Birgit Pfarrkirchner, BSc

Automatic Lower Jawbone Segmentation

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieurin

Master's degree programme: Biomedical Engineering

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg

Institute of Computer Graphics and Vision

Advisor: Dr. Dr. Jan Egger

Graz, October 2017

Abstract

The aim of this master thesis is to segment automatically the lower jawbone in humans' computed tomography (CT) images with the support of trained deep learning networks. To accomplish training, ten CT datasets of the head-neck region were provided by the Department of Oral and Maxillofacial Surgery of the Medical University of Graz. Moreover, physicians segmented manually the mandible in order to generate the ground truths for supervised learning. Ten CT datasets is an absolutely low amount to train neural networks efficiently. Consequently, a module network and a macro module were implemented with the MeVisLab software in order to enable a processing and a synthetic enlargement of the datasets. The data augmentation was realised with the application of affine transformations and noise addition on the original images. Beyond that, the classification and segmentation networks were implemented with Python and the deep learning toolkit TensorFlow, whereby training of the networks was conducted with the processed CT slices. The classification networks ought to decide whether an image shows parts of the lower jawbone or not. Hence, the slices displaying the mandible are delivered to fully convolutional networks, which predict the lower jawbone with the support of an upsampling approach. The results show that the networks perform better if more images are utilised for training. However, the achieved metrics and the visual predictions are quite satisfactorily in fact of the low amount of training data. Nonetheless, a further training and testing with new, unseen CT images is necessarily advised for future investigations.

Keywords: *Deep Learning, Segmentation, Lower Jawbone, MeVisLab, TensorFlow*

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

18.10.2017
Date

Birgit Pfarrkirchner
Signature

Acknowledgments

I would like to thank everyone, who supported me with the establishment of my master thesis and who encouraged me during my years of study.

In particular, I thank my advisor Dr. Dr. Jan Egger, who gave beneficial hints for the progress of my work and he also enabled to acquire an insight into daily scientific work. Additionally, I would like to show gratitude to Univ.-Prof. Dipl.-Ing. Dr.techn. Dieter Schmalstieg, who permitted the accomplishment of a master thesis in the interesting field of medical image processing and deep learning at the Institute of Computer Graphics and Vision.

On top of that, I would like to thank Dr. Dr. Dr. Jürgen Wallner, who provided the medical image data and the ground truth segmentations for the conduction of this thesis and he also assisted me with his sound medical knowledge.

Apart from that, I gratefully thank all my friends and my family, especially my parents and my boyfriend, who supported me during my years in Graz.

Contents

List of Figures

List of Tables

Abbreviations

2D	Two-dimensional
3D	Three-dimensional
CNN	Convolutional Neural Network
CT	Computed Tomography
API	Application Programming Interface
HU	Hounsfield Units
MDL	MeVisLab Definition Language
MRI	Magnetic Resonance Imaging
US	Ultrasound
GPU	Graphics Processing Unit
ReLU	Rectified Linear Unit
CNTK	Computational Network Toolkit
FCN	Fully Convolutional Network
IDE	Integrated Development Environment
CSO	Contour Segmentation Objects
DICOM	Digital Imaging and Communications in Medicine
NRRD	Nearly Raw Raster Data
CSV	Comma-Separated-Values
No.	Number
conv.	convolutional
max-pool.	max-pooling
acc.	accuracy
valid.	validation

min.	minimum
max.	maximum
std. dev.	standard deviation
i. a.	inter alia
e. g.	exempli gratia
μ	Attenuation value of an arbitrary material
μ_{H_2O}	Attenuation value of water
DSC	Dice Coefficient
IoU	Intersection over Union
HD	Hausdorff distance
TP	True Positives
FP	False Positives
FN	False Negatives
w_i	Weight of a neural network
W	Weight matrix
w_0	Bias of a neuron
v_i	Input of a neural network
$f()$	Activation function
y	Output of a neural network
net	Weighted input before activation
η	Learning rate
∇E	Gradient of the loss function
t_{nk}	Label for a distinct entry n and class k
y_{nk}	Predicted output for a distinct entry n and class k
z_k	Output of a neuron before activation for a class k

1 Introduction

Deep learning with neural networks is an increasingly important topic for many research and economic purposes. To emphasize this remarkable gain, Figure 1 illustrates Google's search requests on *"Deep Learning"* during the last five years.

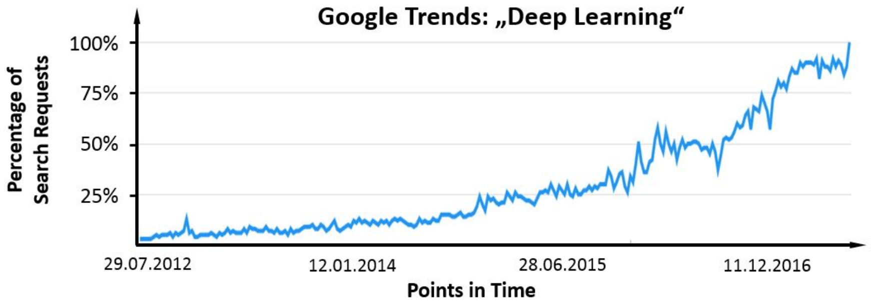


Figure 1: Increasing number of Google's search requests on *"Deep Learning"*. The last five years (2012 – 2017) are the period under consideration, whilst the numbers of search requests are percentage values. Adopted from [25].

Software giants use deep learning networks for the development of their latest technological gadgets. Take, for example, Facebook's face detection, Apple's speech recognition Siri or Google Translate, which all comprise deep learning algorithms [31].

1.1 Motivation and Goal

The motivation of this master thesis is to utilise deep learning networks for medical image processing and analysis. In particular, the aim is to implement convolutional neural networks (CNNs) as well as to train and test them with computed tomography (CT) images in order to enable an automatic segmentation of the lower jawbone.

It has to be noticed that there is a substantial difference between deep learning applications in medicine and general image processing tasks. Public accessible datasets of medical images are diminutive and rare despite to "ordinary" images. As a result, the question arises if deep networks work also satisfactorily with medical images [60].

To train CNNs during this work, ten three-dimensional (3D) CT datasets of the human head-neck region were provided by the Department of Oral and

Maxillofacial Surgery of the Medical University of Graz. Furthermore, two physicians segmented manually the lower jaw of present CT images to generate the ground truths, which were regarded as the correct segmentations. Ten CT datasets is an absolutely small amount to train a network efficiently and to attain a good generalisation ability.

Nevertheless, it is common to artificially enlarge a dataset size in the field of deep learning, which is also known as data augmentation. To increase the amount of training data during this thesis, the initial available images were noised and geometrically transformed. Therefore, a module network and a macro module were implemented within the image processing platform MeVisLab. Moreover, these MeVisLab implementations permit a storage of the CT slices and the ground truth masks in a format that can be processed with deep learning networks.

The deep learning implementations of this work comprise classification as well as segmentation networks. The idea is to mark out with a trained classification net the images, which show parts of the lower jawbone, and to provide those slices to the segmentation networks. The reason for this two-step implementation is that many CT slices occur, which don't display the anatomical region of interest. Hence, various classification and segmentation networks were implemented as well as trained and tested with the deep learning framework TensorFlow and its higher level application programming interfaces (API). The results show that the automatic segmentation of the mandible works adequately for the available CT datasets.

1.2 Thesis Structure

To stay on top of things, the thesis is divided into various chapters. Firstly, an overview of the lower jawbone's anatomical and physiological characteristics is provided in the *Medical Background* chapter. Next, technical aspects, which are fundamental for preceding thesis, are outlined in the *Technical Background* section. Therein, the CT modality, the image processing toolkit MeVisLab, general segmentation and deep learning aspects as well as the framework TensorFlow are introduced. Additionally, the *Related Work* chapter presents publications, that had an impact on the implementations of this thesis. The *Methods* section introduces the available image datasets as well as the MeVisLab implementations are discussed. Moreover, the classification and the segmentation tasks are explained in detail. The *Results* chapter annotates the achieved outcomes and finally, the *Discussion and Future Outlook* section gives the reader a resume and offers advices for further investigations.

2 Medical Background

The following subchapters outline medical fundamentals of the lower jawbone's anatomy and physiology. All instructions mentioned in the two subscript sections rely on the publications of Fanghaenel et al. [18] and Schuenke et al. [53].

2.1 Anatomy of the Mandibula

The lower jawbone's medical term is Mandibula, which corresponds to the Latin expression "mandere" and implies chewing. It is part of the Cranium, which is divided into the Viscerocranium and the Neurocranium. The brain, the hearing organ and further brain tissues are located within the Neurocranium. Despite that, the Viscerocranium forms the basic frame of the human face and it holds the visual organ, the olfactory organ and the teeth. The lower jawbone is also a part of the facial bones.

A human skull is composed of 22 bones, in total. The biggest ones can be seen in Figure 2. The bones indicated by the yellow shaded numbers belong to the Neurocranium, whilst the green shaded ones are part of the Viscerocranium. 21 cranial bones are fixed together with sutures, which are visible in Figure 2 as jagged lines in the surroundings of the Neurocranium. The lower jaw, however, is the only bone that is attached with a joint to the remaining skull by the Articulatio temporomandibularis. Parts of the auditory canal are immediately located behind the socket of this joint. Thus, fractures of the Mandibula may lead to injuries of the auditory canal.

On top of that, Figure 3 illustrates anatomical structures of the mandible, which consists of three main sections. The lower region is termed as Corpus mandibulae (I) and the left and right side of this structure as Angulus mandibulae (II). The superior regions are known as Ramus mandibulae (III). The Corpus mandibulae and the two Anguli mandibulae have a shape like a parable. The Maxilla (No. 6 in Figure 2) forms also a parable, but its arc is greater. Hence, the teeth of the upper jaw protrude lateral beyond the teeth of the lower jaw.

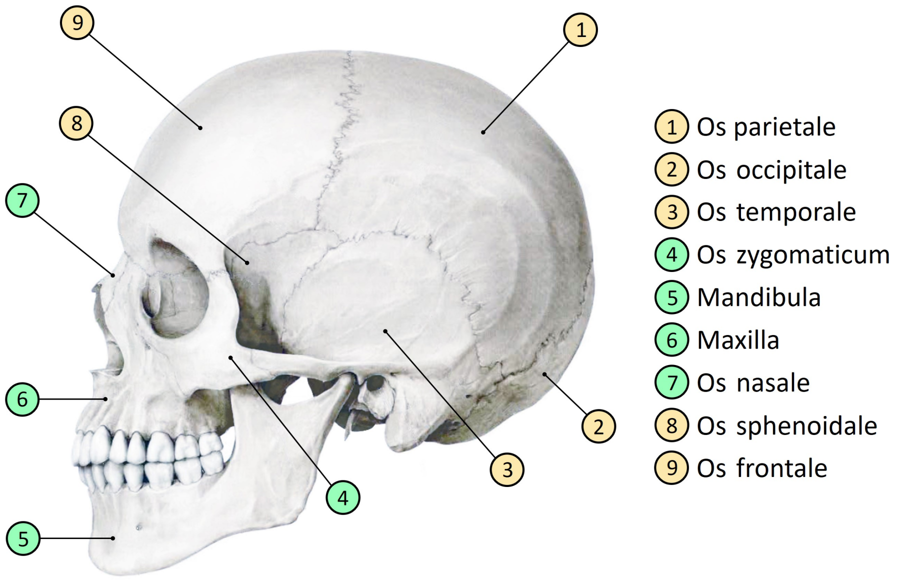


Figure 2: Anatomy of the Cranium. The yellow shaded numbers label structures of the Neurocranium, while the green shaded ones indicate bones of the Viscerocranium. Adopted from [53].

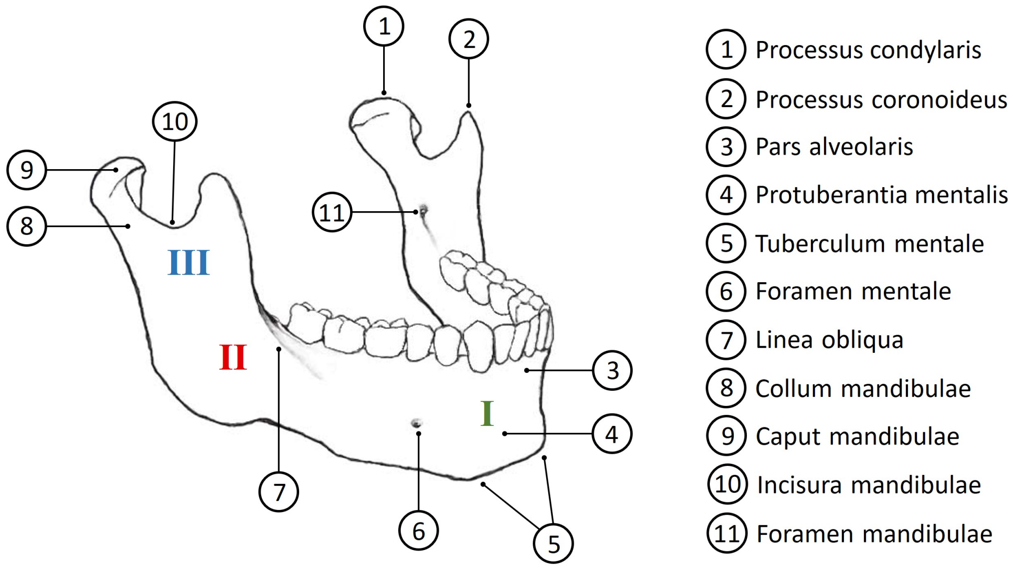


Figure 3: Anatomy of the mandible. The arabic numerals label important structures. Besides that, the Corpus mandibulae (I), the Angulus mandibulae (II) and the Ramus mandibulae (III) are highlighted. Adopted from [18].

At the bottom of the Corpus mandibulae, the Tuberculum mentale (No. 5 in Figure 3) is located, whilst the surface in cranial direction is called Protuberantia mentalis (No. 4). These structures compose the protuberance of a chin. Additionally, the region at the top edge of the lower jaw is labelled as Pars alveolaris (No. 3), which contains recesses (Dental alveoli) for positioning the teeth. A subsequent crista is sited at the dorsal ends of the Pars alveolaris, which is termed as Linea obliqua (No. 7). Furthermore, No. 6 in Figure 3 shows the Foramen mentale and No. 11 the Foramen mandibulae. Both bone cavities are connected by the Canalis mandibulae. This canal encloses vessels and nerves that supply the structures in the surroundings of the mandible. Besides that, the two Anguli mandibulae build a transition from the Corpus mandibulae to the Rami mandibulae. At the top of the Rami mandibulae are the Processus condylaris (No. 1) and the Processus coronoideus (No. 2) located. The Processus condylaris is part of the mandibular joint and consists of the Caput mandibulae (No. 9) and the Collum mandibulae (No. 8). The Caput mandibulae forms the joint head of the Articulatio temporomandibularis. Opposed to this, the Processus coronoideus serves as an attachment of muscles. Furthermore, the edge between the Processus condylaris and the Processus coronoideus is termed as Incisura mandibulae (No. 10).

2.2 Physiology of the Mandibula

The mandible as well as the adjacent muscles and joints are involved in speech synthesis. Furthermore, the teeth, which are held by the lower jaw, are a part of the gastrointestinal tract, as they support the fragmentation of nourishment.

The Mandibula's bone tissue isn't fully evolved from the beginning of life. The jawbone of new-borns consists of two separate fragments that are connected by conjunctive tissue. However, the bone fragments grow together during the first years of life. This development of bone is called ossification. We distinguish between two methods of ossification. The first one is about bone development from mesenchymal conjunctive tissue, which is also known as desmal ossification. The second method involves a conversion from cartilage and is termed as chondral ossification. However, the Mandibula is deployed through the desmal ossification.

Additionally, the appearance of the Pars alveolaris varies with aging. Figure 4 shows the evolving steps of the bone. New-borns don't have teeth, thus the Pars alveolaris doesn't exist. In the course of primary teeth formation, the Pars alveolaris arises. The Pars alveolaris of an adult is fully established. Elderly people often lose their teeth which results in a degradation of the bone. Consequently, the mouth and the face seem to sink if artificial teeth will not be inserted.

Moreover, not only the Pars alveolaris, but also the Anguli mandibulae evolve with aging. In the first years of life those bones show an angle of about 150°. During adult's age, this angle reduces itself to values from about 120° to 130°. With progressing age, the angle finally increases again.

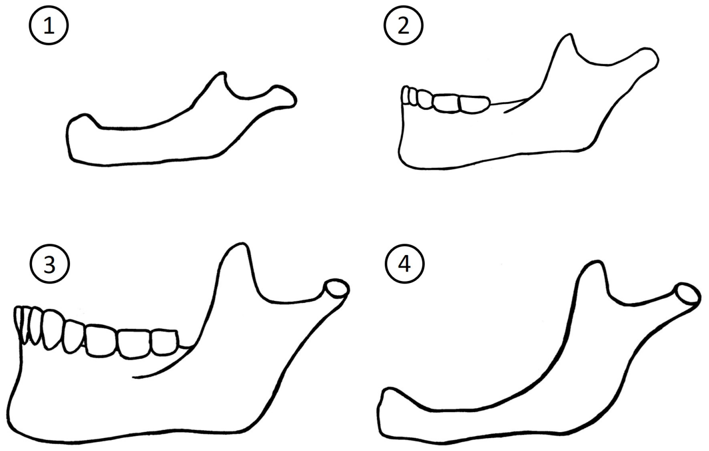


Figure 4: Evolution of the mandible during a human's life. No. 1 shows the lower jaw of a new-born, No. 2 indicates a young child's mandible, No. 3 displays an adult's jaw and No. 4 illustrates the Mandibula of a geriatric human. Adopted from [53].

Apart from that, one distinguishes between three types of jaw movements. The first one is a rotation, which enables closing and opening of the mouth. The joint works here as a hinge joint. The second one is a translation movement that allows to slide the jaw forwards and backwards. The forward movement is termed as Protrusion and the backward movement is known as Retrusion. The third and last one is the possibility to swing the bone to the left or right in order to accomplish grinding. Mostly, a combination of these movement types is conducted.

3 Technical Background

The content of this chapter involves descriptions of technical principles that are used to accomplish the goals of this master thesis. Initially, fundamentals of the image acquisition technique CT are supplied, since the medical images, that are provided for this work, are acquired with this modality. Moreover, basic concepts of the medical image processing platform MeVisLab are outlined, as this framework supports the preparation of the available image datasets to facilitate training and testing of neural networks. Besides that, central principles of segmentation as well as neural networks are outlined and the deep learning toolkit TensorFlow is discussed in detail.

3.1 Computed Tomography

Computed tomography is a medical image acquisition technique that generates sectional images of the human body. A fundamental step of the CT development was made with the detection of X-Rays by W. C. Roentgen in 1895. The mathematical basics to reconstruct images were provided a few years later by J. Radon. Moreover, in 1969 G. Hounsfield developed the first CT scanner for research purposes. After that, the application of CT scanners started to spread in hospitals [9], [8].

Nowadays, the computed tomography is an essential device for diagnosis in clinical routine. A main advantage is its fast image acquisition, which allows a usage in the trauma room. Moreover, it provides images of high quality and supplies tissues' electron density values, which are required for planning radiation therapies. Besides that, current developments enable a gradual reduction of the harmful radiation exposure [8], [24].

3.1.1 General Principle

The image acquisition is based on X-Ray tubes that are used with the classical roentgen, too. For that, the X-Ray tube is fixed at one side of the human body and the detector is placed at the opposite side. This layout can be seen in Figure 5. The collimator forms the shape of the photon beam, whilst the grid in front of the detector allows a reduction of scattered X-Rays, as it absorbs these quanta. All structures that lie in the X-Ray beam path (blue lines in Figure 5) are displayed on top of each other. Thus, there is a superposition of different tissues in the created projection images [9], [1].

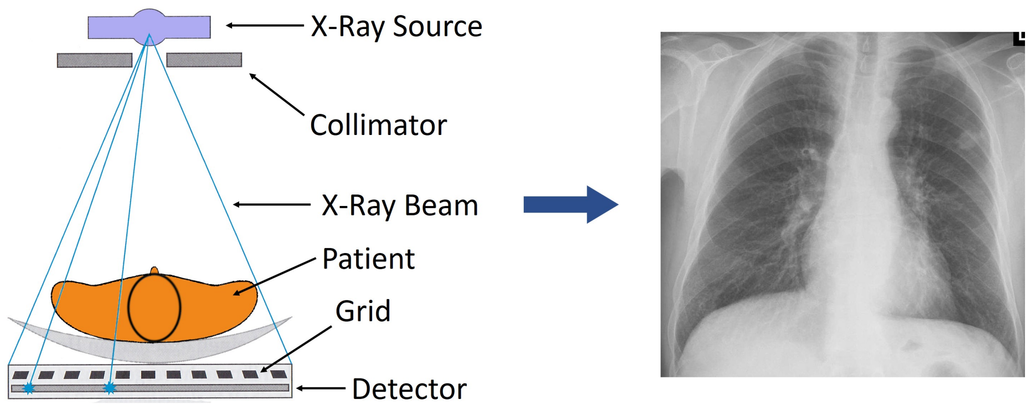


Figure 5: Principle of X-Ray projection and example of a projection image of the thorax. The X-Ray source sends out a wide fan beam (blue lines) and the attenuated intensity is measured by the detector. The detector and the source are fixed. Adopted from [1], [29].

Opposed to this, a CT scanner produces transversal slices of the human body. These transversal cuts don't suffer from a superposition of tissues, which results in a better image quality. Therefore, the X-Ray tube and the opposing detector are similar arranged as it is depicted in Figure 5. However, they move around the patient. There are several methods to transfer the source and the detector. For example, there are hardware realisations with a combined translation and rotation of the X-Ray source and detector, but there are also systems that just conduct a rotation. Furthermore, different forms of the photon beams are possible. The most common one is a fan beam that covers the whole patient [9].

The different hardware approaches are termed as "generations of scanners", whereas the most frequently used type in clinical routine is the CT scanner of the third generation (see Figure 6). The hardware equipment of the third generation conducts a rotation around the patient as the beam is wide enough to cover all body structures. In general, the patient is lying on a table in the centre of the gantry, which is a circular construction in which the devices rotate around the patient. Various image projections are acquired after every small rotation angle of the hardware devices. At the end, the tomographic image can be calculated from these projection images with the filtered backprojection. A typical CT scanner manufactured by Siemens can be seen in Figure 7 [9].

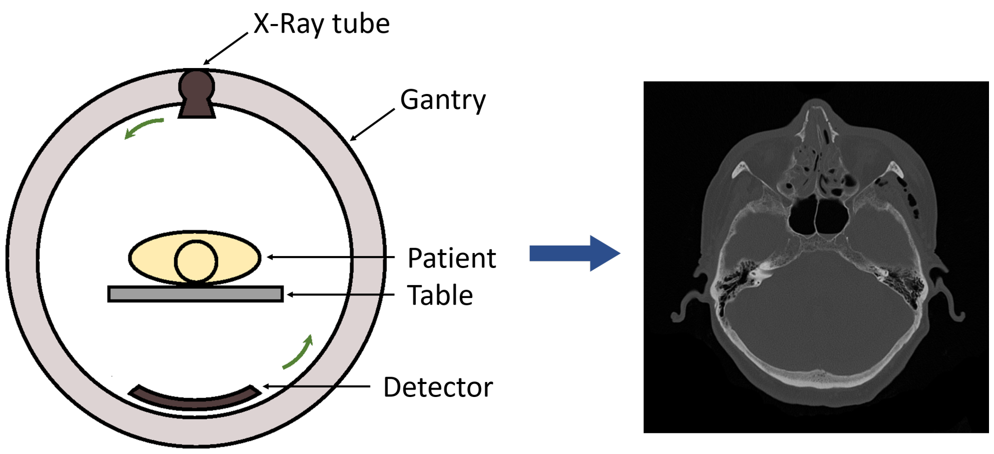


Figure 6: CT principle and example of a CT image of the skull. The X-Ray source and the detector rotate around the patient (left). The calculated image is a transversal cut of the human body (right).

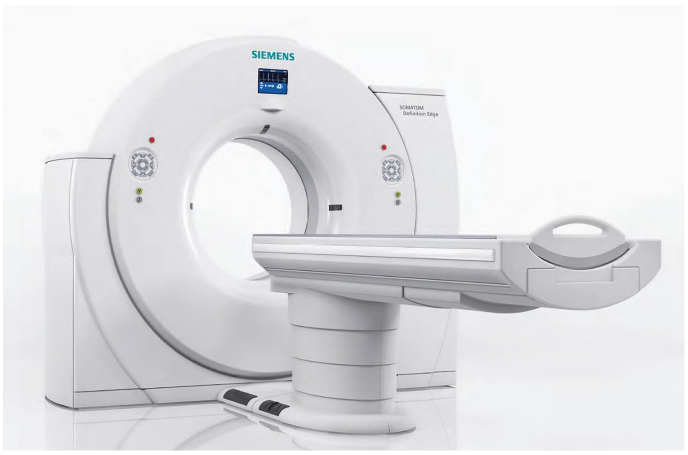


Figure 7: CT scanner SIEMENS Somatom Definition Edge. The circular gantry contains the X-Ray tube, the detector and further electronic devices. The patient is positioned on the table [19].

If the patient table is evenly moved during image acquisition, it is possible to cover a greater volume of the body. However, the generated images do not lie in the axial plane because of the motion of the table. Thus, the image acquisition is conducted in a helical way, which can be imagined with the illustrations of Figure 8. The correct transversal planes are calculated with interpolation [15].

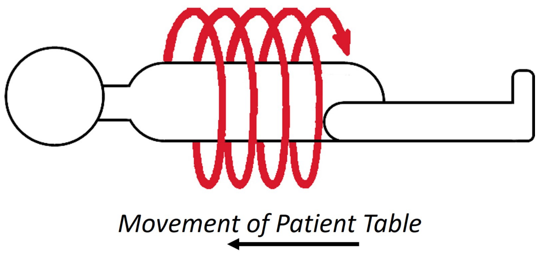


Figure 8: Schematic spiral CT. The table with the patient is moving while the image acquisition is conducted continuously.

Apart from the spiral CT, an acceleration of the image acquisition is also accomplished with the help of multi slice scanners. The detectors of these devices measure simultaneously neighbouring slices. Nowadays, some scanners are able to acquire up to 256 slices coincidentally. Consequently, the X-Ray beam must be widened in longitudinal direction of the patient. The schematic principle can be seen in Figure 9 [9].

