificial Intelligence' (AI) and 'Machine Learning' (ML) received significant rise of media attention. It seems like intelligent machines and human-like robots are both fascinating and worrying for human beings. The dream of creating 'thinking' machines goes back to at least the ancient Greeks [16] but the tremendous growth in machine learning was only after the enhancements of computer processors and the increase in the amount of available data. This evolution improved in particular one machine learning technique drastically: deep learning. Deep learning models are able to learn models from very extensive and complex datasets and make predictions for new data based on this models without the need of user-defined rules describing the task to solve. It is a machine learning method which deals with deep artificial neural networks. To obtain output values for each input value neurons (units) are arranged in layers and each layer takes the output of the former layer as input and passes it forward to the next layer. Different kinds of architectures can be used to handle different input structures e.g. for the classification of images convolutional layers are used in addition to fully connected layers. The aim of this thesis is to apply deep learning to OCT-data.

Optical Coherence Tomography (OCT) is a non-destructive and non-contacting imaging modality developed in the late 1980s. It enables the visualization of the internal structure of materials which is often an important task in biomedical investigations on the one hand and industrial processes on the other hand. It works similar to ultrasound but it uses light waves instead of sound waves and has the great advantage over other imaging modalities that state-of-the-art systems can reach an axial resolution of 1-10μm. However one disadvantage is that because of the scattered light the imaging depth is limited to about 2mm. Medical specialist literature has already covered the successful application of deep learning on OCT-images of the human eye [3, 33, 29]. In this thesis we will examine the use of deep learning on OCT-images in the context of industrial application such as material classification and inspection of coatings and compare this method to other machine learning approaches.

1.1 Problem Description

As already mentioned before OCT-imaging is a very important modality for many industrial processes. One application often used is the classification of different materials as they have different interior structures which can be seen very well in OCT-images. In this case the different materials will be expressed by different color pigments. Looking at them with an OCT-system shows different levels of reflection/absorption and in most of the cases it would also be possible to distinguish the colors manually. The goal of this thesis is to see how well a deep neural network can independently learn a classification model. Another important application of OCT in industrial processes is the inspection of coating thickness. It is for example used in pharmacy to supervise the coatings on pills and in other different fields interested in the analysis of layers.

1.2 Structure of this Work

In Chapter 2 and 3 we will provide an introduction to the theory of optical coherence tomography and machine learning. First we will cover the optical coherence tomography imaging modality and its applications in medicine and industry and also show some typical example images. This Section will be followed by an introduction to the theory of machine learning. As deep learning is a specific machine learning method it is important to understand the concepts and principles of theoretical machine learning, followed by a discussion to the basic principles of deep learning, in particular convolutional neural networks, as well as random forests and support vector machines. In the second part of the thesis we will use deep learning for practical applications, see Chapter 4. Here we will present the two tasks which are to solve, talking about model conventions and occurring problems. In the end there are presented some results and possible extensions and we will also compare the results of deep learning with the results of two other machine learning approaches namely random forest and support vector machine.

2 Optical Coherence Tomography

We briefly introduce some basic theory and applications of optical coherence tomography following [11].
Optical coherence tomography (short **OCT**) was invented in the late 1980s and early 1990s and was demonstrated first by Huang et al. in 1991 [23]. It is a non-destructive and non-contacting imaging technology and can visualize the internal structures of various materials. OCT-imaging is used primarily for biomedical issues e.g. in ophthalmology to visualize the retina layers or in dermatology for the detection of skin diseases. In recent years it also became popular for industrial applications for non-destructive testing of materials which is needed in quality assurance and for the development of new materials.

2.1 Comparison of OCT with Other Imaging Modalities

Optical coherence tomography has similarities with both ultrasound and confocal microscopy. Clinical ultrasound imaging has a great penetration depth and is therefore able to visualize structures inside the human body, for example organs. But because of the sound wave frequency used the resolution is typically limited to 0.1–1 mm. Better resolution is given with a high frequency ultrasound which reaches 15-20 μm but with this technology the imaging depth is limited to a few millimetres. On the other side of OCT imaging there is confocal microscopy which can achieve remarkably high resolution approaching 1 μm but because of the optical scattering the imaging depth is limited to only a few hundred micrometers. The advantage of OCT is that it fills a gap between ultrasound and microscopy as it has a very high resolution of 1-10 μm whereas the imaging depth limits to about 2 mm, see Figure 2.1. Another advantage compared to other imaging technologies is its non-contacting and non-invasive usage.

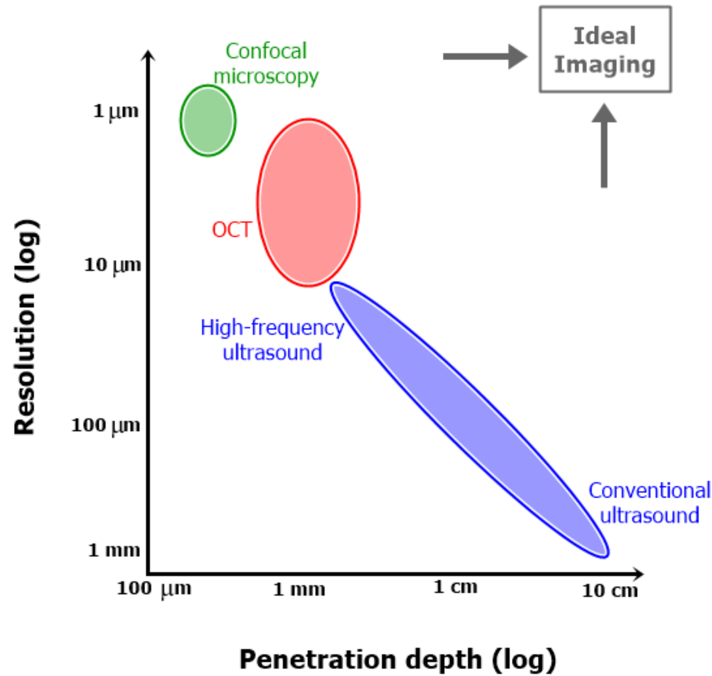


Figure 2.1: Comparison of OCT with other imaging modalities, adapted from [11].

2.2 Technology

OCT-imaging works similar to ultrasound but it uses light waves instead of sound waves. Several methods have been developed so far but in general the magnitude and echo time delay of the light which is back-reflected from the internal micro structures of materials is measured. The scheme is the following: Low coherent light from a broadband light source (e.g. laser) is split into two paths. One path is directed towards a reference arm and the other is directed towards the sample. The backscattered light from the sample is then interfered with the reflected light from the reference arm. The OCT-system used for this thesis has spectral/Fourier domain detection as interferometric detection technique: The light beams of the reference arm and the sample arm have a time delay related to the depth of the sample structure. The interference is acquired with a spectrometer as a function of frequency and then the depth scan can be computed by a Fourier-transform from this spectra. For getting a 2D-image point-wise scanning in lateral direction is performed. Like already mentioned before state-of-the-art OCT-systems can reach an axial resolution of 1-10μm but because of the scattered light the imaging depth limits to about 2mm. For a scheme of an OCT-system see Figure 2.2. OCT imaging has some special features and difficulties compared to other imaging technologies. Here the resulting image depends strongly on the scattering

features of the material used. Surfaces do not always appear as straight and exact lines as coarse structures may scatter the light in another way than smooth surfaces do. Also the depth to which the internal structure of the tissue can be visualized varies with the choice of the material as for example white-coloured objects generally strongly disperse the light and therefore the light can not enter very deep.

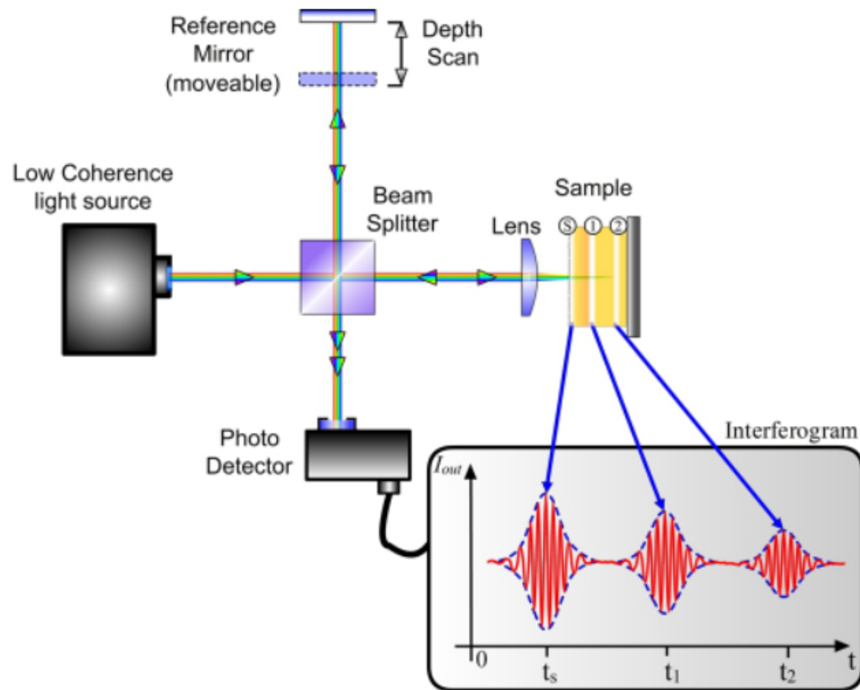


Figure 2.2: Schematic OCT setting, from [63]

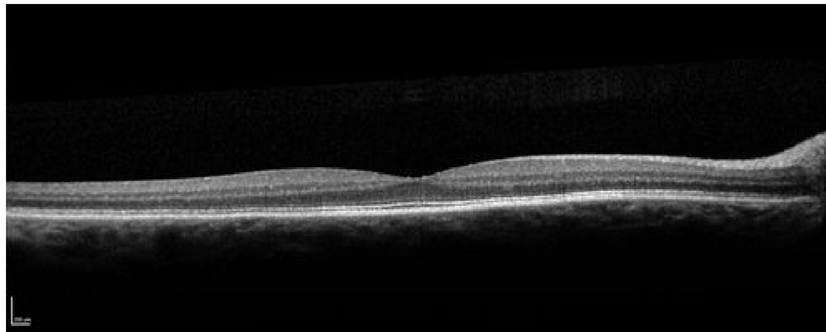
2.3 Introduction and Applications

The main application for OCT-imaging is still in the biomedical field. After the invention in the late 1980s and early 1990s the development of the imaging device in ophthalmology processed quickly given that the first in vivo retinal images were obtained in 1993 by Fercher et al [14] and Swanson et al [48] independently. With the application of OCT for clinical studies in the mid 1990s the imaging modality was investigated for the visualization of many different macular diseases like macular edema, macular holes or age-related macular degeneration. Because OCT allows to detect early stages of diseases before physical symptoms appear, it is a powerful technology in ophatlmology and often used as a standard technique for the monitoring of retinal diseases [12] as well as glaucoma [7]. A sign for the importance of this medical field for the further development of the system is that in 2008 half of

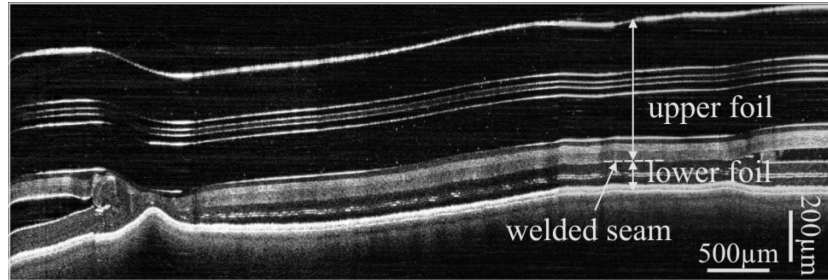
all OCT-publications have been published in ophthalmic journals. As it became clear that the use of longer optical wavelengths can reduce scattering and increase the penetration depth OCT-imaging also became practicable in other biomedical areas. Currently an important area of research is e.g. the detection of atherosclerotic plaque which lead into the clinical development of intra-vascular OCT imaging. Another major application and also an active area for research is the examination of early neoplastic changes which is important for e.g. gastrointestinal, biliary or pulmonary tracts. To enable internal body imaging very early also endoscopic OCT systems were developed. In 1997, the first in vivo endoscopic OCT imaging was performed and demonstrated the importance of OCT also for the observation of organ systems. In contrast to conventional endoscopy OCT endoscopy is able to visualize subsurface tissue morphology which is relevant for the prevalence of gastrointestinal cancers. But since there is a reasonable variation in pathology and there are many types of low incidence cancers a large number of patients are needed to reach statistical significance. Therefore, the usage of OCT for cancer detection will require specific studies and is an ongoing area of research, see e.g. [9]. Other biomedical fields using OCT imaging technology are; e.g.

- Dermatology,
- Laryngology,
- Dentistry,
- Gynecology,
- Development biology.

Because of its advantages compared to other imaging technologies OCT became also popular for industrial applications in recent years, see e.g. [63]. One very important area is the non-destructive testing of materials which is often used for the purpose of quality control in industrial processes and in food industry. This includes the detection of defects for different materials like glass fibre composites or injection moulded polymers [47], the analysis of the distribution of particles in protective coatings, for example in pharmaceutical tablets [32], or of pores in polymer foams. For the packaging industry (among others) an important application is also the examination of the layers in multilayer foils [18] and the detection of an incomplete sealing. Another interesting application is the non-invasive examination of museum paintings [30]. For further reading and other non-biomedical applications see also [13, 21, 35]. With OCT imaging it is possible to visualize internal defects as well as surface cracks or incomplete coatings for various materials. In industry OCT systems can also be used to provide complementary information for the development of new materials. See Figure 2.3 for two examples of OCT-imaging in medicine and industry.



(a) OCT-image of a human retina, from [53].



(b) OCT-image of a multi-layered plastic foil, from [63].

Figure 2.3: Two typical applications for OCT-imaging. (a) Opthalmology: OCT-image of a human retina. (b) Industrial application: OCT-image of a multi-layered plastic foil.

3 Machine Learning

One of the typical Machine learning tasks is concerned with the automatic detection of complex patterns in data and has become very popular in the last decades. Nowadays machine learning is applied in several areas sucht as, e.g. : the advertising industry using personalized promotion, cameras detecting faces in images, anti-spam-software filtering our mails, smart-phones recognizing speech and finally cars driving automatically. Machine learning has become a common method in many different areas where the problem is to detect patterns in the data. On the one hand the tasks can be situations which are intuitively performed by humans such as speech recognition and detecting objects on images with their experience and for these the present machine learning programs reach good results under the assumption that they have enough training data to learn from. On the other hand machine learning is also frequently used in the analysis of very large and complex datasets like weather prediction, search engines or electronic commerce. With their almost unlimited memory capacity and computational power computers can perform tasks which are beyond human capabilities. Broadly speaking one can say that machine learning tries to obtain knowledge out of previous data with the goal to find structure in it or to predict future outcomes. But what means 'learning' for machines? Tom Mitchell stated 1997 in his book 'Machine Learning' [34]: *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."*

3.1 Preliminaries

In this Section we will briefly summarize some basic facts from machine learning following the notations and wordings as introduced in [16, 19, 42].

Supervised Versus Unsupervised Learning

As already stated before machine learning is about finding patterns in a given dataset. For this machine learning methods can be broadly categorized in two types: **Unsupervised learning** deals with a dataset which represents objects or items by several parameters, also called features, and tries to find relationships and structure within the dataset. The goal is for example to find the probability distribution which generated the data (density estimation), to categorize the data automatically in a (unknown) number of clusters (clustering) for the automatic organization of datasets or novelty detection to find atypical or novel points in a dataset (for example spam-mail detection). More detailed information about the mentioned applications can be found in [2].

Supervised learning on the contrary deals with a labelled dataset which means that for each input value or tuple the correct output value is given. The goal is to find a function which maps each input value or tuple to its corresponding target value and to use this function to predict the labels for unlabelled data in the future.

So roughly speaking, both methods observe several examples of a random vector x but unsupervised learning methods try to find the probability distribution $p(x)$ itself or interesting properties of that distribution whereas supervised learning methods observe additionally corresponding target values y and learn to predict y from x and therefore are concerned with determining properties of the conditional density $p(y|x)$. The terminology evolves from the idea that in supervised learning the target value is provided by some 'supervisor' who guides the learning process whereas in unsupervised learning the development has to be made without that guidance. In the following we will only focus on supervised learning tasks.

The Set-up

The domain set is the set X of objects for which we would like to predict some target value in the future based on a set of training data. Usually each instance $x^i \in X$ is represented as a vector of d features $(x_1^i, x_2^i, .., x_d^i)$ where a feature is a numerical or categorical value representing a certain property of the object. The target values, also called **labels**, are given in a set Y of possible values. If the label values are numerical the learning problem is called **regression task** and the desired function is given by $f^* : \mathbb{R}^d \to \mathbb{R}$. If the label values are k distinct classes it is called **classification** and the desired function is given by $f^* : \mathbb{R}^d \to \{1, .., k\}$ (assuming numerical features). Supervised learning methods need as an input a set of labelled data, i.e. , a set $S = \{(x^i, y^i) \mid x^i \in X, y^i \in Y)\}$, and try to find the mapping $f^* : X \to Y$ with $f^*(x^i) = y^i$ for all $(x^i, y^i) \in S$.

For the estimation of generalization abilities the given data set is split into a set of n training samples T and m test samples Z. The training data is used during training to find the model,

i.e. , the pairs $(x^i, y^i) \in T$ are used to obtain an estimate for the true mapping f^*. The test data is needed afterwards to evaluate the trained model on unseen data samples to get an estimate about the prediction abilities of the obtained model. The randomness of the data splitting is very important and the sets should be absolutely independent and identically distributed (i.i.d) (see also Section 'Data Splitting').

The Loss Function

The loss function is also often called **objective function**, **error function** or **cost function**. Assume that f is the mapping between the input and output values which has been estimated from a training set T, so that $f(x)$ gives the predicted output value $\tilde{y}$ for an input x. Then the loss function $L(y, f(x))$ measures the error which occurs for this data sample (x, y) where y is the real target value for x and $f(x)$ is the target value predicted by the trained model. Typical choices for loss functions in regression problems are

$$L(y, f(x)) = (y - f(x))^2, \qquad \text{(squared error)}$$

$$L(y, f(x)) = |y - f(x)|, \qquad \text{(absolute error)}$$

and for classification

$$L(y, f(x)) = \begin{cases} 1, & \text{if } f(x) = y, \\ 0, & \text{else.} \end{cases} \qquad \text{(0-1 loss)}$$

The loss function is used for learning in the following sense: Assume the training set is given by $\{(x^i, y^i) \mid i = 1, ..n\}$. Then the **training error** is the expected loss over the set of training samples with underlying distribution $\hat{p}_{data}$ and therefore is given by

$$\overline{Err} = E_{(x,y)\sim\hat{p}_{data}}[L(y, f(x))] = \frac{1}{n}\sum_{i=1}^{n} L(y^i, f(x^i)) \qquad (3.1)$$

This error is also called **empirical error** or **empirical risk**. As the loss is a value for how far the obtained function f is away from the true function f^* the empirical error measures the quality of f on the samples in the training set. Having selected the training samples independently and identically distributed they should represent the properties of the whole dataset and therefore it makes sense to search for a solution f which minimizes the error on that data. But of course this solution is then biased with the samples in the training set as it is adjusted to exactly those data points and still an estimation for the match of f to new data drawn from the true distribution p_{data} is required. Assume that m test samples $Z = \{(x^{n+1}, y^{n+1}), .., (x^{n+m}, y^{n+m})\}$ independent of the training set are given. Then the expected generalization error is given by

$$Err = E_{(x,y)\sim p_{data}}[L(y, f(x))]$$

and, by the law of large numbers, can be approximated by the average loss over the test samples (i.e **test error**)

$$Err \approx \frac{1}{m} \sum_{i=n+1}^{n+m} L(y^i, f(x^i))$$

for $m \to \infty$ and $(x^i, y^i) \in Z$. As the test error is estimated for new data samples which were not used for training, the test error is usually higher than the training error which is measured for the samples used to fit the model. How well a machine learning algorithm will perform depends on its ability to make the training error and at the same time the gap between training and test error small. This leads to the bias-variance trade-off and the problem of under- respectively overfitting of models.

The Bias-Variance Trade-Off

One important choice regarding the model is its complexity. For an example see Figure 3.1 which shows a series of points generated by a sinus-function which is approximated by polynomials of different orders i.e. different complexity. There were generated in total 17 points in the range of -50 to 50 and only the half of them (black) was used for fitting the polynomials. The minimal squared error reached is 4.16 for the first-order polynomial, 1.94 for the third-order polynomial, 0.03 for the 6th-order polynomial, 0 for the 12th-order polynomial and also 0 for the 18th-order polynomial. Afterwards the error was measured with respect to the other half of the points (red) and the minimal squared error reached is now 4.02 for the first-order polynomial, 1.75 for the third-order polynomial, 0.09 for the 6th-order polynomial, 0.19 for the 12th-order polynomial and 1.29 for the 18th-order polynomial. It can be seen that with increasing complexity (represented by the increasing number of parameters defining the polynomial of a certain degree) the error on the data used for fitting (training data points) decreases. The polynomials of order higher than the number of data points are able to fit the points exactly and therefore reach zero error. But the polynomials which reach a minimal error of zero on the training data points do not reach the minimal error on the test data points and this is due to overfitting: the higher complexity allows the polynomials to fit the training data exactly but it may lead to curves which oscillate in-between them and therefore reach higher error on new data (test data). Note that the range of x-values is chosen quite small. For bigger ranges the error would be much bigger for both training and test data. To fit to all different tasks the model needs to have a certain complexity and the problem is the following: Allowing a high number of parameters of the model (and therefore a high complexity) tends the model to (exactly) fit the training data causing the training error to decrease towards zero. But in this case at some point the model fits too much to the given data (**overfitting**) and from that point on the generalization abilities of the model are

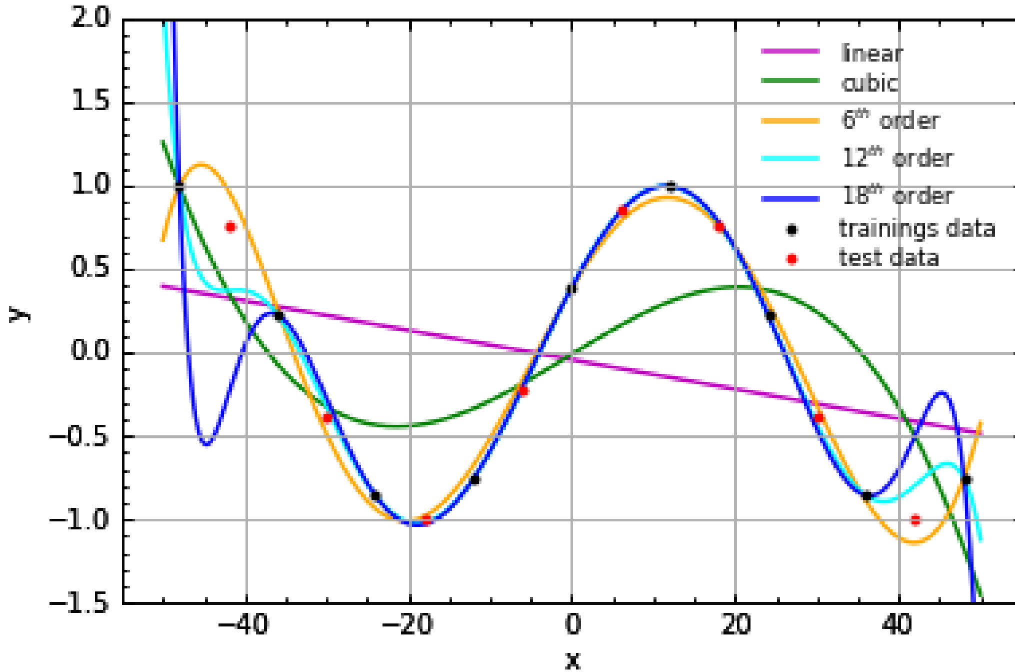


Figure 3.1: This graph shows a series of points (black and red) generated by a sinus function which is approximated by polynomials of different orders. The polynomials are 1st-order (magenta), 3rd-order (green), 6th-order (orange), 12th-order (cyan) and 18th-order (blue).

getting worse and therefore the test error is increasing again. On the other hand if the model complexity is too low the model is not able to fit the data well. This can be seen as a high training error (**underfitting**). As in this case the obtained function f can not fit properly to the wanted function f^* also the test error can not get low, see Figure 3.2.
This can also be examined in a mathematical way [19]: Since $y = f(x)$ and f is just an approximation for the true function f^* we assume $y = f^*(x) + \epsilon$ with $E(\epsilon) = 0$ and $Var(\epsilon) = \sigma_\epsilon^2$. The bias term is defined as a measure for the distance between the average of our estimates and the true mean. Then the expected prediction error of a function $f(x)$ for a given sample $x = x_0$ using the squared-error loss is:

$$
\begin{aligned}
Err_{x_0} &= E[(y - f(x_0))^2 \mid x = x_0)] \\
&= \sigma_\epsilon^2 + [E(f(x_0)) - f^*(x_0)]^2 + E[f(x_0) - E(f(x_0))]^2 \\
&= \sigma_\epsilon^2 + Bias^2(f(x_0)) + Var(f(x_0)) \\
&= \text{Irreducible error} + \text{Bias}^2 + \text{Variance}
\end{aligned}
$$

The first term is the irreducible error and can not be avoided. The second term is the squared

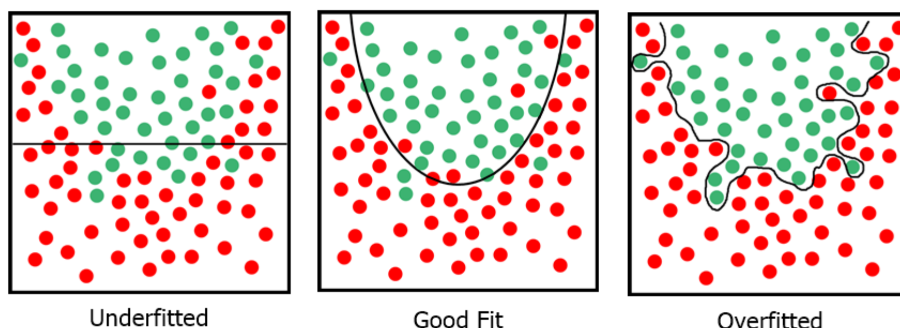


Figure 3.2: Under- and overfitting demonstrated by functions f with different complexities. In the left figure the function f is a constant ($f = c$) resp. linear ($f = a * x + c$) function and therefore it depends just on one resp. two parameters. Since the function can not separate the data well it seems to contain too less information for the given problem. On the other hand in the right figure the function f is of very high degree and therefore is determined by a large number of parameters. Since this function separates the given data very well but may not reach that good results for new data points it seems to have too much data dependence because of its high complexity. In contrast the quadratic function shown in the middle figure seems to be most likely to give a good representation of the problem.

bias which measures the distance between the average of our estimates and the true mean. The last term is the variance of the model at x_0, i.e. the squared deviation of the function $f(x_0)$ around its mean. As learning is concerned with minimizing the training error and at the same time minimizing the gap between training and test error the goal is to optimize bias and variance simultaneously, i.e. , to find the model complexity which obtains the optimal bias-variance trade off. This property can be seen in Figure 3.3.
Another decision which highly affects over- and underfitting is the proportion of samples in training and test sets. As the training set gives partial information about the distribution of the whole data it follows that the larger the training set is the more likely it is to reflect more accurately the real distribution. But on the other hand the goal of learning is the minimization of the generalization error and its approximation gets better for a larger test set. Therefore, one way to reduce overfitting is to use more training data. But the generation of data for real-world applications is oft not that easy as it can be time or money consuming. Therefore a simple alternative which also prevents overfitting is the use of regularization strategies, see Chapter 3.2.3.