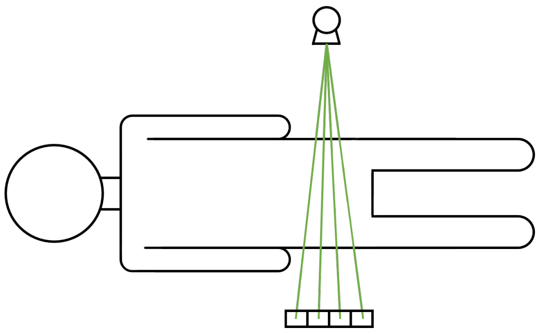


Figure 9: Schematic multi slice CT. The X-Ray tube extracts a beam that is widened in longitudinal direction. The measurement is accomplished with a 4-slice CT detector. Adopted from [9].

The CT image contrast depends on different attenuation properties of the tissues. In general, there is a higher attenuation of the beam's intensity if the X-Ray beam's frequency decreases as well as the atomic number, the thickness or the density of the tissue increase. This is the reason why bones (high atomic number) are high in contrast compared to soft tissues [15].

3.1.2 Hounsfield Units

In the field of computed tomography the Hounsfield units (HU) are introduced, which enable a comparison of grey values between images that are recorded with different scanners but with the same tube voltage. Therefore, the attenuation value of a tissue type is related to water. As a result, the Hounsfield unit (1) can be calculated with

$$HU = \frac{\mu - \mu_{H_2O}}{\mu_{H_2O}} \cdot 1000 \tag{1}$$

where μ indicates the attenuation value of an arbitrary material, μ_{H_2O} is the reference value. According to this definition, water has a HU of zero, air has a value of -1000 and bone shows a value of about 3000. The calculated Hounsfield units are stored as the grey values of an image. Figure 10 displays a graphical interpretation of the Hounsfield units. Soft tissues have a rather small range of HU values, whereas the values of bone protrude clearly. Tissues that contain fat or air exhibit negative values [9].

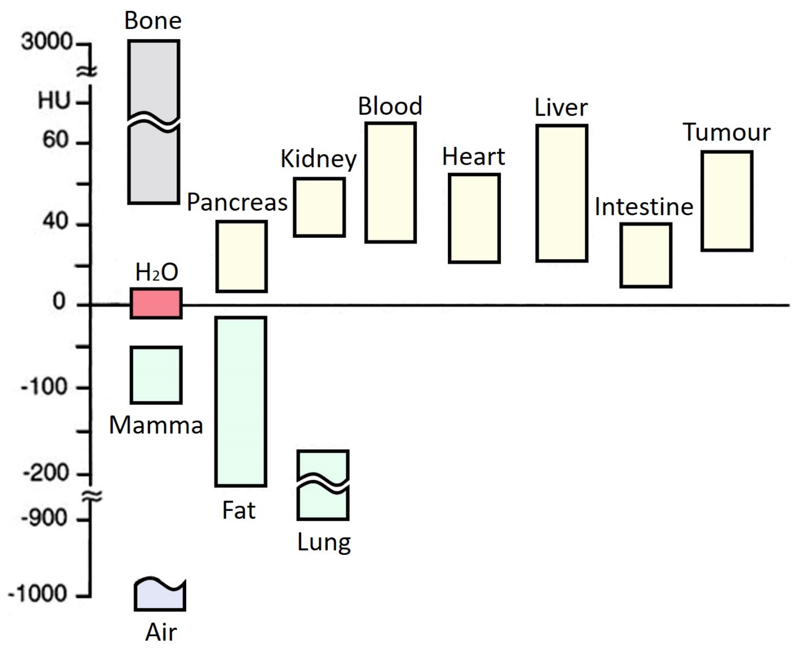


Figure 10: Hounsfield units of different tissue types. Adopted from [15].

3.1.3 Artefacts

Artefacts, which lead to images that do not coincide with real circumstances, are a common issue in CT. Mainly occurring types according to [9] and [15] are listed below.

- *Partial Volume Artefact*: Two different tissues with dissimilar attenuation values, that are placed within the same voxel, lead to a resulting grey value that neither corresponds to the first tissue nor to the second one. As a result, the image seems to be blurred. A possibility to reduce this artefact is to decrease the slice thickness.
- *Metal Artefact*: Metals, such as implants or screws, lead to dark stripes in the image, because of the upcoming beam hardening effect. In general, a photon beam consists of a continuous energy range. Photons with lower energy are absorbed easier. Thus, with an ongoing traversing of the beam, the photons with high energy remain.
- *Motion Artefact*: Patient movements lead to blurred images. A faster acquisition of the slices reduces this artefact.

3.2 MeVisLab

The MeVisLab software tool is used to process image data for training and testing deep learning networks. The instructions of this chapter can also be found in my Master Project [48], although modifications are conducted. All outlined statements depend on the Getting started tutorial [3] and on the MeVisLab Reference Manual [5]. Both references are provided by the MeVis Medical Solutions AG.

MeVisLab [16], [17] is a semi-open source software tool for image processing and visualisation with special focus on medical images. A clear benefit of this software tool is that it enables a graphical user interaction. There are various model components available, which are termed as modules, that can be intuitively connected in order to form a MeVisLab network. These MeVisLab networks are stored as mlab-files. The internal modules already implement basic image processing operations, such as low-pass filtering or segmentation algorithms like thresholding. Thus, the user doesn't have to bother about implementing fundamental image processing steps, but can rather focus on the deployment of new algorithms.

The development of an own MeVisLab network means to select the built-in modules, that should be successively carried out, and to connect them via data connections for passing on the processed information to achieve a desired functionality. MeVisLab provides three different module types that can be joined:

- *ML Modules:* The blue coloured boxes enable a processing of the voxels of an image;
- *Open Inventor Modules:* The green coloured boxes allow a processing of three-dimensional scenes;
- *Macro Modules:* The brown coloured boxes are a combination of fundamental built-in modules.

Specific parameters of the modules, for instance, thresholds, can be adjusted with the modules' graphical panels.

Furthermore, there is a distinction between three types of data connectors, which allow a transfer of information from one module to the next one. The triangles indicate that ML images are passed. Half-circles are available to transfer inventor scenes and data structures are processed by the square connectors. The three module types and the different data connectors are visible in Figure 11.

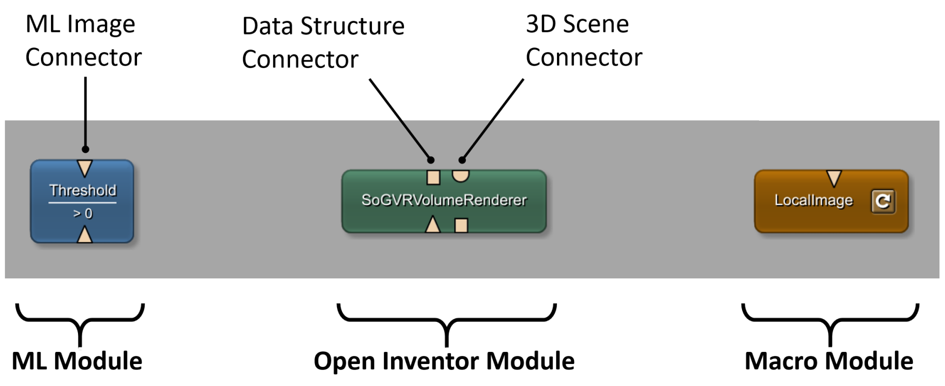


Figure 11: Illustration of the three module types and the data connectors.

The MeVisLab internal implementation defines which connectors are delivered for a distinct module. Data connections are only allowed between connectors of the same type. Typically, input connectors are located at the bottom of a module and output connectors are placed at the top. Hence, processing data in a network starts at the bottom and goes upwards through the linked modules. Additionally, MeVisLab provides a parameter connection, which enables a conjunction of fields between modules. This is a beneficial tool to synchronise values between several modules. In order to get an overview across all modules, they can be grouped. In Figure 12 is an example network with a module group displayed.

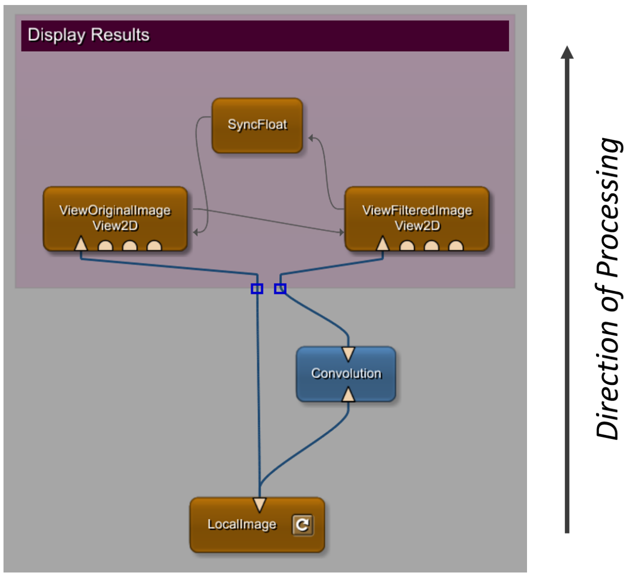


Figure 12: Example of a MeVisLab network. There are several macro modules (brown) and a ML module (blue) visible. The "Display Results" group holds three macro modules. The parameter connections are depicted as grey lines, whereas the data connections are plotted as thick blue lines.

To conclude, MeVisLab provides a lot of internal modules that are used for fundamental image processing. However, it is frequently occurring that more complex problems appear. As a result, it is necessary to generate new user specific functionalities. Two approaches are possible in order to address this issue:

1. Implementation of C++ modules: New ML and Open Inventor modules can be implemented in C++ using Microsoft Visual Studio;

2. Implementation of macro modules: New macro modules based on existing MeVisLab modules are developed with the support of Python or JavaScript.

Hence, if there are completely new image processing algorithms developed, it is obligatory to use C++. If the problem is solvable with already existing MeVisLab modules and further added scripted functionalities, the implementation of a new global macro module is indicated. The desired tasks of this master thesis were feasible with the implementation of a new macro module. As a result, there is an outline of the generation of macro modules.

<u>Macro Modules:</u>

A specific functionality can be attained with a self-implemented global macro module that is built with the MeVisLab Project Wizard. A macro module is a delimitation of processing steps that allows a compact representation in a MeVisLab network. There is a distinction between local and global macros. Local macros can be used in one certain network, but they are inaccessible from others. Global macros are integrated into the MeVisLab module database, which means that they can be called up from every network.

The Project Wizard automatically creates the Script- and the Definition-file. These two files are obligatory in order to build a new global macro. Moreover, a macro module may consist of a Python-file as well as a network of basic MeVisLab modules, termed as the macro network.

The Definition-file is mandatory to integrate the macro module into the MeVisLab module database. The extended functionality is implemented with the Python script. It is also possible to use JavaScript, nevertheless it is recommended to use Python. The requested functions of this master thesis were achieved by Python scripting. Additionally, a graphical user panel is created with the MeVisLab Definition Language (MDL) in the Script-file. Therein is determined which fields are visible in the panel and how they are arranged. Moreover, the input, output and parameter fields of the macro module are declared.

3.3 Segmentation

The following subscripts introduce the principles of segmentation and the necessity of its application in the medical domain. Moreover, methods to validate segmentation algorithms are described.

3.3.1 General Principle

Segmentation is the process of dividing an image into various regions. All pixels that belong to the same object are assigned to one segmented region. Thus, segmentation is strictly speaking a method to extract related areas from an image, but it doesn't comprise a classification of these regions. However, it is not possible to rigorously separate these two steps in practice [30], [36].

Figure 13 displays a schematic representation of segmentation. The left image shows some coins that should be segmented. The right part illustrates these segmented areas, whereby the black colour indicates the background and the red one displays the foreground.

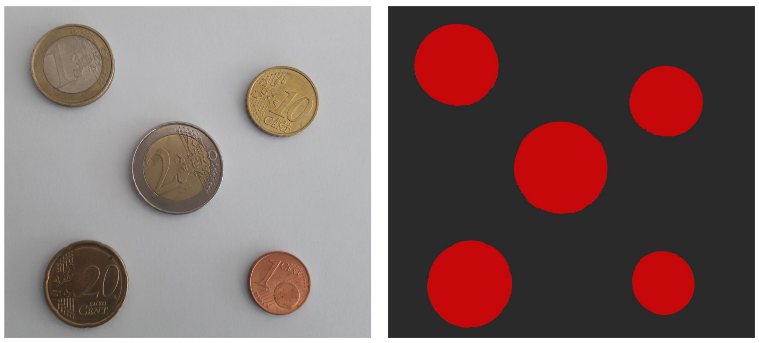


Figure 13: Segmentation principle. The left image demonstrates some coins placed on a surface, whereas the right one displays a corresponding segmentation. Red areas stand for the coins (foreground) and black areas indicate the background.

One possible approach to segment objects in an image is a manual segmentation. Hence, in the case of medical images a physician has to delineate anatomical structures by hand. However, this non-automatic method is tremendously time consuming and it is not objective, which means that two different doctors in general never segment exact same areas [11], [24].

Despite that, there are also automatic and semi-automatic procedures distinguished. Typical approaches are the threshold method, region-based and edge-based algorithms, the watershed transformation or active contours. Nevertheless, it is not possible to declare one segmentation algorithm as the "gold standard". It depends on the examined image types and on the definition of the task, which segmentation procedure is intended to address the problem. In this work, an automatic segmentation should be achieved with the application of neural networks [7].

According to [14], three difficulties, that might appear in medical segmentation tasks, have to be kept in mind:

1. Medical images are heterogeneous: Organs don't only appear equal across different patients and various views, but also across images of the same patient recorded at diverse timepoints.

2. Tissues don't have obvious boundaries in medical images. Consequently, it is difficult to detect certain objects.

3. Segmentation algorithms must be absolutely reliable, because decisions, which are made depending on segmentation results, affect humans' diagnosis and therapy.

3.3.2 Segmentation Applications in Medicine

Segmentation algorithms are applied on images that are acquired with different imaging modalities, like magnetic resonance imaging (MRI), computed tomography, X-Ray, Ultrasound (US) or nuclear medical techniques.

In general, segmentation of anatomical structures is important for diagnostics but also for planning therapies. With the help of segmentation algorithms, pathologic lesions can be separated from healthy structures. For instance, it is possible to delineate a tumour from its surrounding tissue. Therefore, the shape and the size of cancerous tissue can be examined with images, that are acquired at different timepoints, to monitor a treatment outcome. Furthermore, segmentation is necessary to visualise 3D models of certain organs to enable a superior imagination of diseases and injuries. The visualisation is a useful opportunity for physicians to plan a surgical procedure [22].

In addition to this, segmentation is important to plan radiotherapies. During a radiation therapy, electrons or photons are brought into the tumour region to destroy the cancerous tissue. However, the energy should not be applied on the surrounding organs to prevent long-term effects. Hence, the tumour as well as the adjacent regions have to be delineated. In case of the Mandibula, an unintended radiation exposure may lead to a destruction of the teeth and implants [24].

3.3.3 Validation of Segmentation Results

To compare the performance of segmentation approaches, the results of the algorithms must be assessed with standardised methods. Therefore, it is necessary to have a ground truth segmentation, which is equal to the correct segmentation. In this work, the correct segmentation is produced by physicians, who segmented manually CT images of the mandible. As already mentioned, different humans are not able to produce exact same ground truths. The deviation of segmentation results, that are achieved by different physicians, is termed as the inter-observer variability. Despite that, the variance of segmentation contours, that are generated by the same physician at distinct time points, is called as intra-observer variability [22].

To evaluate the segmentation results of this work, the mean intersection over union (IoU), the Dice coefficient (DSC) and the Hausdorff distance (HD) are calculated. Figure 14 displays a schematic interpretation of the accordance of an algorithmic segmentation and the ground truth. The true positives (TP) are the number of correctly segmented voxels by the segmentation algorithm. The false positives (FP) are the voxels, which are incorrectly assumed to be part of the segmentation object, whilst the false negatives (FN) are part of the segmentation object, but the algorithm didn't suppose them as such.

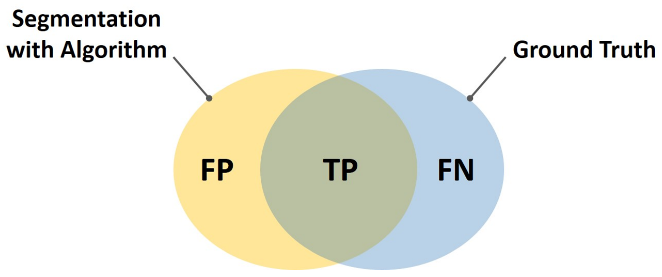


Figure 14: Graphical interpretation of a segmentation result. The yellow area indicates the false positives (FP), the green one the true positives (TP) and the blue one the false negatives (FN).

To determine the IoU (2), the fraction of

$$IoU = \frac{TP}{TP + FP + FN} \tag{2}$$

has to be calculated. The mean IoU is averaged over the different segmentation classes [21].

According to [24], the Dice coefficient (3) is computed with

$$DSC = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \tag{3}$$

where a result of one implies a perfect segmentation. In general, the higher the IoU and the DSC, the better are the segmentation predictions.

Apart from that, the Hausdorff distance (4) is calculated with

$$HD(A, B) = \max(h(A, B), h(B, A)) \tag{4}$$

and

$$h(A, B) = \max_{a \in A} \min_{b \in B} \|a - b\| \tag{5}$$

where A and B are the segmented objects of the ground truth and the automatic segmentation. The symbols a and b are individual voxels of the regions A and B. Hence, for each point of the ground truth is the minimal distance to a point of the algorithmic segmentation identified (and vice versa), whereby the largest appearing range is the Hausdorff distance. A small Hausdorff distance proves that the segmenation result is good [44].

3.4 Deep Learning

The automatic segmentation of the mandible from CT images should be obtained with deep learning networks.

Deep learning with artificial neural networks is a branch of computational intelligence. Neural networks can be imagined as a mathematical system of neurons, which are the basic elements of a network. The neurons are arranged in several layers, whereby the neurons of one layer are connected to those of adjacent layers via weights. However, a connection between the neurons of the same layer is not allowed. The values of these weights have to be learned. Neural networks support the processing of information: The input of the deep learning network is reprocessed by the neurons and their trained connection weights in an abstract manner [56], [57], [34], [35].

Two learning types are distinguished: *supervised* and *unsupervised* learning. Supervised learning means that for an input signal, the correct result of the output is known. Take, for instance, the case of object classification: If images display animals, which should be classified, there exists also a label with the proper class (like "cat", "dog", "mouse", ...) for every input image. Thus, the ideal solution of an input is already offered to learn a network. Despite that, for unsupervised learning, the correct solution of the classification task is not known. The image data of this work is segmented by medical doctors. Therefore, supervised learning is indicated, since the correct solution of the segmentation task is already present [37].

In addition to this, the term *"deep"* refers to the number of layers of a neural network. The more layers are present in a network, the deeper it is. Networks with only a few layers are shallow ones. However, according to the work of J. Schmidhuber [52], there is not a general agreement which layer number divides networks in deep or shallow ones. Nonetheless, it is stated that networks with more than ten layers refer to very deep learning. Deep neural networks were implemented after the spread of shallow ones. The reason is that such capable Graphics Processing Units (GPUs) to process information along the substantial number of layers didn't occur [57].

Furthermore, a lot of training data is necessary to train deep networks and achieve satisfactory results. Thus, these days standardised image data bases, like ImageNet [13] or kaggle [27], are introduced to offer a giant number of images, which promote training of networks [57].

Apart from that, information processing with neural networks offers wide application opportunities. It is possible to utilise deep learning nets with visual image tasks, like classification and segmentation of objects, but they are also relevant for speech processing and audio tasks. Additionally, neural networks are applicable for economic purposes, like forecasting stock prices [35], [32].

3.4.1 Fundamentals of Artificial Neural Networks

The instructions of this section rely on the textbooks of H. Handels [22] and H. Suk [57].

Neural networks in computer science are originally based on a biological analogue: the human brain. The brain consists of up to 100 billion of biological neurons, which are partly connected. Computer scientists tried to simulate these biological structures with mathematical neural networks, because they would like to achieve a similar performance as the brain. For instance, the human brain exhibits an efficient ability to process information. It processes information in a parallel manner and learns new connections between neurons. A biological neuron (see Figure 15) consists of many inputs, termed as the dendrites, as well as of one single output, the axon. The axon is connected to the dendrites of a succeeding neuron with a synaptic body. If enough input signals occur from the dendrites, the neuron emits a signal.

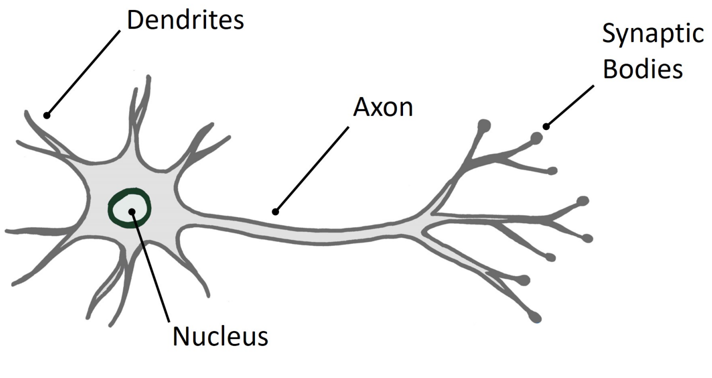


Figure 15: Structure of a biological neuron. The dendrites transfer the input signals to the neuron and the axon passes on the output signal.

The mathematical analogue of a neuron can be seen in Figure 16. The output (6) is calculated with

$$y = f(net) = f(\sum_{i=1}^{D} v_i \cdot w_i) = f(w^T \cdot v) \tag{6}$$

where $v = (v_1, v_2, \dots, v_D)$ are the inputs of the neuron, $w = (w_1, w_2, \dots, w_D)$ are their corresponding weights and $f()$ is the activation function. The values of the weights are determined during the training process. Furthermore, the activation function $f()$ has to be a nonlinear function.

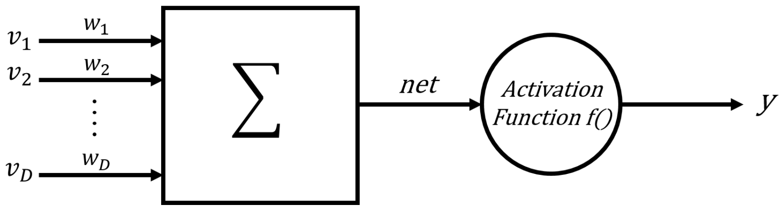


Figure 16: Mathematical analogue of a biological neuron. Adopted from [22], [43].

A neural network is composed of several basic neurons arranged in layers. The smallest network possesses an input and an output layer. It is also termed as a single layer model, as the input layer is not part of the layer number. Moreover, neurons may have a bias, which is not multiplied with the weights. Thus, the output (7) results with

$$y = f(net) = f(\sum_{i=1}^{D} v_i \cdot w_i + w_0) = f(w^T \cdot v + w_0) \tag{7}$$

where the bias is indicated by w_0.

If a network consists of more than one layer, it is termed as a multilayer network. The layers between the input and output layers are known as hidden ones. Moreover, if every neuron of a foregoing layer is connected to each neuron of the succeeding layer, the net is denoted as a fully connected network. Figure 17 shows the structure of a multilayer and fully connected network with three input neurons, three hidden neurons and a single output neuron.

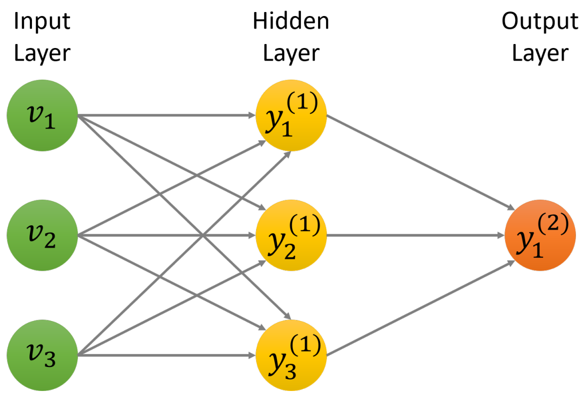


Figure 17: Structure of a multilayer network. There is an input layer with three neurons (green), a hidden layer with three neurons (yellow) and an output layer with one neuron (orange) displayed. The units of the adjoining layers are fully connected. Adopted from [12].

To calculate the output (8) of a two-layer network without a bias,

$$y_k = f^{(2)}(\sum_{j=1}^{M} W_{kj}^{(2)}\ f^{(1)}(\sum_{i=1}^{D} W_{ji}^{(1)} v_i)) \tag{8}$$

has to be computed. The outcome of the calculation of the first layer is the input of the subsequent layer. W_{kj} and W_{ji} are matrices, which comprise the weights between two layers. The first index corresponds to the following neuron and the second one to the previous neuron. As the input is brought in at one side of the network and the calculation is obtained in a stepwise forward way, this net is also labelled as a feed forward network.

The aim of training a net is to determine the values of the weights. Therefore, an input is provided to the network and the resulting solution is compared with the known, ideal solution. The goal is to achieve a difference, as small as possible, between the ideal and the actual output. Thus, an error function (also cost function) is stated and ought to be minimised. In practice, the weights are not calculated analytically. They are rather initialised with arbitrary values at the beginning of the training process and during training, their values change iteratively to minimise the cost function. The adaption of the parameters is conducted with the gradient descent method. Thereby, the gradient of the loss function is calculated for the trainable parameters. The update of this parameter set is accomplished in the way that the negative gradient is followed [12].

According to this, the mathematical formulation (9)

$$W^{t+1} = W^t - \eta \, \nabla E(W^t) \tag{9}$$

can be stated. W indicates trainable parameters, the index t stands for the iteration step and ∇E denotes the derivation of the cost function. Moreover, η signifies the learning rate. Figure 18 displays a loss function and the influence of the learning rate. The best parameter values are those, on which the loss function exhibits the minimum. If the learning rate is small, the steps towards the optimal parameter are also small. However, if the learning rate is large, it is possible that the optimal parameters are overleaped.