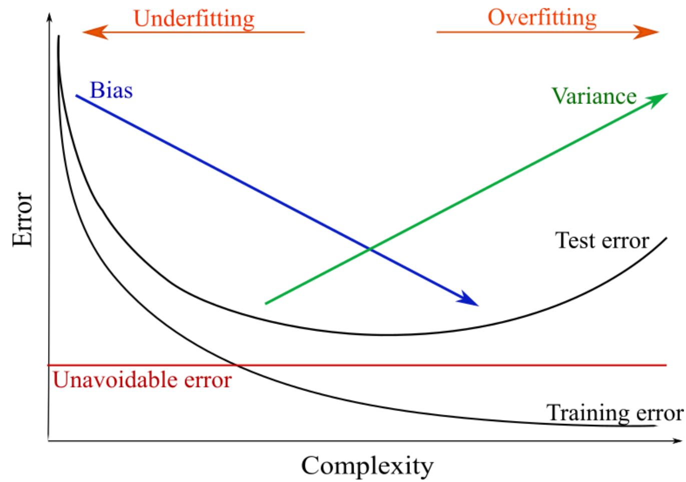


Figure 3.3: The bias-variance trade off.

Performance Measures

In order to compare the strength of different model architectures to each other some performance measures are needed. A common choice is to use some loss function to determine the strength of a model, this is often done for regression tasks. For classifiers one method to visualize the performance is the **confusion matrix**, see Figure 3.4. Note: The number $\#_{i,j}$ states how many samples of class i were classified as class j. Therefore all $\#_{i,j}$ with $i = j$ (e.g. all numbers in the diagonal of the matrix) are the correctly classified samples.

Figure 3.4: Confusion matrix for a classification task with k classes $c_1, .., c_k$.

From the confusion matrix one can easily compute the **accuracy**. Assume that the number of given samples is n and the number of classes is k. Then the accuracy measures the proportion

of correctly classified samples and can be computed by

$$\text{accuracy} = \frac{\sum_{i=1}^{k} \#_{i,i}}{\sum_{i,j=1}^{k} \#_{i,j}} = \frac{1}{n}\sum_{i=1}^{k} \#_{i,i}.$$

For a binary classification task (with classes +1 and -1) the confusion matrix just contains the following values, see Figure 3.5:

- True Positives (#TP): number of samples classified as +1 and actually being +1
- False Positives (#FP): number of samples classified as +1 but actually being-1
- False Negatives (#FN): number of samples classified as -1 but actually being +1
- True Negatives (#TN): number of samples classified as -1 and actually being -1

Correct class \ Predicted class	+1	-1
+1	#TP	#FN
-1	#FP	#TN

Figure 3.5: Confusion matrix for a binary classification task. Again the elements in the diagonal are the numbers of correctly classified samples.

With these values other performance measures like **precision**, **recall** or **F-score** can be easily computed. For a multi-class task it is not that straightforward, but is is possible to compute the values of #TP, #FP, #FN, #TN for each class against all other classes individually.
An important task with performance measures is the choice of the right scale for the given task. If for example the task is to classify between uniformly distributed classes then the accuracy is in most cases an appropriate measure for the performance of the model. But if for example the model should detect a rare disease the samples will be far away from being uniformly distributed. Assume the dataset includes 99% negative samples and just 1% positive samples (with regard to finding the disease or not) then a classifier which returns always 'negative' for every input reaches an accuracy of 99% but in practice does nothing of interest. In this case other performance measures will give better estimates of the actual

performance of the model. Further information can be found in [45]. Since there are no classes as output and therefore no straightforward ’true’ or ’false’ naming for regression tasks all the performance measures based on the values of #TP, #FP, #FN, #TN just refer to classification. For information about common performance measures for regression tasks see [4].

Turning back to the statement of Tom Mitchell: *“A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.”* We may say: In supervised machine learning there is given some learning task T to solve which (internally) means to obtain the best approximation f for the function f^* describing the mapping between input and output. The experience E is given by the set of labelled data and P is some suitable performance measure like accuracy or loss. Then the more samples (experience) are given the better will the performance for the task within the meaning of the performance measure be.

Data Splitting

For a supervised machine learning task the set of given samples $(x, y) \in X \times Y$ has to be split randomly into three disjoint subsets. The **training set**, like already stated before, is used for fitting the model, the **test set** is used to evaluate the model for getting an estimate of its generalization abilities. The test set is not used during the training phase and gets involved only after the selection of the final model. Suppose the test set is used for model selection, i.e. choosing always the model with the lowest test error, then the model is somehow adjusted to the test data. In that case computing the estimate of the generalization error based on the test data may be better than it would be for new and unseen data and therefore it may underestimate the true generalization error badly. Therefore also a **validation set** is needed which is used to select the best model out of different machine learning algorithms, model architectures or parameter-settings. There is no rule on how to choose the ratio between these three sets but a typical split may be to use 50% of the data for training and 25% for validation and testing each, see also Figure 3.6.

Cross-Validation

Cross-validation provides a means to randomize the splitting of the data into training, validation and test set in order to approach i.i.d. samples in each data set. This is in particular of importance for small datasets due to the following reasons: We want to have the training set

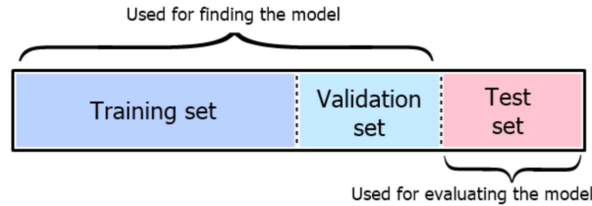


Figure 3.6: The whole set of given labelled data is divided into three disjoint sets. The splitting needs to be random and the samples should be independent and identically distributed.

as large as possible to reach a good approximation for the true function f^*, but if the validation set is too small the performance measured on this data will probably depend too much on that certain selection and if the test set is too small the estimate for the generalization error may be very bad. Also the selection of the data in the sets - randomly or not - could be in such a way that the data in the training set is for example very easy to classify and the data in the validation or test set contains unluckily many hard cases (or the other way around) which will of course give no good model or no good estimate for the model performance. One way to handle this problem is the use of cross-validation where the complete data set is split into folds and then trainings are performed, e.g. , with always one fold put aside. Getting a better estimate of the generalization error can be done in the following way: The whole dataset is split into k folds in advance, then for $j = 1, .., k$ the jth fold is put aside as test set. The rest of the data without this test set is then split into training and validation set. After finding the best model architecture and parameters for each data-split $j = 1, .., k$ with training on the training set of the split excluding fold j and evaluating on the validation set of the split excluding fold j the exact models should be evaluated on the appropriate test sets. The average of the performance scores can then be seen as the estimate for the generalization abilities of the chosen machine learning method for the given kind of data. Therefore, cross-validation performed like that is used to compute an unbiased estimate of the generalization error. But of course as cross-validation requires to run the whole training and evaluation process k times it does take k times longer to get results.

3.2 Neural Networks — Deep Learning

Deep learning is a machine learning method which deals with deep artificial neural networks. As the name indicates artificial neural networks are inspired by the structure of neural networks in the brain where single neurons are connected to each other and pass information on to each other.

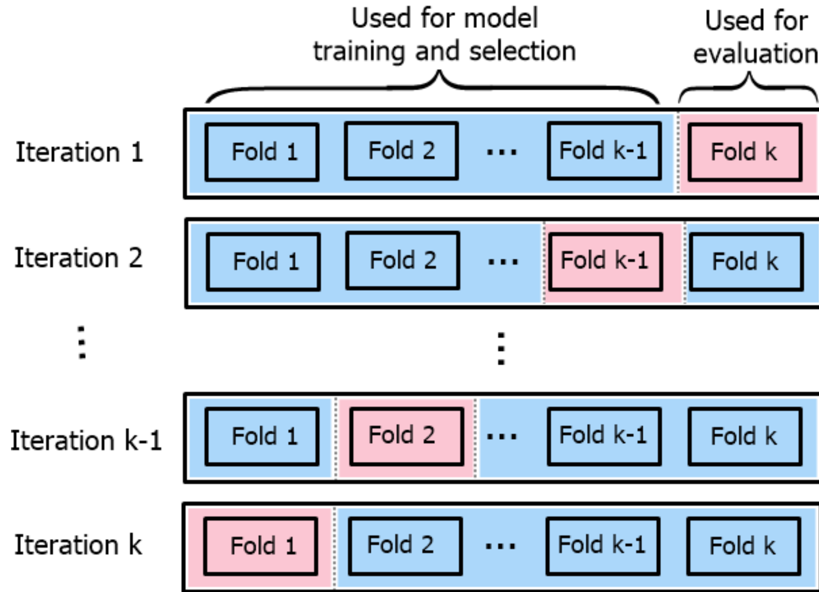


Figure 3.7: k-fold cross-validation. In this setting the data set is split into test, training and validation sets to be able to evaluate different model architectures or parameter settings.

3.2.1 The Structure of Neural Networks

In this Section we introduce some basic notations for neural networks (compare also [16, 1]). In artificial neural networks the neurons (**units**) are arranged in **layers** and each layer takes the output of the former layer as an input and passes it forward to the next layer. The number of units per layer is called the **width** and the number of layers is called the **depth** of the network, so deep learning uses neural networks with several layers. The first layer is called the **input layer** and the last one is the **output layer**, the intermediate layers are called **hidden layers**. For the purpose of passing the inputs forward **activation functions** are needed to decide what should be the output of the neuron given a certain input. These activation functions operate as thresholds to decide whether a neuron should 'fire' or not. A common activation function for the input layer and hidden layers is for example the **ReLU-activation** which gives for the input x of the neuron the output $g(x) = max(0, x)$, see Figure 3.8. For the output layer of a classification problem with c classes it is common to use the **Softmax-activation** which converts c real-valued predictions $v_1, .., v_c$ into output probabilities $o_1, .., o_c$ by

$$o_i = \frac{e^{v_i}}{\sum_{j=1}^{k} e^{v_j}}, \quad i = 1, .., c.$$

To obtain the input for a neuron in the kth layer the outputs of all connected neurons of the $(k-1)^{th}$ layer are each multiplied with a **weight** and then summed up. Then the activation

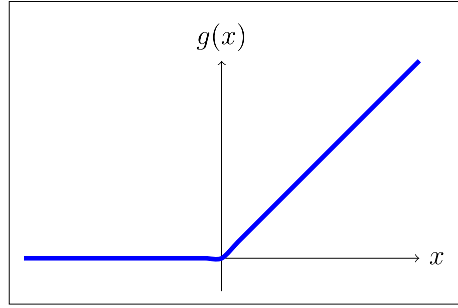


Figure 3.8: The rectified linear unit activation function (ReLU) given by $g(x) = max(0, x)$.

function gets applied to this weighted sum and the obtained result is then the output of the neuron in the kth layer which is again a part of the input for a neuron in the $(k + 1)^{th}$ layer. Therefore, the weight of the connection determines how much a certain neuron in the current layer affects a certain neuron in the next layer. The input of the first layer is the input of the classification problem i.e. a vector x in the domain set X. The output of the last layer yields the predicted label y from the set of possible labels Y. The goal of machine learning – as stated before – is to find a function f which gives the best approximation for the true function f^* describing the mapping between X and Y. In neural networks this approximation function $f(x)$ can be seen as a composed function $f(x) = f^l(..(f^2(f^1(x)))$ where f^1 describes the first layer of the network, f^2 the second layer and so on assuming there are l layers. Each of the subfunctions f^j is determined by the values of the weights w^j and therefore we write $f(x)$ as $f(x; w)$ where w represents the weights $w^1, .., w^l$ of all layers and neurons. For a visual outline of such an artificial neural network see Figure 3.9 and 3.10. For an activation function φ and the representation of the weights of the connections between layer k and $k+1$ by a weight matrix W^k yields

$$h^k = \varphi((W^k)^T * h^{k-1}) \tag{3.2}$$

for the output h^k of the kth layer. In this notation the value $W^k_{i,j}$ (which is the element in the ith row and j^{th} column of the weight matrix W^k) is the weight connecting the ith neuron in the kth layer with the jth neuron in the $(k + 1)^{th}$ layer.
Remark: It is also common to add a bias term b^i in each layer $i = 1, .., l$, then e.g. (3.2) turns into

$$h^k = \varphi((W^k)^T * h^{k-1} + b^k).$$

3.2.2 Learning

To train an artificial neural network the set of given samples $(x, y) \in X \times Y$ has to be split randomly into three disjoint subsets [42]. Like already stated before the **training set** is used

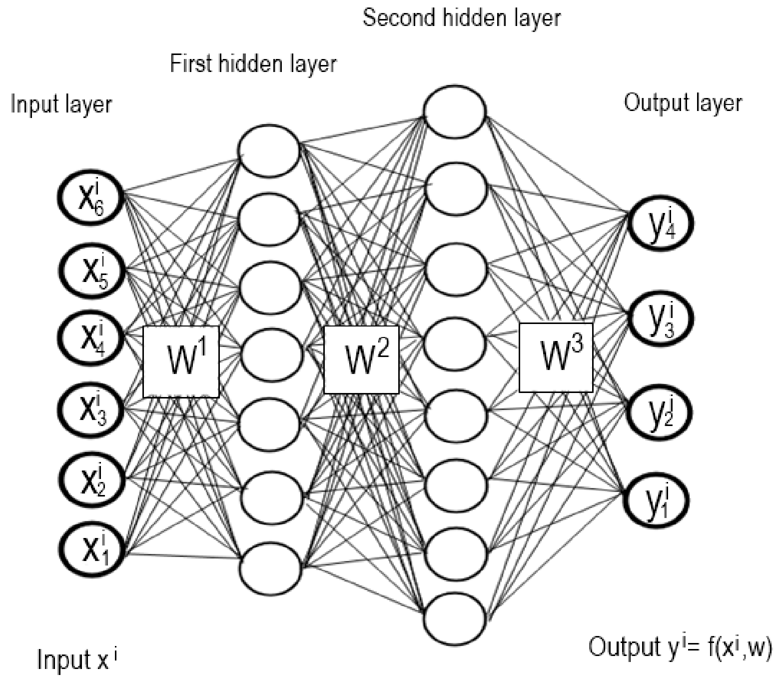


Figure 3.9: Schematic of an artificial neural network with two hidden layers. It has 6 units in the input layer, 7 units in the first and 8 units in the second hidden layer and 4 units in the output layer. The input for the first layer is given by the input values $x_1, .., x_6$ which can be seen as vector $x \in \mathbb{R}^6$ and the weights can be represented by a weight-matrix, e.g. $W^1 \in \mathbb{R}^{7\times 6}$.

for fitting the model, the validation set is used for parameter selection and the **test set** is used to evaluate the model in the end to get an estimate of its generalization abilities.
Starting with random values the network tries to adjust the weights such that the training error for the samples in the training set as defined in Equation (3.1) gets minimized in each training step. There are different functions to use for the loss depending on the application. For regression it is common to use the squared error or the absolute error, for a binary classification the 0-1 loss. For multiclass-classification when dealing with probabilities as network-output a common choice is the **cross-entropy** which measures the distance between the ground-truth distribution and the predictions. By propagating an input sample x^i through the network (**forward propagation**) a prediction label $f(x^i, w)$ is obtained and then compared to the true label y^i to get the loss. To get a good approximation f of the true function f^* the aim is to minimize the loss using the following idea: Let L be the loss function. A set of weights w satisfying the equation

$$\frac{\delta L}{\delta w}(y^i, f(x^i, w)) = 0$$

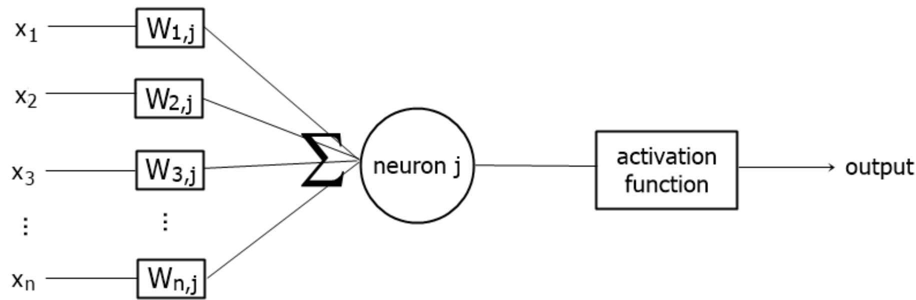


Figure 3.10: Detailed view of one neuron.

minimizes the loss for the given sample (x^i, y^i). Therefore a set of weights w satisfying the equation

$$\sum_{i=1}^{l} \frac{\delta L}{\delta w}(y^i, f(x^i, w)) = 0$$

minimizes the loss for the whole training set $\{(x^i, y^i) \mid i = 1, .., n\}$. To obtain the gradients a computationally efficient method is **back propagation** which uses the chain rule of derivation to compute all required partial derivatives in linear time, see Algorithm 1. These derivatives are used to make small adjustments to the weights with **optimizers** like **stochastic gradient descent** or **AdaDelta** (see Section 3.2.4 for details). In general, the learning process is the modification of the weights with a learning rate γ in some way similar to

$$w \leftarrow w - \gamma \frac{\delta L}{\delta w}$$

Such an updating of the weights for each sample in the training set is called a **training epoch**. This can be done in two different ways concerning the input data: **online learning** updates the weights for each training sample individually whereas **batch learning** updates the weights just once for a certain batch of training samples. Following [25] using mini-batches instead of each sample at a time the gradient over the loss gives a better estimate of the gradient over the whole training set which improves for growing batch size. Also the computation can be much cheaper and therefore the batch size is an important parameter for the training of neural networks.

After each epoch the model is evaluated on the validation set to get an estimate for the general performance of the model. Training stops after a given number of epochs or if the 'best' performance on the validation set is reached. Since the satisfaction with the model performance depends on the application it is needed to choose a **metric** e.g. **accuracy** to be able to obtain the settings for the best model.

With this practice deep learning models are able to learn from very complex datasets and make predictions for new data based on this knowledge, i.e. prior labelled data sets.

Algorithm 1 The Backpropagation Algorithm for computing the gradients for updating the weights. (Compare also [42].)

Input:
Sample (x, y), weight vector w, differentiable activation function $\sigma : \mathbb{R} \to \mathbb{R}$
Initialization:
Assume the layers are given as $V^0, .., V^T$ where $V^t = \{v_1^t, .., v_{k_t}^t\}$ is the tth layer of the network containing k_t units
Define $w_{i,j}^t$ as the weight connecting the units v_j^t and v_i^{t+1}
Forward pass:

Set $o^0 = x$
for $t = 1, .., T$ **do**
  for $i = 1, .., k_t$ **do**
    set $a_i^t = \sum_{j=1}^{k_{t-1}} w_{i,j}^{t-1} o_j^{t-1}$
    set $o_i^t = \sigma(a_i^t)$

Backward pass:

Set $\delta^T = o^T - y$
for $t = T-1, T-2, .., 1$ **do**
  for $i = 1, .., k_t$ **do**
    $\delta_i^t = \sum_{j=1}^{k_{t+1}} w_{j,i}^t \delta_j^{t+1} \sigma'(a_j^{t+1})$

Output:
For the connections v_j^t and v_i^{t+1} between all units of the network set the partial derivative to $\delta_i^t \sigma'(a_i^t) o_j^{t-1}$

3.2.3 Regularization

In this Section we follow mainly the theory from [1] and [16]. Further references are given directly in the subsections.
As stated in [16] *"Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error."*. The bias-variance trade-off we saw in Chapter 3.1 is concerned with finding that model complexity with the optimal trade-off between bias and variance, e.g. between under- and overfitting. With in-

creasing complexity the training error of a model decreases and the test error also decreases at first but then starts increasing again. Therefore, for a low test error, which means high generalization abilities, the complexity of the neural network has to be regulated. To do so there are many so called **regularization** strategies which effectively restrict the parameter space. In the following we only discuss the ones used for the practical part of this thesis.

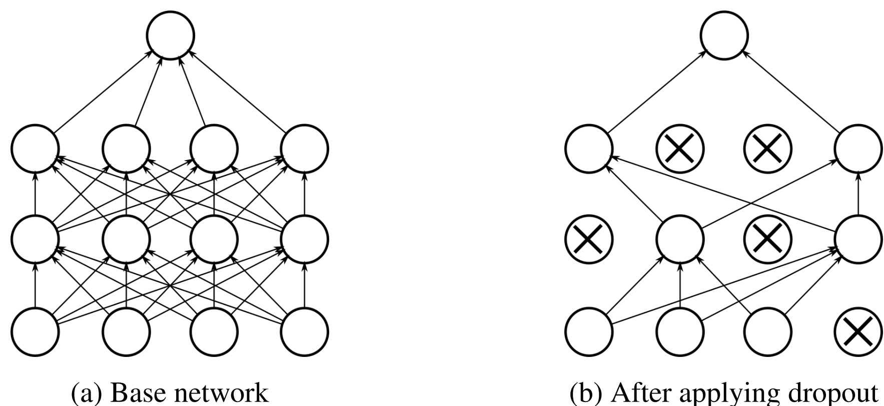
Early Stopping

A common method for regularization is **early stopping**. Here the gradient descent is stopped after only a few iterations. Without early stopping the number of training epochs performed is a fixed number and as training continues the model fits more and more to the training data. The optimal model complexity is reached exactly in that point where the test error starts increasing again and therefore it would be a good idea to stop training right there. If the training is done for a fixed number of epochs it can not be assured that this point is even reached or that it has been surpassed. Therefore the number of training epochs should not be fixed but what should be the criterion to stop training? As mentioned before, the optimal trade-off between over- and underfitting is reached exactly when the performance on the test set starts to get worse again. But using the test set while training to obtain that point would result in the test data affecting the choice of the model and therefore being not independent any more. Therefore the validation set is needed in order to find out the point with the optimal number of training epochs performed: We evaluate the model on the validation data after each training epoch and study the performance on that set hoping that it is an indicator for the performance on the test set. A user chosen performance measure then yields the early stopping point: Assume that accuracy is the selected performance measure. During training the validation accuracy will usually rise at the beginning and after some (unknown) time start to decrease again. In order to prevent the process from stopping too early it is common to wait a certain number of epochs afterwards (patience) to be sure that the performance decrease is unambiguously clear. If the validation accuracy does not rise again the optimal point is reached and training stops. In order not to get stuck with a nearly constant performance it is common to introduce a threshold parameter which defines the minimal improvement the performance measure needs to obtain to continue training.
A disadvantage when applying early stopping is that a copy of the best parameters and the best performance score has to be stored for the final selection but this cost is generally negligible. On the other hand, it has the additional advantage that with stopping the training early it reduces the computational cost of the training procedure. Since the size of the set of possible parameters for the model enlarges with the number of training epochs early stopping acts as a regularizer because it effectively reduces that size to a smaller neighbourhood within the

initial values of the parameters. As early stopping does not change the underlying training strategy and is easy to use, it is a very common and also very effective method to regularize neural networks and improve its performance.

Dropout

Another way to prevent the learning algorithm from overfitting would be the use of ensemble methods which means to train multiple models instead of only one and use a combination of their individual results to make a prediction (see also Chapter 3.3.1). Training and evaluating multiple neural networks is very disadvantageous concerning computational runtime and memory. **Dropout** is a technique which provides a way to approximate ensembling of exponentially many neural network architectures in an efficient and easy manner. More precisely, for neural network architectures, dropout denotes the method of simultaneously training an ensemble of neural networks that can be constructed by dropping out non-output units from that base network. Dropping a unit out means that it is temporarily removed from the network and therefore its incoming and outgoing connections to other units will be removed as it can be seen in Figure 3.11 and 3.12.



(a) Base network (b) After applying dropout

Figure 3.11: Applying dropout to a neural network. (a) Standard neural network with two hidden layers. (b) Network constructed by applying dropout using (a) as base network. The crossed out neurons have been dropped.

In most neural networks this can be done by simply multiplying the output value of the neuron with zero. It is necessary to define the probability for each layer with which the units of this layer will be temporarily removed. The decision which units will be dropped is taken on a random basis. To use the model afterwards for the prediction of new samples it is not necessary to evaluate all generated sub-models but to just use the base model with all units present. For all units the outgoing weights have to be rescaled respecting the probability of

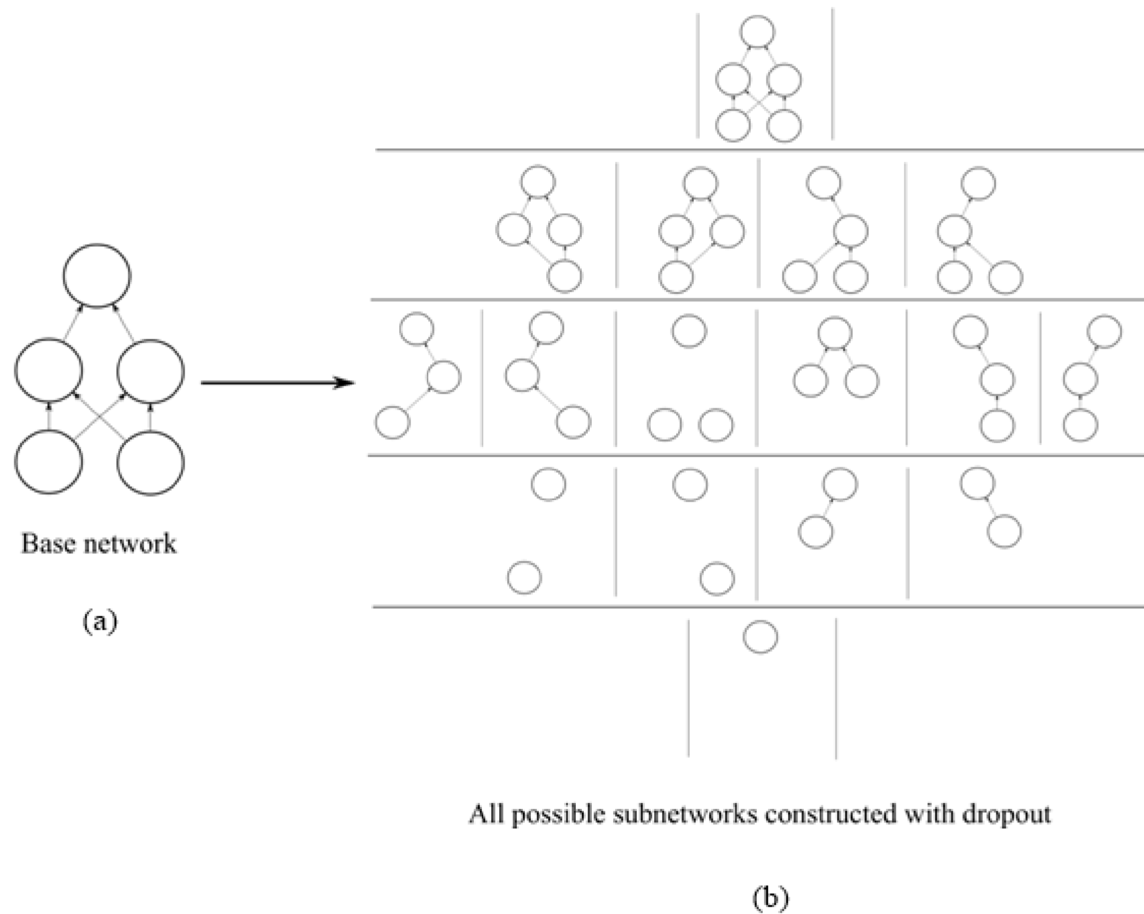


Figure 3.12: (a) The base network. (b) The networks which may be constructed by randomly dropping out one or more units from that base network.

not dropping that unit, this approach is called the weight **scaling inference rule**. Yet there is no theoretical proof for the accuracy of this approach for deep nonlinear networks but in practice it performs really well. Dropout is computationally very cheap but its application is not that effective if the set of training data is limited. As discussed in [46], using dropout to train a network and the weight scaling inference rule for its evaluation leads to significantly lower generalization errors for many different classification problems. Of course dropout may also be combined with other regularization methods to get further improvements.

Batch Normalization

Batch normalization is not primarily used for regularization but for the improvement of optimization and is motivated by some typical problems arising during the training of very deep

networks. One such problem is for instance the **vanishing gradient problem** which is due to the back propagation of the gradients: The repeated multiplications with derivatives of the activation function causes the magnitude of the gradients to decrease exponentially with successive layers and therefore the training process may slow down drastically [22]. Another problem is the **internal covariance shift** which is due to the fact that updating parameters during learning changes the distribution of the layer inputs to subsequent layers which has a negative impact on training convergence as the training data for later layers is not stable. Batch normalization on the one hand is able to reduce this effects but on the other hand it can also regularize neural networks. Further details about vanishing gradients, the internal covariance shift and the effect batch normalization has in that context can be found in [41].
The idea of batch normalization is to add so called **normalization layers** between hidden layers where each unit of such a layer contains two additional parameters β_i and γ_i to be learned during training. For the exact position of the normalization layer there are two different possibilities:

- Post-activation: the normalization is performed just after applying the activation functions to the linear combination of inputs, see Figure 3.13(a).
- Pre-activation: the normalization is performed directly to the linear combination of inputs and the activation function is applied afterwards, see Figure 3.13(b).

For our applications and in the rest of this chapter we shall focus on the second method as it is suggested by [25]. So let $v_i^1, .., v_i^m$ be the input of some unit i for all the samples $1, .., m$ in a specific batch (of batch size m). Then the normalization is done in the following way:

- At first calculate the mean μ_i and the standard deviation σ_i (plus some small $\epsilon > 0$) of these samples for the ith unit, so

$$\mu_i = \frac{1}{m}\sum_{r=1}^{m} v_i^r$$

  and

$$\sigma_i^2 = \frac{1}{m}\sum_{r=1}^{m}(v_i^r - \mu_i)^2 + \epsilon$$

- Then use this values to normalize the unit inputs for all $r = 1, .., m$ samples in the current batch:

$$\hat{v}_i^r = \frac{v_i^r - \mu_i}{\sigma_i}$$

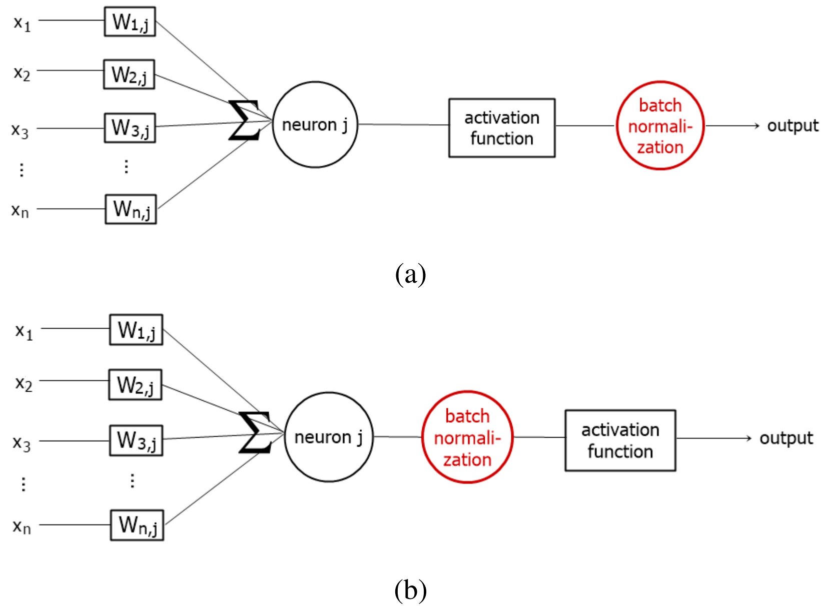


Figure 3.13: Batch normalization applied to (a) post-activation values or (b) pre-activation values.

- Now use the parameters β_i and γ_i to obtain the pre-activation output of the unit i for all $r = 1, .., m$ samples in the current batch:

$$a_i^r = \gamma_i v_i^r + \beta_i$$

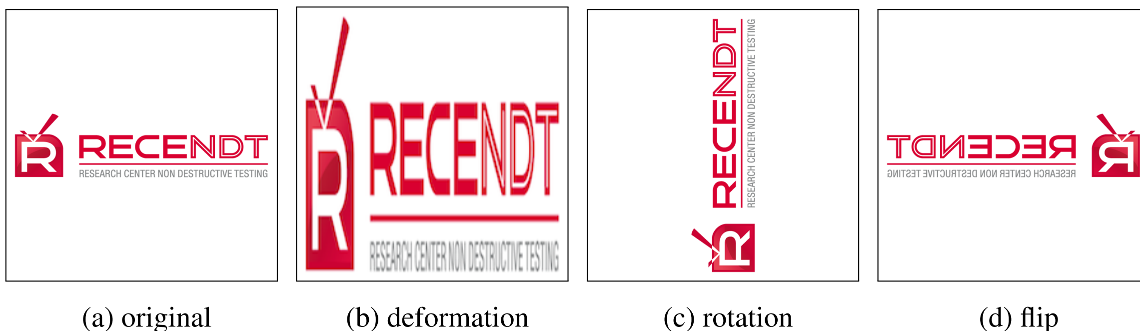
- After these steps the activation function is applied to all a_i^r.

Remark: In the case of bias parameters b, these can be ignored for the application of batch normalization as their effect will be cancelled out by the subtraction of the mean.
The ϵ which is added to the standard deviation is a very small positive value to avoid zero division in cases where all v_i^r are the same. As normalizing the mean and standard deviation of a unit may reduce the expressive power of the network the parameters β_i and γ_i are used to maintain these aspects and they need to be optimized during training. For the use of batch normalization in an artificial neural network this procedure is applied to all units in the network. With the introduction of the new layers which control the mean and the variance of the distribution of the network input this technique can stabilize the distribution of the layer inputs and therefore improve the training of neural networks [41]. To be able to test the network on single samples from the test set without the need of batches it is common to compute the values of μ_i and σ_i in advance by using the whole set of training data and treat them as constants during testing.

Since a particular training sample can cause different weight updates depending on the current batch selection, batch normalization can be seen as a kind of noise added to the training process; and adding noise is a common way to prevent the model from fitting too much to the training data. Therefore, batch normalization can not just be used to speed up training and improve optimization but it also con be seen as a regularization strategy.

Data Augmentation

As already discussed before the performance of machine learning algorithms does increase with the number of training samples used. But since the generation of more data may be either time or money consuming a quite easy and effective method is the use of data augmentation. The idea is to use slightly modified versions of the given data in addition to the original one for training and therefore on the one hand get more training samples and on the other hand also obtain slightly translated input which may lead to model invariant to small modifications. For images as input the data augmentation can be performed in many simple ways: The input images may be flipped horizontally or vertically, a shift or a elastic deformation may be applied, the image may be rotated by a certain degree, be cropped or also shaded with a hue. In Figure 3.14 some image augmentation methods are demonstrated. Since for each input image a duplicate is obtained in that way a dataset of size n can be extended to a dataset of size $2 \cdot n$. For further information see also [37].



(a) original (b) deformation (c) rotation (d) flip

Figure 3.14: Three different data augmentation techniques applied to an image. (a) shows the original image, (b) shows a deformed version of the image, (c) shows the 90° rotated image and (d) shows the horizontally flipped image.

3.2.4 Hyper-Parameter Tuning

We briefly summarize the strategy of hyper-parameter tuning as described in [1] and [16].

The work-flow for machine learning usually comprises the following steps: After selecting an appropriate machine learning method the next important step is to choose a performance metric which is useful for the given task (see Section 3.1) to be able to get an estimate for the abilities of a certain model. Then a decision about the model architecture and parameter setting for the first training with the given data has to be made. If possible a good practice is to find an already successfully solved task which is similar to the task at hand and start with the best-performing model as default baseline model. Then the validation set allows to estimate the generalization abilities of the model after training with the previously chosen performance measure. This value then is used to compare different machine learning architectures and parameter settings to single out the one which performs best on the validation data and therefore will be likely to also perform well on the independent test set. The different model architectures and parameter setting will be determined by the values of the **hyper-parameters** and the parameters which can be tuned are dependent of the machine learning method used. For neural networks common parameters to tune are:

- **Number of layers and units:** The number of layers and units per layer is a very important choice for training a neural network. Too many layers or units yields too high complexity and therefore result in overfitting. But on the other hand having too less complexity will not provide a way to represent the data well enough and therefore results in underfitting. A common way is to first start with a model that just slightly overfits the data and then add regularization methods (see also [8]).

- **Regularization:** Adding regularization strategies to keep control of the model complexity is one method to reduce overfitting. Regularization methods can be used in different intensities. For example the higher the dropout rate the more units are dropped out during training and therefore the lower is the chance to overfit. See also Section 3.2.3 for other regularization methods.

- **Batch-size:** As already mentioned earlier the number of training samples used at once for updating the weights of a model is also an important parameter. In [26] it is stated that in practice larger batch sizes may reduce the generalization abilities of the model. Of course increasing the batch size means using more of the training samples to compute the gradient and therefore getting a better estimate for it. But in [16] it is argued that most of the optimization algorithms converge faster if they quickly compute approximate estimates in comparison to slowly compute the exact gradient. Using a smaller number of samples at the same time also helps to reduce computational memory costs.

- **Optimizer:** Like stated before in Section 3.2.2 the optimizer is used to adjust the

weights of the network with the goal to minimize the derivative of the loss with respect to the weights. To do so many different optimization methods have been developed and it is still a topic of research. The target function for the optimization problem is the training error and the goal is to minimize it with respect to the weights of the neural network. But since the minimal training error may not lead to the model with the best generalization abilities due to overfitting machine learning does not completely follow the scheme of traditional optimization problems. Another issue with machine learning problems in comparison to traditional optimization problems is that the loss usually consists of a sum over all the training examples and in practice optimization algorithms for machine learning compute the weight updates according to the loss just over a subset of the training data, see item 'Batch-size' above. Optimization algorithms may use all the data samples at once or also just single samples but most algorithms are in-between and use mini-batches drawn i.i.d from the data to update the weights. With taking the average gradient of the examples in a mini-batch it is possible to get an unbiased estimate of the gradient. These methods are called **mini-batch** or **stochastic algorithms** and common examples are:

– **Stochastic Gradient Descent (SGD)**: The typical example for a machine learning optimizer. For each mini-batch of m samples drawn i.i.d from the underlying data distribution the gradient estimate is calculated by

$$g = \frac{1}{m}\Delta_w \sum_i L(y^i, f(x^i, w))$$

and then using the learning rate γ the update of the weights is performed by

$$w = w - \gamma g.$$

The learning rate is a very important parameter for the SGD algorithm and in practice it is common to use not a fixed learning rate but to gradually decrease the learning rate over time. As this algorithm can be slow there often is used the momentum method additionally to fasten the learning process. A variable v is introduced which gives the direction and speed at which the weights are updated and is set to

$$v = \alpha v - \gamma g$$

where $\alpha \in [0, 1)$ is a hyperparameter determining how quickly the decay of the previously contributing gradients is. Therefore the larger α is the more previous gradients influence the current direction. The weights are then updated by

$$w = w + v.$$

There are also special momentum methods, e.g. the Nesterov momentum which evaluates the gradient of the loss function after the current parameter v is applied to the weights. Therefore the use of Nesterov momentum may improve the convergence rate.

- **Adam and Adadelta**: It soon has been realized that the learning rate is one of the most important but also hard to set values. On the one hand too small values yield a very slow weight update and will take long to reach an acceptable loss but on the other hand too large values will update the weights too much and maybe an acceptable loss will never be reached. The idea is that parameters with large partial derivatives tend to be oscillating while parameters with small partial derivatives are more consistent but move in the same direction. Therefore, algorithms were developed which individually adapt the learning rates for different parameters. The Adam optimizer for example introduces biased first and second moment estimates of the historical gradients which are decayed in each iteration and then used to scale the weight update δw. The Adadelta optimizer alleviates the challenge of manually choosing a learning rate since it is set according to an exponentially decaying average of the squared gradients. For further reading about the Adadelta method see also [50].

Following [1] there are different kinds of methods for testing certain combinations of all kinds of hyper-parameters. The most common method is called **grid search**, here for each hyper-parameter which should be optimized a set of values is selected. Then for all possible combinations of those values a model is trained and then evaluated on the validation set to find the best setting. Of course this method has the big problem that by trying out many values for the hyper-parameters the number of possible combinations increases rapidly and therefore will take very long. Applying tricks like using bigger step-sizes in the parameter-values or sample them randomly with a certain distribution in a given range can help to reduce the search range in a fast way. Afterwards the set of interesting values can be scaled down to a smaller range and finer step-sizes can be chosen. Here a common method for certain hyper-parameters is also to sample not the values itself with a certain distribution but their logarithms. Following [16], there is also the possibility to manually tune the hyperparameters i.e. to re-set them manually after analyzing the model results. This of course requires a full understanding of the values and the relationship between the hyperparameters, the training and generalization scores and the computational resources and therefore is not very common to use.
To examine whether a model is over- or underfitting on the training data and how to continue the model selection process it is important to always keep a look at the difference between

the performance scores on the training and on the validation data. At first the performance on the training set should be observed. If it is poor the model is not complex enough to fit to the data and therefore the model complexity should be increased. If this does not improve the training scores then the quality of the data may be the problem. If it is too noisy or does not contain the features needed for the task then new and cleaner data should be collected. In the case that the performance on the training data is well enough it is time to study also the score of the validation set. If the performance is much worse here the model seems to be overfitting on that training data. Then possible actions would be on the one hand to remove layers or units or to add regularization such as dropout. On the other hand, it can also be useful to gather more data but in practice the possibility to do so strongly depends on the cost and feasibility of data generation.

Nested Cross-Validation for Parameter Selection

As already mentioned before cross-validation concerns the splitting of data into training, validation and test sets, see Chapter 3.1. Using cross-validation as described aims to get a better estimate of the generalization abilities of a model for the given kind of data. But it is also possible to use cross-validation in a nested way to additionally get a better estimate for the performance on the validation set which is important for the selection of the best hyperparameters: After splitting the data into k folds in advance and putting aside always one fold for testing, for each of the k different training-test data-splits the following is done: The remaining data is again split into k' folds and for $j' = 1, .., k'$ the j'^{th} fold is used as validation set and the remaining data is used for training. Doing this for allj' folds the validation performance is calculated as the mean of the obtained ones. This value can then be used for choosing the best setting of model parameters. After having chosen the best parameters for each of the j training-test splits the models can again be evaluated on the appropriate test set. Summarizing one can say that the inner cross-validation is for the hyperparameter selection whereas the outer cross-validation is used for computing an unbiased estimate of the generalization error.
The greatest advantage of the nested cross-validation is the better estimation of the generalization abilities of the underlying model and its hyperparameter search. Without using nested cross-validation for choosing the best hyperparameters the model will be biased to the dataset which can yield overly-optimistic performance scores. But on the other hand nested cross-validation is computationally very costly as it requires $k * k'$ trainings and as the splitting is done in a nested way, the dataset has to be big enough to still contain enough samples for training in the inner loop. Therefore it is useful to check for the need of cross-validation for the given task by analyzing the variance of results for different data splitting.

3.2.5 Convolutional Neural Networks

We briefly introduce some basic theory of convolutional neural networks following [1], [16] and [36].

Convolutional neural networks (CNN) are similar to traditional artificial neural networks but are primarily used for data with a grid-like topology like images as this input would easily overflow the memory and computational power if used in traditional networks. Assume there is given a 64×64 image with three color channels then the number of weights to be learned on a single neuron in the first layer would be given by 12288 $(64 \cdot 64 \cdot 3)$, and this is not even a big image. Therefore some kind of compressed or sparse representation of the input is needed. Like the structure of traditional neural networks also convolutional neutral networks are inspired by the nature and the idea was already motivated in [24] with the examination of neurons in a cats visual cortex. With images as inputs to the net, each layer of the convolutional network is three dimensional with the two image dimensions and number of color channels as the depth, which is for example 3 for RGB and 1 for greyscale images. Such networks consists of three different kinds of layers which are stacked to form a convolutional neural network: convolutional layers, pooling layers and traditional fully connected / dense layers. The input layer of the network will hold the pixel values of the image. For short a convolutional layer uses the so-called **convolution operation** instead of the matrix multiplication shown in Section 3.2.1 but also applies an activation function (usually ReLU) afterwards. The pooling layer is used to downsample the input to reduce the number of parameters to be learned. The dense layers are then used in the end with activations appropriate to the required output. See Figure 3.15 for an outline of a possible CNN.

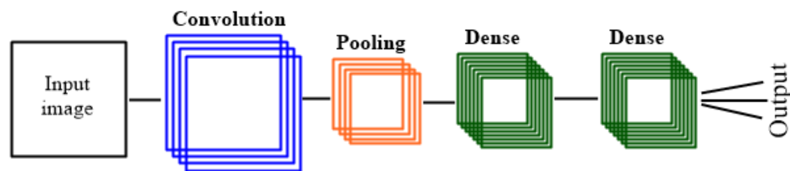


Figure 3.15: Using the image pixels as input and stacking a convolutional layer, a pooling layer and fully connected layers yields a basic convolutional neural network.

Convolutional Layers

As already mentioned before convolutional layers use the so-called convolution operation. In general, this is defined as

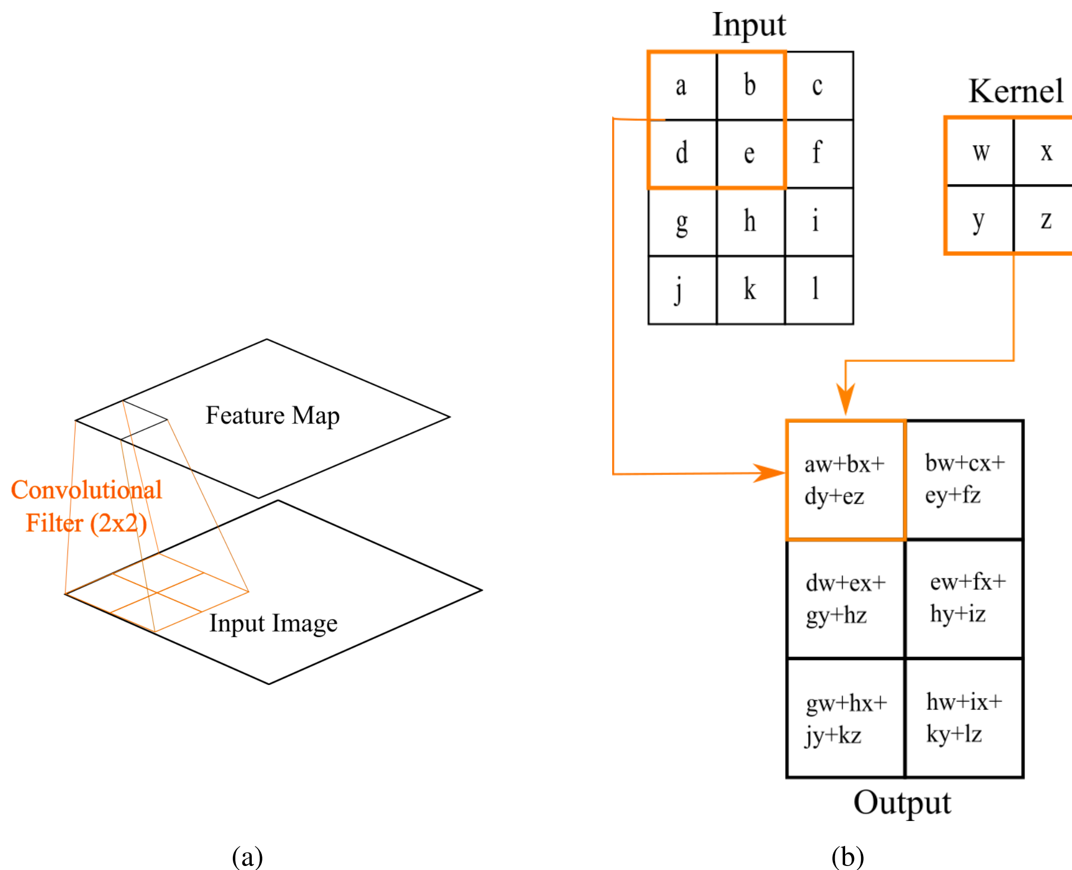
$$s(t) = (x * w)(t) = \int x(a)w(t-a)da$$

for two real-valued functions $x(.)$ and $w(.)$. For neural networks the first function represents the input which is usually a multi-dimensional data array and the second function is a certain filter, often called **kernel** which is usually a multi-dimensional array of parameters to be learned. The kernel has always the same depth and its spatial dimensions are usually squared (**kernel size**) and typically much smaller than those of the data array to which it is applied. The output is called a **feature** or **feature map**. To use the convolution operation for a finite number of points over more than one dimension (see Figure 3.16) as it is needed for example for a 2-dimensional image I as input (in this case the kernel K may also be 2-dimensional) with dimensions m, n the operation can be extended to

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

As it can be seen in Figure 3.16 the convolution operation basically places the filter over all possible positions in the image and and performs a dot product between the filter and the underlying image values. Consider again the number of weights which have to be learned for a given network layer: Assume again a three channel 64×64 pixel image. Applying a filter with kernel size 6×6, the number of weights for each neuron in the first layer will be $6 \cdot 6 \cdot 3 = 108$ instead of the 12288 required for fully connected layers. The output of applying the convolution operation to a data array is the so-called feature map or activation map. Each of the different filters operates as a pattern-detector and tries to detect a certain spatial feature in the image. For an example of a horizontal-edge detector see Figure 3.17.

Other filters will be able to detect other kinds of patterns in the image and therefore it follows that for a larger number of different filters the complexity of patterns detected increases. Concerning the already used term to apply the filter to all 'possible' image patches, this usually means to use just patches that do not cross the image border, as it is done in Figure 3.16. This will of course shrink the size of the data array in the output and to avoid this **(zero) padding** can be used. This technique basically adds a frame of zero values to the image and therefore makes it possible to maintain the size of the data array in the output, for further information see [1]. One can also define the **stride**, this determines the offset of consecutive image patches in order to decrease the overlap. For example in Figure 3.16 no padding is used and the stride is set to 1. When using convolutional layers an important choice is on the one hand the number of filters in a certain layer and on the other hand their kernel-size. As stated in [1] for efficient progressing the number of filters in a layer is often chosen to be a power of two and the kernel size is usually small like 3 or 5. Like fully connected layers, convolutional layers can also use bias values. The activation function used for convolutional layers is usually the ReLU activation as it was shown in [27] that this will fasten training and also increase the performance.



(a) (b)

Figure 3.16: An example for a 2-D convolution operation. The input can be seen as a 2-D image. (a) The convolution filter / kernel is then applied to all possible image patches to calculate the output. (b) illustrates how to apply the filter to all possible positions of the input image.

Max-Pooling Layers

After applying the convolution operation and the activation function usually another kind of layer is used for CNNs: the pooling layer. The pooling layer replaces the output at a certain position with some summary of the outputs of the neighbourhood. For example one common method is the max-pooling, where the maximal value of a given neighbourhood is used as new value, see Figure 3.18. Like in Figure 3.18 it is common to use 2×2 as pool-size for max-pooling layers and the stride is usually set to that pool-size, but it can sometimes also be useful to use slightly overlapping areas.
The goal of the pooling layer is on the one side to further increase the number of parameters to be trained but on the other hand it also helps to make the network translation invariant. This means that having a slightly translated input the output of the pooling layer do not change,

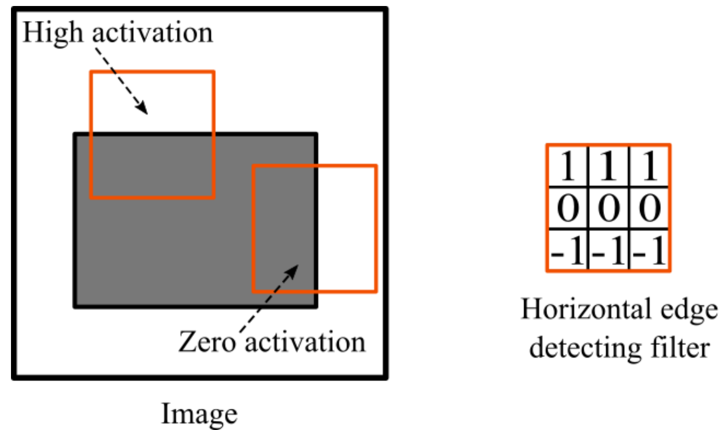


Figure 3.17: (a) The filter can be used to find horizontal edges in an image. Applying it to all possible image patches it yields the highest activation for perfectly horizontal lines and zero activation for perfectly vertical lines. For other edges the activation will be intermediate. With this method the output is a feature map of the image which shows the positions of horizontal edges in the image.

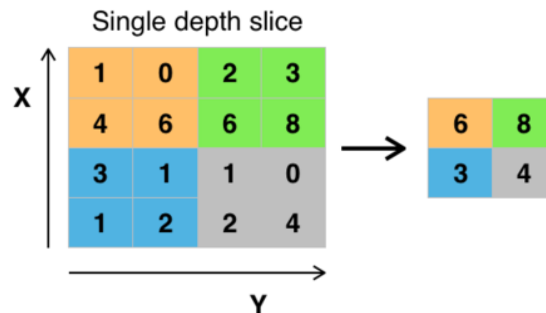


Figure 3.18: The max-pooling with size 2×2 takes the gives the maximal value of each 2×2 image patch as new output (from [61]).

see Figure 3.19.
Stacking these kind of layers several times creates different CNN architectures which determine how the layering could be. Some popular approaches are for example AlexNet [27], VGGNet [44] or ResNet [20]. Convolutional neural networks are one of the most successful and frequently used neural network types. They are used for image classification, object detection or even text recognition and became really popular as the winner of many ImageNet contests [40].