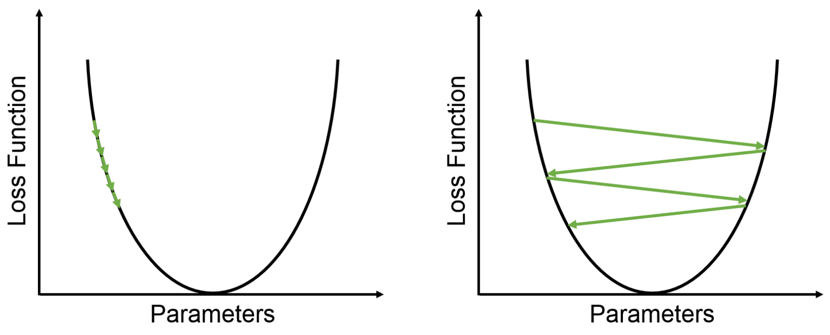


Figure 18: Comparison of the gradient descent method with different learning rates. A small learning rate leads to a slow progress of finding the best solution (left). However, a large learning rate may result in an overleap of the best parameter (right). Adopted from [12].

The iteration is accomplished until the stopping criterion is fulfilled: Either convergence is achieved or the maximum number of iterations is exhausted. The flowchart of Figure 19 visualises these training steps.

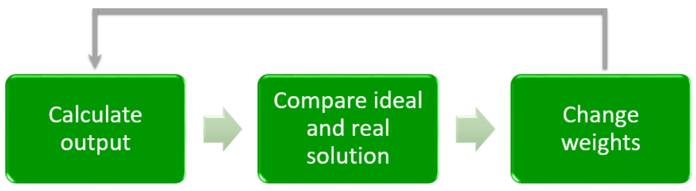


Figure 19: Flowchart of the training process. Firstly, the output is calculated according to a provided input. This computed outcome is compared to the ideal solution. Secondly, the weights of the network are adjusted. These steps are conducted until the stopping criterion is fulfilled.

In addition to this, there are diverse types of neural networks distinguished depending on the topology (arrangement) of the units. For visual tasks, it is common to use convolutional neural networks.

3.4.2 Convolutional Neural Networks

Despite to standard networks characterised in the previous chapter, convolutional neural networks receive 2D and 3D images as an input. As a result, the input data is not flattened into a vector, which leads to a preservation of the compositional information of an image. Two basic layer types are essential to build a CNN: the *convolutional* and the *pooling* layer. Moreover, the final layers of CNNs are frequently fully connected layers. Considering classification networks, each output unit represents one class and its probability of occurrence. The schematic structure of a typical convolutional network can be seen in Figure 20 [57].

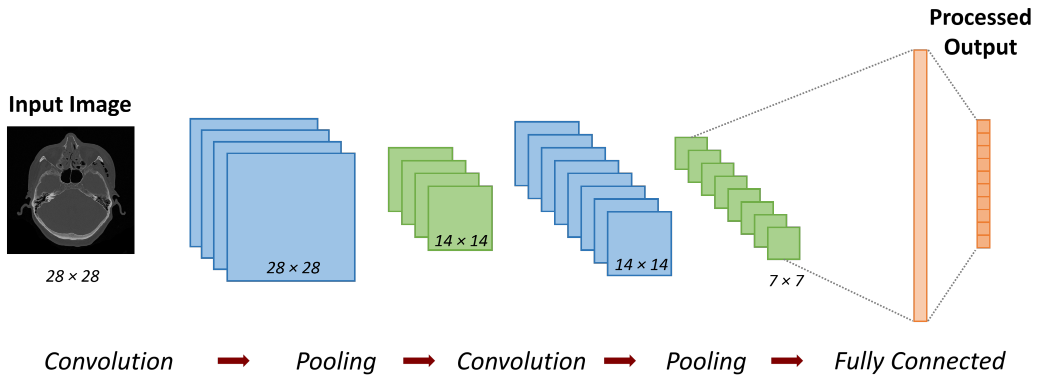


Figure 20: Structure of a convolutional neural network. The first convolution produces four feature maps (blue) with the same size as the image. Max-pooling is used to down-sample the matrix sizes (green). The second convolution is accomplished with eight filter kernels. Following the two convolution and max-pooling layers, a fully connected layer and a final output layer are added. Adopted from [57].

Figure 21 shows an example image, that is convolved with two filters, and afterwards, max-pooling is applied.

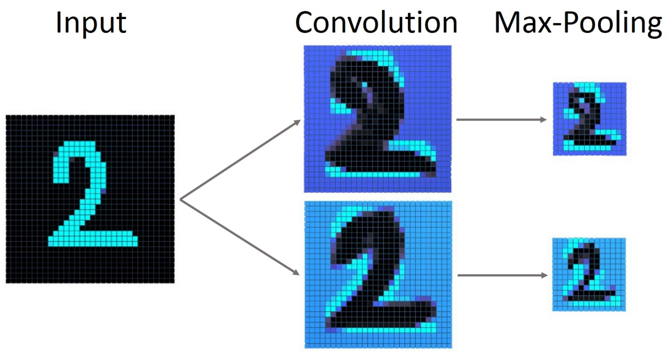


Figure 21: Example image applied to a convolutional and max-pooling layer. The input image is convolved with two different kernels. Adopted from [23].

Convolutional layer: The first layer of a CNN is a convolutional layer, which produces images with the same size as the input image (28×28). The aim of the convolution with different filter kernels is to extract distinct features, for instance, edges or shapes. Therefore, the values of the kernels are determined during the training process. In the example of Figure 20 are four kernels for the first convolution applied, which accordingly generate four feature maps. For the second convolutional layer are eight trainable kernels used [57], [56].

The mathematical principle of a convolution is illustrated in Figure 22. The input image has the same size as the filter kernel in this example. Nevertheless, it has to be noticed that the image size is usually larger than the size of the kernel. This circumstance leads to the effect that there have to be less parameters determined, because there is not an entire conjunction of neurons of adjacent layers (global connectivity). The smaller filter size engenders a local connectivity (see Figure 23). To calculate each matrix entry of the convolution, the kernel (blue dashed box) slides across the input image with a stated stride size. The stride size of this example is one, since the kernel slides in a step of one pixel. Hence, convolutional kernels lead to a weight sharing, because once the kernels are trained, they are just moved over the image and are applied at different positions, but with the same filter weights. Furthermore, zero padding is indicated to ensure a calculation of outer matrix entries [57], [56].

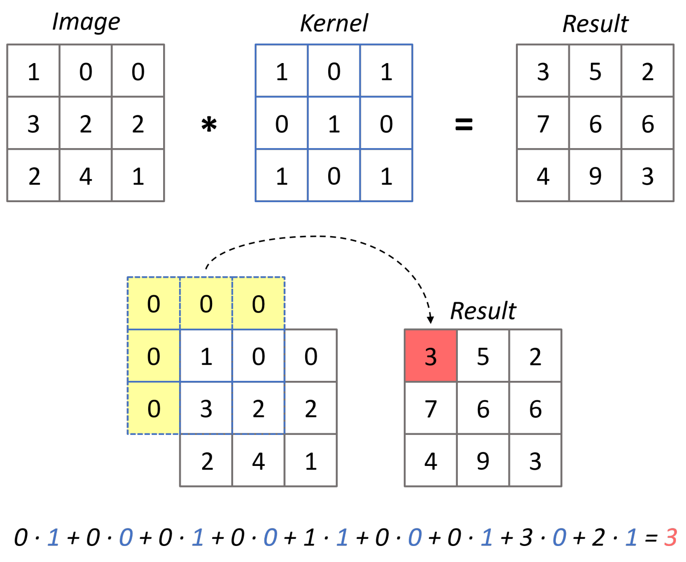


Figure 22: Convolution principle. The convolution of a 3×3 image with a 3×3 kernel is shown above, whilst one single convolution step to calculate the red highlighted value is exemplified below. The blue doted box illustrates the overlay of the filter on the image. At the surroundings of the image are zeros added (yellow) to ensure a calculation of the outer entries. The convolution can be imagined as a multiplication of each voxel value with its overlaid filter value and a summation of these products (equation at the bottom).

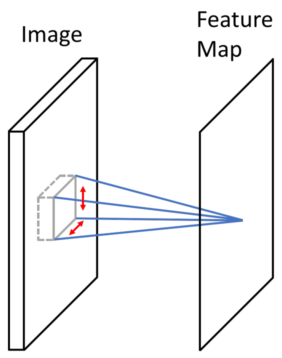


Figure 23: Local connectivity of a convolutional layer. The grey square indicates one extract of the image that is convolved with the filter kernel and slides across the entire image to calculate each element of the feature map. The blue lines imply the local connectivity: Only the elements of the grey area are used to calculate one entry of the feature map.

Pooling layer: A pooling layer usually follows a convolutional layer. Figure 24 displays the concept of a max-pooling layer. Therefore, a receptive field slides with a determined stride size across the matrix. The receptive field has a size of 2×2 and the stride size is scheduled as two in this example. This ensures that from each considered 2×2 area, the maximum value is selected. Consequently, the four matrix entries are replaced by their highest value. As a result, the pooling layer halves the side length of the feature maps. The main advantage of pooling is that less parameters have to be trained in a subsequent convolutional layer. In general, there are also other pooling methods used, for instance, average pooling. However, max-pooling is the most frequently used one [57], [56].

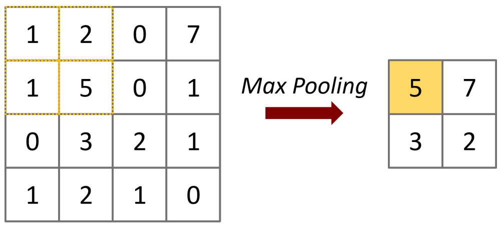


Figure 24: Max-pooling principle. The pooling layer down-samples the size of an image. In case of max-pooling, the highest value of a receptive field is extracted. The orange dotted box indicates the receptive field (2×2) for the computed, orange highlighted (5) result.

Fully connected layer: Finally, fully connected layers are added after the alternation of convolution and pooling operations. This layer type exhibits global connectivity since all units of the latter pooling layer are connected to all neurons of the following fully connected layer [56].

Activation function: Nonlinear activations are also used with CNNs. Thereby, the activation function is applied on the calculations of the convolutional layer. The mainly occurring type for CNNs is the Rectified Linear Unit (ReLU) function. To apply ReLU activation (10) on a value x,

$$f(x) = \max(0, x) \tag{10}$$

has to be calculated [56], [12].

Dropout: Training a network suffers frequently from overfitting, which means that the network produces remarkable results for the available datasets, but not for new test datasets. Dropout is a method to solve this poor generalisation ability. Therefore, some neurons are randomly switched off with a determined probability during each training step. As a result, the network is trained with fewer units. However, testing the network involves all neurons [56].

Deconvolution: On top of that, a segmentation network, as it is implemented during this work, requires upsampling layers, which resize down-sampled feature maps to greater matrix sizes. This procedure can be achieved with deconvolution, also known as transposed convolution. Figure 25 illustrates graphically the upsampling operation. At the left is the down-sampled feature map (2 × 2) shown, whilst at the right is the calculation of the upsampling simulated. For example, the green value in the down-sampled input is the weight that is multiplied with a bilinear filter. The weighted filter has a size of 4 × 4 (green dashed line). Moreover, the stride size, which equals the upsampling factor, indicates the steps for the next deconvolution computation. In this case, a stride size of 2 is defined. Thus, the bilinear filter, that is weighted with the initial blue value, is shifted two pixels downwards. The values of the different filters are added up at the positions where filters overlap (grey striped pixels) [45], [46].

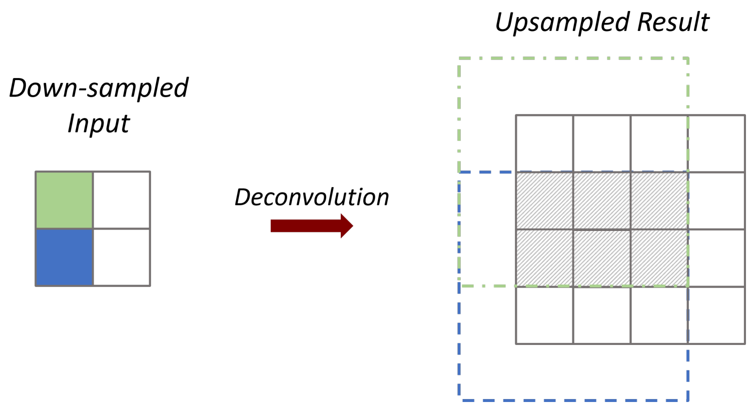


Figure 25: Schematic description of the upsampling operation. At the left is the initial input visible, whilst at the right ist the upsampled output (grey) and examples of two weighted filters (green and blue dashed lines) depicted. Adopted from [45], [46].

3.5 Deep Learning Toolkit: TensorFlow

Deep learning toolkits support the implementation, training and also testing of neural networks. The main advantage is that the software developer has not to code a network and its optimisation from scratch, because these toolkits already offer functions to define layers, train the weights and apply a trained network on new data. A wide variety of toolkits are supplied to accomplish deep learning tasks with different purposes.

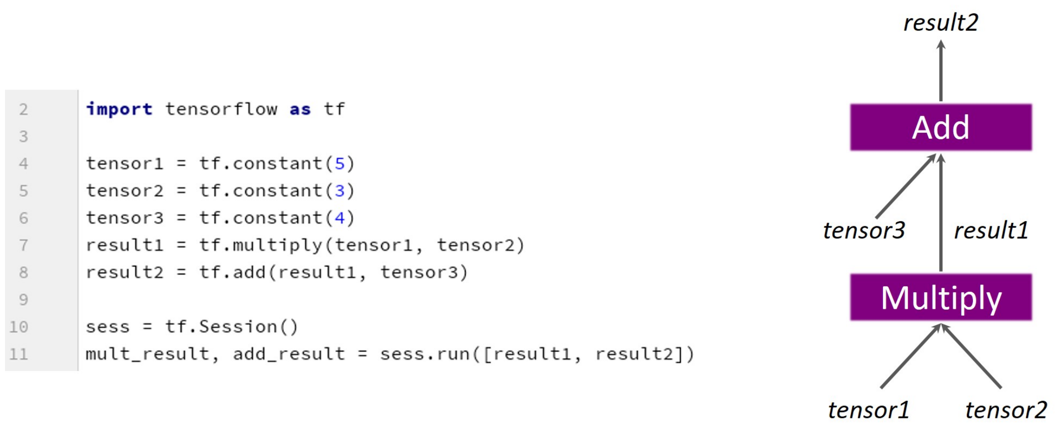
To achieve the aim of this master thesis, the frameworks CNTK [62], Caffe [61] and TensorFlow [2] seemed to be suitable. The Computational Network Toolkit (CNTK) was released by Microsoft and it is utilised with Python or C++ [40], [41]. TensorFlow, which also offers a Python and a C++ binding, was introduced by the Google Brain Team [56], [26]. In contrast, Caffe (Python and MATLAB interface) was developed by the Ph.D. student Jia Yangqing at the University of California, Berkeley [56], [51]. All of these toolkits are open-source and enable image processing with CNNs as it is desired for this work. TensorFlow, however, emerged to be the appropriate library for the thesis tasks. Compared to the other two frameworks, it offers complete documentation as well as pre-trained models.

TensorFlow is Google's successor of their first deep learning framework DistBelief. Both toolkits were developed for deep learning research but also for economic purposes. By now, Google uses its frameworks for their well-known products, for instance, Google Search, Google Maps or Street View. The TensorFlow API was published for anyone in 2015 [2].

During this work, TensorFlow was used with its Python interface. To utilise TensorFlow functions with the Python programming language, this toolkit has to be included as a library into the Python script.

TensorFlow provides a two-step programming principle: Firstly, a computational graph, which is a concatenation of several nodes, is defined by the user and as a second step this graph is executed. The nodes are equivalent to operation types, for instance, mathematical calculations or layer building methods. Moreover, the variables, that are processed within a computational graph, are termed as tensors. The tensors don't have to be initialised within the computational graph definition. To execute the computational graph, a TensorFlow session has to be created and the *run()* method is called up. This method receives variables of the computational graph as arguments and returns their explicit calculated values [2], [26].

Figure 26 shows the programming code of the definition and execution of a TensorFlow session and a schematic representation of the computational graph. The statements of the code lines from four to six create three tensors initialised with constant values. Besides that, the calculations in line seven and eight perform a multiplication and addition with operations provided by TensorFlow. The variables "result1" and "result2", however, don't deliver yet the concrete results of the mathematical operations. Thus, a TensorFlow session is created (line 10) and the *run()* method launches the computation of "result1" and "result2", whereby "mult_result" and "add_result" deliver the demanded results.



```
2   import tensorflow as tf
3
4   tensor1 = tf.constant(5)
5   tensor2 = tf.constant(3)
6   tensor3 = tf.constant(4)
7   result1 = tf.multiply(tensor1, tensor2)
8   result2 = tf.add(result1, tensor3)
9
10  sess = tf.Session()
11  mult_result, add_result = sess.run([result1, result2])
```

Figure 26: Listings of a TensorFlow session (left) and schematic representation of the computational graph (right).

Apart from that, there are several high-level application programming interfaces available to simplify the usage of TensorFlow classes and methods. The APIs TFLearn [58] and TF-Slim [21] were of importance for this work. TFLearn enables a simple layer definition and the user doesn't have to bother about TensorFlow sessions. The library TF-Slim provides network definitions of famous deep learning networks, for example, the VGGnet [55].

In addition to this, TensorFlow offers the visualisation tool TensorBoard. This tool creates considerable overviews of implemented computational graphs and visualises the alteration of tensor values, which change during training progress like the loss or the accuracy. Therefore, these values are stored in a log-file in the course of network training. The resulting TensorBoard visualisation can be observed with a web browser [2].

4 Related Work

This section presents publications, that encourage the implementations of the master thesis. Initially, there are investigations in the field of data augmentation outlined, which is a widely known method to artificially increase the size of a dataset for training neural networks. Beyond that, two approaches for an automatic segmentation with deep learning techniques are discussed.

4.1 Data Augmentation

It is generally accepted that the larger the size of the training dataset, the better are the results of a trained network. P. Y. Simard et al. [54] analysed the influence of an artificially enlarged dataset. They generated synthetic images from basic images with affine transformations including translation, rotation and skewing. This enlarged training dataset led to an improvement of the classification results on the MNIST digit dataset compared to a training with the original dataset. An increase of the initial dataset with elastic deformations, however, offered the best results [54].

Furthermore, K. Chatfield et al. [10] also analysed differences in the network outcomes of various sized training datasets. Firstly, they didn't apply any augmentation method. Secondly, they flipped the images of the initial dataset and finally, they combined cropping and flipping. Their accuracy results have shown that the flipping approach leads to a slightly improvement compared to no augmentation, whereas the combination of cropping and flipping exhibits the best results [10].

Moreover, the artificial increase of the available image amount prevents overfitting, which was outlined in the seminal publication of A. Krizhevsky et al. [33], who applied translations and modifications of RGB values. All in all, extending the datasets is proved to enhance the performance of neural networks. Thus, augmentation approaches are utilised for the lower jawbone datasets of this work, too.

4.2 Segmentation

Two publications emerged to be beneficial to achieve an automatic mandible segmentation with deep learning methods.

4.2.1 Patch-based Segmentation Approach

B. Ibragimov and L. Xing presented in their work [24] “Segmentation of organs-at-risks in head and neck CT images using convolutional neural networks” a tri-planar patch-based segmentation approach. Their goal was to segment automatically several organs of the head and neck region (inter alia the mandible) in CT datasets. For this purpose, they had 50 CT datasets at their disposal, which were manually segmented by medical doctors to produce the ground truth segmentations [24].

In the main, a classification CNN determines the object appearing in an image. The authors of this contribution used a classification network architecture, too. Nevertheless, they classified each voxel of the CT images whether it is part of the organ of interest or not and thus, built up the final segmentation. Therefore, tri-planar patches (small regions) were extracted around the voxel of interest. Tri-planar means that the patches are pulled out from three orthogonal layers and the voxel in the intersection is the voxel of interest. These patches were utilised to train the CNNs. They used positive (center voxel is part of the examined organ) and negative patches (center voxel is part of the background) for training. Moreover, the patches were just extracted from areas where the organ of interest can appear on anatomical grounds. The implementation of the CNNs was achieved with Microsoft’s CNTK framework. On top of that, they did some post processing with applying i. a. dilation and erosion operations. To evaluate the segmentation results, the Dice coefficients were calculated. They achieved the best result for the mandible segmentation (mean DSC of 89.5% $\pm$ 3.6%). The reason for this circumstance is that the lower jaw is very large compared to other segmented structures, its stature is rigid and the bone tissue is clearly visible in CT images [24].

4.2.2 Upsampling-based Segmentation Approach

Another approach to obtain an automatic segmentation is announced by J. Long et al. [38] in their "Fully Convolutional Networks for Semantic Segmentation" contribution. Contrary to the previous presented work, they implemented a fully convolutional network (FCN), which produces an output of the same size as the input image. The output is, consequently, the direct pixel-wise prediction of the segmentation [38].

To create a fully convolutional network, they used the architectures of classification networks (AlexNet, VGGnet and GoogleLeNet), removed the classification layers and replaced the fully connected layers by convolutional ones. Nonetheless, these network adaptations still produce a down-sampled output. Thus, an upsampling layer is added to create the pixel-wise segmentation from the down-sampled network result. The upsampling is conduced with a deconvolution, whereby the deconvolutional filter is learned during training. Figure 27 illustrates the replacement of fully connencted layers with convolutional ones. Beyond that, the vague segmentation result is depicted [38].

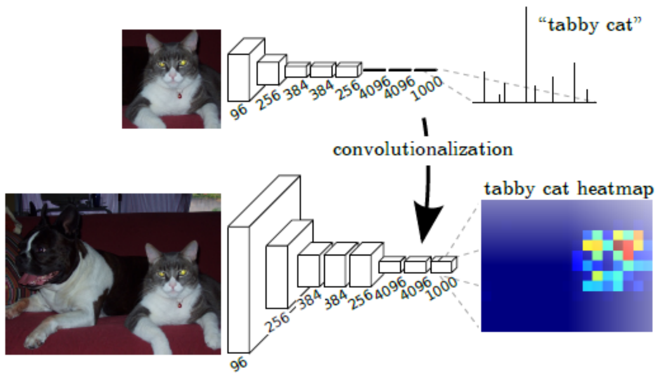


Figure 27: Difference between a classification (above) and a fully convolutional network (below) [38].

To sum up, they used pre-trained classification models, adapted the initial networks to get fully convolutional networks and trained the upsampling layers with the PASCAL VOC 2011 segmentation dataset. Hence, the entire ground truth images were used for supervised training and not only the labels as it is the case for classification networks. The adopted VGGnet, which was initially trained with the dataset from the ImageNet challenge 2014, pointed out to deliver the best segmentation results. They termed this network configuration as FCN-32s. Based on the VGGnet, two further segmentation architectures were introduced, whereby these networks include information from lower layers. The first net is the FCN-16s, which accomplishes upsampling in two steps and includes the fourth pooling layer in one upsampling step. Moreover, the net is initialised with the already trained weights of the FCN-32s network. The second one is the FCN-8s, which conducts the upsampling in three steps and integrates information of the fourth and third pooling layer. The FCN-8s is initialised with trained weights of the FCN-16s network. J. Long et al. also denote the involvement of the max-pooling layers in the upsampling computation as “adding skips”. All in all, their results have shown that the segmentation results improve (see Figure 28) with the integration of pooling layers in the upsampling procedure [38], [55].

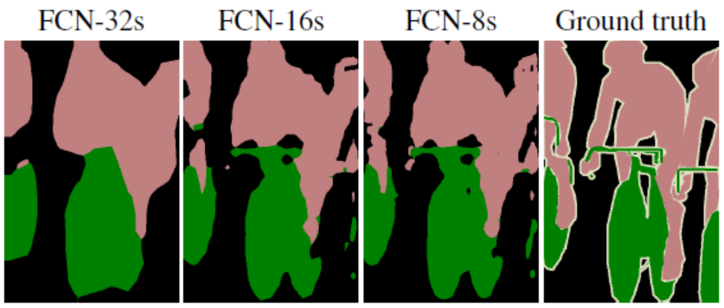


Figure 28: Example image segmented with the three different network architectures (FCN-32s, FCN-16s and FCN-8s) [38].

The segmentation networks of this master thesis are based on the fully convolutional network architecture. Moreover, a method to exclude CT slices from training, that don’t contain the mandible, was implemented.

5 Methods

The succeeding chapter illustrates accomplished work of this master thesis. An overview of available image data is provided as well as the generation of the MeVisLab “Save Slices for Deep Learning” network and the *SaveAsSingleSlices* macro module to ensure data augmentation is described. Apart from that, the implementation of deep learning networks to classify and segment CT images with the toolkit TensorFlow and its high-level APIs TFLearn and TF-Slim is outlined.

All implementations were achieved on a computer with a Intel-Core i5-3470 CPU (3.20 GHz), a RAM of 8 GB and a Windows 8 (64 bit) operating system. Moreover, a NVIDIA GeForce GTX 960 (2 GB memory) was provided. The Python coding was performed with the integrated development environment (IDE) PyCharm.

5.1 Available Computed Tomography Datasets

The CT datasets of the human’s mandible are supplied by the Department of Oral and Maxillofacial Surgery of the Medical University of Graz.

5.1.1 Overview

In total, ten CT datasets were available to train deep learning networks for achieving an automatic segmentation of the mandible. A dataset is saved as one file, but there are several transverse slices stored within this file. It is usual to create lots of tomographic images during one CT examination. However, to apply deep learning networks, it is obligatory to store these image slices as separate files. Therefore, a MeVisLab macro module was developed (see ensuing chapter 5.2).

Additionally, the available tomographic images cover various sections of the head-neck region of a human. Hence, the datasets encompass tissues below and above the lower jawbone. Figure 29 displays three examples of the available image slices that show parts of the Mandibula (red shaded areas).

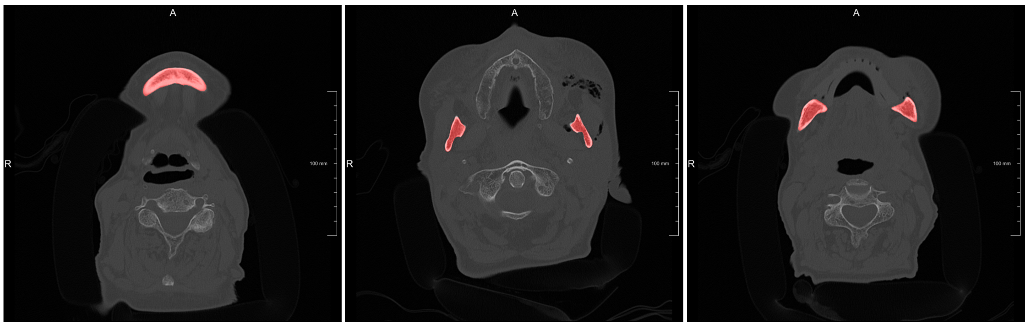


Figure 29: Examples of CT slices displaying parts of the mandible (red).

5.1.2 Ground Truth Segmentations

It is necessary to produce ground truth segmentations to enable a supervised training of deep networks. Therefore, two physicians segmented manually the lower jawbone from all slices of the present CT datasets. Exceptionally, the mandible of dataset nine was segmented by three physicians. All segmentation contours are stored as Contour Segmentation Objects (CSO) files.

Table 1 lists the timespans, that were required to create the ground truth contours, in order to accentuate the enormous effort of segmenting manually anatomical structures. The different physicians are distinguished by the letters A, B and C. Besides that, the average value of timespan, that is needed to segment one dataset manually, results in a duration of 38 minutes and 33 seconds.

Table 1: Duration of manual segmentations to generate the ground truth contours. The datasets were processed by two doctors, exceptionally dataset nine was segmented by three physicians.

Dataset	**Doctor A**		**Doctor B**		**Doctor C**	
	min.	*sec.*	*min.*	*sec.*	*min.*	*sec.*
1	36	15	39	43		
2	45	55	40	18		
3	38	9	39	27		
4	37	36	38	2		
5	37	24	35	5		
6	42	46	40	1		
7	38	5	41	48		
8	35	48	37	27		
9	38	10	37	57	38	40
10	35	52	35	17		

The segmentation contours, that are created by the doctors, were converted into voxelised images during this work. This conversion was achieved with the MeVisLab toolkit. The complete implementation steps are outlined in the succeeding chapter 5.2.

5.1.3 Acquisition Parameters

The CT image acquisition was accomplished with various recording parameters. Table 2 shows an overview of these values. For instance, the slice thicknesses differentiate from each other. There are datasets recorded with 1 mm slice thickness, but there are also images with a thickness of up to 2 mm present. Thus, the image datasets show different resolutions in longitudinal direction. This is also the reason why the slice numbers of the datasets, that indeed contain parts of the mandible, suffer from obvious variations. Besides that, thicker slices lead to a worse image quality. The effect of increasing slice thicknesses causing blurred images is termed as the partial volume effect (chapter 3.1.3). Moreover, the in-plane resolution varies towards different datasets. All voxels have a square base, but the side lengths are dissimilar. Nonetheless, every available CT slice consists of a quadratic 512×512 image matrix.

Table 2: Acquisition parameters of the CT images. The in-plane resolution is indicated by the "Side length of voxels" column. Two slice numbers in one entry indicate that the doctors segmented a diverse number of slices.

Dataset	**Side Length of Voxels** *mm*	**Slice Thickness** *mm*	**Number of Slices**	**Number of Slices incl. the Mandible**
1	0.428	1.0	217	90
2	0.383	1.0	154	92/91
3	0.420	2.0	136	39
4	0.523	1.999	141	39
5	0.510	1.0	232	97/98
6	0.461	1.5	168	66/65
7	0.465	1.0	200	73
8	0.451	2.0	110	49
9	0.383	1.0	154	91
10	0.461	1.5	168	66

5.1.4 Dataset Size

To sum up, 1680 CT slices and 3514 ground truths exist for training and testing deep learning networks. Considering the image slices that display parts of the mandible, then there are only 1494 ground truths available. However, training deep learning networks requires a larger amount of datasets. Take, for example, the work of B. Ibragimov and L. Xing [24], who had 50 CT datasets at their disposal to train and test a network for lower jawbone segmentation. Accordingly, there is a lack of medical images that can be used for this work. Several reasons can be marked out for this circumstance: One issue is that all ground truths are generated manually by doctors. In daily routine, they don't have plenty of time to conduct these manual segmentations. Another reason is that images, that are acquired during daily routine in hospital, are usually stored as Digital Imaging and Communications in Medicine (DICOM) files [42], which comprise private patient information like the patient's name or date of birth. Thus, these types of images are not public accessible. To fulfil ethical demands, the images must be anonymised. Therefore, the CT images, that are used for this master thesis, were provided as Nearly Raw Raster Data (NRRD) files. NRRD-files don't include any personal information about the patient. Finally, the CT images, that are used for this work, must not have any teeth. Teeth often contain metallic implants, which produce distinct artefacts (see Figure 30). Hence, a segmentation of the lower jawbone is difficult to accomplish.

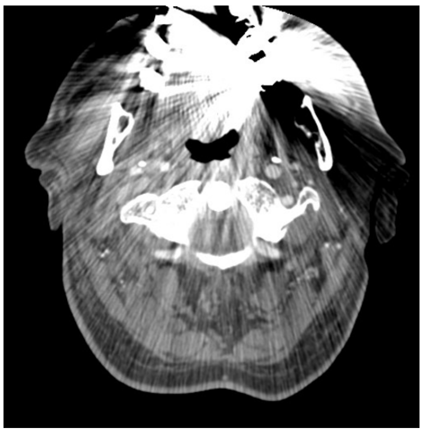


Figure 30: CT image corrupted by metal artefacts. The artefacts are visible as stripes. Adopted from [28].

However, to circumvent the problem of too few available images, data augmentation is an opportunity to enlarge artificially the size of the dataset. To increase the magnitude of the datasets of this work, the augmentation methods noise addition and affine transformation were implemented with MeVisLab.

5.2 Implementations in MeVisLab

It was necessary to achieve three desired functionalities with the medical image processing platform MeVisLab, before the training of the neural networks could be accomplished:

1. Conversion of the ground truth segmentation contours into voxelised images;
2. Automatic storage of the image slices as separate files;
3. Application of data augmentation methods.

Therefore, the MeVisLab module network "Save Slices for Deep Learning" and the macro module *SaveAsSingleSlices* were implemented. The generated network enables the conversion of a segmentation contour into a binary mask and a depiction of the patients' CT images. In addition to this, the macro module *SaveAsSingleSlices* stores the slices as separate and perhaps modified image files.

Following subchapters depend on the MeVisLab modules' HTML help. The network and the macro module implementations were obtained with the support of the MeVisLab Definition Language (MDL) Reference [4] and the MeVisLab Scripting Reference [6]. A part of the conducted work was already achieved during my Master Project [48], [49], [47], [20]. Thus, similar instructions can also be found in the documentations of this preceding work. However, modifications of the network and the documentations were conducted.

5.2.1 "Save Slices for Deep Learning" Network

As a first step, the MeVisLab network "Save Slices for Deep Learning", which can be seen in Figure 31, was created. The functions of the particular modules are explained in the paragraphs below.

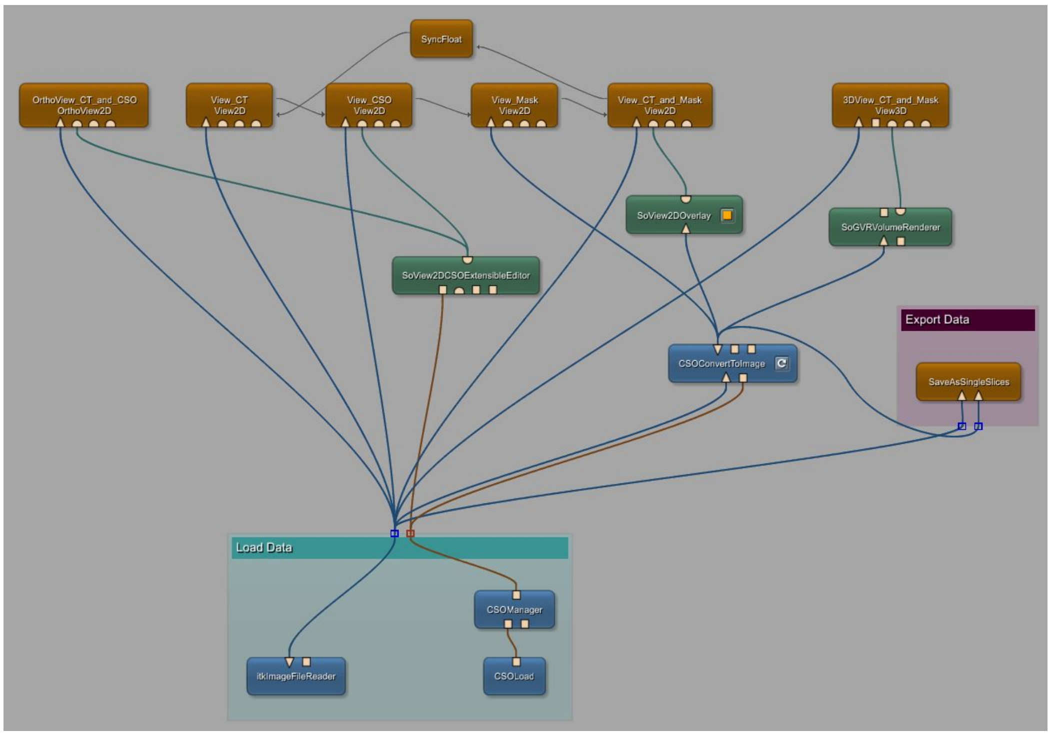


Figure 31: Implemented "Save Slices for Deep Learning" network. The "Load Data" group enables the integration of the CT images and the CSO files into the network. The *View2D* and *View3D* modules visualise medical data and the macro module *SaveAsSingleSlices* provides the storage of separate and probably deformed images.

At the lower section of the network is the "Load Data" group visible. The patient's CT dataset is imported into the MeVisLab environment with the *itkImageFileReader* module. The contours of the lower jawbone are loaded into MeVisLab with the *CSOLoad* module. A CSO file encloses all contours of a CT dataset as a CSOList. To make these segmented contours available in MeVisLab, the *CSOManager* module is used. The output of the *CSOManager* module isn't the ground truth information for deep learning networks yet, because the CSOs represent the border of the mandible in each image slice, but they aren't voxelised images.

Nevertheless, neural networks for image segmentation require voxelised masks for training. As a consequence, the CSOs are converted into voxelised images with the *CSOConvertToImage* module. This module requires the original CT images as well as the CSOLists as an input, while the output delivers binary images. The segmented areas, that belong to the lower jaw, are indicated as the foreground with a voxel value of one, whilst the background is set to zero. It is central to tick the "Voxelise Border" checkbox at the module's panel to add the path points of the CSOs to the generated image. Furthermore, it was adjusted to fill the segmentation borders. The output of the *CSOConvertToImage* module is composed of multiple slices, too. There are not only output masks with visible segmentation regions, but there are also slices without any foreground voxels. The reason for this circumstance is that the input CT includes slices below as well as above the mandible. Thus, there aren't any segmentation contours of the mandible at these slices present, which results in output masks containing only voxel values of zero.

The four *View2D* modules are selected to display the patient CT dataset and the generated segmentation masks. It is possible to scroll through the individual slices within the viewer panels. Three resulting representations of these modules are visible in Figure 32. All displayed images show slice No. 32 from patient CT one and the contours are segmented by physician A.

The *View_CT* module depicts a two-dimensional (2D) representation of the transversal CT slices (No. 1 of Figure 32). Moreover, the module *View_CSO* displays the original CT image and the segmentation contour. Therefore, the CT images and the contours are provided as inputs to this module. Before the CSOs are brought to the *View2D* module, the CSOList is added to a *SoView2DCSOExtensibleEditor* module. This module ensures a visualisation of the single contours. Furthermore, the module *View_Mask* shows the output of the *CSOConvertToImage* module. As a result, there is the binary mask of the segmented areas visible (No. 2 of Figure 32). Finally, the *View_CT_and_Mask* module is added, which illustrates the CT images and the overlaid masks (No. 3 of Figure 32). Again, there are two inputs for the *View2D* module necessary: the original CT image and the output of the *SoView2DOverlay* module. The latter module supports an overlay of a 2D image over another one. In this case, the output binary mask of the *CSOConvertToImage* module should be superposed on the original patient CT. The *SoView2DOverlay* module parameter “Base Color” is set to orange and for the “Alpha Factor” is a value of 0.4 chosen. The alpha factor indicates the transparency of the superposed mask. An alpha factor of zero means complete transparency, whereas greater values lead to higher opacities.

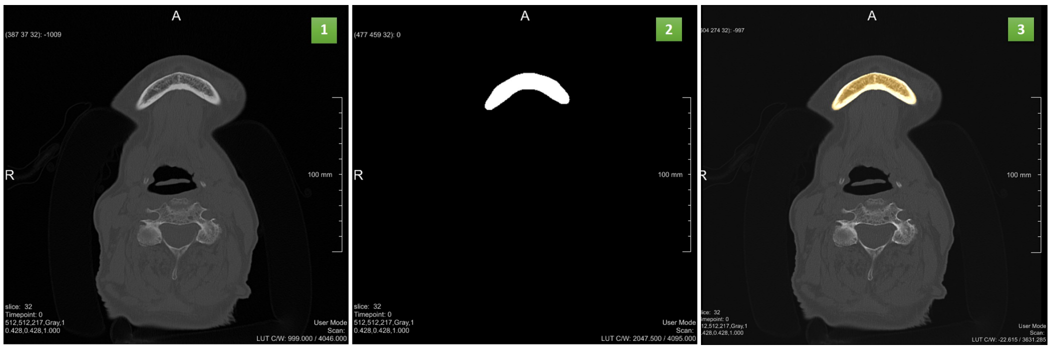


Figure 32: Various representations created with the *View2D* modules. No. 1 shows the original CT, No. 2 the binary mask (white) and No. 3 the CT with the overlaid binary mask (orange).

SyncFloat is a module that synchronises the currently shown image slices between various *View2D* modules. As a consequence, a comparison between different representations of one slice is automatically possible. It is noticeable that MeVisLab maps specific image information, for example, the in-plane resolution, slice thickness or grey values in the viewer panels of the modules.

Additionally, the *OrthoView2D* module, termed as *OrthoView_CT_and_CSO*, shows orthogonal views of the patient and the contours. There is the standard transversal plane (No. 1 of Figure 33), the sagittal plane (No. 2 of Figure 33) and the frontal plane (No. 3 of Figure 33) visible. The yellow crossed lines in one image indicate the location of the other two planes. This module needs the imported CT image and the output of the *SoView2DCSOExtensibleEditor* as an input.

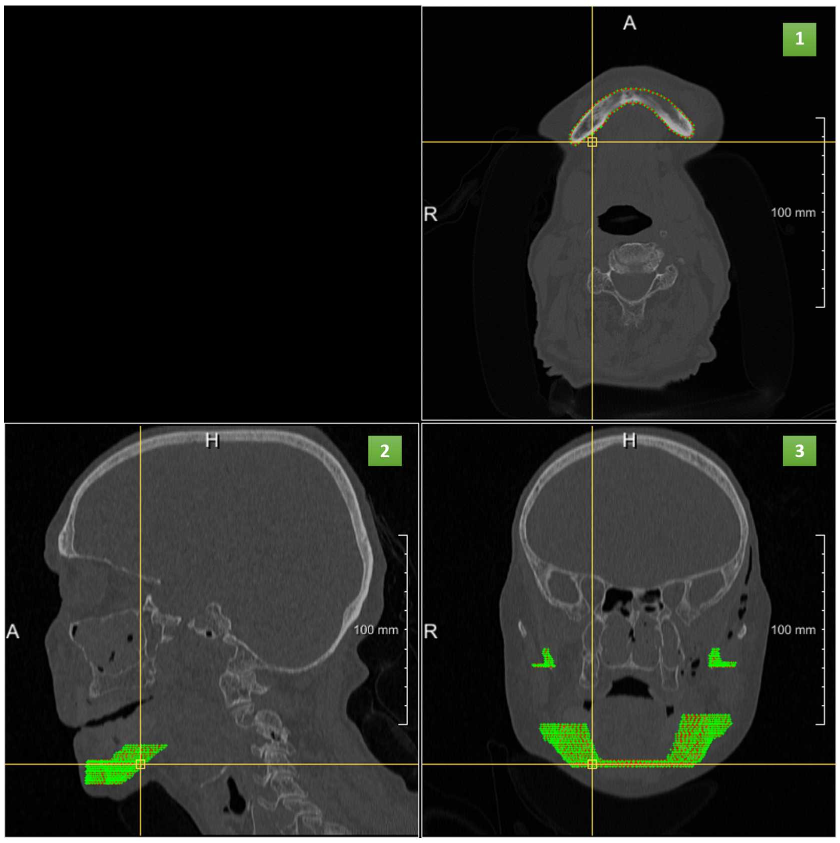


Figure 33: Depiction with *OrthoView2D*. No. 1 shows the original transversal plane, No. 2 the sagittal plane and No. 3 the frontal plane. The green areas correspond to the segmentation contours.

Last, the *View3D* module, named as *3DView_CT_and_Mask*, is added to the network in order to present a 3D visualisation of the CT dataset and the overlaid masks (see Figure 34). The *SoGVRVolumeRenderer* module enables the rendering of the generated masks. There are gradations along the longitudinal axis of the body visible, but they can't be seen along the sagittal or the transversal axis. The reason for this fact is that the in-plane pixel size has a value of 0.428 mm, but the out-of-plane resolution is worse, as the slice thickness has a value of 1 mm.

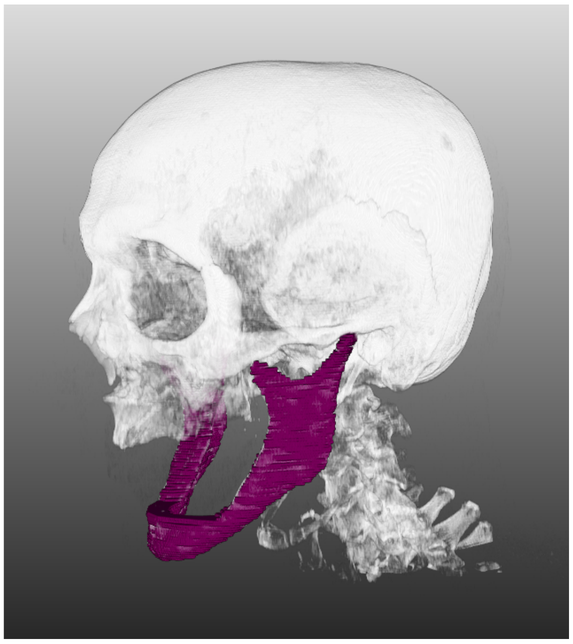


Figure 34: 3D visualisation of a CT dataset and the segmented mandible.

Finally, each slice of the CT and mask datasets ought to be saved as single slices in detached files. As there aren't any MeVisLab internal modules available to accomplish this task, an own macro module was implemented. This macro module, termed as *SaveAsSingleSlices*, is also able to apply geometric transformations and noise on the original datasets to increase the amount of training and testing data.

5.2.2 ***SaveAsSingleSlices*** Macro Module

As a next step, the global macro module *SaveAsSingleSlices* was developed in order to achieve a separate storage of image slices and to apply data augmentation. Both claimed functionalities were implemented within one macro module. A quite small part of the module is based on forum posts of the Fraunhofer MeVisLab forum [39].

As already mentioned in section 3.2, a macro module consists of a macro network, a Definition-, a Script- and a Python-file. The Definition-file was created automatically by the MeVisLab Project Wizard, hence this file wasn't changed. The implementation of the other three parts is outlined in the following paragraphs.

Macro Network

The *SaveAsSingleSlices* module appearance with its two ML image connectors is shown on the left side of Figure 35, whilst the macro network itself can be seen in the right part.

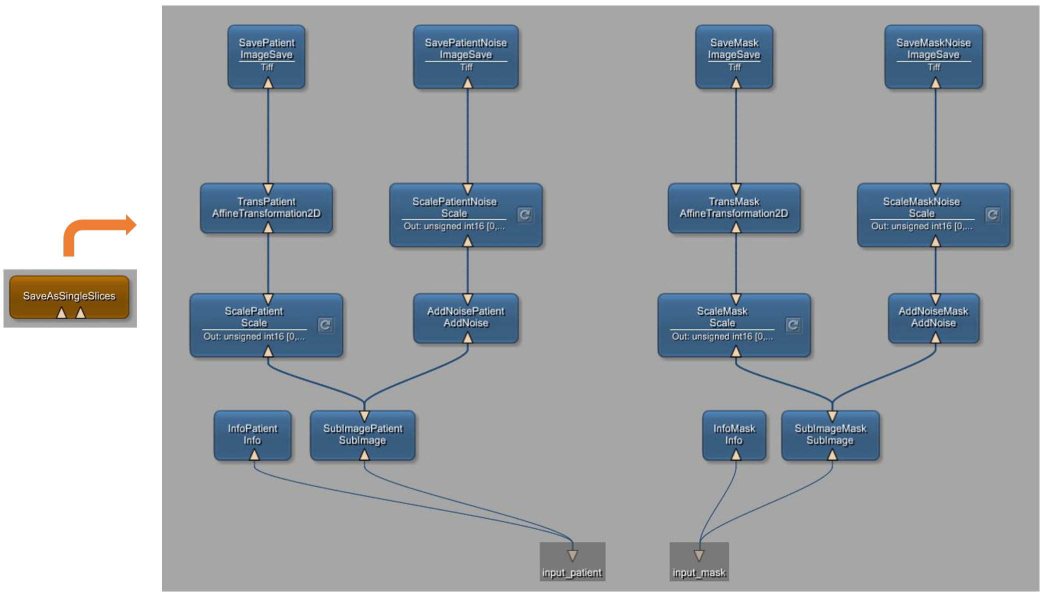

SaveAsSingleSlices
SavePatient ImageSave Tiff
SavePatientNoise ImageSave Tiff
SaveMask ImageSave Tiff
SaveMaskNoise ImageSave Tiff
TransPatient AffineTransformation2D
ScalePatientNoise Scale Out: unsigned int16 [0,...
TransMask AffineTransformation2D
ScaleMaskNoise Scale Out: unsigned int16 [0,...
ScalePatient Scale Out: unsigned int16 [0,...
AddNoisePatient AddNoise
ScaleMask Scale Out: unsigned int16 [0,...
AddNoiseMask AddNoise
InfoPatient Info
SubImagePatient SubImage
InfoMask Info
SubImageMask SubImage
input_patient
input_mask

Figure 35: Internal network of the *SaveAsSingleSlices* module. On the left side is the module surface (brown) visible, whereas the right side shows the internal network consisting of basic MeVisLab modules.

The implemented macro offers two module inputs: The first ML image input is required for the CT images and the second one is needed for the created masks. The processing steps are the same for the CTs and the masks. Accordingly, the modules and their chosen settings are identical for the left and the right "network wing".

The *Info* modules (*InfoPatient* and *InfoMask*) list specific information about the input images, for instance, image size or bit depth. To ensure further processing steps, the slice number of a dataset was identified with this module.

The *SubImage* modules (*SubImagePatient* and *SubImageMask*) are required to select one distinct slice from the image stack. Beyond the *SubImage* modules, the network divides into two further processing paths. One path is utilised for affine transformations and the other one is used for adding noise. Moreover, four *Scale* modules (*ScalePatient*, *ScalePatient-Noise*, *ScaleMask* and *ScaleMaskNoise*) are added in order to scale the grey values onto a bit depth of UINT16 (65536 grey values) or onto a depth of UINT8 (256 grey values).

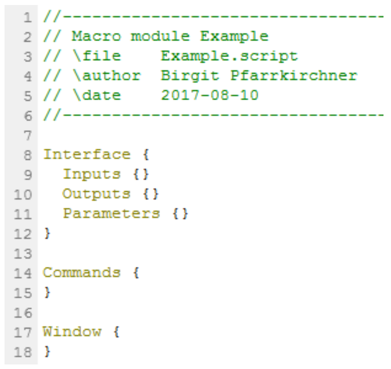
To save a single slice, the *ImageSave* modules (*SavePatient*, *SavePatientNoise*, *SaveMask*, *SaveMaskNoise*) are appended to the macro network. Generally, it is possible to select between numerous datatypes to save the slices with this module, for example, TIFF or DICOM. For this macro module, the datatypes TIFF (default), JPEG, PNG and DICOM are enabled. The *ImageSave* module is able to export only one slice. To ensure that all slices are saved without any user interaction, an iteration loop was implemented within the Python script.

In addition to this, it is claimed to implement opportunities for data augmentation. Firstly, the modules *AffineTransformation2D* (*TransPatient* and *TransMask*) qualify the macro to create new transformed slices, that are also separately saved. In the main, this module type involves the transformations translation, rotation, scaling and shearing as well as combinations of them depending on the parameters that are set. Nevertheless, the transformations scaling and rotation are supported for the *SaveAsSingleSlices* macro. Moreover, a mirroring of the image around the sagittal axis is provided by setting the scaling parameter along the x-axis to a value of -1. These transformation types are enabled, because they are physiological: The head can be turned a view grades to the left or right and it is possible that there are small size variations of the human anatomy. The flipping of the image is a practical occasion to double the number of images. Shearing isn't provided, because this transformation doesn't produce physiological deformations. Translation wasn't supported, since a human is positioned in the isocentre of a computed tomograph. Thus, a translation is not absolutely necessary. Furthermore, it is possible to apply several geometric transformations simultaneously to achieve combined deformations like flipping-rotation, flipping-scaling, rotation-scaling and flipping-rotation-scaling.

Finally, the *AddNoise* modules (*AddNoisePatient* and *AddNoiseMask*) append noise on the CT images and perhaps on the masks. It can be distinguished between various types including Uniform noise, Gaussian noise and Salt and Pepper noise. However, it is possible to apply one noise type, but not a superposition of these noise options.