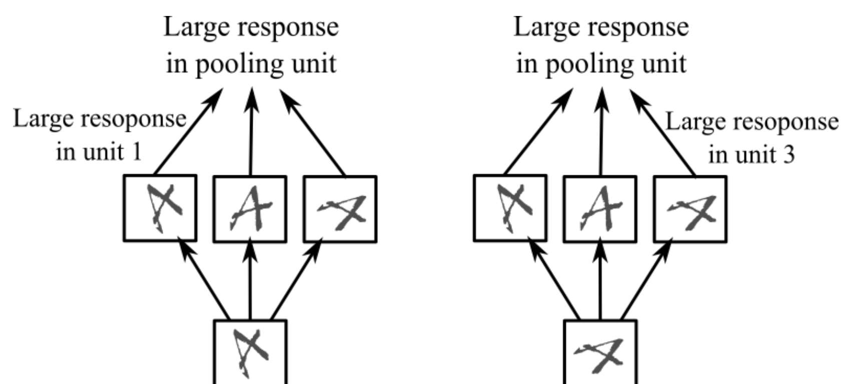


Figure 3.19: Max-pooling makes the network translation invariant.

3.3 Selection of Feature-Based Machine Learning Methods

To compare the results of deep learning to results from of standard machine learning methods we choose two common techniques, namely random forest and support vector machine. The main difference between these standard methods and deep learning is the input needed for training and classification. For deep learning the input usually is the raw data like time series or images. But for most of the other machine learning methods there is the additional need to manually extract features from the raw data and then input just the vector of features. Therefore, deep learning has the great advantage that the specificities of each class are extracted automatically from the raw data during learning and so the network determines and generates the features itself. On the other hand, it can never been said for sure that the network learns exactly what it should learn as there is no insight to the nature and variety of determined features. For example there could be a small dot in the left corner of each image belonging to a certain class due to some recording fault which does not appear on the images of the other class. Then the network may learn just the appearance of this point as indication for the first class and gives a high prediction score while the important features which really describe each class are not used for the decision. Because of this there is also an advantage in selecting the features manually and to have the opportunity to give the machine learning algorithm just those values which are important for the given classification task.

3.3.1 Random Forests

This Section is following [15] and [42] in the main ideas about random forests.

Random forest is an ensemble method. These methods are very common in machine learning and it means that instead of a single model, multiple such models are trained simultaneously; to make predictions the results of all these models are used (for example averaged or

with voting). Random forests consist of decision trees. A **decision tree** has a tree structure with a root node, in-between nodes and leaves and it classifies a sample x by travelling from the root node to a leaf. At each node in the tree the successive node is chosen with some rule based on the features of the training samples. A leaf contains a specific label which is then associated to the sample x. Decision trees are primarily used for classification but can also be extended for regression tasks.
During the training phase the tree is constructed. Let $S = \{(x^j, y^j) \mid j = 1, ..n\}$ be the set of n training samples where $x^j \in \mathbb{R}^d$ and $y^j \in \mathbb{R}^k$ i.e. each sample contains d features and there are k different labels given. Starting from the root node the successive nodes are determined by splitting the set of training samples S based on one or some of the features of the samples or on pre-defined splitting rules. A very common splitting rule for numerical features is to find a threshold for a single feature and for the successive nodes splitting the dataset according to that threshold. Assume we are given numerical features. For applying the threshold c to the ith feature the data splitting would deliver the subsets

$$S_L = \{(x, y) \in S \mid x_i < c\} \textit{ and } S_R = \{(x, y) \in S \mid x_i \geq c\}$$

For categorical features it is either possible to make a binary split i.e. check for the label c in the ith feature, obtaining

$$S_L = \{(x, y) \in S \mid x_i = c\} \textit{ and } S_R = \{(x, y) \in S \mid x_i \neq c\}$$

or to make a split according to the entire feature which means making a split for all possible labels of the ith feature obtaining

$$S_j = \{(x, y) \in S \mid x_i = c_j\}$$

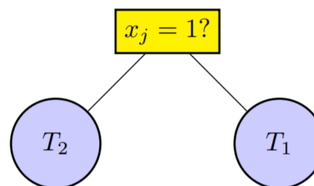
Compare also Figure 3.20. Note that the case of numerical features can be reduced to the case of binary features by defining a set of thresholds Θ s.t. for given sorted values $x_{1,i} \leq .. \leq x_{m,i}$ of the ith feature $\Theta_{j,i} \in (x_{j,i}, x_{j+1,i})$ and splitting like shown before.
In each step there are of course many different kinds of splits possible and the decision which split is performed is determined by some **gain** measure. For this all possible splits are examined and then the split with the maximum **splitting criterion** is performed. That means the data is divided into subsets according to the chosen split and then for all newly generated subsets which are not empty there is generated a new node and the procedure is repeated. One common algorithm for decision trees is the **Iterative Dichotomizer 3** algorithm, short **ID3 algorithm**. In the following pseudocode (Algorithm 2) we assume just binary features, i.e. $X = \{0, 1\}^d$. (This algorithm can be used equivalently for numerical features as they can be seen as binary features). The algorithm is recursive with the initial call ID3(S, [d])

Algorithm 2 The ID3 algorithm, following [42].

Input:

Training set S, feature subset $A \subseteq [d]$

if all examples in S are labelled with 1:
  return a leaf 1
end if
if all examples in S are labelled with 0:
  return a leaf 0
end if
if $A = \emptyset$:
  return a leaf whose value = majority of labels in S
else
  Let $j = argmax_{i \in A} Gain(S, i)$
  if all examples in S have the same labels:
    return a leaf whose value = majority of labels in S
  else
    Let T1 be the tree returned by ID3($\{(x, y) \in S : x_j = 1\}$, A j).
    Let T2 be the tree returned by ID3($\{(x, y) \in S : x_j = 0\}$, A j).
    Return the tree:



  end if
end if

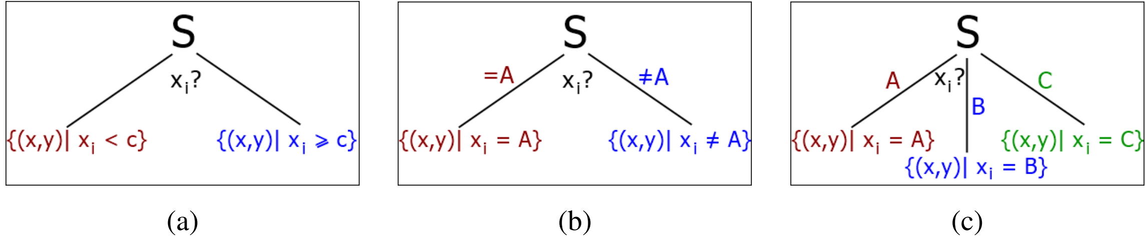


Figure 3.20: (a) For numerical features it is common to apply a threshold c to one of the feature values. For categorical features is is common to make either a binary split (b) or to make a split separating all possible feature values (c).

and it returns a decision tree for the set S of training samples. A possible realization of the procedure Gain(S,i) will follow afterwards.
To determine the best split for a certain state of the tree construction some **gain** measure is needed. For this we follow [5] and assume again binary features. Define an **impurity function** $i(t)$ which measures the goodness of a node t and reaches its maximum if the samples are uniformly distributed between the classes and its minimum if all samples belong to one class. So the smaller the impurity of a node, the 'purer' (regarding the different labels assigned to the remaining samples) is that node and therefore the goal is to minimize it with splitting. To measure the goodness of a split the decrease in impurity at node t for a split s is calculated as follows:

$$\Delta i(t,s) = i(t) - p_L i(t_L) - p_R i(t_R) \qquad \text{(binary split)}$$

$$\Delta i(t,s) = i(t) - \sum_j p_j i(t_j) \qquad \text{(split of entire feature)}$$

where t_L, t_R or t_j are the new nodes generated by the split s and p_L, p_R or p_j are the proportions of samples belonging to the new nodes. With $\bar{T}$ the current set of terminal nodes and $p(t)$ the proportions of samples in node t define the tree impurity $I(T) = \sum_{t\in\bar{T}} i(t)p(t)$. Now selecting the splits which minimize this is equivalent to selecting the splits which maximize the decrease in impurity $\Delta i(t,s)$. The most common impurity functions for classification are the following:

- the Gini criterion:

$$i(t) = 1 - \sum_{k=1}^{K} p(k|t)^2$$

- the Shannon entropy:

$$i(t) = -\sum_{k=1}^{K} p(k|t) log(p(k|t))$$

Here $p(k|t)$ is the proportion of samples labelled with class k in the node t and can be interpreted as an estimated probability of class c in node t.
Of course it would be possible to split until each sample has its own leaf, this would end in perfect scores on the training set. But the generalization abilities of the model could be poor as in this case the model may have overfitted to the data probably containing noise. Because of this there is the need to define some rules when to stop the further growing of the tree to find the optimal trade off between tree complexity and generalization abilities. Some common **stopping criteria** to prevent overfitting are the following [31]:

- Fix the value N_{min} to determine the minimal number of samples which need to be in a node. So if a node t contains less than N_{min} samples it is set as terminal node.
- Fix the value d_{max} which determines the maximal depth the tree can reach.
- Fix a threshold β and monitor the decrease in the impurity measure. If the total decrease $p(t)\delta i(s^*, t) < \beta$ for a node t with remaining sample set s^* then t is set as terminal node.
- Fix the value N_{leaf} to determine the minimal number of samples which need to be contained in a leaf. So if there is no split s.t. all resulting nodes will contain at least N_{leaf} samples the node is set as terminal node.

The strategy of early stopping for decision trees is often also called **pre-pruning.**
Another strategy to avoid overfitting for decision trees is **post-pruning** [42]. Here a much too large tree is grown and afterwards it is pruned. This is usually performed in a bottom-up walk through the tree where each node might be replaced with one of its subtrees or with a leaf with the goal to minimize the generalization error.
A reduction of the risk of overfitting can also be achieved by using many decision trees at once as an ensemble method [42]. A random forest is a classifier consisting of many individually constructed decision trees. The prediction for a new data sample is made by navigating it through each tree and in the end averaging the results. Random forests are not pruned.

3.3.2 Support Vector Machines

In this Section we will briefly summarize some basic theory about support vector machines following the notations and wordings as introduced in [42].
The **support vector machines (SVM)** are very useful and powerful machine learning techniques for learning a classification problem with high dimensional feature spaces. It is used primarily for binary classification but can also be extended to more than two classes.

The idea of SVM is the following: the $d-$dimensional sample features are represented in a $d-$dimensional space and the goal is to find a large margin separator. A separator is a hyperplane which separates the data such that all the samples are on the correct side of the hyperplane. As there may be many such hyperplanes the idea is to choose that one where the samples have the greatest distance to it i.e. the margin is large. So let $S = \{(x^i, y^i) \mid i = 1, ..n\}$ be the set of n samples where $x^i \in \mathbb{R}^d$ for d features given and $y^i = \{+1, -1\}$. Then the set of samples S is linear separable if there exists $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$ such that for all $i = 1, .., n$:

$$y^i = sign(\langle w, x^i \rangle + b)$$

which can be rewritten as

$$y^i(\langle w, x^i \rangle + b) > 0$$

The hyperplane separating the data samples is then given by

$$(\langle w, x^i \rangle + b) = 0$$

For any separable set of samples there are many separating hyperplanes and therefore the question is which one to choose, see Figure 3.21.

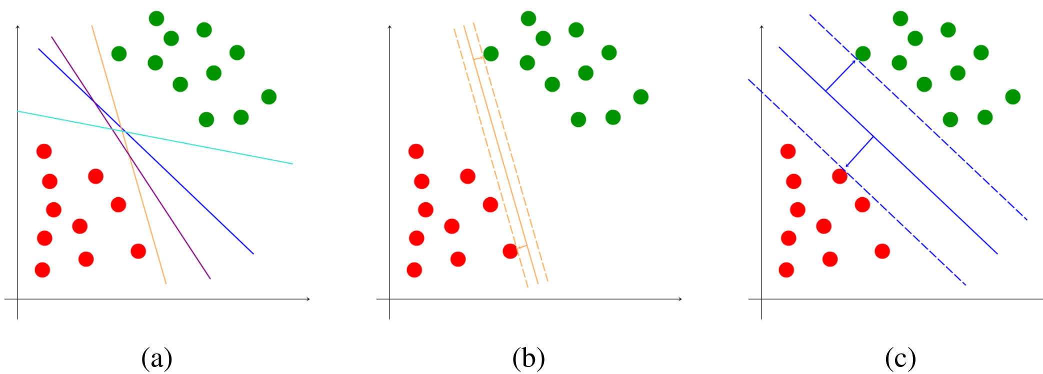


Figure 3.21: (a) There are usually many separating hyperplanes for a set of samples which should be classified to red, green. (b) and (c) show that there are hyperplanes with a smaller and larger distance to the points. Intutially one would choose the one with the larger distance, therefore (c), as that hyperplane which separates the data points better.

For this the **margin** is considered. The margin of a separating hyperplane is the minimal distance of the boundary function $(\langle w, x^i \rangle + b) = 0$ to all data points. Having a small margin means that slightly moving some data points may lead to a set which is no longer separated by the hyperplane. But the higher the margin the more likely it is to be able to still separate the data after slightly displacing some points. In terms of the classification problem this

means that with data points which are different from the training samples the probability to have them on the right side of the hyperplane (i.e. in the correct class) increases for a larger margin as they can vary more. For the next steps we follow [2].

Claim. *The distance between a point* x *and the hyperplane defined by* (w, b) *is given by* $\frac{|\langle w,x\rangle+b|}{||w||}$ *and hence the margin is given by*

$$\min_{i=1,..,n} \frac{|\langle w, x\rangle + b|}{||w||}$$

Now fix the scaling of w by assuming $\min_{i=1,..,n} |\langle w, x\rangle + b| = 1$, then the margin becomes simply $\frac{1}{||w||}$ and

$$\langle w, x^i\rangle + b \geq 1 \, \textit{for} \; y^i = +1$$

and

$$\langle w, x^i\rangle + b \leq -1 \, \textit{for} \; y^i = -1$$

For this see Figure 3.22.

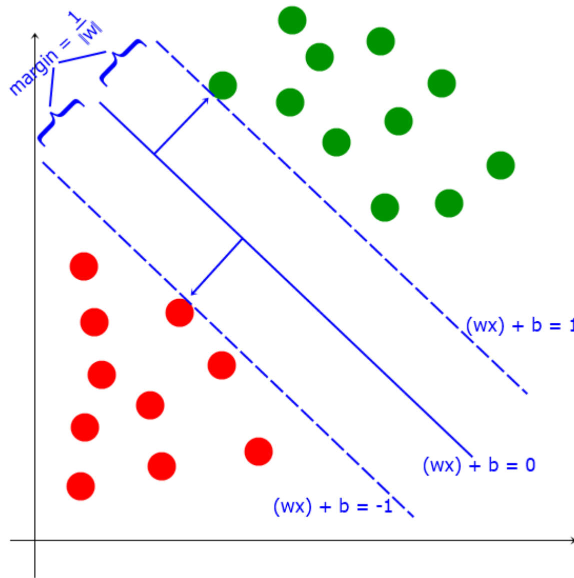


Figure 3.22: The margin of a separating hyperplane.

Therefore the margin maximization problem is given by

$$\begin{aligned} &\max_{w,b} \frac{1}{||w||} \\ \textit{s.t.} \quad &y^i(\langle w, x^i\rangle + b) \geq 1 \, \textit{for all} \; i = 1, .., n \end{aligned} \tag{3.3}$$

Since actually we are interested in the values of w and b which maximize the margin w.r.t the given constraints and not in the maximal value itself we define the arg max.

Definition. *For a function $f : X \to Y$ the arg max is defined by*

$$\arg\max_x f(x) = \{x \mid x \in X \wedge \forall y \in X : f(y) \leq f(x)\}$$

The definition of arg min is in complete analogy

$$\arg\min_x f(x) = \{x \mid x \in X \wedge \forall y \in X : f(x) \leq f(y)\}$$

Claim. *For $f(x) > 0$ the following holds*

$$\arg\min_x f(x) = \arg\max_x \frac{1}{f(x)}.$$

Proof. Assume $f(x) > 0$. Then

$$\begin{aligned}\arg\min_x f(x) &= \{x \mid x \in X \wedge \forall y \in X : f(x) \leq f(y)\} \\ &= \{x \mid x \in X \wedge \forall y \in X : 1 \leq \frac{f(y)}{f(x)}\} \\ &= \{x \mid x \in X \wedge \forall y \in X : \frac{1}{f(y)} \leq \frac{1}{f(x)}\} = \arg\max_x \frac{1}{f(x)}.\end{aligned}$$

□

Claim. *For $f(x) > 0$ the following holds*

$$\arg\min_x f(x) = \arg\min_x f(x)^2.$$

Proof. Assume $f(x) > 0$. Then

$$\begin{aligned}\arg\min_x f(x) &= \{x \mid x \in X \wedge \forall y \in X : f(x) \leq f(y)\} \\ &= \{x \mid x \in X \wedge \forall y \in X : f(x)^2 \leq f(y) * f(x)\} \\ &= \{x \mid x \in X \wedge \forall y \in X : f(x)^2 \leq f(y)^2\} = \arg\min_x f(x)^2\end{aligned}$$

since $f(x) \leq f(y)$ for all $y \in X$. □

Claim.

$$\arg\min_x f(x) = \arg\min_x \frac{1}{2} f(x)$$

Proof. Since $0 < \frac{1}{2} < 1$ it follows that

$$\begin{aligned}\arg\min_x f(x) &= \{x \mid x \in X \wedge \forall y \in X : f(x) \leq f(y)\} \\ &= \{x \mid x \in X \wedge \forall y \in X : \frac{1}{2} f(x) \leq \frac{1}{2} f(y)\} \\ &= \{x \mid x \in X \wedge \forall y \in X : \frac{1}{2} f(x) \leq f(y)\} = \arg\min_x \frac{1}{2} f(x)\end{aligned}$$

□

Since $||w|| > 0$ holds for all $w \in \mathbb{R}^d$ we can apply the claims to (3.3) and obtain $\arg\max_{w,b} \frac{1}{||w||} = \arg\min_{w,b} \frac{1}{2}||w||^2$. Therefore, the original problem can be rewritten equivalently as a convex quadratic optimization problem with linear constraints which has one unique solution.

$$\begin{aligned} &\min_{w,b} \frac{1}{2}||w||^2 \\ \textit{s.t.} \quad & y^i(\langle w, x^i\rangle + b) \geq 1 \quad \textit{for all } i = 1, .., n \end{aligned} \tag{3.4}$$

For this it was explicitly assumed that the data samples ARE linearly separable which means that there exists such a separating hyperplane. In that case the classifier is called a **hard margin classifier**. But in most applications the samples will not be separable exactly by a hyperplane and in that case the optimization problem will have no solution. To solve this problem an idea is to relax the constraints with the use of so called **slack variables** $\xi_i \geq 0$ in the following way (**soft margin classifier**):

$$y^i(\langle w, x^i\rangle + b) \geq 1 - \xi_i \quad \textit{for all } i = 1, .., n$$

Now the optimization problem (3.4) can be extended to

$$\min_{w,b} \frac{1}{2}||w||^2 + \frac{C}{n}\sum_{i=1}^{n} \xi_i$$

$$\textit{s.t.} \quad y^i(\langle w, x^i\rangle + b) \geq 1 - \xi_i \quad \textit{for all } i = 1, .., n$$

$$\textit{and} \quad \xi_i \geq 0$$

Here $C > 0$ is a penalty parameter controlling the size of the slack variables. Using the Lagrangian with Lagrange multipliers $\alpha_i, \beta_i \geq 0$ and defining an $n \times n$ matrix H with entries $H_{i,j} = y^i y^j \langle x^i, x^j\rangle$ and the vectors α and β consisting of α_i and β_i the following dual optimization problem can be obtained:

$$\min_{\alpha} \frac{1}{2}\alpha^T H\alpha - \alpha^T e$$

$$\textit{s.t.} \quad \alpha^T y = 0$$

$$\textit{and} \quad 0 \leq \alpha_i \leq \frac{C}{n}$$

For the exact computation see [2].

Up to now there were just linear models considered but in some cases non linear models may be more appropriate. Therefore, it is common to use the **kernel trick** which maps the input data to a higher dimensional feature space i.e. a Reproducing Kernel Hilbert Space (RKHS)

via a feature map ϕ. Then $H_{i,j} = y^i y^j \langle \phi(x^i), \phi(x^j) \rangle$ and we can still solve the optimization problem and with the optimal α obtain the decision boundary. With

$$w = \sum_{i=1}^{n} \alpha_i y^i \phi(x^i)$$

the decision boundary $\langle w, x \rangle + b$ is given by

$$\sum_{i=1}^{n} \alpha_i y^i k(x^i, x) + b$$

where k is the kernel associated with the RKHS, (so $k(x, x') = \langle \phi(x), \phi(x') \rangle$ for the feature map ϕ). For the computation of b there is the need to look at the different types of sample points, see Figure 3.23.

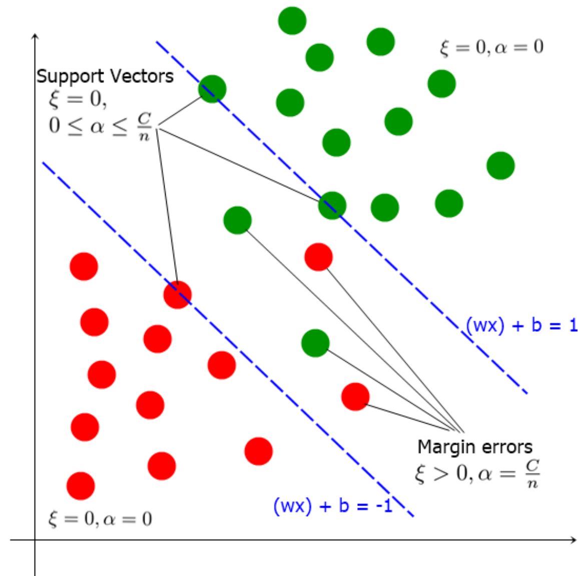


Figure 3.23: In this case three samples are given which lie exactly on the borders. Theses are the support vectors. Inbetween the borders there are some margin errors and on the outside there are the well classified points which do not affect the decision boundary.

- **Margin errors**: If $y^i(\langle w, x^i \rangle + b < 1$ there must be $\xi_i > 0$ and hence $\beta_i = 0$ and $\alpha_i = \frac{C}{m}$
- **Well classified points**: If $y^i(\langle w, x^i \rangle + b > 1$ then $\xi_i = 0$ and $\alpha_i = 0$ and the points do not affect the decision boundary.

- **Support vectors**: If $y^i(\langle w, x^i \rangle + b = 1$ then the points lie exactly on the decision border. Therefore $\xi_i = 0$ and $0 \leq \alpha_i \leq \frac{C}{n}$

Therefore we can compute b by taking the average of

$$y^i - \sum_i \langle w, x^i \rangle$$

over all support vector points.
Common choices for the kernel are [10]:

- **Linear Kernel**: $k(x, x') = \langle x, x' \rangle$
- **Polynomial Kernel**: $k(x, x') = (\gamma \langle x, x' \rangle + c)^d$
- **RBF Kernel**: $k(x, x') = exp(-\gamma ||x - x'||^2)$

4 Deep Learning for OCT-Images

4.1 State of the Art

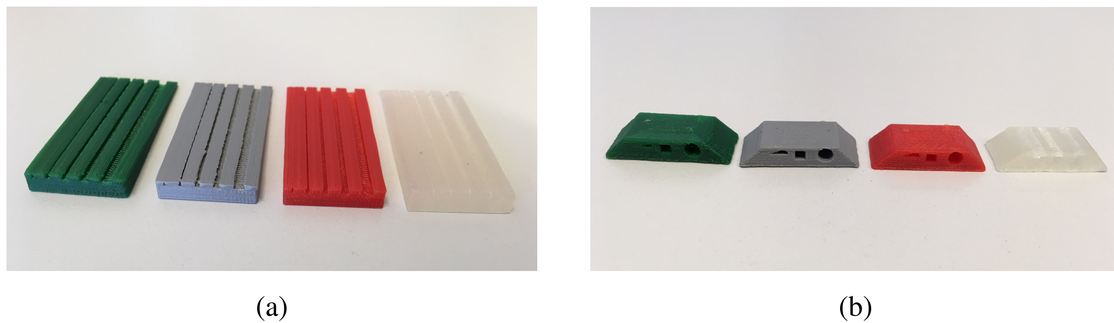
Recent articles on medical studies report on the successful application of deep learning on OCT-images for biomedical issues. Since the main field of application for this imaging technology is ophthalmology most of the applications of deep learning for OCT-images are in this field. Many papers cover the analysis of images of the retina to discover diseases like Diabetic Macular Edema (DME) [3], glaucomatous damage [33] or age-related macular degeneration [29]. As deep learning has been a useful and successful method for the detection of these diseases, we will extend the application of this machine learning technique to OCT-images for the classification of technical materials in an industrial context. We study the automatic classification of different materials represented by different color pigments in 3D-printed poly-lactate-acid (PLA) objects. The second task is the inspection of materials to automatically distinguish between coated and uncoated areas on different kinds of papers. We compare the results of deep learning with results obtained by other machine learning techniques namely random forest and support vector machine.

4.2 The First Task – 'Material' Classification

4.2.1 Problem Description

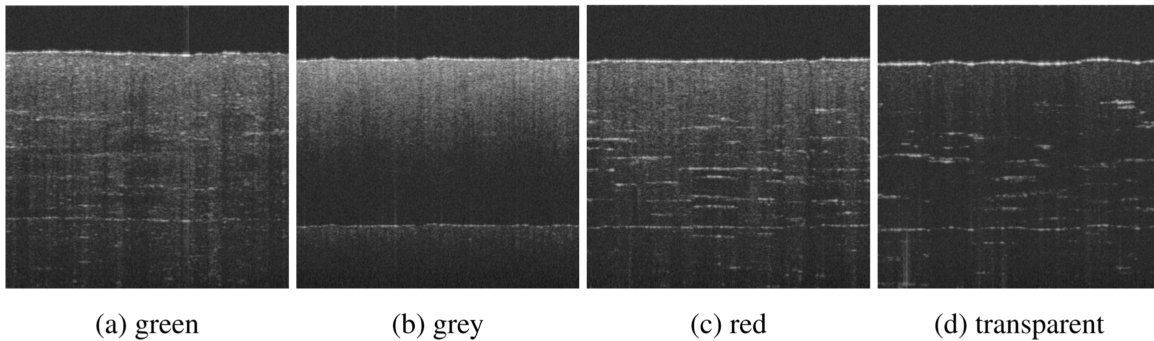
The first task has been the classification of OCT-images of different objects with respect to their 'material'. The objects have been 3D-prints from the base material poly-lactate-acid (PLA) and the different 'materials' were represented by different color pigments in the PLA matrix. The goal has been to train a model which is able to distinguish between four different materials – represented by the color pigments of green, grey, red and transparent color – with only their (greyscale) OCT-image as an input. For each color two differently shaped objects have been printed, a small dovetail-shaped one and a larger wedge. The objects have been designed in such a way that they include many different kinds of structures to record, such as

plain surfaces, slopes of $5°$ and $45°$, interior defects and surface defects. Figure 4.1 provides a short glimpse on the objects. Because of the different scattering of the color pigments included, human experts can in most cases distinguish the OCT-images, compare also Figure 4.2. One can see that the green material gives coarse images where the printed layers are hardly visible. The grey material absorbs the light very fast, the structure is fine and the layers are not apparent. For the red material the images look also very fine but the light is not absorbed as quickly as for the grey material and the layers are very well visible. And finally the transparent material reflects the light mostly on the surface and the layers whereas the internal structure is nearly black.