Script-file

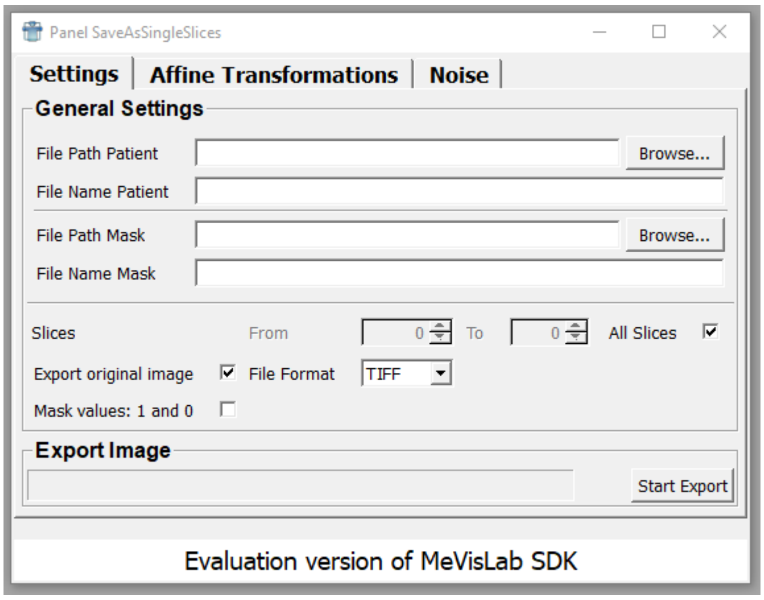
The Script-file is obligatory to create a macro module. Figure 36 shows the basic programming code of this file without own specifications. This file is mandatory to define the interface of the *SaveAsSingleSlices* module block. The inputs of the module are declared in the Interface section of Figure 36. As module outputs are not provided, there isn't any programming code added at the Outputs section. Nevertheless, lots of parameters are declared that facilitate the user to define settings of the macro module. These parameters involve general adjustments, such as the target folder, but also parameters for data augmentation. Furthermore, a conjunction to the Python script (Commands section in line 14) was implemented and a Field-Listener was added to trigger the Python script in case of a started image exportation by the user. Finally, the Window section in line 17 defines the appearance of the user panel to set the module's parameters for saving original slices and applying data augmentation.



```
 1 //--------------------------------
 2 // Macro module Example
 3 // \file    Example.script
 4 // \author  Birgit Pfarrkirchner
 5 // \date    2017-08-10
 6 //--------------------------------
 7
 8 Interface {
 9   Inputs {}
10   Outputs {}
11   Parameters {}
12 }
13
14 Commands {
15 }
16
17 Window {
18 }
```

Figure 36: Basic framework code of a Script-file.

Figure 37 shows the resulting user panel of the implemented *SaveAsSingleSlices* macro module with the selected Settings-tab. The General Settings area allows to assess the parameters for an exportation of the slices. The path for saving the CT and mask slices must be determined. It is possible to save the masks and the CT images in separate folders. Furthermore, templates for the filenames have to be stated. There is the occasion to export all slices (default) or just a distinct number of slices. The checkbox "Export original image" has to be ticked to store the original dataset, as it is also possible to export only modified images. Moreover, the user can select between exporting TIFF, PNG, DICOM or JPEG files. If "Mask values: 1 and 0" is checked, the generated masks have grey values of zero and one. Otherwise, the foreground is represented with pixels of the maximum grey value (255 for UINT8 and 65535 for UINT16). To verify the entered parameters, some checks are implemented in the Python script (e.g. meaningful numbers or existing file paths).


Panel SaveAsSingleSlices

Settings | Affine Transformations | Noise

General Settings

File Path Patient [ ] Browse...
File Name Patient [ ]
File Path Mask [ ] Browse...
File Name Mask [ ]

Slices From 0 To 0 All Slices ☑
Export original image ☑ File Format TIFF
Mask values: 1 and 0 ☐

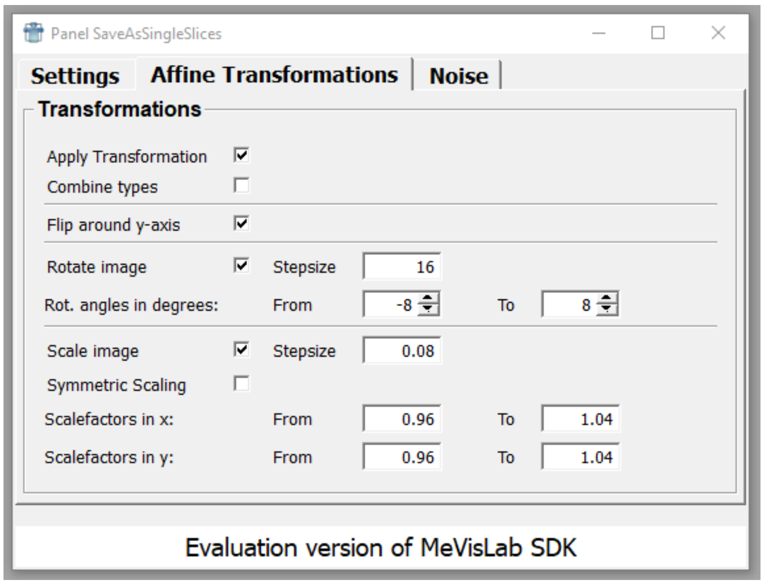
Export Image

Start Export

Evaluation version of MeVisLab SDK

Figure 37: Panel of the *SaveAsSingleSlices* module with selected Settings-tab.

If it is intended to save affine transformed slices, the user has to open the Affine transformations-tab, which can be seen in Figure 38, and tick the “Apply Transformation” checkbox. Provided that “Flip around y-axis” is checked, the code of the Python script sets the scaling factors along the x-axis of the *AffineTransformation2D* modules to -1. Ticking “Rotate image” allows to enter parameters for saving rotated images. The user is able to state the start and stop rotation angles in degrees and the incremental step size of the rotation. Thus, it is possible to save several differently rotated images with one exportation. Additionally, checking the “Scale image” box permits to save scaled images. In this case, the user has to enter the start and stop scale factors in x- and y-direction and the step size for the adjustment of these parameters. If “Symmetric Scaling” is true, the same scaling parameters are applied in x- and y-direction. Hence, there is a smaller number of scaled images created. As described, all these transformation types are applied apart from each other. Nevertheless, there is the possibility to combine two or three transformation types. Therefore, the Checkbox “Combine types” should be ticked. After applying a transformation, it was important to ensure that the transformation parameters are set to initial values, otherwise the changed parameters are saved and they are also applied during a following function call, even though this isn't desired.



Panel SaveAsSingleSlices

Settings | Affine Transformations | Noise

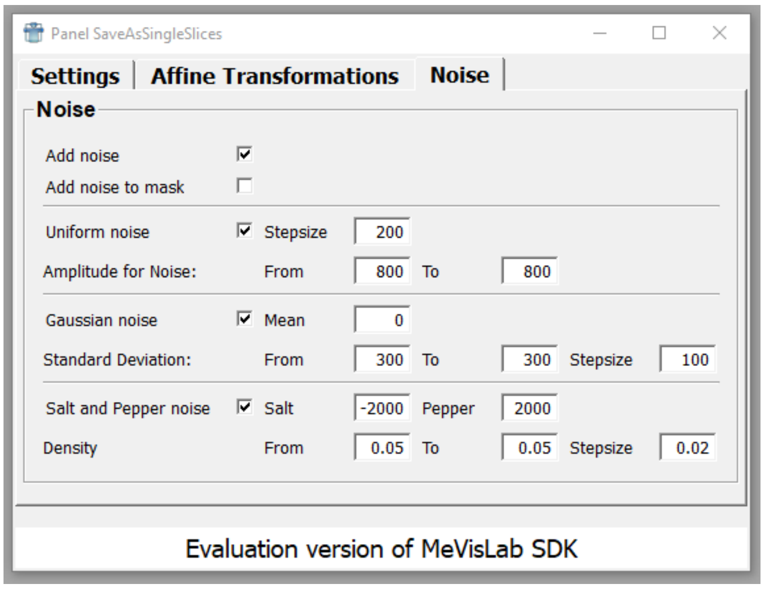
Transformations

Apply Transformation	✓				
Combine types	☐				
Flip around y-axis	✓				
Rotate image	✓	Stepsize	16		
Rot. angles in degrees:		From	-8	To	8
Scale image	✓	Stepsize	0.08		
Symmetric Scaling	☐				
Scalefactors in x:		From	0.96	To	1.04
Scalefactors in y:		From	0.96	To	1.04

Evaluation version of MeVisLab SDK

Figure 38: Panel of the *SaveAsSingleSlices* module with selected Affine Transformations-tab.

Finally, adding noise for data augmentation is possible by ticking the "Add noise" checkbox at the Noise-tab of the implemented panel (see Figure 39). As default, the noise is added on the CT images but not on the generated masks. However, if the checkbox "Add noise to mask" is ticked, the noise is also applied on the masks. To add Uniform, Gaussian or Salt and Pepper noise, the respective checkboxes have to be marked. If Uniform noise is chosen, a start and stop value for the amplitude has to be defined. Moreover, the step size must be stated to generate automatically multiple noisy images. The parameters of Gaussian noise cover the mean value and the standard deviation, which can be varied with a chosen step size. Provided that Salt and Pepper noise is selected, the amplitudes of salt and pepper have to be specified. Besides that, the start and stop values of the density and also the step size must be declared. Adding various noise types is, however, possible separately, but not in combination as it is the case for the affine transformations.


Panel SaveAsSingleSlices

Settings | Affine Transformations | Noise

Noise

Add noise	✓					
Add noise to mask	☐					
Uniform noise	✓	Stepsize	200			
Amplitude for Noise:		From	800	To	800	
Gaussian noise	✓	Mean	0			
Standard Deviation:		From	300	To	300	Stepsize 100
Salt and Pepper noise	✓	Salt	-2000	Pepper	2000	
Density		From	0.05	To	0.05	Stepsize 0.02

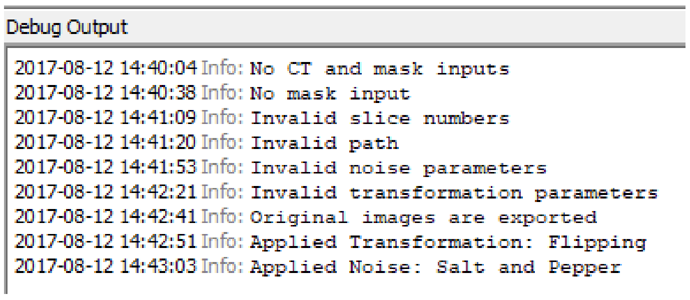
Evaluation version of MeVisLab SDK

Figure 39: Panel of the *SaveAsSingleSlices* module with selected Noise-tab.

If all necessary parameters are entered, the data exportation is launched with the "Start Export" button at the Settings-tab (see Figure 37). A progress bar shows the progression of the exportation of an image dataset and its corresponding mask dataset. Examples of the synthetic generated transformed and noisy images can be found in the Results section 6.1.

Python-file

The Python script controls the functionality of the macro module. It passes on by the user entered parameters to the modules of the macro network. Thereby, the plausibility of the stated values is checked. Beyond that, the Python script permits a "communication" with the user. Messages, which inform about executed transformations or incorrect input parameters, are displayed in the debug output of the MeVisLab surface. For example, Figure 40 shows several types of messages.



Debug Output

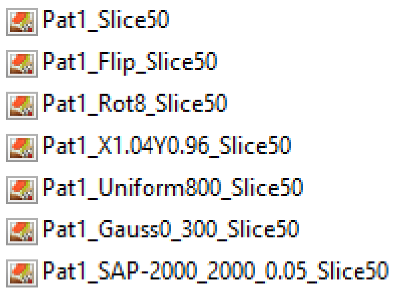
```
2017-08-12 14:40:04 Info: No CT and mask inputs
2017-08-12 14:40:38 Info: No mask input
2017-08-12 14:41:09 Info: Invalid slice numbers
2017-08-12 14:41:20 Info: Invalid path
2017-08-12 14:41:53 Info: Invalid noise parameters
2017-08-12 14:42:21 Info: Invalid transformation parameters
2017-08-12 14:42:41 Info: Original images are exported
2017-08-12 14:42:51 Info: Applied Transformation: Flipping
2017-08-12 14:43:03 Info: Applied Noise: Salt and Pepper
```

Figure 40: Messages to inform the user.

- "No CT and mask inputs" means that there isn't any data connection to the inputs of the module joined, which leads to an abortion. However, the exportation with one connected input is allowed.
- "No mask input" informs that there isn't a mask input offered. Nevertheless, a CT input is fixed.
- "Invalid slice numbers" indicates that there are wrong slice numbers, for instance, negative values entered. An exportation is disabled.
- "Invalid path" warns that there isn't an existing path for storage stated. Thus, the slices aren't saved.
- "Invalid noise parameters" implies that a non-meaningful noise parameter, for example, a negative density is stated. Hence, adding noise is not possible.
- "Invalid transformation parameters" indicates that there isn't a reasonable transformation parameter, such as a positive step size, entered. Consequently, a transformation is not executable.
- "Original images are exported" informs that the original version of the input is successfully stored.
- "Applied Transformation: Flipping" communicates that a deformed version of the original image is stored. In this case, the applied transformation is flipping.
- "Applied Noise: Salt and Pepper" informs that a noisy version of the original image is saved. In this case, the applied noise type is Salt and Pepper.

Moreover, the implementations of the Python-file support iterations over the individual slices as well as the automatic variation of parameters like the rotation angle. Finally, the Python script manages the self-acting generation of the filenames of the exported slices. Therefore, the software adds name extensions to the chosen template filename in order to distinguish between the slices and the applied data augmentation methods as well as to prevent an overwriting of files.

Figure 41 lists exported files of slice 50 of image dataset one. “Pat1” is the template filename (stated by the user) and “_Slice50” is automatically attached for the slice number by the Python script. Furthermore, if flipping is applied, “_Flip” is added to the file name. The rotation is indicated by “Rot8” and the scaling by “X1.04Y0.96”, whereby the numbers after the abbreviations represent the applied transformation parameters: “Rot8” means that the image is rotated by an angle of 8°. The last three examples of Figure 41 illustrate the filenames for adding noise. The numbers for Gaussian noise indicate the mean and the standard deviation, those for Salt and Pepper noise stand for the two amplitudes and the noise density and the value of Uniform noise indicates the amplitude.



Pat1_Slice50
Pat1_Flip_Slice50
Pat1_Rot8_Slice50
Pat1_X1.04Y0.96_Slice50
Pat1_Uniform800_Slice50
Pat1_Gauss0_300_Slice50
Pat1_SAP-2000_2000_0.05_Slice50

Figure 41: Generated filenames of exported slices. “Pat1” corresponds to the entered template filename and “Slice50” designates the number of the exported image slice. The further name extensions indicate applied deformations.

The generation of these standardised filenames is important to ensure the succeeding deep learning implementations.

5.3 Classification of the CT Images

Succeeding deep learning implementations comprise a classification of the CT slices. The idea is to achieve a self-acting decision whether the mandible appears in an image or not, because many images occur that don't show the lower jawbone. These CT slices should be eliminated with a trained classification network and consequently, the segmentation network was just trained with images that show parts of the lower jaw. B. Ibragimov and L. Xing removed in their work [24] also CT slices, that don't include the mandible, from training and testing neural networks. They applied, however, geometrical methods for slice exclusion.

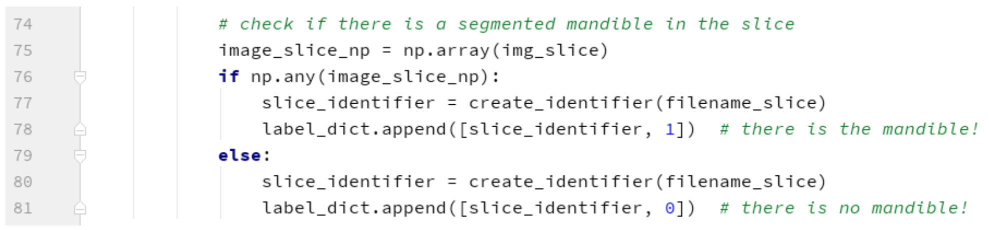
During this work, various classification networks were trained with TensorFlow and its API TFLearn. The CNN, which offers the best performance, was selected for the further segmentation task.

5.3.1 Dictionary of Classification Labels

Before a training of classification networks can be accomplished, it is necessary to know the label of every image. These labels indicate which class an image is attending. In the course of this work, two classes exist: There is the case that the lower jawbone appears in a CT slice and alternatively, that the mandible doesn't occur. The labels of the images can be extracted from the ground truth segmentations. Therefore, the masks, which are generated from the manual segmentations of physician A, were exported with the MeVisLab implementations. Those masks are binary images, as there appear only black and white pixels. If a CT slice contains the mandible, its corresponding mask encompasses white pixels. Otherwise, the mask exhibits only pixel values of zero. Thus, it is possible to infer the classification label of a CT slice from the mask's pixel values.

The implemented Python script *Dictionary_Labels_Classification.py* permitted the extraction of the labels and the storage of them as a "Labels Dictionary" in a Comma-Separated-Values (CSV) file. Within this dictionary are the labels of the original images and their unique identifiers stored. The identifier enables a look up of the label of a distinct image slice of a dataset. Besides that, the storage of the labels in a dictionary offers the advantage that they don't have to be determined from the masks every time before a new network is trained.

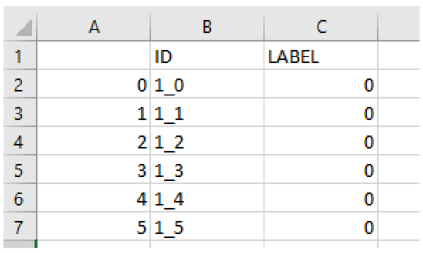
The listings in Figure 42 demonstrate the determination of the labels. The code within the if-statement is executed if any pixel of an examined image is white. If this is the case, the image identifier is stored with a label of 1, which stands for an appearing mandible. The implemented *create_identifier()* function generates this identifier. In contrast, the else-statement leads to a label of 0. These investigations are executed for each of the 1680 masks.



```python
# check if there is a segmented mandible in the slice
image_slice_np = np.array(img_slice)
if np.any(image_slice_np):
    slice_identifier = create_identifier(filename_slice)
    label_dict.append([slice_identifier, 1])  # there is the mandible!
else:
    slice_identifier = create_identifier(filename_slice)
    label_dict.append([slice_identifier, 0])  # there is no mandible!
```

Figure 42: Listings to figure out the label of an examined mask.

To generate the identifiers with the *create_identifier()* function, the filenames of the exported masks are used. The masks' filenames have a format of the type: *Mask1a_Slice1.tif.* The first numeral indicates the number of the dataset and the second one stands for the slice number. Both numbers were extracted from the filenames with the assistance of string processing functions in Python. Moreover, the character "a" stands for ground truth masks, which were generated by physician A. The resulting identifier has a formatting of *dataset_slice.* Figure 43 illustrates examples of the identifiers and their determined labels stored in the CSV-file.



	A	B	C
1		ID	LABEL
2	0	1_0	0
3	1	1_1	0
4	2	1_2	0
5	3	1_3	0
6	4	1_4	0
7	5	1_5	0

Figure 43: Exemplary entries of the created "Labels Dictionary". The first column contains consecutive numbers, the second one comprises the identifiers (ID) with the format of *dataset_slice* and the LABEL column lists the related labels.

Apart from that, it is sufficient to store the labels of the original images even if networks are trained with artificial enlarged datasets. Take, for example, an image slice that is rotated. This rotated instance exhibits the same label as the original one. As a result, it is not necessary to determine the label for every augmented image from its ground truth mask. The labels of the transformed images can be marked out from the dictionary with the support of the identifier, too. Thereby, the ID is extracted from the augmented images' filenames, since they also enclose the numbers of the dataset and of the slice.

5.3.2 Data Processing

To train neural networks in TensorFlow and TFLearn, the image data has to be processed and put into a distinct format. Therefore, the Python programme *Data_Processing_Classification.py* was implemented.

As a first step, this script permits the loading of a set of images, which should be used for training or testing a network. For that, the CT images are stored within one folder. Moreover, the order of the particular image slices is shuffled to ensure, for instance, a random division of the images into a training and a validation dataset. Another issue is that the images might be down-sampled for a processing with neural networks. Down-sampling of images is a common trick, because smaller images lead to less input nodes of the network and as a consequence, less parameters have to be trained.

As a next step, the labels of the images are looked up in the previously generated "Labels Dictionary". Therefore, the IDs (*dataset_slice*) of all images are extracted from their filenames (see Figure 44). Afterwards, the labels are converted into one-hot vectors, which is necessary for supervised training. A one-hot vector has as many entries as classes are distinguished and each vector entry stands for a distinct class. This vector type has only zero-valued entries, except the entry, that represents the occurring class, is set to one. Hence, the one-hot vectors of this task have two entries, whereby $[1, 0]$ indicates that there isn't the mandible and $[0, 1]$ means that a part of the lower jaw appears in the image. Besides that, each one-hot vector entry can be imagined as one output unit of the neural net.

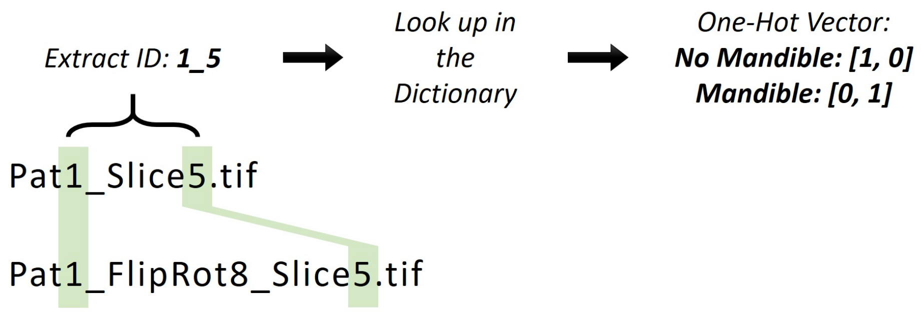


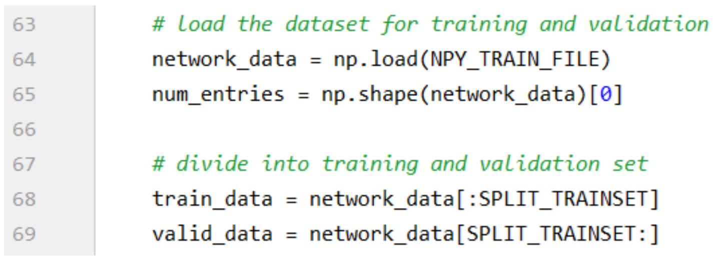
Figure 44: Steps to get an image's one-hot vector. The identifier for the look up is extracted from the number of the dataset and the slice (green). The determined label is converted into a one-hot vector.

At last, the images and their corresponding one-hot vectors are converted into NumPy arrays, which are merged into one list. Afterwards, this list is stored as a NPY-file, since this format is specific to save NumPy arrays on a storage medium and it can also be processed by TensorFlow. Apart from that, a CSV-file with a register of the filenames and their labels in the shuffled order is stored to permit a manual check during training, since the NPY-file comprises only numerical data.

5.3.3 Implementation of the Code Framework

It is required to implement the code framework, which encompasses the structure as well as the training and testing options of the net. Therefore, the classification network was implemented in the Python-file *Classification_Network.py* with TFLearn. This library offers many practical functions to build the scaffold of a convolutional neural network as well as to train and test it. The documentation of TFLearn [58] and also the online tutorial [50] supported the implementation of following CNNs.

Firstly, the dataset, which is stored as a NPY-file and covers the images and their one-hot vectors, is loaded into the Python script. Thereafter, the shuffled images are divided into a training set and a validation set (see Figure 45), whereby the variable "SPLIT_TRAINSET" defines the subdivision of the loaded dataset. The training set permits the calculation of the network's weights. Despite that, the TFLearn network uses the validation set during training to review the training process, since the accuracy of the validation dataset should not decrease after weight updates.

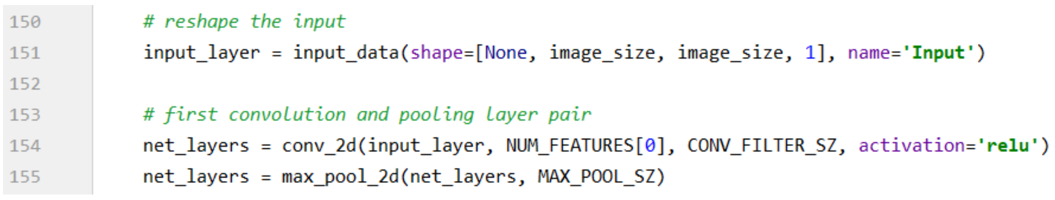


```
# load the dataset for training and validation
network_data = np.load(NPY_TRAIN_FILE)
num_entries = np.shape(network_data)[0]

# divide into training and validation set
train_data = network_data[:SPLIT_TRAINSET]
valid_data = network_data[SPLIT_TRAINSET:]
```

Figure 45: Division of the dataset into a training and a validation set.

As a next step, the topology of the network is defined. Figure 46 displays the programming code of the input and of the following convolutional and max-pooling layers. The TFLearn function *input_data()* generates the first layer and requires the size of the input images as a parameter. Furthermore, the convolutional layer is created with the *conv_2d()* function. The number of the feature maps, which should be generated, is determined with the “NUM_FEATURES[0]” variable and the size of the convolutional filter is defined with “CONV_FILTER_SZ”. The convolutional filter is shifted with the default stride size of one. Besides that, the typical ReLU function is chosen as the activation function. The max-pooling layer is constituted with *max_pool_2d()*, whereby the variable “MAX_POOL_SZ” stands for the size of the receptive field. Again, the receptive field is shifted with the default stride size, which equals the size of the receptive field itself. To achieve a connection between adjacent layers, each preceding layer is a parameter of the layer-function of the succeeding one. For an extension of the network with further convolutional and pooling layers, the functions of the lines 154 and 155 have to be copied and attached to the present programming code.



```
150    # reshape the input
151    input_layer = input_data(shape=[None, image_size, image_size, 1], name='Input')
152
153    # first convolution and pooling layer pair
154    net_layers = conv_2d(input_layer, NUM_FEATURES[0], CONV_FILTER_SZ, activation='relu')
155    net_layers = max_pool_2d(net_layers, MAX_POOL_SZ)
```

Figure 46: Listings of the implemented input layer and of the first convolutional and pooling layer pair.