(a) (b)

Figure 4.1: 3D-print objects: (a) a wedge (short k = "Keil") and (b) a dovetail (short s = "Schwalbenschwanz"). Each object is given in four colors: green, grey, red and transparent.



(a) green (b) grey (c) red (d) transparent

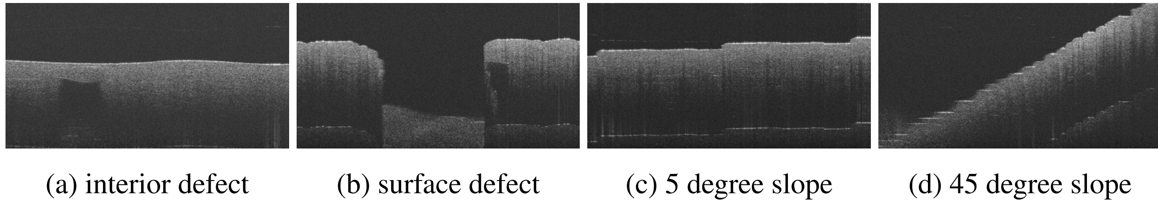
Figure 4.2: The OCT-images of the different colored objects are well distinguishable in most of the cases.

For the investigation a total of 12800 images (dimension 512×1000 pixels) have been recorded, evenly distributed among the four classes. The recordings were done on different spots on the objects and with and without defects. See Table 4.1 for the distribution of images and Figure 4.3 for OCT images of the different areas on the objects. One important issue for this task has been that the various surfaces and the defects should not confuse the

material classification s.t. for example all recordings in Figure 4.3 should have been classified as ’grey’ even though the surface looks quite different. The recordings were done in a manner to prevent from a main problem of neural networks: learning the false features. To get the data as clean as possible the position of the object-surface in the OCT image has been alternated in height for all classes. Furthermore on each recording-day all classes have been recorded for a certain object-area to exclude possible changes in the ambient light or OCT-system settings. The final data set has been randomly split into 6400 training samples, 3200 validation samples and 3200 test samples. Finally three different machine learning methods have been applied for solving the classification task: deep learning, random forests and support vector machine.

	green	grey	red	trans	sum	green	grey	red	trans	sum
plain	500	500	500	500	2000	500	500	500	500	2000
5 degree	250	250	250	250	1000	250	250	250	250	1000
other slopes	500	500	500	500	2000	400	400	400	400	1600
interior defects	-	-	-	-	-	400	400	400	400	1600
surface defects	-	-	-	-	-	400	400	400	400	1600
sum	1250	1250	1250	1250	**5000**	1950	1950	1950	1950	**7800**

Table 4.1: The numbers of samples recorded for the different settings. The left table summarizes the data about the small, dovetail-shaped object and the right table is about the larger, wedge-shaped object. Here ’other slopes’ means that the record was not made in the direction of the steepest descent (5 degree or 45 degree) but in some other directions.



(a) interior defect (b) surface defect (c) 5 degree slope (d) 45 degree slope

Figure 4.3: The recordings of different surfaces and defects.

4.2.2 Deep Learning

Having images as input for the given task the most straightforward idea has been to use a convolutional neural network architecture. As suggested in [16] we have started the analysis

with a CNN model architecture that is known to perform well for one of the most common machine learning problems: the MNIST dataset [55]. Using the Keras library [54] the implementation to start with has been taken from [51]. Implementing a classifier for the MNIST dataset the number of classes is 10, namely the numbers from 0 to 9. The training images are provided in size 28×28 pixels. For the given task the number of classes has been changed to four and the image dimensions have been reduced to 64×125 pixels to maintain the image proportions while preventing memory overflow. The need of cross-validation has been tested by observing the variance of results for different data splits, see Chapter 4.2.5. To prevent overfitting some data augmentation - namely the horizontal flip - and additionally early stopping has been used from the beginning. As argued in Chapter 3.1 the test set of 3200 images was put aside right away and the tuning of the model architecture and hyperparameters has been performed with the 6400 training samples and 3200 validation samples. The different settings include:

1. Add one or two convolutional layers.
2. Try different numbers of units in the layers, taking multiples of 20 from 20 to 400.
3. Use batch normalization after all convolutional and dense layers, following the implementation of [52].
4. Vary the batch size, taking powers of two in the range of 16 to 512.
5. Tune the dropout rates for convolutional and dense layers, going from 0 to 1 with stepsize 0.05.
6. Change the optimizer and try Adadelta, Adam and Stochastic Gradient Descent (with Nesterov momentum).

As for this task the distribution of samples between the classes has been balanced and none of the classes has been more important than the others the most appropriate performance measure is accuracy. Choosing a random parameter setting the network has been trained on the training data and afterwards evaluated on the validation data. The training for a certain parameter setting has taken between five and ten minutes in average when performed on the GPU. The scores of many different settings have then been compared to select the best parameter combinations. After training many different models and adjusting the parameter ranges a few times the parameter setting which reaches the best validation accuracy has been the following, see also Figure 4.4:

Ad 1.) One convolutional layer has been added to the architecture based on the MNIST example from [51]. So now there have been three convolutional layers in the network.

Ad 2.) As the standard setting of units per layer has still been the best one, we have used 32 units in the first convolutional layer, 64 units in the second and 128 in the third one. The first dense layer has had 256 units and the second of course just the number of classes which is four.

Ad 3.) Having used batch normalization, as it has improved performance and training time.

Ad 4.) The batch size which has been giving best scores for the given task is 32.

Ad 5.) For convolutional layers the dropout has been performing best with dropout rate at 0.1 and for the dense layer with dropout rate at 0.05. Therefore, 10% of the convolutional layer's units and 5% of the dense layer's units have been dropped out randomly.

Ad 6.) The best optimizer for the given task has been Adadelta with its default settings, see [54].

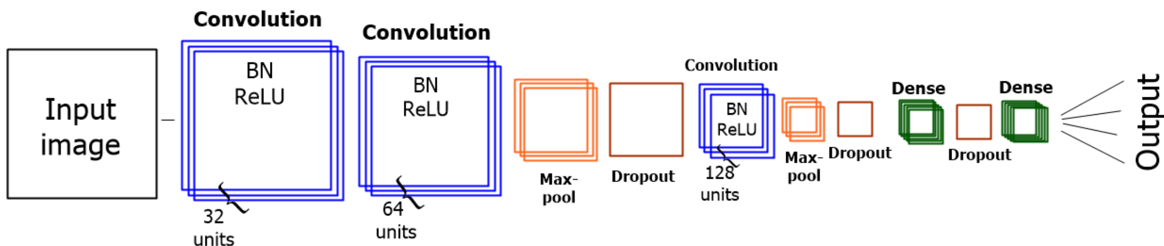
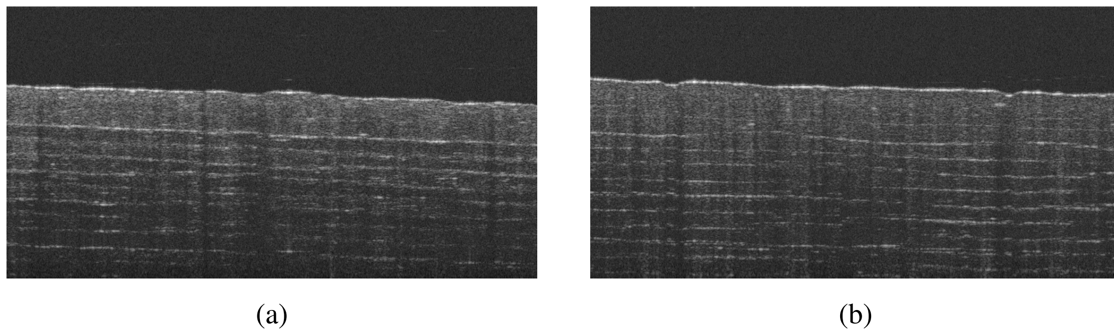


Figure 4.4: The network architecture of the final model.

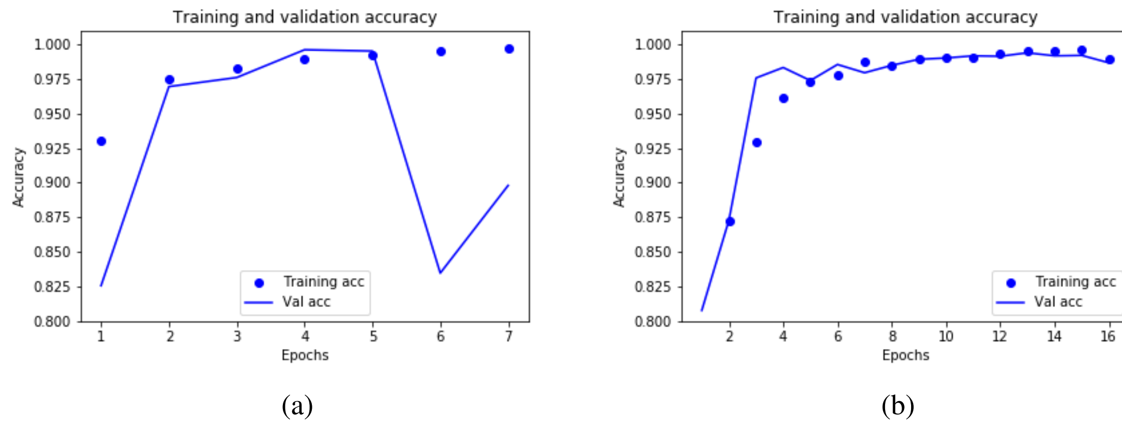
This parameter setting has reached a training accuracy of 98,91% and a validation accuracy of 99,59%. On the independent test set an accuracy of **99,47%** has been reached, this corresponds to just 17 false predicted images out of 3200. A closer look on these misclassified images shows that the problematic cases have been objects with red and green color particles. For them on some special areas of the objects the classes look quite similar and can also not be easily distinguished by an expert. See Figure 4.5 for false predicted images of the classes 'red' and 'green'.

To visualize the progress of learning during the epochs it is common to plot the accuracy scores of training and validation set for each epoch. As the chosen classifier uses batch normalization the maximum of the validation accuracy has been reached very quickly and was starting decreasing again but because of early stopping the classifier which is saved has been the one with the maximal validation accuracy. Without batch normalization the training has usually taken longer and because of that the accuracy curve looks nicer, but in that case the validation accuracy has not reached as nice results. See Figure 4.6 for two possible accuracy curves with and without the use of batch normalization.



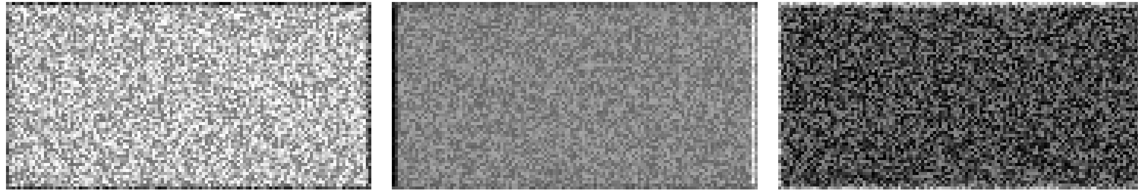
(a) (b)

Figure 4.5: Two examples of OCT-images which are wrongly predicted by the classifier. (a) shows an image of an object with green color pigments which is classified as red. (b) shows an image of an object with red color pigments which is classified as green.

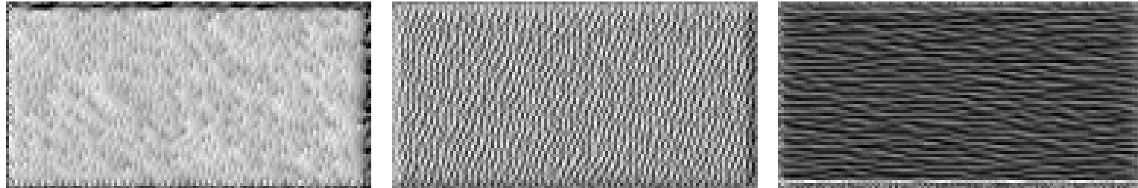


(a) (b)

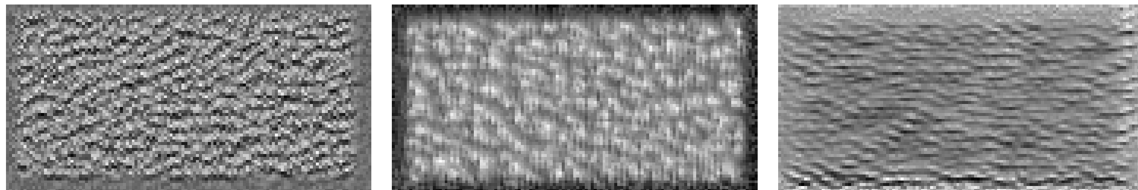
Figure 4.6: The training and validation accuracy curves of two different classifiers. (a) is the training of the finally chosen classifier with batch normalization. As described before the maximal validation accuracy has been reached very early and then starts zigzagging. (b) shows the training of a classifier without batch normalization, in this case it has taken nearly twice as many epochs to reach the maximal validation score but the curve looks much nicer here. It can be seen here that when training started both training and validation scores first have been quite bad but with more and more epochs the accuracies have raised until at some point they have started decreasing again. That is the point where further training has been stopped.



(a) First Convolutional Layer



(b) Second Convolutional Layer



(c) Third Convolutional Layer

Figure 4.7: Some filters of the convolutional layers i.e. the images certain units of convolutional layers have been maximally responsive to. It can be seen that the later layers combine the structures learned in earlier layers and therefore units of the third convolutional layer have much complexer structures than the units of the first convolutional layer.

Filter Visualization

As already explained in Chapter 3.3 the main advantage of neural networks is the possibility to use 'raw data' like images or time series as an input without the need to manually extract features. On method to get some insight into a trained model rather than accepting them as 'black boxes' is to display the visual patterns that the filters (units) are meant to respond to. For each unit in a convolutional or the output layer of the network starting with a random greyscale image the pixel values are adjusted such that the activation of the unit is maximized. With this method it has been possible to obtain the image to which the filter is maximally responsive. In Figure 4.7 the visualization of three different filters for each convolutional layer is illustrated. Figure 4.8 shows the visualization of the four units in the output layer.

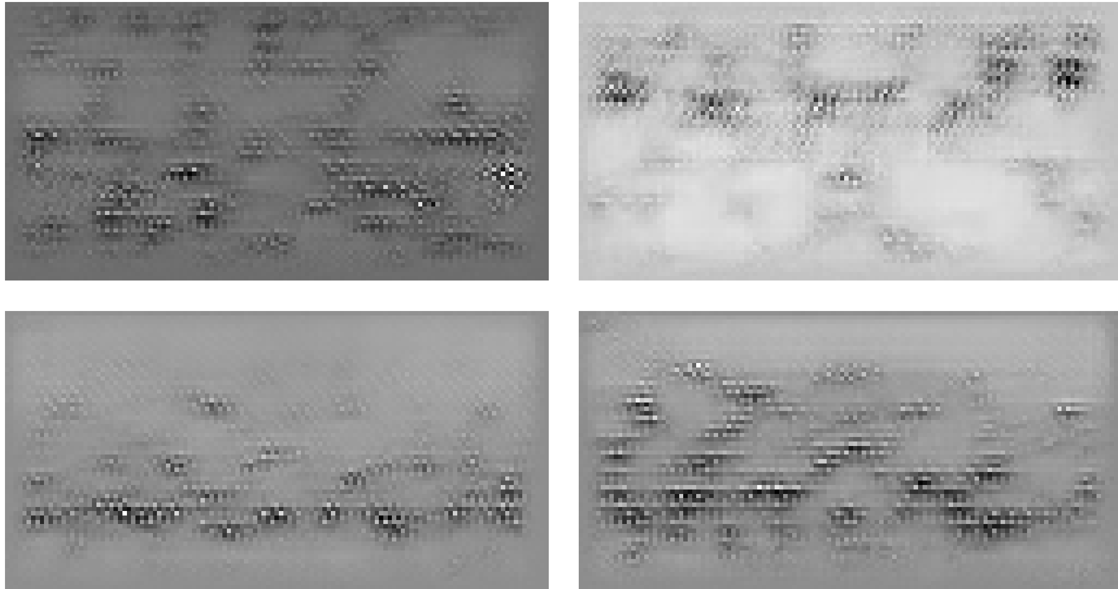


Figure 4.8: Visualization of the four units in the output layer. Here each of the images represents how an input image should look like such that it obtains a maximal activation for a certain class in the output layer.

4.2.3 Feature-Based Methods – Random Forests and Support Vector Machines

As already discussed in Section 3.3 the main difference between neural networks and standard machine learning methods like random forest and support vector machine is the input needed for training. For the networks the OCT-images have been used as input, but for random forests and support vector machines just a 'small' vector of features is needed and therefore it has been required to think about appropriate features representing and characterizing the images. As the difference between the four classes lies just beneath the object surface and the part above does not say anything the first step has been to normalize the OCT-images such that the flattened surface is on the upper border of the image. Additionally the images have been randomly flipped horizontally. As it can be seen in Figure 4.2 the information distinguishing the classes best are just beneath that border. To become more robust against defects and other flaws in the structure the image has been split into several patches. One row of 50×50 pixel patches have been taken from the image starting from the object border but as there is not much information beneath the surface of the lower steps of a slope such patches have been excluded.

The Features

The selection of appropriate features is essential for the success of standard machine learning methods. For the current task of distinguishing between the four material classes it has been important to look at the inner structure of the objects. Differences in brightness and texture seem to be essential; based on recommendations of the company partner RECENDT who have dealt with similar problems earlier, the chosen features have been: mean, median, variance, skewness, kurtosis and histogram entropy. In the following we will show how these values are computed and what they reveal regarding image analysis [28, 38, 43].

Mean: The mean is the first central moment. The mean over the N data points $x_1, .., x_N$ is calculated by adding up all the values and dividing that sum by N.

$$\bar{x} = \frac{1}{N} \sum x_i$$

The mean therefore indicates the mid-point of a set of values but it is easily distorted by outliers. In image analysis the mean provides general information about the brightness of the image patch but small areas with either extremely bright or dark pixel values will have a big influence on the mean.

Median: The median is similar to the mean but it is more robust against outliers. It is the value separating the lower half from the higher half for a given data sample. For N sorted values $x_1, .., x_N$ the median is defined as

$$mean = \begin{cases} \frac{x_{N/2} + x_{(N+1)/2}}{2}, & N \text{ even,} \\ x_{(N+1)/2}, & N \text{ odd.} \end{cases}$$

Therefore the median also gives information about the general brightness of an image patch but it is not as easily distorted by a small number of extremely bright or dark pixels.

Variance: The variance is the second central moment. It measures how far the data points are spread out from their mean and is calculated for N data points $x_1, .., x_N$ by

$$var = \frac{1}{N} \sum (x_i - \bar{x})^2$$

For image processing the variance expresses how much dispersion exists from the mean. A low value indicates that the pixel values are very similar to the mean whereas a large variance indicated that the pixel values are spread out over a large range.

Skewness: The skewness is the third moment and it is usually normalized. Regarding the histogram of data values the skewness measures the lopsidedness of the distribution. It is often computed as the Fisher-Pearson coefficient of skewness

$$skewness = \frac{m_3}{(m_2)^{3/2}}$$

where m_i is the ith central moment

$$m_i = \frac{1}{N}\sum(x_i - \bar{x})^i$$

Skewness gives information about the data symmetry. A negative skewness means that the left tail of the data histogram is longer or bigger than the right one and therefore in image analysis it means that the frequency is wider spread over the dark pixel values.

Kurtosis: Kurtosis is the fourth moment and it is also usually normalized. It measures the peakedness of a histogram of data values and is calculated by

$$kurtosis = \frac{m_4}{m_2^2}$$

with m_i is defined as above. For this thesis the Fisher kurtosis was used i.e. the value 3.0 is subtracted from the result to give 0.0 for a normal distribution. Since the kurtosis measures the shape of the probability distribution for images it indicates the number of pixels in the dominant gray level in comparison the the number of pixels in other gray levels. The higher the kurtosis the more uniform the pixel values are with less noise.

Entropy: Entropy is a measure for the information content i.e. for the uncertainty in the values. Taking the relative frequency of a value x_i as $p(x_i)$ the entropy can be computed by

$$entropy = -\sum p(x_i) * log(p(x_i)).$$

The values of $p(x_i)$ can be computed easily from the histogram by dividing the number of elements in the bin x_i by the total number of elements. Since images contain many pixels with unique values for this thesis the histogram was constructed with just ten bins. In image analysis entropy measures the complexity of pixel values contained in a certain neighborhood and therefore it can detect variations in the distribution of the values.

Taking the first moments for the texture analysis of the image is proposed also in [49] where it helps to classify medical CT-images. Additionally to the mean, the median has been used to be more robust against outliers. The entropy of the histogram can be used to classify

textures since a certain texture has repeating patterns which might lead to a certain entropy. Therefore, the input for the learning-process has been a 6-dimensional feature vector which was extracted for all image patches drawn from the input images like described before. The feature vectors of the images in the test set have again not been used for training. **Remark:** In the following 'accuracy' means the rate of correctly classified images, not correctly classified patches. The training has been performed with the features of the single patches but for the evaluation the results of the patches of an image have been accumulated to get a prediction for the whole image.

Random Forest

Using the random forest classifier method from the scikit-learn library [59] the parameters to be tuned have been taken from [56]: The number of estimators (i.e. the number of trees in the forest), the maximal depth of a tree, the maximal number of features considered for a split, the minimal number of samples in a split and the minimal number of samples in a leaf. After having adjusted the parameter ranges a few times during training the parameter setting which has reached the best accuracy of 95,84% on the validation data has been the following: The number of estimators has been 32 and the maximal depth has been 36. The maximal number of features considered for a split has just been 1, the minimal number of samples in a split has been 4 and the minimal number of samples in a leaf has been 2. With that parameter setting the random forest has been trained again on the training and validation data and on the independent test set it has reached an accuracy of **95,29%**. The training for one certain parameter setting has usually taken less than a minute for the random forest classifier. The visualization of a random forest classifier is more complicated as it consists of many single decision trees. We have tried to graphically represent one of the decision trees and got an extremely wide tree. In Figure 4.9 a little part (about $\frac{1}{16}$) of the tree width can be seen. **Remark:** Due to the zooming the connection lines between the colored nodes have vanished. In Figure 4.10 just the first three depth-levels of the tree are visualized in detail.

Figure 4.9: Visualization of one of the decision trees contained in the final random forest model. Inside the nodes there are given the ranges for the different feature values which lead to that node. The colors represent the output classes.

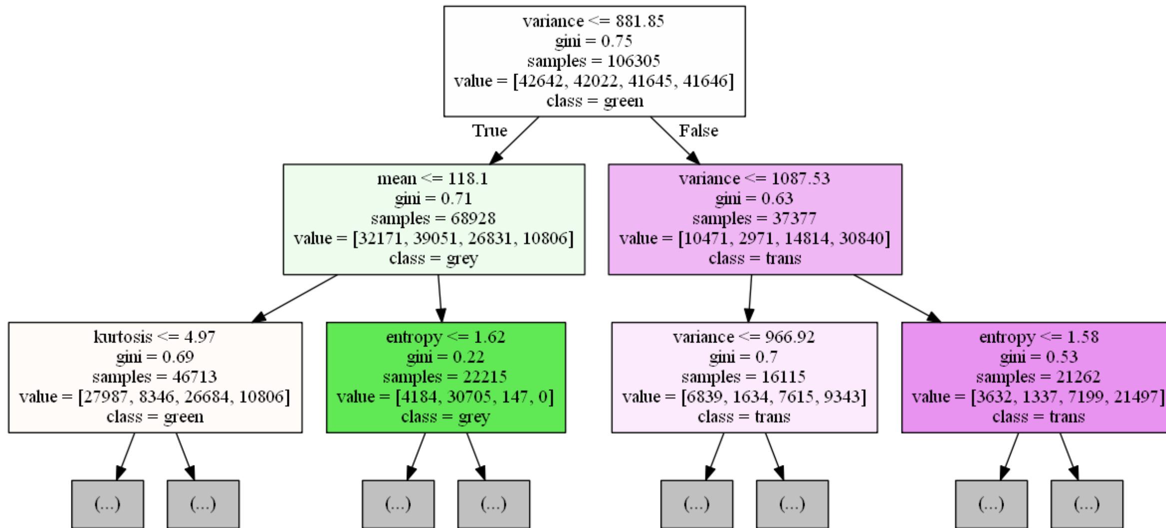


Figure 4.10: Detailed visualization of the first three depth-levels.

Support Vector Machine

Using the C-support vector classifier for the rbf kernel and the Linar-SVC for the linear kernel from the scikit-learn library [60], [58] the parameters for this method have been tuned following [57]: the penalty parameter C of the error term, the kernel to be used, for rbf kernel the kernel coefficient γ and for the linear kernel the penalty norm and whether to use dual optimization. Like for deep learning and random forests the parameter ranges have been adjusted a few times and the the best validation accuracy has been reached with $C = 500$ and taking the rbf-kernel with $\gamma = 0.01$. With this parameter setting the classifier has been retrained using the training and validation data. It has reached an accuracy of **95,19%** on the independent test set. The training for one certain parameter setting has taken about one minute in average for a linear kernel but for the rbf kernel it has taken about 150 minutes in average. A decent visualization of the SVM model has not been possible for this task as the features have been six-dimensional and therefore it would have been necessary to visualize the six-dimensional space.

4.2.4 Comparison

In Table 4.2 an overview of the results is provided and allows to compare the different machine learning methods. All of them were performing very well for the given task, but the results show that deep learning has reached the best accuracy on the test set whereas the average time for training one parameter setting on the GPU has been within a good scope.

	Deep Learning	Random Forest	Support Vector Machine
Accuracy on the independent test set	99,47%	95,29%	95,19%
Average training time for one parameter setting	5-10 minutes	<1 minute	$\sim$ 1 min (linear) $\sim$ 150 minutes (rbf)

Table 4.2: Accuracies and average training times for the three different machine learning methods applied to the first task.

4.2.5 Remarks

(Nested) Cross-Validation

To test the need of (nested) cross-validation for the given task it has been important to explore the variance of the results for the given dataset. Therefore, we have used the MNIST-model architecture [51] but slightly modified it as the given data is more complex (add one layer). As nested 5-fold cross-validation would result in test sets containing approximately 16% of the data it has been split five times randomly into 42% training and validation data and 16% test data. For each split we have trained a neural network and evaluated it on the test set. The variance over these test accuracies can give an estimate for the general variance of the data splitting. To be even more sure about the need of cross-validation we have done this for a model with dropout, without dropout and with stochastic gradient descent optimizer. The results can be seen in Figure 4.11. As the variances for the evaluation of different test-sets have been very small in all the cases it can be assumed that the cross-validation would also not have led to a high variance. Therefore, due to the very small variances for each of the three models and the intense rise in computation costs cross-validation has neither been used nested nor single for this task.

Colour Pigments in Injection Moulded Parts

To compare the color pigments and internal structure of the 3D-printed objects with the color pigments and internal structure of everyday life goods additionally OCT-images of injection moulded parts (mostly bottle caps) in the same four colors have been recorded. Then the finally obtained deep learning classifier has been used to predict the class of the new injection moulded part's OCT-images. Per class there have been recorded 20 images and the classifier has predicted the right class for nearly all of them, just one of the images of the part with red color pigments has been classified wrongly. Therefore, the classifier is not just useful to distinguish between color pigments of 3D printed objects but also for other plastic parts.

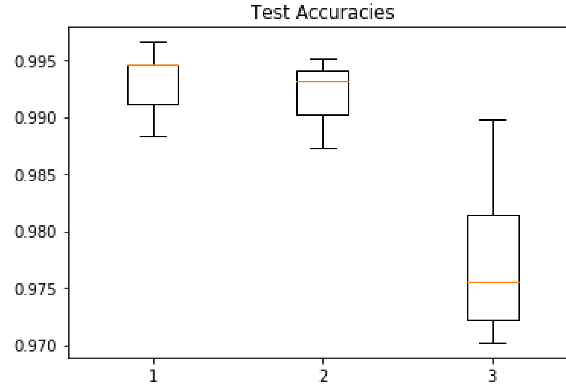


Figure 4.11: The test accuracies of the tests for analyzing the need of cross-validation. Left: The slightly modified MNIST model using dropout, variance = $8.717e^{-6}$. Middle: The model without using dropout, variance = $8.163e^{-6}$. Right: the model using stochastic gradient descent as optimizer, variance = $5.003e^{-5}$.

4.3 The Second Task – Inspection of Coatings

4.3.1 Problem Description

The second task has been the identification of coatings on different types of paper. Five different materials with coated and uncoated areas have been examined: three different envelopes and two different cigarette papers. The coated samples were given by the glued surfaces on the objects and there have been two with self-adhesive areas and three with glue which needs to be moistened. The goal has been to train a model which is able to distinguish between OCT-images of coated and uncoated areas on the objects. The coating can be seen as a small line above the object surface in the OCT-image (compare also Figure 4.12).

In total there have been recorded 9000 images (dimension 400×1000) evenly distributed between the two classes. Again the imaging was done in a manner to prevent from learning the false features (see also Section 4.2.1). To keep the data as clean as possible again the position of the object-surface in the OCT image has been alternated in height for both coated and uncoated areas and the different classes for one object type have always been handled on the same day. Additionally all objects have been flipped 90 degree once to be independent of possible coating-directions or differences in the underlying material. To examine the need for cross-validation again a test for the variance of the data-splitting was performed in the same way as it had been done for the first task: The whole data set has been split five times differently into 42% training and validation and 16% test data. Using each for three different model architectures has yielded a too high variance within the data splits which confirmed

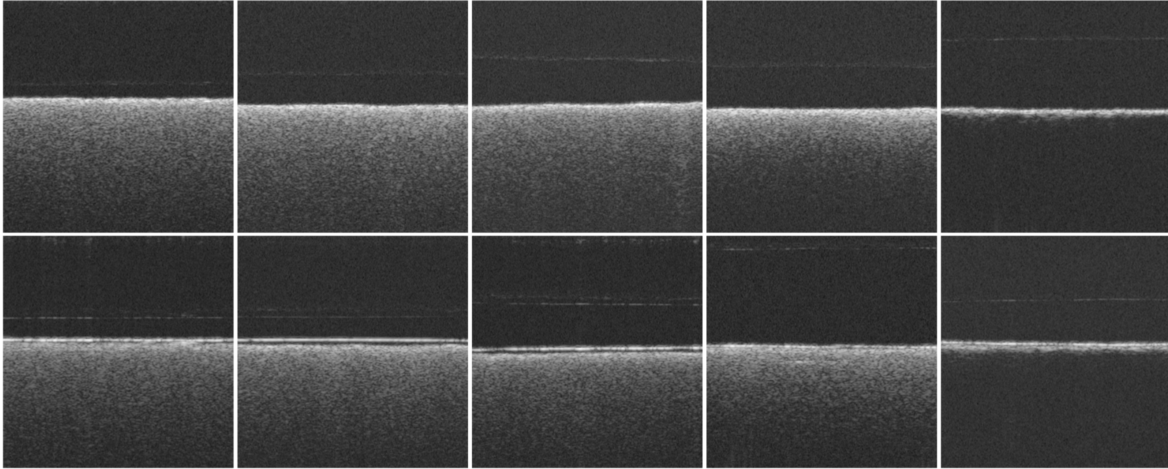


Figure 4.12: Some sample OCT-images of the second task. Given were five different materials where a recording of an uncoated area is at the top and a recording of a coated area is at the bottom. The first three columns are the envelopes and the last two columns are the cigarette papers. The coating can be seen as a small white line above the object surface and its visibility depends on the different objects as on some the coating may be thicker than on others or it may diffuse more into the material.

cross-validation as necessary, see Section 4.3.5 for exact results. To perform five-fold cross-validation the data has been split into five folds containing each 1800 samples. As already explained in Section 3.1 each of the folds has then been used as the independent test set once while the remaining four folds have been split equally into training and validation set. This has resulted in five different data-splits with each 3600 training and validation samples and 1800 test samples. The parameter selection was now performed for each data-splitting individually and in the end the five results have been used to get an estimate for the generalization abilities of the three different machine learning methods used: deep learning, random forest and support vector machine.

4.3.2 Deep Learning

Again the Keras library [54] has been used. Having once more OCT-images as an input the idea has been to start with the convolutional neural network architecture which has performed best for the first task and start to tune the hyper-parameters from that base. To use it for the new task it was necessary to change the number of classes to two and the image dimensions to 50×125 to fit the proportions of the new data. Again the horizontal flip and early stopping has been used against overfitting. For each of the five different data split-

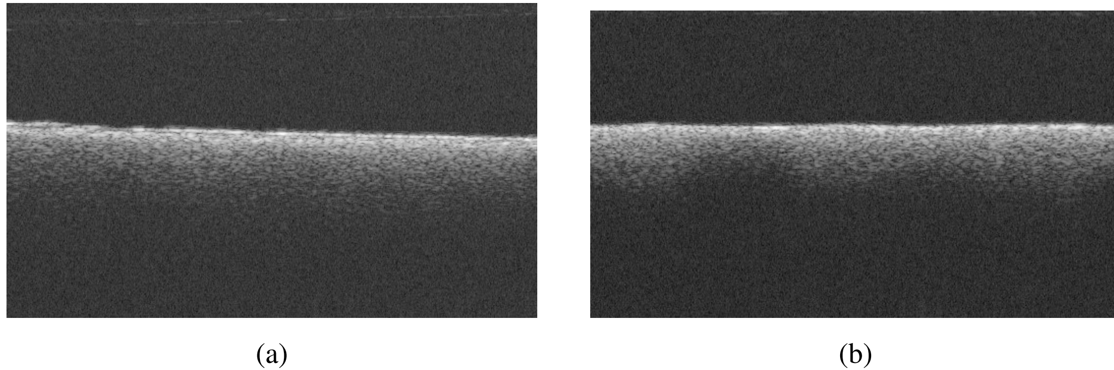
	Split 1	Split 2	Split 3	Split 4	Split 5	Average
Validation Accuracy	98,81%	98,81%	99%	98,61%	99%	**98,84%**
Test Accuracy	98,28%	98,61%	98,22%	98,77%	98,99%	**98,58%**

Table 4.3: The validation and test accuracies for the five different data-splits obtained with deep learning.

tings the hyper-parameters have then been tuned independently by training the network on the current training data and evaluating it on the validation data. The test set has been put aside right away for each splitting. The hyper-parameters to be tuned have basically been the same as for the first task but as the base network contains already three convolutional layers now the different settings have included removing or adding one convolutional layer. Again using accuracy as performance measure and selecting the best set of hyper-parameters like described earlier for each of the data-splits the best model has been obtained. For this task the training time for one parameter setting has in average been about 5 minutes and therefore a bit less than it had been for the first task due to a minimal decrease in the number of data samples. In nearly all splits the number of convolutional layers has remained the same as for the color-task, namely three, just in one data split the best performance score has been reached with four convolutional layers. Now not only the Adadelta optimizer but in two cases also the Adam optimizer has been the best choice to use. Batch normalization has improved the accuracy for three out of the five data-splits. With the appropriate parameter settings the maximal validation accuracies reached for each data-splitting with its corresponding test set accuracies are provided in Table 4.3.
Therefore, the average validation accuracy has been 98,8% and the average test accuracy has been **98,6%** which corresponds to about 26 false predicted images out of the 1800 in the test set. Further analyzing the wrongly predicted images for all the folds two-thirds of the mistakes have happened as images of coated areas are classified as uncoated. More than a half of the problematic images have been from the thick cigarette paper whereas the thin cigarette paper seems to be well distinguishable. See Figure 4.13 for two examples of false predicted images.

Filter Visualization

As five fold cross-validation has been used for the current task there are five different resulting models, one for each data-splitting. Therefore, the filter images of all these models can be obtained individually. In Figure 4.14 we present again three selected filters for each convolutional layer and in Figure 4.15 the two filters of the output layer for the models which



(a) (b)

Figure 4.13: Two examples of OCT-images which are wrongly predicted by the classifier. Both images show the thicker cigarette paper, (a) shows an actually coated area on the paper which has been misclassified as uncoated. (b) on the other hand shows an actually uncoated area which has been misclassified as coated. Here it can be seen that the images of the two classes are sometimes very similar and hard to distinguish.

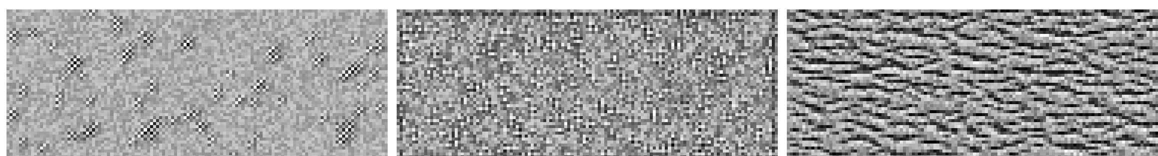
has been trained on one certain data-split. The filter images of the models which have been trained on the other data-splits may look similar.

4.3.3 Feature-Based Methods – Random Forests and Support Vector Machines

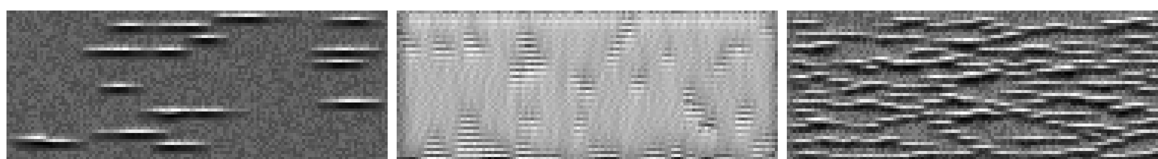
The goal has been to compare the deep learning results from the coating task also to the standard machine learning methods random forest and support vector machine. Therefore, it has been required to again extract appropriate features from the OCT-images. As the only difference between recordings of coated and uncoated areas lies just beneath the object surface the code from the first task which had been used to flatten that border has been used and the images have been flipped randomly horizontally. Now the image patches which were taken from the normalized image have just been 15×50 pixels as for this issue the information about the classes lies right on the surface of the objects.

The Features

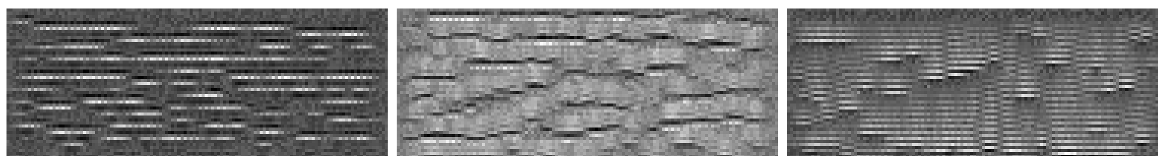
In contrast to the 3D printed objects the difference between the classes now does not appear in the inner structure of the objects but right on the border. Accordingly the idea has been to extract that part of the OCT-image and to not consider the remaining part of the recording which should not contain any information about the class. Studying the data in detail has led to the assumption that 15 pixels should be sufficient to contain the information about the



(a) First Convolutional Layer



(b) Second Convolutional Layer



(c) Third Convolutional Layer

Figure 4.14: Some filters of the convolutional layers i.e. the images certain units of convolutional layers are maximally responsive to. It can be seen again that the later layers combine the structures learned in earlier layers and therefore units of the third convolutional layer have much complexer structures than the units of the first convolutional layer.

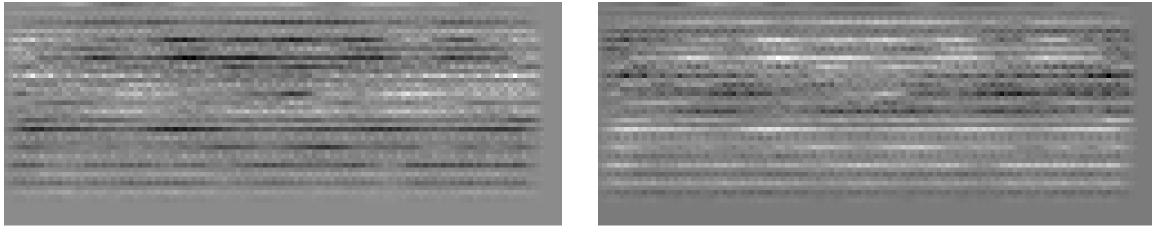


Figure 4.15: Visualization of the two units in the output layer. Here the images represent how an input image should look like such that it obtains a maximal activation for the coated or uncoated class in the output layer. In comparison with the output filter images of the first task - see Figure 4.8 - it can be seen that now there has been some activation on the object surface and the structures which to model searches for to distinguish between images of coated and uncoated areas are primarily horizontal. Therefore it can be argued that the model seems to have learned the right features as the coating can be seen in OCT-images also as a horizontal structure.

properties distinguishing the classes. On the one hand a part of the base material appearance was included for both classes s.t. that the line between the coating and the material is fully visible. But on the other hand not too much of the base material should have been in the image as it does not contain knowledge about the class. For each of the 15×50 patches obtained in that way the following features have been computed: mean, median, variance, minimal brightness and maximal brightness. The calculations and meanings of the first three features is specified in Section 3.3. Minimal and maximal brightness refers to the darkest respectively brightest pixel value in the image patch. The intention has been that for the coated areas the image patches contain the dark border between the coating and the base material and should therefore contain darker values which can be seen in the mean, median and minimal brightness. Analogously the uncoated areas may contain brighter values that influence the mean, median and maximal brightness values and the variance can be helpful to observe how far the values diverge. Following this the input to the machine learning procedure now has been a five dimensional feature vector which was extracted for each image patch attained like described before. **Remark:** Like already mentioned for the first task 'accuracy' in the following means the rate of the correctly classified images, not the correctly classified patches. Of course the classifier has been trained with and does classify the feature vectors of the individual patches but to compute the accuracy value the results for all patches of an image have been accumulated to get the results for the whole image.

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Average
Validation Accuracy	95,64%	96,22%	95,36%,	95,14%	96,17%	**95,71%**
Test Accuracy	96,05%	95,28%	94,61%	93,89%	95,72%	**95,11%**

Table 4.4: The validation and test accuracies for the five different data-splits obtained with random forest classifiers.

Random Forest

Again the random forest classifier method from the scikit-learn library [59] has been used and the parameters to be tuned were taken from [56]. Thus the values of the number of estimators, the maximal depth, the maximal numbers of features considered for a split, the minimal number of samples for a split and the minimal number of samples in a leaf have been varied and the value ranges have been adjusted a few times to maximize the performance score on the appropriate validation data. According to Section 4.3.5 cross-validation has been needed and therefore the parameter tuning has been done individually for each of the five data-splits. The parameter settings which reach the best accuracies on the correct validation sets have ranged from one to 128 trees in the forest with a maximal depth from 17 to 43 nodes. The maximal number of features considered for a split have either been two or four, the minimal number of samples required for a split has been between five and nine and the minimal number of samples required in a leaf has either been one or two. For each data split with the fitting parameter settings the models trained on the training data have achieved in average 95,7% validation accuracy. After finding the best parameter setting for each split the model has been retrained on the training and validation data with the corresponding parameters and evaluated on the independent test set where they have reached **95,1%** average test accuracy. The detailed results are shown in Table 4.4.
Again the training for one certain parameter setting has usually taken less than a minute for the random forest classifier. Since cross-validation has been used we have obtained five final random forest classifiers for this task. To visualize one decision tree we have choosen the random forest classifier with the minimal tree-depth, namely 17. Therefore the graphical representation of one decision tree in that random forest is again extremely wide but not that deep any more as it had been for the first task. In Figure 4.16 a little part (about $\frac{1}{16}$) of the tree width can be seen. In Figure 4.17 just the first three depth-levels of the tree are visualized in detail.

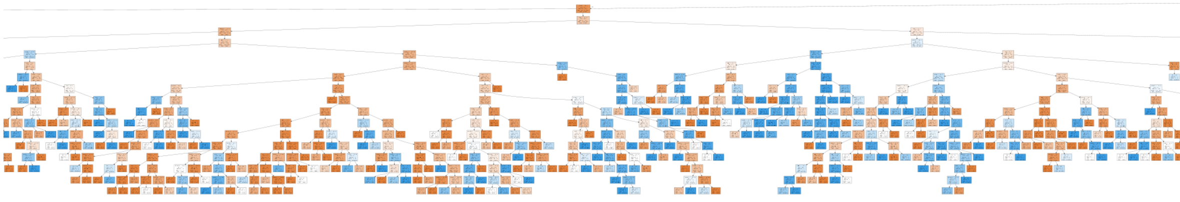


Figure 4.16: Visualization of one of the decision trees contained in one of the final random forest classifiers trained for the coating task. Inside the nodes there are given the ranges for the different feature values which lead to that node. The colors represent the output classes.

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Average
Validation Accuracy	94,47%	95,17%	94,83%	94,39%	94,36%	**94,64%**
Test Accuracy	94,55%	94,44%	93,33%	93,16%	94,72%	**94,04%**

Table 4.5: The validation and test accuracies for the five different data-splits obtained with support vector machine classifiers.

Support Vector Machine

Again the C-support vector classifier and the Linear-SVC from the Scikit-learn library [60], [58] have been used and the following parameters were tuned: the penalty parameter of the error C, the kernel coefficient γ for the rbf-kernel and the penalty norm and whether to use dual optimization for the Linear-SVC. Cross-validation has been used repeatedly in a five-fold way and thus the parameter settings were obtained separately for each data split. In all five cases the best results on the corresponding validation set have been reached with the linear kernel and the l_2 penalty norm. The error parameter C has varied from 0,5 to 1000 and in just one case the solving of the primal instead of the dual optimization problem has given better results. The obtained validation and test accuracies for the five data-splits are shown in detail in Table 4.5. Accordingly the average validation accuracy for this task has been 94,6% and the test accuracy has been **94%** for the support vector machine .
The training for one certain parameter setting has taken less than a minute for a linear kernel. For the rbf kernel it has taken about 6 minutes in average which is much faster than it had been for the first task. This has been due to the reduction of the feature vector dimension and the decrease in the number of training samples. The visualization of the SVM model was again not possible for this task as the features have been five-dimensional and therefore it would have been necessary to visualize the five-dimensional space.

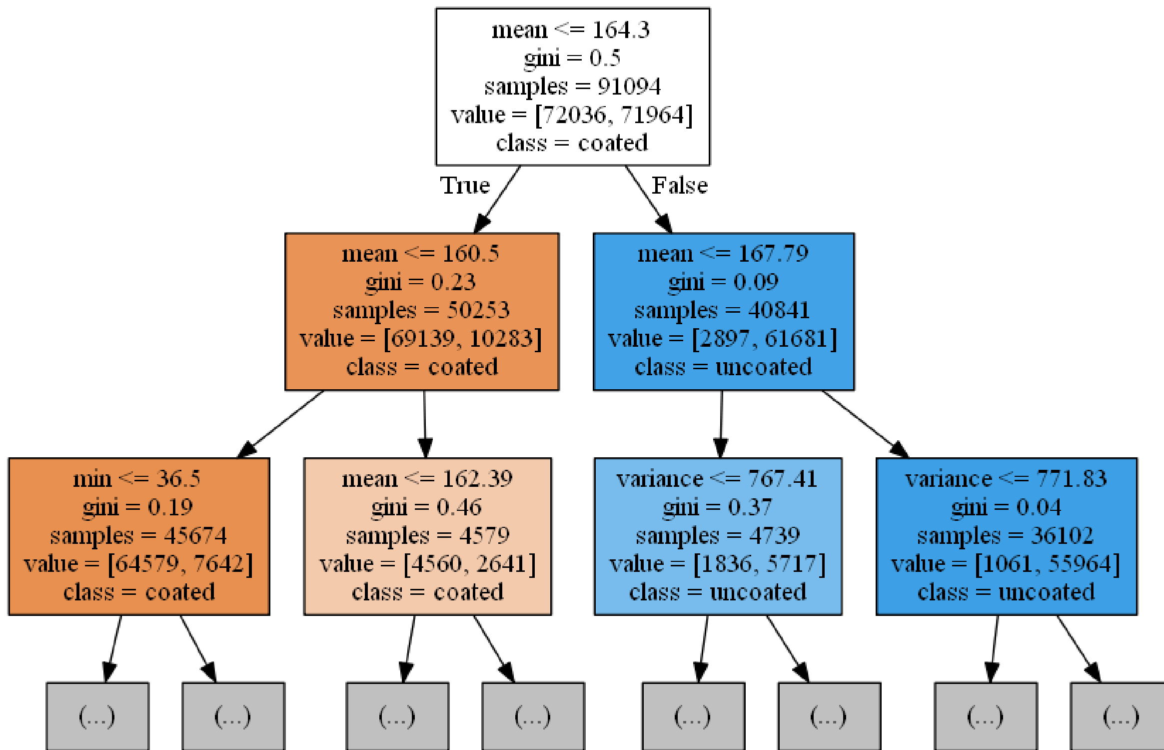


Figure 4.17: Detailed visualization of the first three depth-levels.

4.3.4 Comparison

In Table 4.6 an overview of the results is provided and allows to compare the different machine learning methods. All of them have performed well for the given task, but the results show that deep learning has again reached the best accuracy on the test set whereas the average time for training one parameter setting on the GPU has been within a good scope.

4.3.5 Remarks

(Nested) Cross-Validation

Again the need of (nested) cross-validation has been tested by examining the variance of the results for different test-set selections in the same way as before. Thus the whole dataset has been split into 42% training and validation samples and 16% test samples five times randomly. Using each data-split for the three different model architectures the variance of the test accuracies have been computed to get aware of how far the results of the different selection of training/validation/test samples diverge. The results can be seen in Figure 4.18. Note that in contrast to the boxplots of the first task shown in Figure 4.11 the range of accuracy

	Deep Learning	**Random Forest**	**Support Vector Machine**
Average accuracy on the independent test set	98,58%	95,11%	94,04%
Average training time for one parameter setting	$\sim$ 5 minutes	$<$1 minute	$<$1 min (linear) $\sim$ 6 minutes (rbf)

Table 4.6: Average accuracies and training times for the three different machine learning methods applied to the second task.

values now has been much bigger since it has spread from about 60% to about 99% whereas in the first task the range had just been from about 97% to about 99%. Since the variance now has been quite big for the plain model and the model with SGD-optimizer in comparison to the first task now using cross-validation has been required. As already stated in Section 3.1 and 3.2.4 cross-validation usually is computationally costly especially when it is performed in a nested way. Therefore we have decided to practice just the outer cross-validation for this task as it has been more important for us to get a better estimate for the generalization error that the selection of the hyper-parameters would have been.

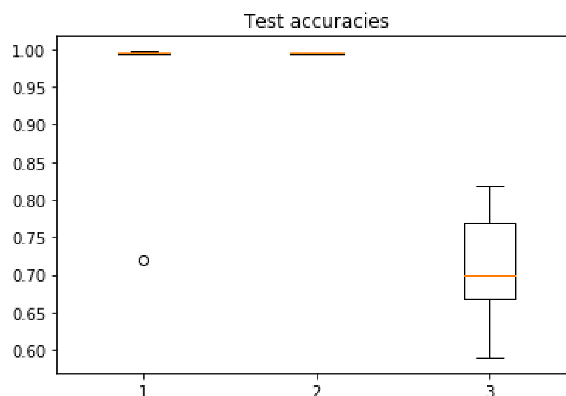


Figure 4.18: The test accuracies for the analyzation of the need of cross-validation. Left: The slightly modified MNIST model using dropout, variance = 0.012. Middle: The model without using dropout, variance = $9.645e^{-7}$. Right: the model using stochastic gradient descent as optimizer, variance = 0.006.

	Deep Learning	**Random Forest**	**Support Vector Machine**
Task 1: Material classification	99,47%	95,29%	95,19%
Task 2: Coating inspection	98,58%	95,11%	94,04%

Table 4.7: Accuracies for the three different machine learning methods on the independent test set respectively the average over those of the different data-splits needed for cross-validation for the second task.

4.4 Comparison and Conclusions

In Table 4.7 an overview of all the results is provided and allows to compare the different methods for both tasks. All machine learning methods have performed very well for the given tasks, but the results show that deep learning in both cases has reached the best accuracy on the test set.
As the range of possible parameters for the different machine learning methods is unlimited the tuning of this parameters could have always been continued further. Therefore, to make the three methods comparable, we have chosen to restrict the computational time used for the parameter tuning, as this may also be a restriction if the methods were used in an industrial context. The time which was needed for the training of a certain parameter setting differ strongly between the methods: A random forest classifier and the support vector machine with the linear kernel has usually taken less than a minute whereas a support vector machine classifier with a rbf-kernel has taken about 200 min in average. Neural networks can also run on the GPU which speeds up training and has taken just about 5-10 minutes in average. Better results for the standard methods therefore could have been reached by longer parameter-tuning runs on the one hand (especially for SVM) and on the other hand the extracted features may not have been good enough and should have been extended / replaced. Therefore, if computational time has not been a restriction and the extracted features have been optimized, then SVM and RF may have also reached as good results as DL. But spending much more time on the manual feature selection confirms even more the advantages of neural networks with their built-in feature selection capabilities.