In addition to this, Figure 47 shows the listings of the fully connected, the dropout and the output layer. The TFLearn function *fully_connected()* supplies the fully connected layer after alternating convolutional and max-pooling ones. The variable “NUM_FULLY_NODES[0]” holds the unit number and ReLU is used for the activation of this layer type. Furthermore, *dropout()* enables the exclusion of random nodes during the training process, whereby the variable “DROPOUT” defines the percentage of remaining units during a training step. The output layer is also generated with the *fully_connected()* function. Nevertheless, the number of output neurons is set to a value of two, since there are two classes discriminated in this task. The activation of the output layer is a softmax function, which is the frequent case for classification tasks [57]. A softmax function (11) is calculated with

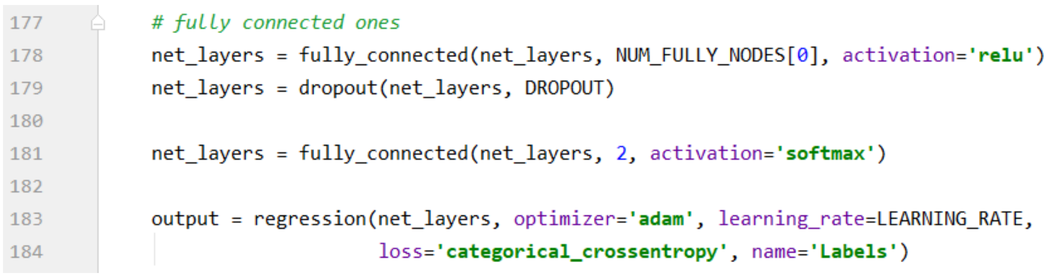
$$s(z_k) = \frac{exp(z_k)}{\sum\limits_{l=1}^{K} exp(z_l)} \tag{11}$$

where z_k denotes the output of a neuron before activation for a distinct class k, whilst the symbol K stands for the occurring classes. Despite that, z_l is the output of a neuron before activation for any class l out of K [57].

Finally, the function *regression()* is needed to define a gradient descent optimizer, the learning rate and the loss function. In this case, the default TensorFlow “adam” optimizer and the default loss function “categorical_crossentropy” were adjusted. In contrast, the learning rate is defined by the user. The cross entropy cost function (12) is defined as

$$E(parameters) = -\frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln y_{nk} \tag{12}$$

where N is the number of data entries and K is the amount of appearing classes. Moreover, t_{nk}, which stands for the ground truth label of a data entry, is multiplied with the logarithmic of y_{nk}. The formular symbol y_{nk} indicates the predicted output of the data entry [57].



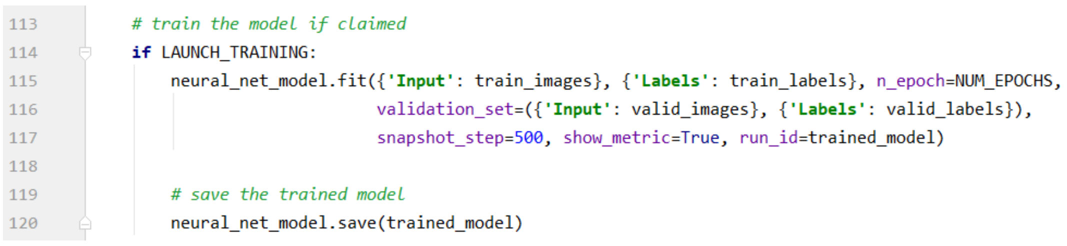
```
# fully connected ones
net_layers = fully_connected(net_layers, NUM_FULLY_NODES[0], activation='relu')
net_layers = dropout(net_layers, DROPOUT)

net_layers = fully_connected(net_layers, 2, activation='softmax')

output = regression(net_layers, optimizer='adam', learning_rate=LEARNING_RATE,
                    loss='categorical_crossentropy', name='Labels')
```

Figure 47: Listings of the fully connected, the dropout and the output layer.

Following the definition of the net topology, the CNN training can be launched. For this purpose, TFLearn provides the *fit()* method (see Figure 48). The training and the validation set are stated with Python dictionarys. Moreover, the user defines the number of epochs with the "NUM_EPOCHS" variable. The higher the number of epochs, the more often is the network run with the training data [32]. Furthermore, if the parameter "show_metric" is set to True, the user is informed about important metrics during the training process like the loss or the accuracy. A loss value of almost zero means that the predicted results are nearly the same as the ideal ones. Moreover, the accuracy states the amount of correct predicted items of a dataset. Hence, a value of one implies that all images are predicted correctly with a trained network. After accomplished training, the network weights are stored. As a result, it is possible to reload this model and to train it again.

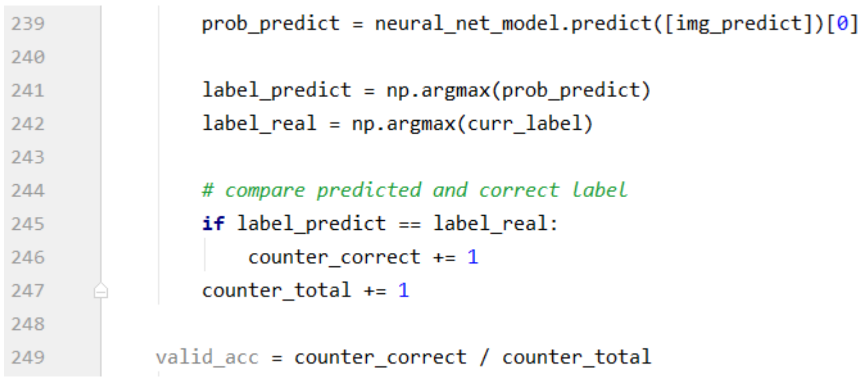


```python
# train the model if claimed
if LAUNCH_TRAINING:
    neural_net_model.fit({'Input': train_images}, {'Labels': train_labels}, n_epoch=NUM_EPOCHS,
                         validation_set=({'Input': valid_images}, {'Labels': valid_labels}),
                         snapshot_step=500, show_metric=True, run_id=trained_model)

    # save the trained model
    neural_net_model.save(trained_model)
```

Figure 48: Launching the training process of the configured neural network with the *fit()* function and thereafter, saving the trained network.

To keep track of the defined settings and achieved accuracies of various trained networks, a CSV-file with all those values is automatically stored after each launched training process. For this purpose, a function was implemented to calculate manually the accuracy of a dataset (see Figure 49). The method *predict()* forecasts the class of a delivered input according to the trained model. Hence, a vector with two entries is returned, whereby the first entry stands for the class of no appearing mandible and the second one stands for the opposite case. These two values are the class probabilities and the sum of them must be one. Thus, the class, that is more likely, has the higher probability. The NumPy *argmax()* function allows an extraction of the index of the highest value and in this way, a comparison between the predicted and the real label is possible. The accuracy is calculated with the code in line 249 of the listings displayed below.



```
        prob_predict = neural_net_model.predict([img_predict])[0]

        label_predict = np.argmax(prob_predict)
        label_real = np.argmax(curr_label)

        # compare predicted and correct label
        if label_predict == label_real:
            counter_correct += 1
        counter_total += 1

    valid_acc = counter_correct / counter_total
```

Figure 49: Manual calculation of the accuracy.

5.3.4 Training Datasets

The implemented code framework of a CNN was trained with four different sized datasets, which were exported with the generated MeVisLab network and macro module. Each dataset contains a diverse number of images, as there were different augmentation methods applied. As a result, it is possible to examine the behaviour of the network performance according to the artificially enlarged datasets. Table 3 lists these datasets and the applied data augmentation methods. The first image set involves the initial CT images, the second one is enlarged with noisy images and the third one with affine transformed images. Dataset four covers both data augmentation types.

Table 3: Datasets to train classification networks. The original images were exported and various combinations of data augmentation methods were applied.

Dataset	**Settings**	**Number of Generated Images**
1	*Original*	1680
2	*Original and Noise*	6720
3	*Original and Affine Transformations*	13440
4	*Original, Noise and Affine Transformations*	18480

Table 4 shows the declared MeVisLab settings to produce the augmented training datasets. These exportation parameters are the default values of the MeVisLab implementations. If synthetic images are generated with affine transformations, seven new images can be created per one original slice. The geometric transformations are applied separately from each other. If noise is added, three new images can be exported.

Table 4: Exportation settings in MeVisLab to create augmented images for training CNNs. The affine transformations are applied independently from each other.

Data Augmentation	**Type**	**Settings**	**Adjusted Values**	**Created Images per Slice**
Separate Affine Transf.	Flipping	-	-	1
	Rotation	Rot. Angles:	±8°	2
	Scaling	Scal. Factors:	0.96 & 1.04	4
Noise	Uniform	Amplitude:	800	1
	Gaussian	Mean:	0	1
		Std. Dev.:	300	
	Salt & Pepper	Amplitudes:	±2000	1
		Density:	0.05	

As already mentioned, the TFLearn function *fit()* requires a division into training and validation data. According to [59], the validation set (13) should have a minimum proportion of

$$Val = \frac{1}{\sqrt{2T}} \tag{13}$$

of the total available amount of training data T. Thus, the fragmentation of the four generated datasets happened the following way (Table 5):

Table 5: Subdivision into training and validation data for classification.

Dataset	**Total Number of Images**	**Minimum Size of Valid. Set**	**Actual Size of Training Dataset**	**Actual Size of Valid. Dataset**
1	1680	28.98	1650	30
2	6720	57.97	6600	120
3	13440	81.98	13300	140
4	18480	96.12	18300	180

5.3.5 Training of the CNNs

The goal of this section was to find a network, which is able to discriminate CT images according to an appearing mandible. Moreover, the metrics ought to deliver optimal values, which means that the accuracy should be high and the loss should be low.

To produce meaningful results with a trained network, it is required to figure out optimal training and network parameters. The choice of these values influence eminently the results of the net. Figure 50 gives an overview of factors that can be varied and consequently, lead to completely different results. For instance, dataset properties (red arrows in Figure 50) and also the arrangement of diverse layer types (orange) influence the behaviour of a net. Additionally, parameters concerning the training process (green) and parameters of the network layers itself (black) affect the results.

Unfortunately, there aren't any generally applicable rules to select these parameters. The seek for appropriate values is more about trying out. Neural networks are also often declared as "black magic" [31]. As a result, the aim was to find out values of the listed parameters, that deliver adequate network results [43].

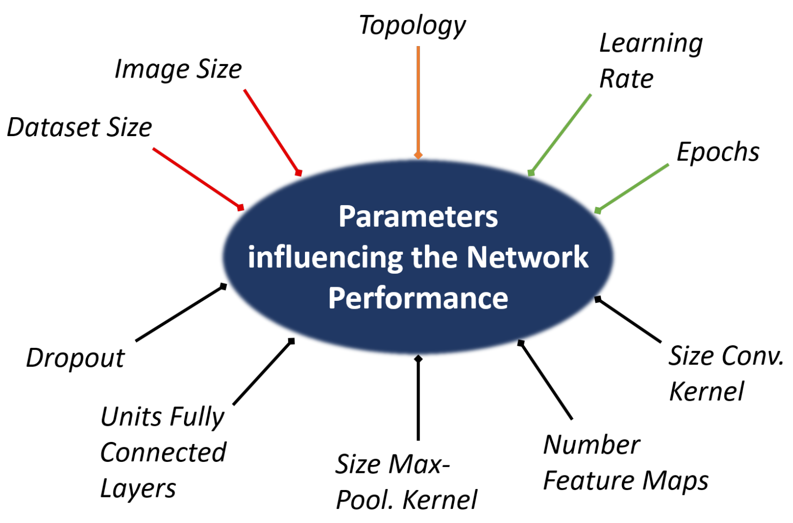
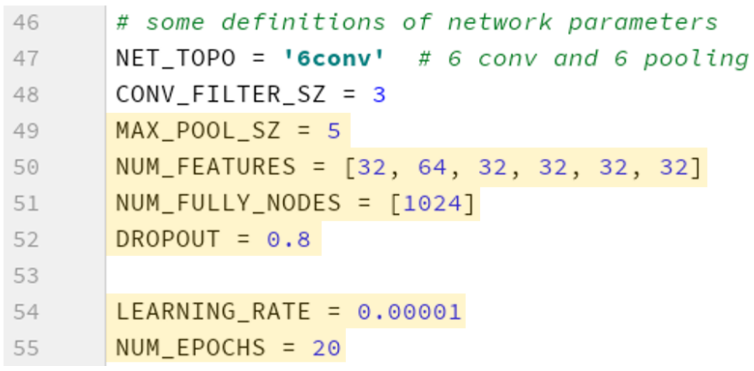


Figure 50: Parameters influencing the network performance. Dataset properties (red), the topology of the net (orange), training settings (green) and parameters of the network layers itself (black) influence a network's behaviour.

Figure 51 shows values of some of these variables that emerged to be assistant to train an appropriate classification network. Good accuracies were achieved with a max-pooling filter size of five, with 1024 nodes of the fully connected layer and feature map numbers of 32 and 64. Moreover, the dropout rate is defined with 0.8, the learning rate was set to 0.00001 and the number of epochs per launched training is stated as 20.



```
# some definitions of network parameters
NET_TOPO = '6conv'  # 6 conv and 6 pooling
CONV_FILTER_SZ = 3
MAX_POOL_SZ = 5
NUM_FEATURES = [32, 64, 32, 32, 32, 32]
NUM_FULLY_NODES = [1024]
DROPOUT = 0.8

LEARNING_RATE = 0.00001
NUM_EPOCHS = 20
```

Figure 51: Listings of defined training parameters. The values of the yellow shaded variables are stated for further examinations of deep networks.

The remaining influencing factors (dataset size, image size, topology and convolutional kernel size) were examined in more detail. Hence, networks with miscellaneous values of these parameters were trained. In total, the training was conducted four times for each configured model.

As a first investigation, the impacts of the network topology and the dataset size were surveyed. To enable a faster training, the images were down-sampled to a size of 50 $\times$ 50. Initially, a CNN was trained with three convolutional and three max-pooling layers and with the settings of Table 6. Furthermore, the network exhibited an input layer, one fully connected layer and the output layer. The learning rate and the epochs were defined with the values of Figure 51. Beyond that, this network configuration was trained with the four different sized datasets.

Table 6: Training configuration of a CNN with three convolutional and three max-pooling layers.

3 Conv. and 3 Max-pooling Layers

Dataset Size	*Image Size*	*Conv. Kernel*	*Max-Pool. Kernel*	*Feature Maps*	*Fully Nodes*	*Dropout*
1680	50	3	5	32 64 32	1024	0.8
6720	50	3	5	33 64 32	1024	0.8
13440	50	3	5	34 64 32	1024	0.8
18480	50	3	5	35 64 32	1024	0.8

Two further CNN configurations were trained with the same parameter settings as the networks of the previous example. Nevertheless, one network was composed of four convolutional and max-pooling layer pairs (see Table 7) and the second one consisted of six convolutional and six max-pooling layers (see Table 8). Again, four datasets were used as training data.

Table 7: Training configuration of a CNN with four convolutional and four max-pooling layers.

4 Conv. and 4 Max-pooling Layers

Dataset Size	*Image Size*	*Conv. Kernel*	*Max-Pool. Kernel*	*Feature Maps*	*Fully Nodes*	*Dropout*
1680	50	3	5	32 64 32 32	1024	0.8
6720	50	3	5	33 64 32 32	1024	0.8
13440	50	3	5	34 64 32 32	1024	0.8
18480	50	3	5	35 64 32 32	1024	0.8

Table 8: Training configuration of a CNN with six convolutional and six max-pooling layers.

6 Conv. and 6 Max-pooling Layers

Dataset Size	*Image Size*	*Conv. Kernel*	*Max-Pool. Kernel*	*Feature Maps*	*Fully Nodes*	*Dropout*
1680	50	3	5	32 64 32 32 32 32	1024	0.8
6720	50	3	5	33 64 32 32 32 32	1024	0.8
13440	50	3	5	34 64 32 32 32 32	1024	0.8
18480	50	3	5	35 64 32 32 32 32	1024	0.8

The previous network topologies were trained with a convolutional filter size of three. The CNN topology with six convolutional and six pooling layers was also examined with a filter size of five and seven. Table 9 lists the respective training settings.

Table 9: Training configuration of a CNN with six convolutional and six max-pooling layers and various convolutional filter sizes.

6 Conv. and 6 Max-pooling Layers

Dataset Size	*Image Size*	*Conv. Kernel*	*Max-Pool. Kernel*	*Feature Maps*	*Fully Nodes*	*Dropout*
1680	50	**5**	5	32 64 32 32 32 32	1024	0.8
6720	50	**5**	5	32 64 32 32 32 32	1024	0.8
13440	50	**5**	5	32 64 32 32 32 32	1024	0.8
18480	50	**5**	5	32 64 32 32 32 32	1024	0.8
1680	50	**7**	5	33 64 32 32 32 32	1024	0.8
6720	50	**7**	5	34 64 32 32 32 32	1024	0.8
13440	50	**7**	5	35 64 32 32 32 32	1024	0.8
18480	50	**7**	5	36 64 32 32 32 32	1024	0.8

After all, the datasets were down-sampled to a larger image size of 128×128 in order to explore the influence of a larger matrix size on the network results. Therefore, a training with the topology of six convolutional and six max-pooling layers was conducted. The convolutional filter size was again assigned as seven. Table 10 lists an overview of the settings of this training process.

Table 10: Training configuration of a CNN with six convolutional and six max-pooling layers and image sizes of 128×128.

6 Conv. and 6 Max-pooling Layers

Dataset Size	*Image Size*	*Conv. Kernel*	*Max-Pool. Kernel*	*Feature Maps*	*Fully Nodes*	*Dropout*
1680	**128**	7	5	32 64 32 32 32 32	1024	0.8
6720	**128**	7	5	33 64 32 32 32 32	1024	0.8
13440	**128**	7	5	34 64 32 32 32 32	1024	0.8
18480	**128**	7	5	35 64 32 32 32 32	1024	0.8

Apart from that, it was not feasible to train the delineated network configurations with the initial image size of 512×512. The reason for this circumstance is a run out of memory of the GPU while allocating the TensorFlow tensors with the larger image size.

5.3.6 Testing of the trained CNNs

After training the various network configurations, the nets had to be tested with a new dataset. Therefore, novel images were generated with the MeVisLab macro module *SaveAsSingleSlices*, whereby the chosen exportation parameters can be seen in Table 11. The test dataset includes noisy images (Gaussian as well as Salt and Pepper) and geometric transformed ones, but not the initial CT images. The selected affine transformation types were combined, but the scaling was symmetrically applied. As a result, six images were generated per one original slice. In total, 10080 test images were created with MeVisLab.

Table 11: Exportation settings in MeVisLab to create augmented images for testing CNNs. The affine transformations were combined. Thus, flipping, rotation and scaling are simultaneously applied.

Data Augmentation	Type	Settings	Adjusted Values	Created Images per Slice
Combined Affine Transf.	Flipping	-	-	4
	Rotation	Rot. Angles:	±6°	
	Scaling	Scal. Factors:	0.97 & 1.03	
Noise	Gaussian	Mean:	0	1
		Std. Dev.:	200	
	Salt & Pepper	Amplitudes:	±1000	1
		Density:	0.05	

Comparing all trained models according to their achieved loss values and accuracies leads to the conclusion that the network with the topology of six convolutional and six max-pooling layers as well as the filter size of seven provides the best results. As a consequence, the model, which was trained with the 50×50 sized images, was utilised for the further image segmentation. A detailed delineation of the results can be found in chapter 6.2.

5.4 Segmentation of the Mandibula

In the succeeding sections is the implementation and also the training and testing of the segmentation networks described. The programming of the deep networks was conducted with TensorFlow and its API TF-Slim.

Beyond that, the realised segmentation method follows the upsampling principle presented by J. Long et al. [38] in their "Fully Convolutional Networks for Semantic Segmentation" contribution. The implementation of the code framework and also the present outline rely on the TensorFlow [26] and the TF-Slim [21] documentation as well as on the GitHub tutorial "TF Image Segmentation: Image Segmentation framework" [45], [46]. Nevertheless, the programming code of the tutorial was adapted according to specific requirements of this thesis.

5.4.1 Adaption of the Code Framework

As already outlined, J. Long et al. [38] recommended a 3-step training principle of a FCN. Figure 52 illustrates the workflow of the model implementations. The first segmentation network is the FCN-32s, which is an adjustment of the VGG-16 net. The VGG-16 model is originally trained for a classification with the ImageNet dataset of the ILSVRC 2014 challenge [13], whereby the weights for FCN-32s initialisation were provided by the tutorial [45], [46]. Moreover, the second segmentation net is the FCN-16s, which assumes weights of the trained FCN-32s model. The third and last one is the FCN-8s, which is again initialised with the weights of the previous network.

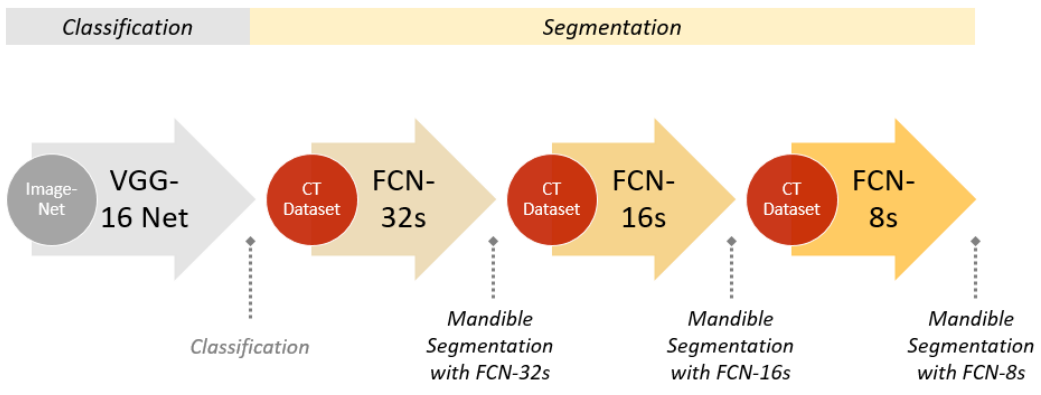


Figure 52: Workflow of the segmentation network implementations. The classification part was provided by the TF-Slim library, whereas the segmentation part was trained with the CT datasets during this work.

FCN-32s

The first segmentation network, termed as FCN-32s, was implemented within the Python script *Training_Segmentation_Network.py.* To create the FCN-32s net, the VGG-16 model definition is required. The TF-Slim library already offers the implemented VGG-16 layers in order to use them for fully convolutional networks.

Figure 53 shows a graphical interpretation of the VGG-16 architecture supplied by the TF-Slim library, whereby the layers Fc6, Fc7 and Fc8 are implemented as convolutional ones. Originally, these three layers are fully connected ones.

Moreover, the red box in Figure 53 indicates an image with a size of a$\times$a, which is provided as an input to the VGG-16 model. The blue boxes stand for the convolutional layers, whereby two or three convolutional layers are stacked. The filter sizes of the convolutional layers, that are illustrated by the blue coloured boxes, have a value of three. Despite that, the blue ellipses indicate the max-pooling layers, which hold receptive fields of 2×2. Thus, the feature maps' side lengths are halved after each max-pooling step. In the end, the matrices of the feature maps have a side length of 1/32 of the original input image. This is the reason why the first segmentation network is known as the FCN-32s. As a result, to gain a segmentation with the initial image size, the down-sampled feature maps have to be upsampled with a factor of 32.

Apart from that, the grey blocks in Figure 53 constitute the convolutional layers that replace the original fully connected ones. The Fc6 layer exhibits a convolutional filter size of seven, whilst the other two layers have a filter size of one.

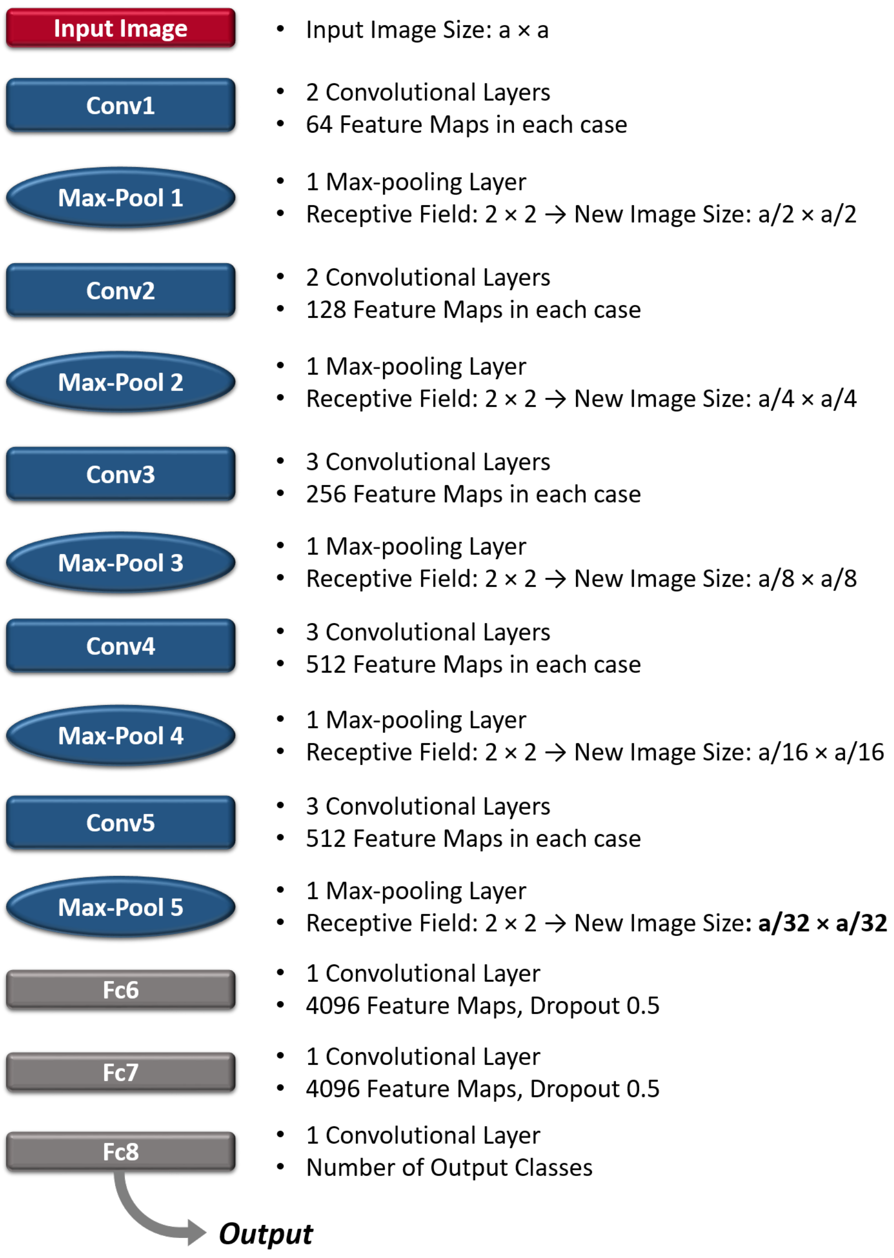
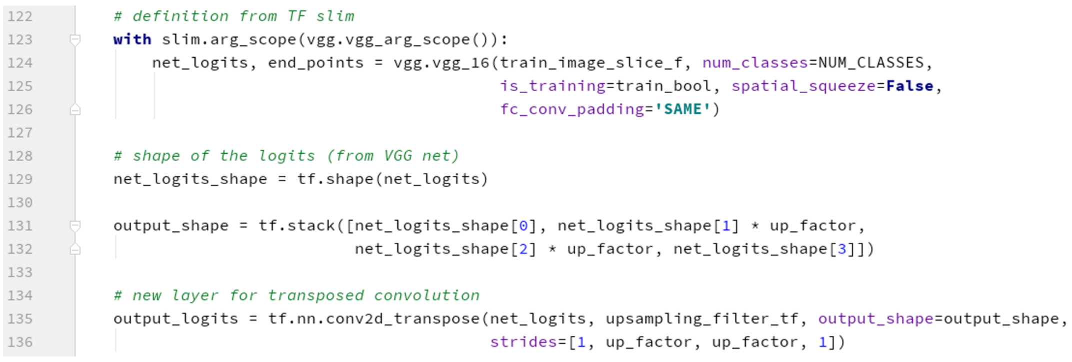

Input Image
Input Image Size: a × a
Conv1
2 Convolutional Layers
64 Feature Maps in each case
Max-Pool 1
1 Max-pooling Layer
Receptive Field: 2 × 2 → New Image Size: a/2 × a/2
Conv2
2 Convolutional Layers
128 Feature Maps in each case
Max-Pool 2
1 Max-pooling Layer
Receptive Field: 2 × 2 → New Image Size: a/4 × a/4
Conv3
3 Convolutional Layers
256 Feature Maps in each case
Max-Pool 3
1 Max-pooling Layer
Receptive Field: 2 × 2 → New Image Size: a/8 × a/8
Conv4
3 Convolutional Layers
512 Feature Maps in each case
Max-Pool 4
1 Max-pooling Layer
Receptive Field: 2 × 2 → New Image Size: a/16 × a/16
Conv5
3 Convolutional Layers
512 Feature Maps in each case
Max-Pool 5
1 Max-pooling Layer
Receptive Field: 2 × 2 → New Image Size: a/32 × a/32
Fc6
1 Convolutional Layer
4096 Feature Maps, Dropout 0.5
Fc7
1 Convolutional Layer
4096 Feature Maps, Dropout 0.5
Fc8
1 Convolutional Layer
Number of Output Classes
Output

Figure 53: Network architecture of the TF-Slim VGG-16 model.

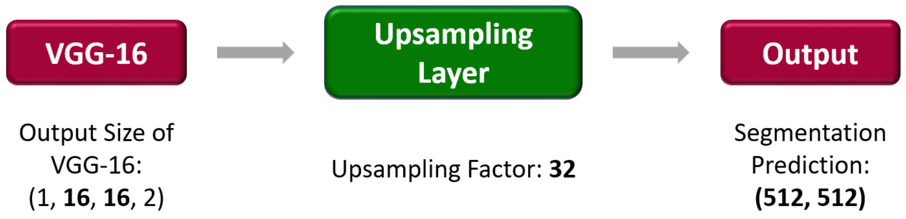
On top of that, Figure 54 shows the extension of the VGG-16 architecture and consequently, the generation of the FCN-32s network. The code lines from 124 to 126 import the described VGG-16 model from TF-Slim, whereby the variable "net_logits" holds the output for an input image “train_image_slice_f”. The number of classes is two, since the voxels are distinguished if they are part of the mandible or not. The upsampling of the VGG-16 output is accomplished with the TensorFlow function *conv2d_transpose()* (code line 135). This function requires the output of the VGG-16 network, the upsampling filter, the size of the final output and the filter strides as parameters. The upsampling filter is implemented as a bilinear filter, which is shifted with the upsampling factor of 32. To get the demanded final output size, the bilinear filter must have a size of the product of two times the upsampling factor, which can be imagined with Figure 25.



```
122   # definition from TF slim
123   with slim.arg_scope(vgg.vgg_arg_scope()):
124       net_logits, end_points = vgg.vgg_16(train_image_slice_f, num_classes=NUM_CLASSES,
125                                           is_training=train_bool, spatial_squeeze=False,
126                                           fc_conv_padding='SAME')
127
128   # shape of the logits (from VGG net)
129   net_logits_shape = tf.shape(net_logits)
130
131   output_shape = tf.stack([net_logits_shape[0], net_logits_shape[1] * up_factor,
132                            net_logits_shape[2] * up_factor, net_logits_shape[3]])
133
134   # new layer for transposed convolution
135   output_logits = tf.nn.conv2d_transpose(net_logits, upsampling_filter_tf, output_shape=output_shape,
136                                          strides=[1, up_factor, up_factor, 1])
```

Figure 54: Listings of the FCN-32s network topology.

Furthermore, Figure 55 depicts the FCN-32s net topology assuming that the input images have a size of 512×512. The output of the VGG-16 model has a size of (1, 16, 16, 2), where the first numeral indicates the batch size, which means that one image is used for the computations of a training step. The last value stands for the two voxel classes and the remaining numerals imply that feature maps with side lengths of 16×16 are produced. Thus, an upsampling with a factor of 32 is necessary to gain a segmentation of the initial image size. The upsampling is conducted with one additional layer (green).



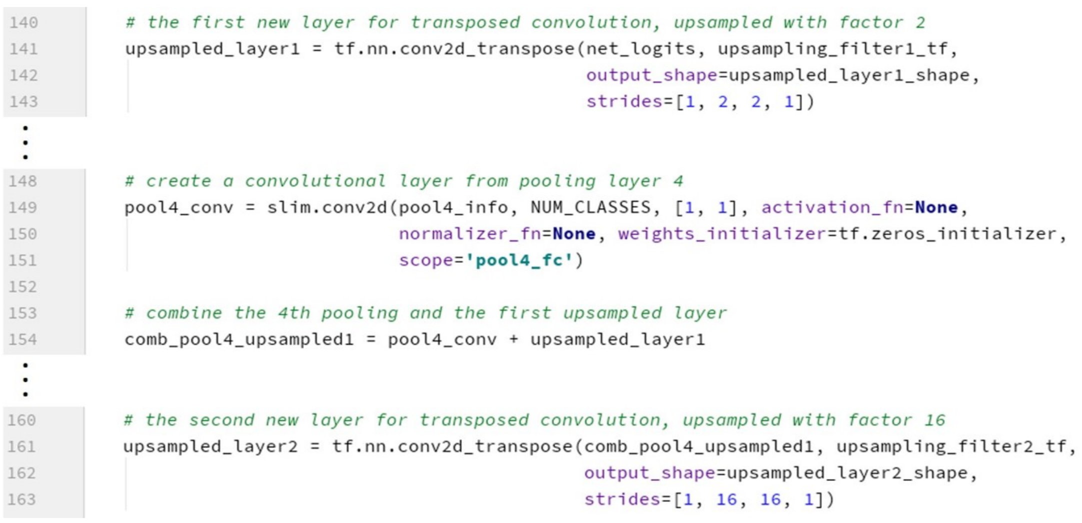
VGG-16
Upsampling Layer
Output
Output Size of VGG-16: (1, **16**, **16**, 2)
Upsampling Factor: **32**
Segmentation Prediction: **(512, 512)**

Figure 55: FCN-32s network topology for input images with size of 512×512.

FCN-16s

The FCN-16s network was implemented within the Python-file *Training_Segmentation_Network_Step2.py.* The main programming steps, that implement the network topology, are outlined in Figure 56. Despite to the previous introduced net, the FCN-16s achieves upsampling in two steps. Therefore, the function *conv2d_transpose()* (code line 141) accomplishes the first upsampling with a factor of two. Again, the down-sampled output of the VGG-16 is provided as an input to the layer function.

Additionally, the output of the max-pooling layer four is involved in the upsampling process. For this purpose, a convolutional layer is created with the TF-Slim function *conv2d()*, which receives the result of the max-pooling layer four. The convolutional filter has a size of one and the amount of created feature maps equals two for the appearing classes of this task. According to [38], the additional convolutional layer is added to determine voxel predictions after the fourth pooling layer. This outlined information is combined with the output of the first upsampling layer (code line 154). The second and last upsampling step is conducted with the implementations from lines 161 to 163, whereby a factor of 16 is necessary to reach the initial image size for the final output.



```
140  # the first new layer for transposed convolution, upsampled with factor 2
141  upsampled_layer1 = tf.nn.conv2d_transpose(net_logits, upsampling_filter1_tf,
142                                            output_shape=upsampled_layer1_shape,
143                                            strides=[1, 2, 2, 1])
 ⋮
148  # create a convolutional layer from pooling layer 4
149  pool4_conv = slim.conv2d(pool4_info, NUM_CLASSES, [1, 1], activation_fn=None,
150                           normalizer_fn=None, weights_initializer=tf.zeros_initializer,
151                           scope='pool4_fc')
152
153  # combine the 4th pooling and the first upsampled layer
154  comb_pool4_upsampled1 = pool4_conv + upsampled_layer1
 ⋮
160  # the second new layer for transposed convolution, upsampled with factor 16
161  upsampled_layer2 = tf.nn.conv2d_transpose(comb_pool4_upsampled1, upsampling_filter2_tf,
162                                            output_shape=upsampled_layer2_shape,
163                                            strides=[1, 16, 16, 1])
```

Figure 56: Listings of the FCN-16s network topology.

Figure 57 depicts the schematic of the implemented FCN-16s topology for input images with a size of 512 × 512. The output of the first upsampling layer produces feature maps with a size of (1, 32, 32, 2). The fourth max-pooling layer of the VGG-16 network creates an output of the same size. Hence, a combination of those two layers is feasible. The convolutional layer, which was appended to the fourth max-pooling layer for upsampling (blue box), is not depicted for reasons of clarity. The result of the combination is supplied to the second upsampling step. To create the final segmentation prediction, an upsampling factor of 16 is indicated.

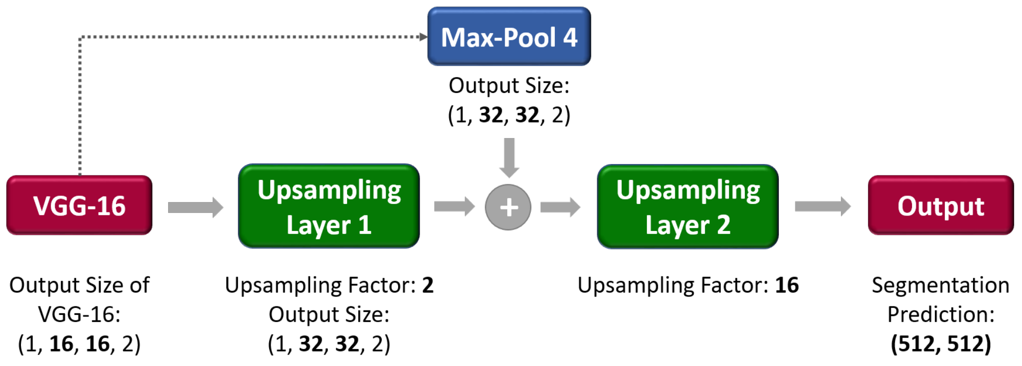
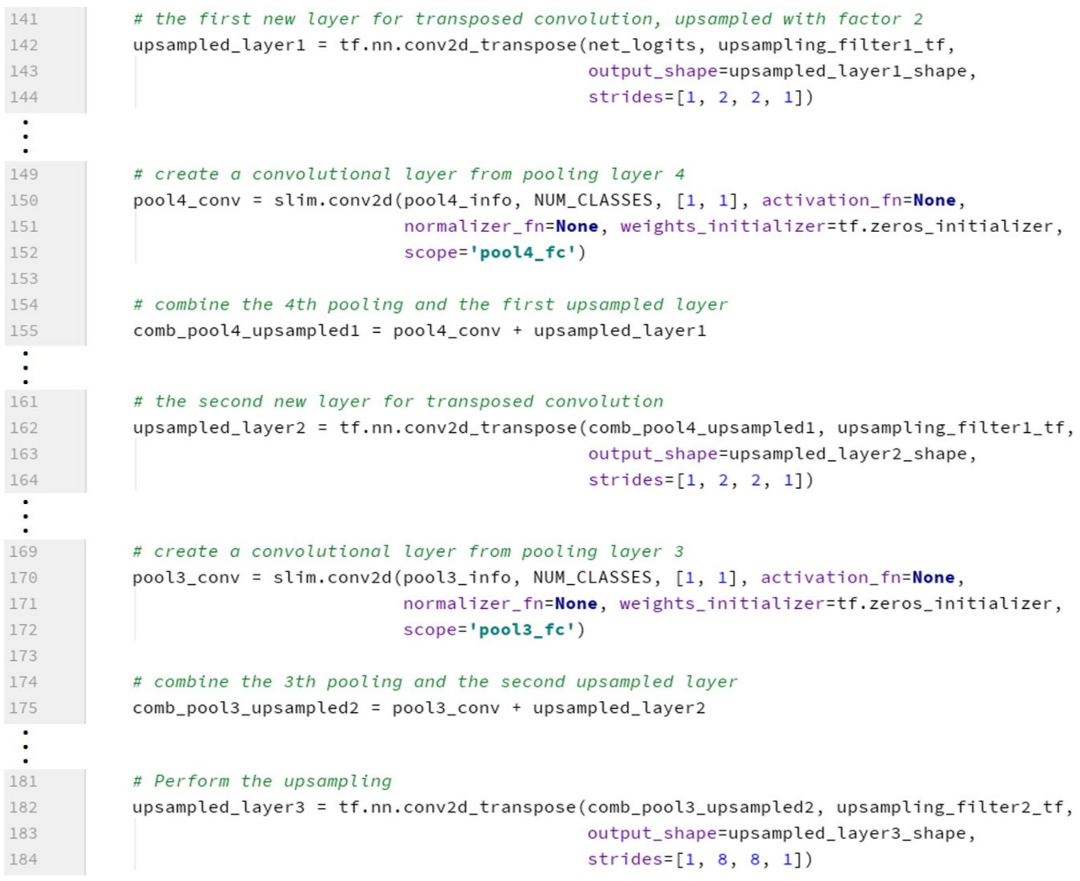

Max-Pool 4
Output Size: (1, **32**, **32**, 2)
VGG-16
Upsampling Layer 1
+
Upsampling Layer 2
Output
Output Size of VGG-16: (1, **16**, **16**, 2)
Upsampling Factor: **2**
Output Size: (1, **32**, **32**, 2)
Upsampling Factor: **16**
Segmentation Prediction: **(512, 512)**

Figure 57: FCN-16s network topology for input images with size of 512 × 512.

<u>FCN-8s</u>

The final segmentation network is the FCN-8s, which is initialised with trained weights of the FCN-16s model. This network was implemented within the Python-file *Training_Segmentation_Network_Step3.py*, whereby the upsampling is performed with three steps.

Figure 58 introduces the programming code of the FCN-8s net topology. The first upsampling is conducted in the same manner as it was achieved with the FCN-16s network. Hence, the output of the VGG-16 model is delivered as an input to the first upsampling layer (code line 142), which executes a resize with a factor of two. Moreover, the involvement of the fourth max-pooling layer (code lines 150 and 155) is achieved in the same way as it was done for the FCN-16s network. The resize factor of the second upsampling layer exhibits, however, a value of two. Finally, the third upsampling layer involves the output of the third max-pooling layer of the VGG-16 network. Therefore, a convolutional layer (code line 170) is appended to the max-pooling layer three in order to provide the voxel predictions. The implementations of line 175 combine the information of the third max-pooling layer and the output of the second upsampling step. To gain the original matrix size for the final segmentation prediction, an upsampling factor of eight is essential. The last upsampling layer is generated in line 182 with the TensorFlow function *conv2d_transpose()*.



```
141     # the first new layer for transposed convolution, upsampled with factor 2
142     upsampled_layer1 = tf.nn.conv2d_transpose(net_logits, upsampling_filter1_tf,
143                                               output_shape=upsampled_layer1_shape,
144                                               strides=[1, 2, 2, 1])
 :
149     # create a convolutional layer from pooling layer 4
150     pool4_conv = slim.conv2d(pool4_info, NUM_CLASSES, [1, 1], activation_fn=None,
151                              normalizer_fn=None, weights_initializer=tf.zeros_initializer,
152                              scope='pool4_fc')
153
154     # combine the 4th pooling and the first upsampled layer
155     comb_pool4_upsampled1 = pool4_conv + upsampled_layer1
 :
161     # the second new layer for transposed convolution
162     upsampled_layer2 = tf.nn.conv2d_transpose(comb_pool4_upsampled1, upsampling_filter1_tf,
163                                               output_shape=upsampled_layer2_shape,
164                                               strides=[1, 2, 2, 1])
 :
169     # create a convolutional layer from pooling layer 3
170     pool3_conv = slim.conv2d(pool3_info, NUM_CLASSES, [1, 1], activation_fn=None,
171                              normalizer_fn=None, weights_initializer=tf.zeros_initializer,
172                              scope='pool3_fc')
173
174     # combine the 3th pooling and the second upsampled layer
175     comb_pool3_upsampled2 = pool3_conv + upsampled_layer2
 :
181     # Perform the upsampling
182     upsampled_layer3 = tf.nn.conv2d_transpose(comb_pool3_upsampled2, upsampling_filter2_tf,
183                                               output_shape=upsampled_layer3_shape,
184                                               strides=[1, 8, 8, 1])
```

Figure 58: Listings of the FCN-8s network topology.

Furthermore, Figure 59 shows a visualisation of the implemented FCN-8s topology. The blue rectangles illustrate the max-pooling layers that are utilised for upsampling. Nevertheless, the convolutional layers, which are added at the max-pooling ones, are not displayed. The first upsampling layer generates feature maps with a size of (1, 32, 32, 2), which allows a combination with the fourth max-pooling layer. Besides that, the second upsampling is conducted with a factor of two. Consequently, an addition of the second upsampling output and the third max-pooling layer is possible. The final upsampling is achieved with a factor of eight.

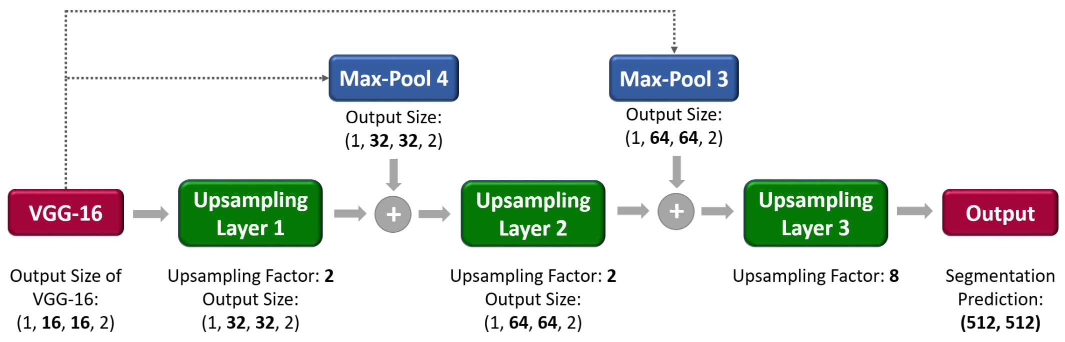


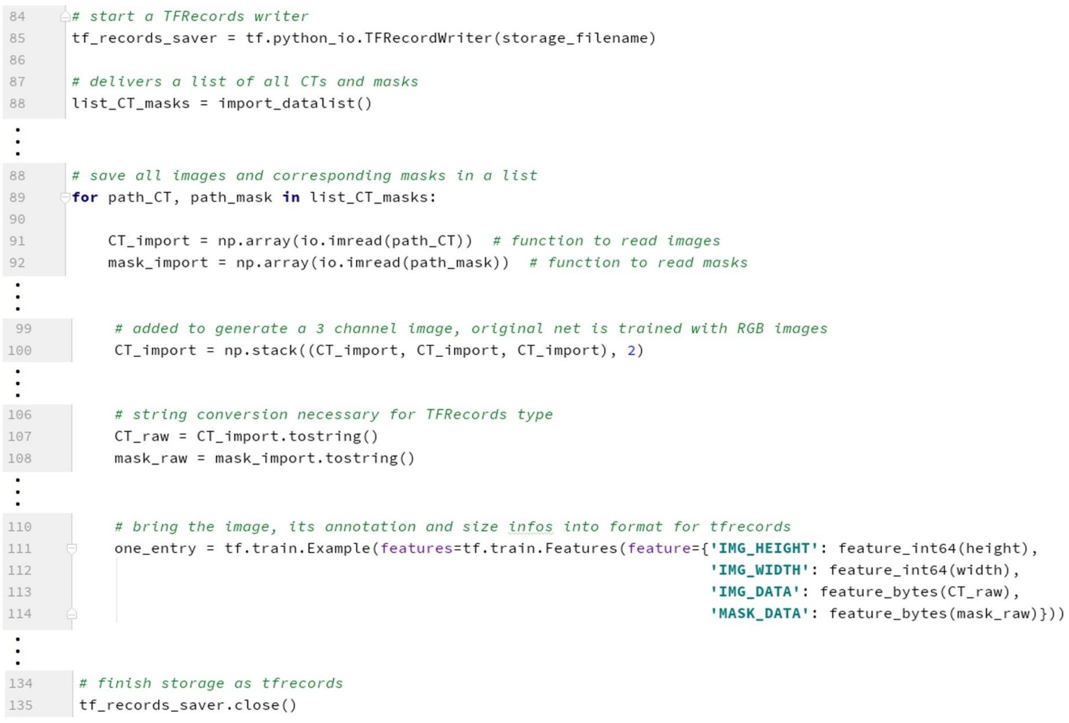
Figure 59: FCN-8s network topology for input images with size of 512×512.

5.4.2 Storage of the Datasets as TFRecords Files

TensorFlow and TF-Slim are not able to process the images with their stored PNG file formats. Hence, it is necessary to convert the CT slices and also the masks into file types that can be utilised with TensorFlow. For the classification task of this thesis, all images of a dataset were converted into NumPy arrays and stored as one NPY-file.

A further method is to save the image data in a TFRecords file. TensorFlow promotes this specific file format to address problems that might appear with large amounts of training data. For instance, a separate importation of every image and the corresponding mask during one iteration step is time-consuming. Hence, TensorFlow introduced the TFRecords format in order to load the data efficiently and to embed the data importation in the computational graph. Therefore, all images and masks are stored within one TFRecords file. Another feature is that TensorFlow is able to batch the images from this file type. This means that a defined number of images are used for one training iteration, whereby the images are selected randomly. Moreover, it is important to mention that the information about image dimensions is lost if the training data is stored as such a binary file. Thus, the width and height must be stored within the TFRecords file for every data entry [45], [26].

To accomplish the conversion of the mandible images and masks into the TFRecords format, the Python file *SaveAsTFRecords.py* was constituted. Figure 60 presents the essential parts of this script. Initially, a TensorFlow TFRecordWriter has to be launched (line 85). The implemented function *import_datalist()* stores automatically the filenames of the CT slices and the names of their corresponding masks in a Python list. As a result, it is possible to import the matching training images and masks with the functions of the lines 91 and 92. It should be noticed that the CT slices have originally one channel as they are grey-value images. The VGG-16 network, however, is trained with images featuring three color channels. Thus, the CT slices were converted into images containing three channels, too. Therefore, the original grey value channel was copied twice (line 100). Additionally, the image data has to be converted into string format to enable a storage with the binary file (lines 107 and 108). Finally, one entry of the image data is generated with the TensorFlow function of line 111. A TFRecords entry involves the CT slice and the mask saved as strings as well as the height and width of the images. To finish the TFRecords generation, the TFRecordWriter has to be closed (line 135). Apart from that, the implemented Python file permits a down-sampling of the training data.



```
84   # start a TFRecords writer
85   tf_records_saver = tf.python_io.TFRecordWriter(storage_filename)
86
87   # delivers a list of all CTs and masks
88   list_CT_masks = import_datalist()
⋮
88   # save all images and corresponding masks in a list
89   for path_CT, path_mask in list_CT_masks:
90
91       CT_import = np.array(io.imread(path_CT))  # function to read images
92       mask_import = np.array(io.imread(path_mask))  # function to read masks
⋮
 99       # added to generate a 3 channel image, original net is trained with RGB images
100       CT_import = np.stack((CT_import, CT_import, CT_import), 2)
⋮
106       # string conversion necessary for TFRecords type
107       CT_raw = CT_import.tostring()
108       mask_raw = mask_import.tostring()
⋮
110       # bring the image, its annotation and size infos into format for tfrecords
111       one_entry = tf.train.Example(features=tf.train.Features(feature={'IMG_HEIGHT': feature_int64(height),
112                                                                       'IMG_WIDTH': feature_int64(width),
113                                                                       'IMG_DATA': feature_bytes(CT_raw),
114                                                                       'MASK_DATA': feature_bytes(mask_raw)}))
⋮
134  # finish storage as tfrecords
135  tf_records_saver.close()
```

Figure 60: Listings of the generation of a TFRecords file.

5.4.3 Training Datasets

During this thesis, the segmentation networks were trained with four different datasets, which were exported with the MeVisLab implementations and converted into the TFRecords format with the previous presented *SaveAsTFRecords.py* script. Table 12 lists the generated training datasets. Two of the training sets (I and II) contain the original images, whereby the images of the first dataset are down-sampled to a size of 256 × 256. The other two datasets (III and IV) cover the original images and also artificially generated ones. Again, one dataset comprises the original sized images, while the other one contains down-sampled CT slices. It has to be noticed that only slices, which show parts of the lower jawbone, were used to train the segmentation networks. Hence, the number of available training images reduces compared to the training data of the classification networks. The extraction of the slices, that don't comprise the mandible, was executed manually.

Table 12: Datasets to train segmentation networks. The original images were exported and augmented datasets were generated.