4.5 Possible Extensions and Outlook

In this work we have described three different machine learning methods: deep learning, random forest and support vector machine. Furthermore these methods have been applied to two different classification tasks in the context of industrial materials with the use of OCT-images. For the future there are several ways to extend and maybe improve the work:

Other OCT-system

For the results presented in this thesis there had been used an industrial OCT-system which reaches an axial resolution of $7,5\mu m$. But there is also the opportunity to do the recordings on an in-house OCT-system which reaches an axial resolution of $2\mu m$. As the penetration depth is smaller for the in-house system it is only recommended to use it for the second task as in contrast to the first task here the information about the class lies on the object surface. Because of the higher resolution it can be assumed that the dark border between the coating and the base material appears stronger in OCT-recording of the in-house system than it does on the recordings of the industrial system. Accordingly having better images may lead to better results.

Use Google's MorphNet

For the optimization of deep learning models Google has developed a technique called MorphNet. It is an approach which iteratively shrinks and expands existing neural networks with several interesting goals: On the one hand it can optimize a network structure with respect to FLOPs or model size while keeping the performance high or on the other hand obtaining higher performance while keeping or even reducing the resource usage. Additionally it is scalable to large datasets and large models and is simple and fast to apply. This year Google released an open source version of MorphNet on github [62]. Applying MorphNet to the models obtained for this work may lead to better results. Further information can be found in [17].

Scattering Vectors

In [6] and [39] the application of scattering transform to image handwritten digit recognition and texture classification is demonstrated. By iterating over wavelet decompositions and complex modulus a scattering vector can be computed which holds local multi-scale and multi-direction co-occurrence information. It is also translation invariant and linearizes small deformations and therefore gives knowledge about the texture of an image without being

confused by small errors. The scattering vectors can be applied as features to standard machine learning methods and in that way yield state of the art results. One future approach for the given tasks, especially for the first task, can be to obtain the features for support vector machine and random forests with scattering transform.

5 Implementation

In the following we will present the most important parts of the code used for the practical part of this thesis. The code has been written in the Python programming language and is tested with WinPython 3.7.0.2. For the deep learning part the Keras library with Tensorflow as backend has been used. For the random forests and support vector machines the corresponding implementation from Scikit-learn have been used. Note that the following code snippets are just samples of the originally used code as it needs to be modified to correspond to the different tasks.

The first step is to split the data into training, validation and test set. This is done in two different ways depending on whether cross validation is needed.

```
"""
Splits and copies images of different classes to training, validation and test datasets.
Returns paths of the datasets.

Tested with WinPython64-3.7.0.2.

Author: Laura Peham <laura.peham@recendt.at>
"""
import os
import shutil
import random


def splitCopy(data_name, green_dirs, grey_dirs, red_dirs, trans_dirs, seed):
    '''
    Creats a folder 'data_name' in 'training-data' and randomly splits the given datasets
    into 50% training data, 25% validation and 25% test data.
    Then copies these as subfolders into 'training_name' s.t. the structure can be used
    by the ImageDataGenerator.
    '''
    train_prop = 0.5
    val_prop = 0.25
    # Test-prop is 1 - (train_prop + val_prop)
    random.seed(seed)

    base_dir = os.path.join('training-data', data_name)
    if not os.path.exists(base_dir):
```

```
        os.mkdir(base_dir)


    train_dir = os.path.join(base_dir, 'train')
    val_dir = os.path.join(base_dir, 'validation')
    test_dir = os.path.join(base_dir, 'test')
    # If the 'data_name' already exists it just returns the paths
    if os.path.exists(train_dir):
        return([train_dir, val_dir, test_dir])

    os.mkdir(train_dir)
    os.mkdir(val_dir)
    os.mkdir(test_dir)

    classes = {'green': green_dirs, 'grey': grey_dirs, 'red': red_dirs, 'trans': trans_dirs}

    for class_ in classes:
        train_class_dir = os.path.join(train_dir, class_)
        os.mkdir(train_class_dir)
        val_class_dir = os.path.join(val_dir, class_)
        os.mkdir(val_class_dir)
        test_class_dir = os.path.join(test_dir, class_)
        os.mkdir(test_class_dir)

        for d in classes[class_]:
            d = os.path.join('data', d)
            d = os.path.join(d, 'Image_data')
            # Since the OCT-images have 'intensity' in their name that's how they can be
    seperated from possible other images.
            files = [f for f in os.listdir(d) if 'intensity' in f]
            random.shuffle(files)
            n = len(files)
            for i in range(0, int(n*train_prop)):
                shutil.copyfile(os.path.join(d, files[i]), os.path.join(train_class_dir,
    files[i]))
            for i in range(int(n*train_prop), int(n*(train_prop+val_prop))):
                shutil.copyfile(os.path.join(d, files[i]), os.path.join(val_class_dir, files
    [i]))
            for i in range(int(n*(train_prop+val_prop)), n):
                shutil.copyfile(os.path.join(d, files[i]), os.path.join(test_class_dir,
    files[i]))


    return([train_dir, val_dir, test_dir])


# Splits the data in kFolds sets to use for cross validation
def splitCV(data_name, kFolds, green_dirs, grey_dirs, red_dirs, trans_dirs, seed):
    train_dirs = []
    val_dirs = []
    test_dirs = []
    random.seed(seed)

    base_dir = os.path.join('training-data', data_name)
    if not os.path.exists(base_dir):
        os.mkdir(base_dir)
```

```
classes = {'green': green_dirs, 'grey': grey_dirs, 'red': red_dirs, 'trans': trans_dirs}

files = {}
for class_ in classes:
    classFiles = []
    for d in classes[class_]:
        d = os.path.join('data', d)
        d = os.path.join(d, 'Image_data')
        # Since the OCT-images have 'intensity' in their name that's how they can be
seperated from possible other images.
        classFiles.extend(os.path.join(d,f) for f in os.listdir(d) if 'intensity' in f)
    random.shuffle(classFiles)
    files[class_] = classFiles

for i in range (1, kFolds+1):
    fold_dir = os.path.join(base_dir, 'Fold{}'.format(i))
    if not os.path.exists(fold_dir):
        os.mkdir(fold_dir)

    train_dir = os.path.join(fold_dir, 'train')
    val_dir = os.path.join(fold_dir, 'validation')
    test_dir = os.path.join(fold_dir, 'test')
    # If the 'data_name' already exists it just returns the paths
    if os.path.exists(train_dir):
        train_dirs.append(train_dir)
        val_dirs.append(val_dir)
        test_dirs.append(test_dir)
    else:
        os.mkdir(train_dir)
        os.mkdir(val_dir)
        os.mkdir(test_dir)

        for class_ in classes:
            train_class_dir = os.path.join(train_dir, class_)
            os.mkdir(train_class_dir)
            val_class_dir = os.path.join(val_dir, class_)
            os.mkdir(val_class_dir)
            test_class_dir = os.path.join(test_dir, class_)
            os.mkdir(test_class_dir)

            classFiles = files[class_]
            n = len(classFiles)//kFolds
            for j in range((i-1)*n, i*n):
                shutil.copyfile(classFiles[j], os.path.join(test_class_dir, os.path.
basename(classFiles[j])))
            classFiles = classFiles[:(i-1)*n] + classFiles[i*n:]

            random.shuffle(classFiles)
            n = len(classFiles)
            for j in range(n//2):
                shutil.copyfile(classFiles[j], os.path.join(train_class_dir, os.path.
basename(classFiles[j])))
            for j in range(n//2, n):
                shutil.copyfile(classFiles[j], os.path.join(val_class_dir, os.path.
basename(classFiles[j])))
```

```
            train_dirs.append(train_dir)
            val_dirs.append(val_dir)
            test_dirs.append(test_dir)


    return([train_dirs, val_dirs, test_dirs])
```

The next script shows how to randomly select the hyperparameters of a model, train the model on a given training set and evaluate it on the validation set. This script can be used to find the best hyperparameters for a certain task and the ranges for the parameters should be adjusted a few times according to the results.

```
"""
Script to select the best hyperparameters of a model.

Tested with WinPython64-3.7.0.2

Author: Laura Peham <laura.peham@recendt.at>
Based on a Script written by Robert Pollak
"""

import os
import time
import datetime
import numpy as np
from random import seed, randrange
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from keras.layers import Dense, Dropout, Flatten, BatchNormalization, Activation
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
from keras.models import Sequential
from keras.callbacks import CSVLogger, ModelCheckpoint, EarlyStopping
from keras.optimizers import Adadelta, SGD, Adam
from keras.preprocessing.image import ImageDataGenerator
from keras.losses import categorical_crossentropy
import statistics

import tensorflow as tf
from keras.backend.tensorflow_backend import set_session
config = tf.ConfigProto()
config.gpu_options.allow_growth = True  # dynamically grow the memory used on the GPU
sess = tf.Session(config=config)
set_session(sess)  # set this TensorFlow session as the default session for Keras

program = os.path.basename(__file__)
version = 'v1-0-0'

write_fit_files = bool(1)

use_optimal_hyperparams = bool(0)
```

```
if not use_optimal_hyperparams:
    version += '-randompars'

optilog_dir = 'opti-logs/'
optilog_id = ('opti_' + '_' + datetime.datetime.now().strftime('%Y-%m-%d_%H-%M') + '_' +
    version + '_')
optilog_name = optilog_dir + optilog_id + '.csv'
os.mkdir(optilog_dir + optilog_id)

# Dataset
train_dir = 'data_name\train'
val_dir = r'data_name\validation'
# Input image dimensions
img_dims = (50, 125)
num_classes = 4
units = [32, 64, 128, 128]
unit_dense = 256
n_conv = 3 # The number of convolutional layers
# The best acc result unitl now:
optimal_max_val_acc = 0.95


n = 1
nmax = 0
val_acc_list = []

header = 'n\tnmax\tunits0\tunits1\tunits2\tunits3\tunitDense\tconv_drop\tdense_drop\topt\
    tbatch_size\tBN\tval_acc\tmedian\tmins\tepochs'

while True:
    if n % 6 == 1: # s.t. even with CUDA log messages, there's always one on screen.
        print(program + ' ' + version + ', logs:', optilog_id)
        print(header)
    if n == 1:
        with open(optilog_name, 'a') as logfile:
            print(program + '-' + version, file=logfile)
            print(header, file=logfile, flush=True)
    log_filebase = optilog_id + '_{:03d}'.format(n)
    log_base = optilog_dir + optilog_id + '/' + log_filebase

    if use_optimal_hyperparams:
        np.random.seed(1)
        seed(1)
        # Hyperparameters:
        n_conv = 3
        units = [32, 64, 128, 0]
        unit_dense = 256
        conv_dropout = 0.1
        dense_dropout = 0.05
        opt_name = 'Adadelta'; optimizer = Adadelta()
        batch_Size = 32
        batch_Normalization = False

    else:
        np.random.seed(n)
        seed(log_filebase)
```

```
    # Randomly choose hyper-parameters:

    # Try different numbers of conv-layers: 2, 3, and 4
    n_conv = randrange(2, 5)

    #try different unit sizes for all layers (max 5 layers with variable number of units
)
    units = [0,0,0,0]
    for i in range(n_conv):
        k = randrange(1,11)
        units[i] = (k*20) # use just multiples of 20, from 20 to 200
    k = randrange(1,21)
    unit_dense = k*20 # From 20 to 400

    conv_dropout = randrange(10)/20
    dense_dropout = randrange(20)/20

    opt_n = randrange(3)
    if opt_n == 0:
        opt_name = 'Adam'; optimizer = Adam()
    elif opt_n == 1:
        opt_name = 'SGD'; optimizer = SGD(nesterov = True)
    else:
        opt_name = 'Adadelta'; optimizer = Adadelta()

    # As there are big gaps between the batch sizes, one can use 0.5 stepsize, but then
you need to round
    power = randrange(4,8)
    batch_Size = 2**power

    if randrange(2) == 0:
        batch_Normalization = True
    else:
        batch_Normalization = False


line = '{:3d}\t{:3d}\t{:2d}\t{:2d}\t{:2d}\t{:2d}\t{:2d}\t{:.2f}\t{:.2f}\t{:8s}\t{:2d}\t
{:3d}'.format(n, nmax, units[0], units[1], units[2], units[3], unit_dense, conv_dropout,
 dense_dropout, opt_name, batch_Size, batch_Normalization)

#line = '{:3d}\t{:3d}\t{:.2f}\t{:.2f}\t{:8s}\t{:2d}\t{:3d}'.format(n, nmax, conv_dropout
, dense_dropout, opt_name, batch_Size, batch_Normalization)
print(line, end = '')
with open(optilog_name, 'a') as logfile:
    print(line, end='', file=logfile, flush=True)


train_datagen = ImageDataGenerator(rescale = 1./255, horizontal_flip = True)
test_datagen = ImageDataGenerator(rescale = 1./255)

train_generator = train_datagen.flow_from_directory(train_dir, target_size = img_dims,
color_mode = 'grayscale', batch_size = batch_Size, class_mode = 'categorical')
val_generator = test_datagen.flow_from_directory(val_dir, target_size = img_dims,
color_mode = 'grayscale', batch_size = batch_Size, class_mode = 'categorical')
```

```
if K.image_data_format() == 'channels_first':
    input_shape = (1, img_dims[0], img_dims[1])
else:
    input_shape = (img_dims[0], img_dims[1], 1)


model = Sequential()

if batch_Normalization:
    model.add(Conv2D(units[0], kernel_size=(3, 3),
                     use_bias=False,
                     input_shape=input_shape))
    model.add(BatchNormalization())
    model.add(Activation("relu"))
    model.add(Conv2D(units[1], (3, 3), use_bias=False))
    model.add(BatchNormalization())
    model.add(Activation("relu"))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(conv_dropout))
    if not n_conv == 2:
        model.add(Conv2D(units[2], (3, 3), use_bias=False))
        model.add(BatchNormalization())
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=(2,2)))
        model.add(Dropout(conv_dropout))
        if n_conv == 4:
            model.add(Conv2D(units[3], (3, 3), use_bias=False))
            model.add(BatchNormalization())
            model.add(Activation("relu"))
            model.add(MaxPooling2D(pool_size=(2,2)))
            model.add(Dropout(conv_dropout))
    model.add(Flatten())
    model.add(Dense(unit_dense, use_bias=False))
    model.add(BatchNormalization())
    model.add(Activation("relu"))
    model.add(Dropout(dense_dropout))

else:
    model.add(Conv2D(units[0], kernel_size=(3, 3),
                     activation='relu',
                     input_shape=input_shape))
    model.add(Conv2D(units[1], (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(conv_dropout))
    if not n_conv == 2:
        model.add(Conv2D(units[2], (3, 3), activation='relu'))
        model.add(MaxPooling2D(pool_size=(2,2)))
        model.add(Dropout(conv_dropout))
        if n_conv == 4:
            model.add(Conv2D(units[3], (3, 3), activation='relu'))
            model.add(MaxPooling2D(pool_size=(2,2)))
            model.add(Dropout(conv_dropout))
    model.add(Flatten())
    model.add(Dense(unit_dense, activation='relu'))
    model.add(Dropout(dense_dropout))
```

```
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=categorical_crossentropy, optimizer=optimizer, metrics=['accuracy'])


callbacks=[
    # Stop at overfitting.
    EarlyStopping(monitor='val_acc', min_delta = 0.001, patience=3, verbose=1),
    # Save the best model.
    ModelCheckpoint(log_base + '.hdf5', monitor='val_acc', verbose=1, save_best_only=
True)
    ]

if write_fit_files:
    csv_logger = CSVLogger(log_base + '-acc.csv', separator='\t')
    callbacks.insert(0, csv_logger)

start = time.time()
history = model.fit_generator(train_generator, steps_per_epoch = len(train_generator),
epochs =100, verbose = 0,
                              validation_data = val_generator, validation_steps = len(
val_generator), shuffle = False,
                              callbacks=callbacks)

duration = round((time.time() - start)/60)


acc = history.history['acc']
try:
    val_acc = history.history['val_acc']
    max_val_acc = max(val_acc)
    epoch_of_max = np.argmax(val_acc) + 1

    val_acc_list.append(max_val_acc)
    median = statistics.median(val_acc_list)

    if max_val_acc > optimal_max_val_acc:
        nmax = n
        optimal_max_val_acc = max_val_acc


    # Plot results.
    fig = plt.figure(figsize=(10,10), dpi=75)
    epochs = np.arange(len(acc)) + 1
    plt.plot(epochs, acc, 'bo', label='Training acc')
    plt.plot(epochs, val_acc, 'b', label='Validation acc')
    plt.xlabel('Epoch')
    plt.ylabel('Accury')
    plt.grid()
    plt.title('Training and validation accuracy')
    plt.legend()
    plt.tight_layout()
    if write_fit_files:
        plt.savefig(log_base + '-acc.png')
    plt.close(fig)
```

```
    except KeyError:
        max_val_acc = np.nan

        if n==1: # else it just stays the same.
            median = np.nan

        epoch_of_max = -1

    result_line = '\t{:.5f}\t{:.5f}\t{:3d}\t{:3d}'.format(
         max_val_acc, median, duration, epoch_of_max)

    print('' + result_line)
    with open(optilog_name, 'a') as logfile:
        print(result_line, file=logfile, flush=True)

    # Prevent memory leaks
    K.clear_session()
    tf.reset_default_graph()

    n += 1
    #break
```

After finding the model with the best validation accuracy scores it needs to be evaluated on the test set. The functions in the following script can load pre-trained models and evaluate certain test sets with them. There is also a function the get the wrongly classified images of a test set.

```
'''
Evaluate a pretrained model on the test set

Tested with WinPython64-3.7.0.2.

Author: Laura Peham <laura.peham@recendt.at>
'''

import keras
from keras.preprocessing.image import ImageDataGenerator
from PIL import Image
import numpy as np
from skimage import transform
import os
from statistics import mean

# Evaluate the model on a test set
# The image dimensions need to be equivalent to the ones the model has been trained with
def evaluate(model_path, test_dir, img_rows, img_cols):
    model = keras.models.load_model(model_path)

    datagen = ImageDataGenerator(rescale = 1./255)
    test_generator = datagen.flow_from_directory(test_dir, target_size = (img_rows, img_cols
    ), color_mode = 'grayscale', batch_size = 32, class_mode = 'categorical')

    test_loss, test_acc = model.evaluate_generator(test_generator, steps = len(
```

```
    test_generator), verbose = 1)
    print('test_loss = ' , test_loss)
    print('test_acc = ', test_acc)
    return([test_loss , test_acc])


# Returns the class labels and it's prediciton probabilities of a test set predicted by a
    pre-trained model
# The image dimensions need to be equivalent to the ones the model has been trained with
def predict(model_path , test_dir , img_rows , img_cols):
    model = keras.models.load_model(model_path)

    datagen = ImageDataGenerator(rescale = 1./255)
    test_generator = datagen.flow_from_directory(test_dir , target_size = (img_rows, img_cols
    ), color_mode = 'grayscale', batch_size = 32, class_mode = 'categorical', shuffle =
    False)

    pred = model.predict_generator(test_generator , steps = len(test_generator), verbose = 1)
    labels = test_generator.classes
    return(pred , labels)


# Returns the class labels and it's prediciton probabilities of one single input image
    predicted by a pre-trained model
# The image dimensions need to be equivalent to the ones the model has been trained with
def predictOne(model_path , image_path , img_rows , img_cols):
    model = keras.models.load_model(model_path)

    np_image = Image.open(image_path)
    np_image = np.array(np_image).astype('float32')/255
    np_image = transform.resize(np_image , (img_rows , img_cols , 3))
    np_image = np.expand_dims(np_image , axis=0)

    print(model.predict(np_image))


# Returns a list with the false predicted images
# The input 'npArr' needs to be the first output array of 'predict()'
def getFalseImg(npArr , testSet):
    n,m = npArr.shape
    classes = ['green','grey', 'red', 'trans']
    falseImg = []
    i = 0
    classN = 0
    for class_ in classes:
        classP = os.path.join(testSet , class_)
        files = [f for f in os.listdir(classP)]
        for j in range(i, i+len(files)):
            if(npArr[j][classN] < 0.5):
                falseImg.append(files[j-i])
        i += len(files)
        classN += 1

    return(falseImg)
```

To train a random forest and support vector classifier the chosen features needs to be extracted from the image patches. For an OCT-image first the object border is flattened and bad areas are cropped and than it is split into several patches of a certain size. The features of these patches are then used to train a model.

```
"""
This script is for classifying OCT images w.r.t. their material with Random Forests and
    Support Vector Machines
Therefore features need to be extracted out of image patches

Tested with WinPython64-3.7.0.2.

Author: Laura Peham <laura.peham@recendt.at>
"""
import numpy as np
from PIL import Image
import statistics
import os
from random import seed, randrange
seed(1)

# Normalize the OCT-image s.t. the upper border is for every column the first pixel brighter
     than 100
# and the image is only 50 pixels high as no more is needed to create the 50x50 patches
# Sort out lower slopes and defects as there is no important information under the border
    there
def normalizeImg(img):
    image = np.array(Image.open(img))
    # In the upper image there are often very bright areas, so crop the first 40 lines
    image = image[40: , : ]


    thresholds = []
    for i in range(0, image.shape[1]):
        for j in range(0, image.shape[0]):
            if image[j][i] > 100:  # grayscale threshold, found out after looking at
    different images
                thresholds.append(j)
                break
            if j == image.shape[0]-1: # if it never gets bright enough
                thresholds.append(image.shape[0])

    median = statistics.median(thresholds) # to get an idea what is too low for threshold
    thres = thresholds[0]
    haveArray = False
    for i in range(0, image.shape[1]):
        thres = thresholds[i]
        if thres < (median + 100) and thres + 50 < image.shape[0]:  # To sort out too low
    values (just 100 here as surface is more or less straight)
            if haveArray == False:
                normalImg = np.array(image[thres:thres+50, i])
                haveArray = True
            else:
                normalImg = np.column_stack((normalImg, image[thres:thres+50, i]))
```

```
    return(normalImg)


# Split the (normalized) image into 50x50 patches (as many as will fit there)
def getPatches(normalImg):
    if randrange(2) == 0:
            normalImg = np.fliplr(normalImg)  # Random flipping the image horizontally

    patches = np.zeros((50,50), dtype=np.uint8)
    for i in range((normalImg.shape[1])//50):
        patches = np.dstack((patches, normalImg[:50, 50*i : 50*(i+1)]))
    patches = patches[:, :, 1:] # delete zero array

    return(patches)


# Return the image features given a 50x50 patch
def getPatchFeat(patch):
    mean_ = np.mean(patch)
    median_ = np.median(patch)
    var_ = np.var(patch)
    max_ = np.amax(patch)
    min_ = np.amin(patch)

    return([mean_, median_, var_, max_, min_])


# Return a list of image features and write them in a log file for all patches of a given
    image
def getImgFeat(imgPath, dataName, classSet):
    name = os.path.splitext(os.path.basename(imgPath))[0]
    trainDir = os.path.join('image-features', dataName)
    if not os.path.exists(trainDir):
        os.mkdir(trainDir)

    save = os.path.join(trainDir, classSet)
    if not os.path.exists(save):
        os.mkdir(save)

    logName =  os.path.join(save, 'features.csv')

    header = 'name\tn\tmean\tmedian\tvariance\tmax\tmin'
    with open(logName, 'a') as logfile:
        print(header, file=logfile, flush=True)

    imgFeatures = []
    imgPatches = getPatches(normalizeImg(imgPath))
    for i in range(imgPatches.shape[2]):
            patch = imgPatches[:,:,i]

            # save the features
            feat = getPatchFeat(patch)
            imgFeatures.append(feat)
            line = '{}\t{:3d}\t{:.2f}\t{:.2f}\t{:.2f}\t{:.2f}\t{:.2f}'.format(name, i, feat
    [0], feat[1], feat[2], feat[3], feat[4])
            with open(logName, 'a') as logfile:
```

```
                print(line, file=logfile, flush=True)

            # save the image patches
            patchImg = Image.fromarray(patch).convert("L")
            patchImg.save(os.path.join(save, name + str(i) + '.png'))

    return((imgFeatures, imgPatches.shape[2]))


# Get a list of the images features for a whole set of images
def getFeatures(className, path, dataName, setName):
    classPath = os.path.join(path, className)
    colorImgs = [f for f in os.listdir(classPath)]
    features = []
    numberPatch = [] # Save the numer of patches per image
    classSet = className + '-' + setName  # e.g. green-train, red-validation
    print('Evaluating ' + classSet + '...')
    for img in colorImgs:
        path = os.path.join(classPath, img)
        imgFeat = getImgFeat(path, dataName, classSet)
        features.extend(imgFeat[0])
        numberPatch.append(imgFeat[1])
    return((features, numberPatch))


# Create the feature-arrays for training, testing and validation for all classes
# given an already splitted dataset
def createData(data, dataName):
    classes = ['green', 'grey', 'red', 'trans']

    dataSave = os.path.join('training-data', dataName)
    if not os.path.exists(dataSave):
        os.mkdir(dataSave)

    # Check if the arrays already exist -> just load them instead of creating them again
    npdata = os.path.join(dataSave, 'trainX.npy')
    if os.path.isfile(npdata): #Assume that if one exists all will exist
        trainX = np.load(os.path.join(dataSave, 'trainX.npy'))
        trainY = np.load(os.path.join(dataSave, 'trainY.npy'))
        trainPatNr = np.load(os.path.join(dataSave, 'trainPatNr.npy'))
        valX = np.load(os.path.join(dataSave, 'valX.npy'))
        valY = np.load(os.path.join(dataSave, 'valY.npy'))
        valPatNr = np.load(os.path.join(dataSave, 'valPatNr.npy'))
        testX = np.load(os.path.join(dataSave, 'testX.npy'))
        testY = np.load(os.path.join(dataSave, 'testY.npy'))
        testPatNr = np.load(os.path.join(dataSave, 'testPatNr.npy'))

    else:
        trainX = []
        trainY = []
        valX = []
        valY = []
        testX = []
        testY = []
        trainPatNr = []
        valPatNr = []
```

```
        testPatNr = []
        i = 0
        for class_ in classes:
            trainPath = os.path.join(data, 'train')
            feat = getFeatures(class_, trainPath, dataName, 'train')
            trainF = np.array(feat[0])
            trainPatNr.extend(feat[1])

            valPath = os.path.join(data, 'validation')
            feat = getFeatures(class_, valPath, dataName, 'validation')
            valF = np.array(feat[0])
            valPatNr.extend(feat[1])

            testPath = os.path.join(data, 'test')
            feat = getFeatures(class_, testPath, dataName, 'test')
            testF = np.array(feat[0])
            testPatNr.extend(feat[1])

            trainX.extend(trainF)
            trainY.extend([i]*np.shape(trainF)[0])
            valX.extend(valF)
            valY.extend([i]*np.shape(valF)[0])
            testX.extend(testF)
            testY.extend([i]*np.shape(testF)[0])

            i+=1
        # Save the arrays to be able to use them again later
        np.save(os.path.join(dataSave, 'trainX.npy'), trainX)
        np.save(os.path.join(dataSave, 'trainY.npy'), trainY)
        np.save(os.path.join(dataSave, 'trainPatNr.npy'), trainPatNr)
        np.save(os.path.join(dataSave, 'valX.npy'), valX)
        np.save(os.path.join(dataSave, 'valY.npy'), valY)
        np.save(os.path.join(dataSave, 'valPatNr.npy'), valPatNr)
        np.save(os.path.join(dataSave, 'testX.npy'), testX)
        np.save(os.path.join(dataSave, 'testY.npy'), testY)
        np.save(os.path.join(dataSave, 'testPatNr.npy'), testPatNr)

    return([trainX, trainY, trainPatNr, valX, valY, valPatNr, testX, testY, testPatNr])
```

The script for the random forest classfier includes two functions: the tuning function is similar to the one of deep learning and it can be used to find the best hyperparamters for a given task. The train function can be used to retrain a model with certain parameters and then instantly evaluate it on the test set. For this thesis after finding the best hyperparameter setting the model has been retrained on the training and validation data and then evaluated on the test set.

```
"""
This script is for classifying OCT images w.r.t. their material with a random forest
    classifier

Tested with WinPython64-3.7.0.2.
```

```
Author: Laura Peham <laura.peham@recendt.at>
"""

from sklearn.ensemble import RandomForestClassifier
from image_features import createData
from statistics import median
from sklearn.externals import joblib
import os
import time
import datetime
from random import seed, randrange
from numpy import nan
import numpy as np



def getMaxIndex(array):
    maxIndex = 0
    for i in range(1, len(array)):
        if array[i] > array[maxIndex]:
            maxIndex = i
    return(maxIndex)

def trainRF():
    data = 'path/to/data'
    trainName = 'training_name'
    dataName = 'data_name'

    trainX, trainY, trainPatNr, valX, valY, valPatNr, testX, testY, testPatNr = createData(
    data, dataName)

    # Uncomment the following lines and comment out the next two if a pretrained model
    should be load instead of training a new one
    #model_path = 'path/to/pretrained/model'
    #clf = joblib.load(model_path)

    clf = RandomForestClassifier(n_estimators=32, max_depth = 36, min_samples_split = 1,
    min_samples_leaf = 4, max_features = 2, random_state=0)
    clf.fit(trainX, trainY)

    save = os.path.join('image-features', trainName + '.sav')
    joblib.dump(clf, save)

    print(clf.feature_importances_)

    i = 0
    valImgProb = []
    valImgY = []
    for patNr in valPatNr:
        valImgY.append(valY[i])

        valImgX = valX[i:i+patNr]
        valProb = clf.predict_proba(valImgX)
        class0 = [p[0] for p in valProb]
        class1 = [p[1] for p in valProb]
        class2 = [p[2] for p in valProb]
```

```
        class3 = [p[3] for p in valProb]
        pred = [median(class0), median(class1), median(class2), median(class3)]
        valImgProb.append(pred)
        i += patNr

    valImgPred = []
    for i in range(len(valImgProb)):
        valImgPred.append(getMaxIndex(valImgProb[i]))

    i = 0
    testImgProb = []
    testImgY = []
    for patNr in testPatNr:
        testImgY.append(testY[i])

        testImgX = testX[i:i+patNr]
        testProb = clf.predict_proba(testImgX)
        class0 = [p[0] for p in testProb]
        class1 = [p[1] for p in testProb]
        class2 = [p[2] for p in valProb]
        class3 = [p[3] for p in valProb]
        pred = [median(class0), median(class1), median(class2), median(class3)]
        testImgProb.append(pred)
        i += patNr

    testImgPred = []
    for i in range(len(testImgProb)):
        testImgPred.append(getMaxIndex(testImgProb[i]))

    val_acc = sum([i == j for i, j in zip(valImgPred, valImgY)]) / len(valImgPred)
    print('Validation accuracy: ' + str(val_acc))
    patch_acc = clf.score(valX, valY)
    print('Validation accuracy for individual patches: ' + str(patch_acc))
    patch_acc_train = clf.score(trainX, trainY)
    print('Training accuracy for individual patches: ' + str(patch_acc_train))
    test_acc = sum([i == j for i, j in zip(testImgPred, testImgY)]) / len(testImgPred)
    print('Test accuracy: ' + str(test_acc))
    patch_acc_test = clf.score(testX, testY)
    print('Test accuracy for individual patches: ' + str(patch_acc_test))

    file = open(os.path.join('image-features', trainName + '.txt'),'w')
    file.write('Feature Importances: ' + str(clf.feature_importances_) + '\n')
    file.write('Validation Accuracy: ' + str(val_acc) + '\n')
    file.write('Validation Accuracy for individual patches: ' + str(patch_acc) + '\n')
    file.write('Training accuracy for individual patches: ' + str(patch_acc_train) + '\n')
    file.write('Test Accuracy: ' + str(test_acc) + '\n')
    file.write('Test Accuracy for individual patches: ' + str(patch_acc_test))
    file.close()



# Tuning parameters according to: https://medium.com/all-things-ai/in-depth-parameter-tuning
    -for-random-forest-d67bb7e920d
def tuningRF():
    data = 'path/to/data'
    dataName = 'data_name'
```

```
program = os.path.basename(__file__)
version = 'v1-0-0'

use_optimal_hyperparams = bool(0)
if not use_optimal_hyperparams:
    version += '-randompars'

optilog_dir = 'opti-logs/'
optilog_id = ('opti_RF_' +  '_' + datetime.datetime.now().strftime('%Y-%m-%d_%H-%M') + '
_' + version )
optilog_name = optilog_dir + optilog_id + '.csv'
os.mkdir(optilog_dir + optilog_id)

optimal_max_val_acc = 0.90
n = 1
nmax = 0
val_acc_list = []
header = 'n\tnmax\tn_estimators\tmax_depth\tmin_samples_split\tmin_samples_leaf\
tmax_features\tval_acc\tval_acc-patches\ttraining_acc-patches\tmins'
while True:
    if n % 6 == 1: # s.t. even with CUDA log messages, there's always one on screen.
        print(program + ' ' + version + ', logs:', optilog_id)
        print(header)
    if n == 1:
        with open(optilog_name, 'a') as logfile:
            print(program + '-' + version, file=logfile)
            print(header, file=logfile, flush=True)
    log_filebase = optilog_id + '_{:03d}'.format(n)
    log_base = optilog_dir + optilog_id + '/' + log_filebase

    if use_optimal_hyperparams:
        seed(n)

        # Hyperparameters:
        n_estimator_ = 32
        max_depth_ = 36
        min_samples_split_ = 1
        min_samples_leaf_ = 4
        max_feature_ = 2

    else:
        seed(log_filebase)

        n_estimators = [1, 2, 4, 8, 16, 32, 64, 100, 128, 200, 256]
        max_depths = np.linspace(1, 30, 30, endpoint=True)
        min_samples_splits = [2,3,4,5,6,7,8,9,10]
        min_samples_leafs = [1,2,3,4,5,6,7,8,9,10]
        max_features = [2,3,4,5]

        # Randomly choose hyper-parameters:
        r = randrange(11)
        n_estimator_ = n_estimators[r]
        r = randrange(30)
        max_depth_ = max_depths[r]
        r = randrange(9)
```

```
        min_samples_split_ = min_samples_splits[r]
        r = randrange(10)
        min_samples_leaf_ = min_samples_leafs[r]
        r = randrange(4)
        max_feature_ = max_features[r]


    line = '{:3d}\t{:3d}\t{:3d}\t{:8s}\t{:3d}\t{:3d}\t{:8s}'.format(n, nmax,
n_estimator_, str(max_depth_), min_samples_split_, min_samples_leaf_, str(max_feature_))
    print(line, end = '')
    with open(optilog_name, 'a') as logfile:
        print(line, end='', file=logfile, flush=True)

    trainX, trainY, trainPatNr, valX, valY, valPatNr, testX, testY, testPatNr =
createData(data, dataName)
    clf = RandomForestClassifier(n_estimators = n_estimator_, max_depth = max_depth_,
min_samples_split = min_samples_split_, min_samples_leaf = min_samples_leaf_,
max_features = max_feature_,  random_state=0)

    start = time.time()
    clf.fit(trainX, trainY)
    duration = round((time.time() - start)/60)

    try:
        i = 0
        valImgProb = []
        valImgY = []
        for patNr in valPatNr:
            valImgY.append(valY[i])

            valImgX = valX[i:i+patNr]
            valProb = clf.predict_proba(valImgX)
            class0 = [p[0] for p in valProb]
            class1 = [p[1] for p in valProb]
            class2 = [p[2] for p in valProb]
            class3 = [p[3] for p in valProb]
            pred = [median(class0), median(class1), median(class2), median(class3)]
            valImgProb.append(pred)
            i += patNr

        valImgPred = []
        for i in range(len(valImgProb)):
            valImgPred.append(getMaxIndex(valImgProb[i]))

        val_acc = sum([i == j for i, j in zip(valImgPred, valImgY)]) / len(valImgPred)
        patch_acc = clf.score(valX, valY)
        patch_acc_train = clf.score(trainX, trainY)

        val_acc_list.append(val_acc)

        if val_acc > optimal_max_val_acc:
            nmax = n
            optimal_max_val_acc = val_acc


    except KeyError:
```

```
            val_acc = nan

        # Save the model
        joblib.dump(clf, log_base + '.hdf5')

        result_line = '\t{:.5f}\t{:.5f}\t{:.5f}\t{:3d}'.format(
            val_acc, patch_acc, patch_acc_train, duration)

        print('' + result_line)
        with open(optilog_name, 'a') as logfile:
            print(result_line, file=logfile, flush=True)

        n += 1
```

The script for the support vector machine is nearly equivalent to the one of random forests.

```
"""
This script is for classifying OCT images w.r.t. their material with a SVM classifier

Tested with WinPython64-3.7.0.2.

Author: Laura Peham <laura.peham@recendt.at>
"""

from sklearn.svm import SVC, LinearSVC
from sklearn.preprocessing import StandardScaler
from image_features import createData
from statistics import median, mean
import joblib
import os
import time
import datetime
from random import seed, randrange
from numpy import nan


def getMaxIndex(array):
    maxIndex = 0
    for i in range(1, len(array)):
        if array[i] > array[maxIndex]:
            maxIndex = i
    return(maxIndex)


def trainSVC():
    data = 'path/to/data'
    trainName = 'training_name'
    dataName = 'data_name'

    trainX, trainY, trainPatNr, valX, valY, valPatNr, testX, testY, testPatNr = createData(
    data, dataName)

    # Uncomment the following lines and comment out the next two if a pretrained model
    should be load instead of training a new one
    #model_path = 'path/to/pretrained/model'
```

```
#clf = joblib.load(model_path)

clf = SVC(C = 500, gamma = 0.01, verbose = True)
clf.fit(trainX, trainY)

save = os.path.join('image-features', trainName + '.sav')
joblib.dump(clf, save)

i = 0
valImgProb = []
valImgY = []
for patNr in valPatNr:
    valImgY.append(valY[i])

    valImgX = valX[i:i+patNr]
    valProb = clf.predict_proba(valImgX)
    class0 = [p[0] for p in valProb]
    class1 = [p[1] for p in valProb]
    class2 = [p[2] for p in valProb]
    class3 = [p[3] for p in valProb]
    pred = [median(class0), median(class1), median(class2), median(class3)]
    valImgProb.append(pred)
    i += patNr

valImgPred = []
for i in range(len(valImgProb)):
        valImgPred.append(getMaxIndex(valImgProb[i]))

i = 0
testImgProb = []
testImgY = []
for patNr in testPatNr:
    testImgY.append(testY[i])

    testImgX = testX[i:i+patNr]
    testProb = clf.predict_proba(testImgX)
    class0 = [p[0] for p in valProb]
    class1 = [p[1] for p in valProb]
    class2 = [p[2] for p in valProb]
    class3 = [p[3] for p in valProb]
    pred = [median(class0), median(class1), median(class2), median(class3)]
    testImgProb.append(pred)
    i += patNr

testImgPred = []
for i in range(len(testImgProb)):
        testImgPred.append(getMaxIndex(testImgProb[i]))

val_acc = sum([i == j for i, j in zip(valImgPred, valImgY)]) / len(valImgPred)
print('Validation accuracy: ' + str(val_acc))
patch_acc = clf.score(valX, valY)
print('Validation accuracy for individual patches: ' + str(patch_acc))
patch_acc_train = clf.score(trainX, trainY)
print('Training accuracy for individual patches: ' + str(patch_acc_train))
test_acc = sum([i == j for i, j in zip(testImgPred, testImgY)]) / len(testImgPred)
print('Test accuracy: ' + str(test_acc))
```

```
    patch_acc_test = clf.score(testX, testY)
    print('Test accuracy for individual patches: ' + str(patch_acc_test))

    file = open(os.path.join('image-features', trainName + '.txt'),'w')
    file.write('Validation Accuracy: ' + str(val_acc) + '\n')
    file.write('Validation Accuracy for individual patches: ' + str(patch_acc) + '\n')
    file.write('Training accuracy for individual patches: ' + str(patch_acc_train) + '\n')
    file.write('Test Accuracy: ' + str(test_acc) + '\n')
    file.write('Test Accuracy for individual patches: ' + str(patch_acc_test))
    file.close()




# Tuning parameters according to: https://medium.com/all-things-ai/in-depth-parameter-tuning
    -for-svc-758215394769
def tuningSVC():
    data = 'path/to/data'
    dataName = 'data_name'

    program = os.path.basename(__file__)
    version = 'v1-0-0'

    use_optimal_hyperparams = bool(0)
    if not use_optimal_hyperparams:
        version += '-randompars'

    optilog_dir = 'opti-logs/'
    optilog_id = ('opti_SVC_' +  '_' + datetime.datetime.now().strftime('%Y-%m-%d_%H-%M') +
    '_' + version )
    optilog_name = optilog_dir + optilog_id + '.csv'
    os.mkdir(optilog_dir + optilog_id)

    optimal_max_val_acc = 0.90
    n = 1
    nmax = 0
    val_acc_list = []
    header = 'n\tnmax\tc\tkernel\tgammaORpenalty\tdual(linear)\tval_acc\tval_acc-patches\
    ttraining_acc-patches\tmins'
    while True:
        if n % 6 == 1: # s.t. even with CUDA log messages, there's always one on screen.
            print(program + ' ' + version + ', logs:', optilog_id)
            print(header)
        if n == 1:
            with open(optilog_name, 'a') as logfile:
                print(program + '-' + version, file=logfile)
                print(header, file=logfile, flush=True)
        log_filebase = optilog_id + '_{:03d}'.format(n)
        log_base = optilog_dir + optilog_id + '/' + log_filebase

        if use_optimal_hyperparams:
            seed(n)

            # Hyperparameters:
            c_ = 500
            kernel_ = 'rbf'
```

```
        gamma_ = 0.01
        #penalty_ = 'l2'
        #dual_ = 0

    else:
        seed(log_filebase)

        # Randomly choose hyper-parameters:
        cs = [0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000]
        dual_ = randrange(2)
        gammas = [0.01, 0.1, 1, 10, 100]

        r = randrange(13)
        c_ = cs[r]
        r = randrange(2)
        if r == 0:
            kernel_ = 'linear'
        else:
            kernel_ = 'rbf'
        r = randrange(5)
        gamma_ = gammas[r]
        r = randrange(2)
        if r == 0:
            penalty_ = 'l1'
        else:
            penalty_ = 'l2'


    if kernel_ == 'linear':
        line = '{:3d}\t{:3d}\t{:.2f}\t{:8s}\t{:8s}\t{:3d}'.format(n, nmax, c_, kernel_,
 penalty_, dual_)
    else:
        line = '{:3d}\t{:3d}\t{:.2f}\t{:8s}\t{:8s}\t'.format(n, nmax, c_, kernel_, str(
gamma_))
    print(line, end = '')
    with open(optilog_name, 'a') as logfile:
        print(line, end='', file=logfile, flush=True)

    trainX, trainY, trainPatNr, valX, valY, valPatNr, testX, testY, testPatNr =
createData(data, dataName)
    if kernel_ == 'linear':
        clf = LinearSVC(C = c_, penalty = penalty_, dual = bool(dual_), verbose = True)
    else:
        clf = SVC(C = c_, kernel = kernel_, gamma = gamma_, verbose = True)

    scaler = StandardScaler().fit(trainX)
    scaler.transform(trainX)
    scaler.transform(valX)

    start = time.time()
    clf.fit(trainX, trainY)
    duration = round((time.time() - start)/60)

    try:
        i = 0
        valImgProb = []
```

```
        valImgY = []
        for patNr in valPatNr:
            valImgY.append(valY[i])

            valImgX = valX[i:i+patNr]
            valProb = clf.predict_proba(valImgX)
            class0 = [p[0] for p in valProb]
            class1 = [p[1] for p in valProb]
            class2 = [p[2] for p in valProb]
            class3 = [p[3] for p in valProb]
            pred = [median(class0), median(class1), median(class2), median(class3)]
            valImgProb.append(pred)
            i += patNr

        valImgPred = []
        for i in range(len(valImgProb)):
            valImgPred.append(getMaxIndex(valImgProb[i]))

        val_acc = sum([i == j for i, j in zip(valImgPred, valImgY)]) / len(valImgPred)
        patch_acc = clf.score(valX, valY)
        patch_acc_train = clf.score(trainX, trainY)

        val_acc_list.append(val_acc)

        if val_acc > optimal_max_val_acc:
            nmax = n
            optimal_max_val_acc = val_acc


    except KeyError:
        val_acc = nan

    joblib.dump(clf, log_base + '.hdf5')


    result_line = '\t{:.5f}\t{:.5f}\t{:.5f}\t{:3d}'.format(
         val_acc, patch_acc, patch_acc_train, duration)

    print('' + result_line)
    with open(optilog_name, 'a') as logfile:
        print(result_line, file=logfile, flush=True)

    n += 1
```

Bibliography

[1] C. C. Aggarwal. *Neural Networks and Deep Learning. A Textbook.* Springer, Heidelberg, 2018.

[2] S. V. Alex Smola. *Introduction to Machine Learning*. Cambridge University Press, Cambridge, 2008.

[3] M. Awais, H. Müller, and F. Meriaudeau. Classification of SD-OCT Images Using Deep Learning Approach. In *2017 IEEE International Conference on Signal and Image Processing Applications (ICSIPA)*, pages 489–492, 2017.

[4] A. Botchkarev. Performance Metrics (Error Measures) in Machine Learning Regression, Forecasting and Prognostics: Properties and Typology. *ArXiv*, abs/1809.03006, 2018.

[5] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Taylor & Francis, London, 1984.

[6] J. Bruna and S. Mallat. Classification with Scattering Operators. *ArXiv*, abs/1011.3023, 2010.

[7] I. Bussel, G. Wollstein, and J. Schuman. OCT for Glaucoma Diagnosis, Screening and Detection of Glaucoma Progression. *The British journal of ophthalmology*, 98, 2013.

[8] F. Chollet. *Deep Learning with Python*. Manning Publications Co., Greenwich, CT, USA, 2017.

[9] W. Cong, X. Intes, and G. Wang. Optical Tomographic Imaging for Breast Cancer Detection. *Journal of Biomedical Optics*, 22, 2017.

[10] R. F. de Mello and M. A. Ponti. *Machine Learning. A Practical Approach on the Statistical Learning Theory.* Springer, Heidelberg, 2018.

[11] W. Drexler and J. Fujimoto. *Optical Coherence Tomography: Technology and Applications*. Springer, Heidelberg, 2008.

[12] J. Duker, N. Waheed, and D. Goldman. *Handbook of Retinal OCT: Optical Coherence Tomography*. Elsevier Health Sciences, Canada, 2014.

[13] T. Engen-Skaugen, J. Erik Haugen, and O. Tveito. Beyond Biomedicine: A Review of Alternative Applications and Developments for Optical Coherence Tomography. *Applied Physics B: Lasers and Optics*, 88:337–357, 2007.

[14] A. F. Fercher, C. K. Hitzenberger, W. Drexler, G. Kamp, and H. Sattmann. In Vivo Optical Coherence Tomography. *American Journal of Ophthalmology*, 116(3):113–114, 1993.

[15] S. Gollapudi. *Practical Machine Learning*. Packt Publishing, Birmingham, 2016.

[16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning.* MIT Press, Cambridge, 2016.

[17] A. Gordon, E. Eban, O. Nachum, B. Chen, H. Wu, T.-J. Yang, and E. Choi. MorphNet: Fast & Simple Resource-Constrained Structure Learning of Deep Networks. In *ArXiv*, volume abs/1711.06798, 2017.

[18] G. Hannesschläger, A. Nemeth, C. Hofer, C. Goetzloff, J. Reussner, K. Wiesauer, and M. Leitner. Optical Coherence Tomography as a Tool for Non-destructive Quality Control of Multi-layered Foils. *Proceedings of the 6th NDT in progress*, 2011.

[19] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning. Data Mining, Inference and Prediction. 2nd ed.* Springer, New York, 2009.

[20] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[21] B. Heise, S. Schausberger, and D. Stifter. *Full Field Optical Coherence Microscopy: Imaging and Image Processing for Micro-Material Research Applications*. IntechOpen, London, 2013.

[22] S. Hochreiter. The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6:107–116, 04 1998.

[23] D. Huang, E. Swanson, C. Lin, J. Schuman, W. Stinson, W. Chang, M. Hee, T. Flotte, K. Gregory, C. Puliafito, and a. et. Optical Coherence Tomography. *Science*, 254(5035):1178–1181, 1991.

[24] D. H. Hubel and T. N. Wiesel. Receptive Fields of Single Neurones in the Cat's Striate Cortex. *The Journal of physiology*, 148(3), 1959.

[25] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37, pages 448–456, Lille, France, 2015.

[26] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp . *ArXiv*, abs/1609.04836, 2016.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Neural Information Processing Systems*, 25, 2012.

[28] V. Kumar and P. Gupta. Importance of Statistical Measures in Digital Image Processing. *International Journal of Emerging Technology and Advanced Engineering*, 2, 08 2012.

[29] C. Lee, D. M. Baughman, and A. Lee. Deep Learning Is Effective for Classifying Normal versus Age-Related Macular Degeneration Optical Coherence Tomography Images. *Ophthalmology Retina*, 1:322 – 327, 2017.

[30] H. Liang, M. G. Cid, R. G. Cucu, G. M. Dobre, A. G. Podoleanu, J. Pedro, and D. Saunders. En-face Optical Coherence Tomography - a Novel Application of Non-invasive Imaging to Art Conservation. *Opt. Express*, 13(16):6133–6144, 2005.

[31] G. Louppe. *Understanding Random Forests: From Theory to Practice*. PhD thesis, University of Liege, 2014.

[32] D. Markl, G. Hannesschläger, S. Sacher, J. G. Khinast, and M. Leitner. Optical Coherence Tomography for Non-destructive Analysis of Coatings in Pharmaceutical Tablets. *Proceedings of SPIE - The International Society for Optical Engineering*, 8792, 2013.

[33] F. A. Medeiros, A. A. Jammal, and A. C. Thompson. From Machine to Machine: An OCT-trained Deep Learning Algorithm for Objective Quantification of Glaucomatous Damage in Fundus Photographs. *ArXiv*, abs/1810.10343, 2018.

[34] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, USA, 1997.

[35] A. Nemeth, G. Hannesschläger, E. Leiss-Holzinger, K. Wiesauer, and M. Leitner. Optical Coherence Tomography – Applications in Non- Destructive Testing and Evaluation. In *Optical Coherence Tomography*, chapter 9. IntechOpen, Rijeka, 2013.

[36] K. O'Shea and R. Nash. An Introduction to Convolutional Neural Networks. *ArXiv*, abs/1511.08458, 2015.

[37] L. Perez and J. Wang. The Effectiveness of Data Augmentation in Image Classification Using Deep Learning. *ArXiv*, abs/1712.04621, 2017.

[38] J. B. Ramsey, H. J. Newton, and J. L. Harvill. *The Elements of Statistics: With Applications to Economics and the Social Sciences*. Duxbury Press, North Scituate, MA, 2002.

[39] P. Rasti, A. Ahmad, S. Samiei, E. Belin, and D. Rousseau. Supervised Image Classification by Scattering Transform with Application to Weed Detection in Culture Crops of High Density. *Remote Sensing*, 11(3), 2019.

[40] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[41] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry. How Does Batch Normalization Help Optimization? *arXiv preprint arXiv:1805.11604*, 2018.

[42] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning. From Theory to Algorithms.* Cambridge University Press, Cambridge, 2014.

[43] C. E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[44] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *ArXiv*, abs/1409.1556, 2015.

[45] M. Sokolova and G. Lapalme. A Systematic Analysis of Performance Measures for Classification Tasks. *Information Processing and Management*, 45(4):427–437, 2009.

[46] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[47] D. Stifter, P. Burgholzer, O. Höglinger, E. Götzinger, and C. Hitzenberger. Polarisation-sensitive Optical Coherence Tomography for Material Characterisation and Strain-field Mapping. *Applied Physics*, 76:947–951, 2003.

[48] E. A. Swanson, J. A. Izatt, M. R. Hee, D. Huang, C. P. Lin, J. S. Schuman, C. A. Puliafito, and J. G. Fujimoto. In Vivo Retinal Imaging by Optical Coherence Tomography. *Opt. Lett.*, 18(21):1864–1866, 1993.

[49] K. Wu, C. Garnier, J.-L. Coatrieux, and H. Shu. A Preliminary Study of Moment-based Texture Analysis for Medical Images. *Conference proceedings : Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, 2010:5581–4, 2010.

[50] M. D. Zeiler. Adadelta: An Adaptive Learning Rate Method. *ArXiv*, abs/1212.5701, 2012.

[51] Mnist Example Code. `https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py`, (last accessed July 09, 2019).

[52] One Simple Trick to Train Keras Model Faster with Batch Normalization. `https://www.dlology.com/blog/one-simple-trick-to-train-keras-model-faster-with-batch-normalization`, (last accessed July 09, 2019).

[53] Retina Gallery. `http://retinagallery.com/displayimage.php?album=1205&pid=12160#top_display_media`, (last accessed July 09, 2019).

[54] The Keras Library. `https://keras.io/`, (last accessed July 09, 2019).

[55] The Mnist Database. `http://yann.lecun.com/exdb/mnist/`, (last accessed July 09, 2019).

[56] In Depth Parameter Tuning for Random Forests. `https://medium.com/all-things-ai/in-depth-parameter-tuning-for-random-forest-d67bb7e920d`, (last accessed July 11, 2019).

[57] In Depth Parameter Tuning for SVC. `https://medium.com/all-things-ai/in-depth-parameter-tuning-for-svc-758215394769`, (last accessed July 11, 2019).

[58] Scikit-learn: Linear Support Vector Classifier. `https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html`, (last accessed July 11, 2019).

[59] Scikit-learn: Random Forest Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html/, (last accessed July 11, 2019).

[60] Scikit-learn: Support Vector Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC, (last accessed July 11, 2019).

[61] Wikipedia: Convolutional Neural Network. https://en.wikipedia.org/wiki/Convolutional_neural_network, (last accessed July 18, 2019).

[62] Google's MorphNet. https://github.com/google-research/morph-net, (last accessed July 25, 2019).

[63] Homepage of Recendt. https://www.recendt.at/en/OCT.html, (last accessed June 27, 2019).