Dataset	**Settings**	**Image Sizes**	**Number of Generated Images**
I	*Original*	256	702
II	*Original*	512	702
III	*Original, Noise and Affine Transformations*	256	4212
IV	*Original, Noise and Affine Transformations*	512	4212

On top of that, Table 13 lists the MeVisLab settings for the exportation of the augmented image slices. The scaling factors 0.96 and 1.04 were applied in different directions. Thus, the original slices were scaled with 0.96 in x- and 1.04 in y-direction and vice versa. In total, five synthetic images were generated per one original CT slice.

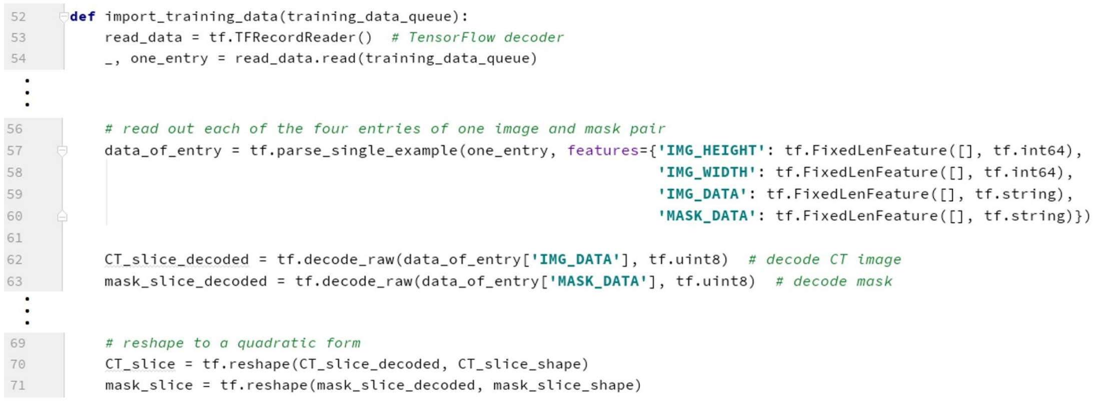
Table 13: Exportation settings in MeVisLab to create augmented images for training segmentation networks. The affine transformations are applied independently from each other.

Data Augmentation	**Type**	**Settings**	**Adjusted Values**	**Created Images per Slice**
Separate	Flipping	-	-	1
Affine	Rotation	Rot. Angle:	8°	1
Transf.	Scaling	Scal. Factors:	0.96 & 1.04	2
Noise	Salt & Pepper	Ampl.: Density:	±2000 0.05	1

5.4.4 Training of the Networks

The training of the three network architectures was accomplished within the *Training_Segmentation_Network.py* (FCN-32s), the *Training_Segmentation_Network_Step2.py* (FCN-16s) and the *Training_Segmentation_Network_Step3.py* (FCN-8s), which comprise the introduced network definitions of preceding chapter 5.4.1. Nevertheless, the training by itself was conducted similarly according to the three different network topologies. Thus, the following instructions pertain to all segmentation implementations of this work.

As a first step of the training process, the data, which was stored as a TFRecords file, was imported into the training script. Therefore, the *import_training_data()* function was implemented to decode the images and masks (see Figure 61). TensorFlow provides the *TFRecordReader()* (line 53) and the *decode_raw()* functions (lines 62 and 63) to convert the strings into numerals. The numerals are, however, arranged in a “line”. Thus, a reshape into square images was conducted with the functions of the lines 70 and 71.



```python
def import_training_data(training_data_queue):
    read_data = tf.TFRecordReader()  # TensorFlow decoder
    _, one_entry = read_data.read(training_data_queue)
⋮
    # read out each of the four entries of one image and mask pair
    data_of_entry = tf.parse_single_example(one_entry, features={'IMG_HEIGHT': tf.FixedLenFeature([], tf.int64),
                                                                 'IMG_WIDTH': tf.FixedLenFeature([], tf.int64),
                                                                 'IMG_DATA': tf.FixedLenFeature([], tf.string),
                                                                 'MASK_DATA': tf.FixedLenFeature([], tf.string)})

    CT_slice_decoded = tf.decode_raw(data_of_entry['IMG_DATA'], tf.uint8)  # decode CT image
    mask_slice_decoded = tf.decode_raw(data_of_entry['MASK_DATA'], tf.uint8)  # decode mask
⋮
    # reshape to a quadratic form
    CT_slice = tf.reshape(CT_slice_decoded, CT_slice_shape)
    mask_slice = tf.reshape(mask_slice_decoded, mask_slice_shape)
```

Figure 61: Listings of the decoding of a TFRecords file.

Apart from that, TensorFlow offers the function *shuffle_batch()* to ensure that a random image mask pair is used for the calculations of every training iteration. The number of images, that are utilised for one iteration, is also termed as the batch size. In this case, one image mask pair is used for the computations of the weight updates. On top of that, the number of epochs was set to ten for the three implemented networks, whereas the learning rate changed with the various topologies. The learning rate of the FCN-32s net had a value of 0.0001, while the rates of the FCN-16s and FCN-8s networks were set to 0.000001 and 0.0000001. The decrease of these values was conducted with the same factors as it was defined in the original experiments by J. Long et al. [38].

Moreover, the cross entropy was calculated for the minimisation task. TensorFlow supplies the *softmax_cross_entropy_with_logits()* function, which receives the ground truth and the upsampled prediction as input parameters (see line 267 in Figure 62). The variable "logits_reshaped" comprises the output of the network topology (FCN-32s, FCN-16s or FCN-8s), whilst the variable "labels_reshaped" holds the manual segmentation. The *reduce_sum()* function (line 269) defines the calculation of the final result of the cross entropy. Furthermore, the minimisation is achieved with TensorFlow's Adam Optimizer, whereby "Adam" stands for denotation of the optimisation algorithm. The value, which should be minimised with this optimiser, is the "cross_entropy_sum". Despite to the TFLearn library, which was used for the classification task, TensorFlow requires a manual retention of data for the toolkit TensorBoard. Therefore, the function of line 252 permits the storage of scalar values for TensorBoard illustrations. During this work, the computed cross entropy is stored for every iteration.

```
# calculate cross entropy
cross_entropies = tf.nn.softmax_cross_entropy_with_logits(logits=logits_reshaped,
                                                          labels=labels_reshaped)
cross_entropy_sum = tf.reduce_sum(cross_entropies)
⋮
# the minimisation step
with tf.variable_scope("adam_vars"):
    train_step = tf.train.AdamOptimizer(learning_rate=LEARNING_RATE).minimize(cross_entropy_sum)
⋮
# storage of data for TensorBoard
tf.summary.scalar('cross_entropy_loss', cross_entropy_sum)
```

Figure 62: Listings of the cross entropy calculation.

Before a TensorFlow session could be started, the trained weights of the previous network were loaded with the TF-Slim function *assign_from_checkpoint_fn()*. The values of the network weights are stored in a so-called checkpoint-file. In the case of the listings of Figure 63, the values of the VGG-16 net were restored for the FCN-32s configuration. The TensorFlow session for network training is stated in line 279 , whereby the number of iteration steps depends on the product of the epochs times the number of training images. After accomplished training, the weights were stored in a new checkpoint-file with the support of a TensorFlow saver (line 262 and 296).

```
245  # initialise variables from trained model (checkpoint), but not the last layer
246  initialisation_checkp = slim.assign_from_checkpoint_fn(model_path=checkpoint_vgg16,
247                                                         var_list=vgg_16_variables_excl_fc8)
 ⋮
260  # the saver and restorer of the model
261  model_variables = slim.get_model_variables()
262  model_saver = tf.train.Saver(model_variables)
 ⋮
276      # iterations depending on epochs
277      for counter_it in range(NUM_IMAGES * EPOCHS):
278          # calculation of cross entropy, save for tensorboard and do the training
279          cross_entropy, summary_string, _ = sess.run([cross_entropy_sum,
280                                                       merged_summary_op,
281                                                       train_step])
 ⋮
295      # final storage of the model
296      final_storage_path = model_saver.save(sess, final_checkpoint)
```

Figure 63: Listings of the weights recovery and storage of new trained weights.

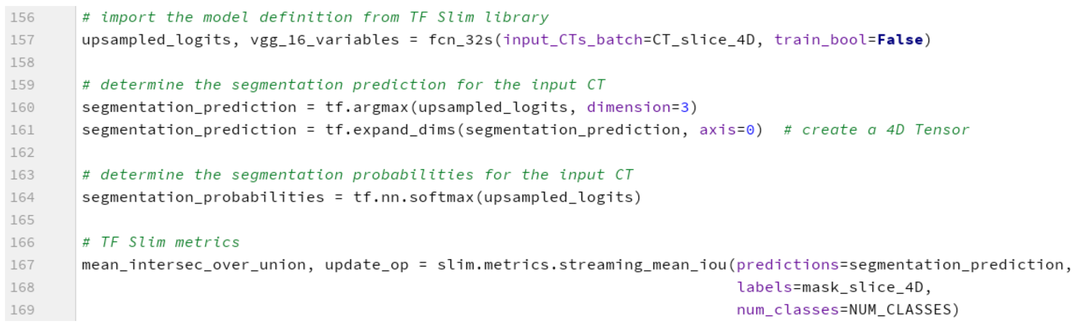
The segmentation networks were trained on the CUDA04 Server delivered by the TU Graz for datasets comprising images with a size of 512×512. The training with the down-sampled images was conducted on a PC providing a Intel-Core i7-6700 CPU (3.40 GHz) and 8 GB RAM. However, as there was such a small amount of data available, it was also achievable to train the segmentation networks on a CPU. The consecutive training of the FCN-32s, the FCN-16s and the FCN-8s models took in total about one day and a half for the smaller sized datasets (I and II), while training with the datasets III and IV lasted about five days. The stated values are estimations, because the times were not exactly stopped.

5.4.5 Testing of the Networks

The three trained segmentation networks were tested with the implemented Python files *Test_Segmentation_Network.py* (FCN-32s), *Test_Segmentation_Network_Step2.py* (FCN-16s) and *Test_Segmentation_Network_Step3.py* (FCN-8s). The implementations of these scripts are similar with the exception of the topology definitions.

During testing of the networks, the image data was converted into the TFRecords format with the introduced script of section 5.4.2. Furthermore, the network topology was defined in the same way as it was realised within the training scripts. Thus, an outline of the network structure is not conducted once more.

Apart from the training implementations, the testing scripts calculate the mean intersection over union between predicted segmentations and the ground truths of a whole test dataset (see Figure 64). Therefore, the output is generated with the implemented network structure (line 157), whereby the function *fcn_32s()* defines the net topology and the variable "upsampled_logits" holds the forecasted segmentation. It is important to set the "train_bool" variable of this function to false in order to prevent the launch of a new training. TensorFlow's *argmax()* function returns the class labels for each voxel. If the case "background" is more likely for a voxel, zero is returned. Otherwise, if the voxel is presumably part of the mandible, the function returns one. The class probabilities are directly computed with the implemented code in line 164. Furthermore, the *expand_dims()* function extends the dimension of the predicted network output. This has to be done in order to allow the computation of the mean IoU with the TF-Slim *straming_mean_iou()* function in line 167.



```python
# import the model definition from TF Slim library
upsampled_logits, vgg_16_variables = fcn_32s(input_CTs_batch=CT_slice_4D, train_bool=False)

# determine the segmentation prediction for the input CT
segmentation_prediction = tf.argmax(upsampled_logits, dimension=3)
segmentation_prediction = tf.expand_dims(segmentation_prediction, axis=0)  # create a 4D Tensor

# determine the segmentation probabilities for the input CT
segmentation_probabilities = tf.nn.softmax(upsampled_logits)

# TF Slim metrics
mean_intersec_over_union, update_op = slim.metrics.streaming_mean_iou(predictions=segmentation_prediction,
                                                                    labels=mask_slice_4D,
                                                                    num_classes=NUM_CLASSES)
```

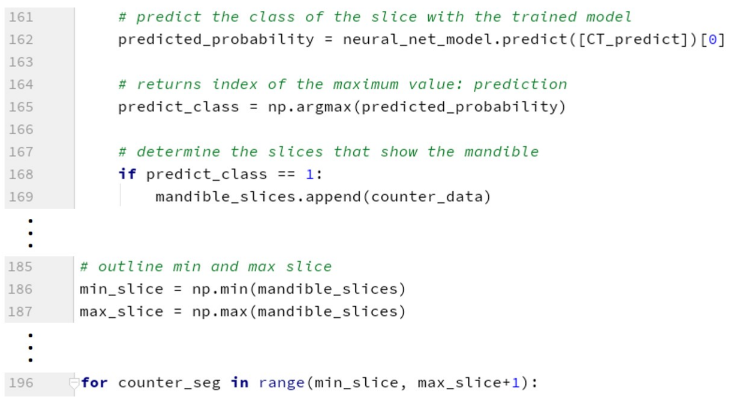
Figure 64: Listings of the segmentation prediction and calculation of the mean IoU of a test dataset.

In addition to this, the predicted segmentations might be stored automatically as PNG-files if this is desired by the user. Moreover, the concrete calculations are conducted with launching the TensorFlow sessions.

5.4.6 Combination of the Segmentation and Classification Networks

To bring to mind, the classification CNNs were trained to distinguish if an image slice contains parts of the lower jawbone or not. The segmentation networks were just trained with images that comprise the mandible. To join the two networks, the *First_Step_Testing.py* script was implemented. It has to be noticed that following investigations were always deployed on a patient's dataset comprising all slices of the head-neck region.

Firstly, the CT images were imported and down-sampled to a size of 50×50 to ensure a prediction with the classification CNNs. As a next step, the classification topology, which emerged to deliver the best results, and also the trained weights were loaded. The "neural_net_model" of line 162 in Figure 65 holds the classification network and the TFLearn method *predict()* forecasts the label of an image slice. The returned "predicted_probability" contains the likelihoods of the two classes, whereby NumPy's *argmax()* function determines the absolute labels zero or one. If a slice is showing the mandible, the slice number is appended to a list. After testing all slices of a dataset, the minimum and the maximum slice displaying the Mandibula are established from the stored list ("min_slice" and "max_slice"). These two slices build the limitation of the images that are converted into a TFRecords file for the succeeding segmentations (for-loop of line 196). The segmentation is achieved with the scripts of section 5.4.5.



```python
    # predict the class of the slice with the trained model
    predicted_probability = neural_net_model.predict([CT_predict])[0]

    # returns index of the maximum value: prediction
    predict_class = np.argmax(predicted_probability)

    # determine the slices that show the mandible
    if predict_class == 1:
        mandible_slices.append(counter_data)
⋮
# outline min and max slice
min_slice = np.min(mandible_slices)
max_slice = np.max(mandible_slices)
⋮
for counter_seg in range(min_slice, max_slice+1):
```

Figure 65: Integration of the classification into the segmentation process.

In addition to this, the Python scripts for testing the predicted segmentations permit a storage of every result as a PNG-file. To calculate the 3D Dice-coefficients and the Hausdorff distances, my advisor provided a MeVisLab network that accomplishes these calculations. This module network receives, however, all ground truths and predicted segmentations of one patient as 3D NRRD-files. Thus, the segmentation slices, that were stored as single slices after the performed segmentation, have to be stacked to re-build a 3D image set.

On these grounds, MeVisLab modules were combined to carry out this conversion (see Figure 66). The module *Compose3DFrom2DFiles* stacks images of a determined folder in order to get a 3D dataset. The single images are arranged alphabetically according to their names, where the *View2D* module allows a review if the slices are correctly combined. Furthermore, the *itkImageFileWriter* module stores the image piles as one NRRD-file.

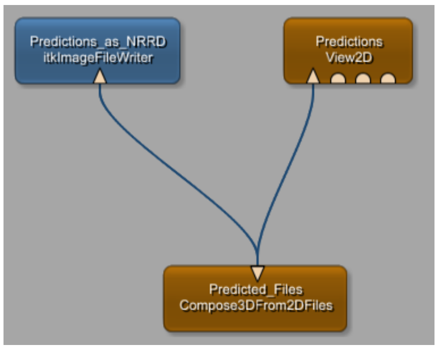


Figure 66: MeVisLab network for the generation of 3D NRRD-files.

The MeVisLab network for the computations of the DSC and the Hausdorff distance is shown in Figure 67, whereby the emphasised modules are of most importance. The red highlighted *itkImageFileReader* modules are utilised to import the created NRRD stacks of the ground truths (left) and the predictions (right). Moreover, the yellow framed module *itkHausdorffDistanceImageFilter* calculates the Hausdorff distance between the two inputs. The result of the Dice coefficient is visible within the yellow highlighted *Arithemtic0* module's viewer panel.

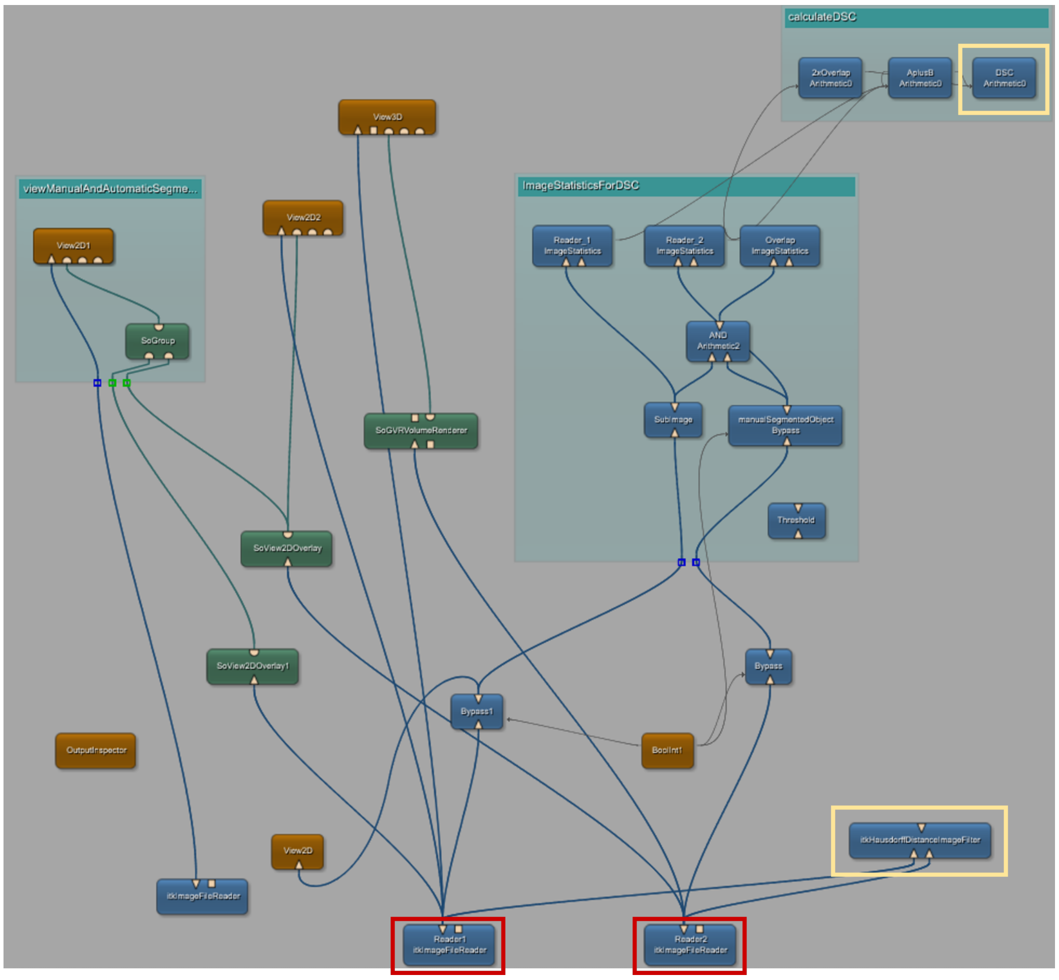


Figure 67: MeVisLab network to compute the Dice Scores and the Hausdorff distances. The ground truths and predictions are imported with the modules at the bottom (red), whereas the results of the DSCs and the Hausdorff distances are shown within the user panels of the yellow framed modules.

Beyond that, detailed information about the datasets, which were utilised for testing, is outlined in Table 14. The test images were generated with the MeVisLab implementations, too. Nevertheless, they don't encompass the original images, but they comprise affine transformed and noisy ones, which were not used for training the networks. The affine transformations were simultaneously applied, whereby the scaling was conducted with the same factors in x- and y-direction (symmetric scaling). It was possible to generate six new images per one original slice.

Table 14: Exportation settings in MeVisLab to create augmented images for testing the segmentation networks. The affine transformations were combined. Thus, flipping, rotation and scaling were simultaneously applied.

Data Augmentation	**Type**	**Settings**	**Adjusted Values**	**Created Images per Slice**
Combined Affine Transf.	Flipping	-	-	4
	Rotation	Rot. Angles:	$\pm 6°$	
	Scaling	Scal. Factors:	0.97 & 1.03	
Noise	Gaussian	Mean:	0	1
		Std. Dev.:	200	
	Uniform	Amplitude:	800	1

All in all, it has to be mentioned that it was pledged that new and independent images will be delivered for testing the networks. Unfortunately, it was too difficult to acquire new CT datasets, which exhibit non-fractured bones and teeth without metallic implants. As a consequence, the networks were tested with augmented images.

6 Results

Following chapters present the results of this master thesis. Some artificially generated CT images are shown as well as the classification and also the segmentation results are outlined. All listed tables were analysed with the support of Microsoft Excel, which delivers functions to compute the mean values, the standard deviations as well as the minimum and maximum values.

6.1 Data Generation with the MeVisLab Implementations

The displayed figures depict examples of synthetic generated CT images, which were created with the implemented MeVisLab network and macro module. All images show slice 44 of dataset seven. Figure 68 illustrates the initial acquired CT slice, whilst Figure 69 displays the mirrored version.

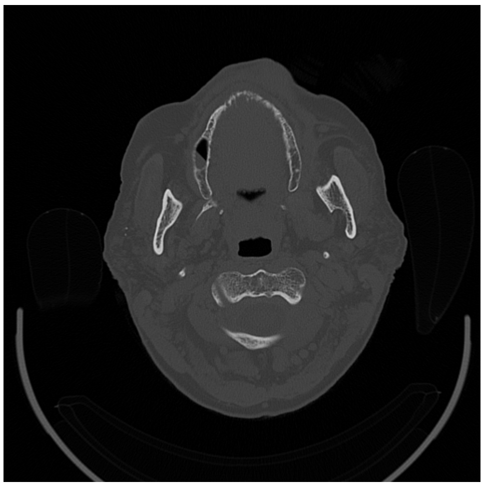


Figure 68: Original CT image.

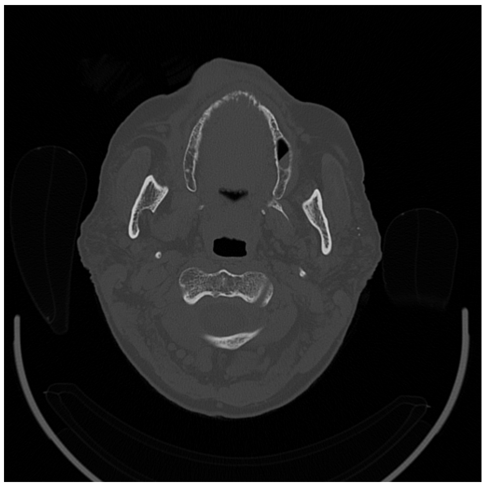


Figure 69: Flipped CT image.

Additionally, the ground truth mask, which pertains to the CT slice of Figure 68, is visible in Figure 70. Figure 71 displays the with $-8°$ rotated segmentation mask, as the affine transformations are also applied on the masks.

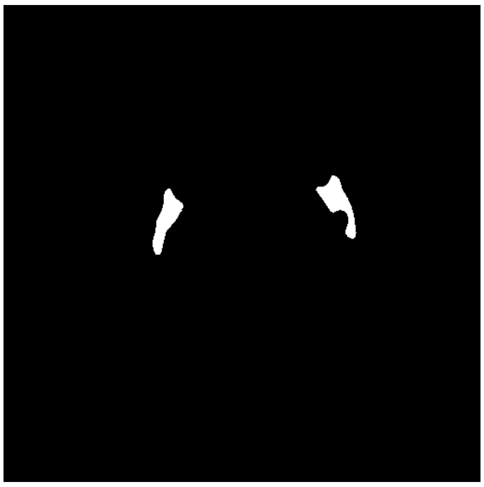


Figure 70: Original mask.

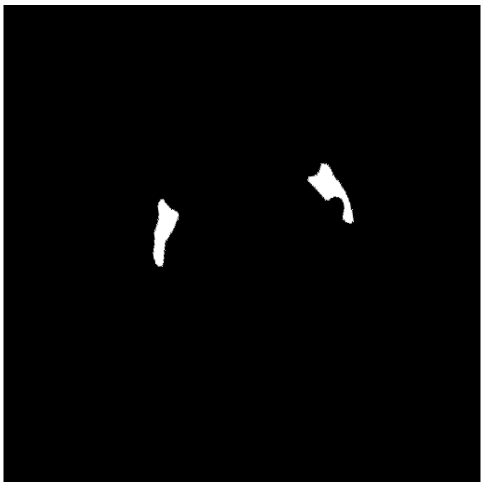


Figure 71: To $-8°$ rotated mask.

Furthermore, Figure 72 illustrates a scaled image with different applied scaling factors in x- and y-direction. It has to be noticed that scaling factors greater than one lead to a shrinking and factors smaller than one lead to an enlargement of the anatomical structures. Figure 73 shows the application of a combination of all implemented geometric transformation types. Hence, flipping, symmetric scaling and rotation are simultaneously deployed.

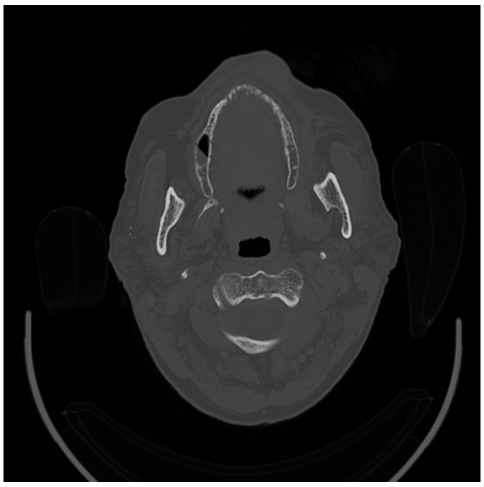


Figure 72: Scaled image with scaling factors of 1.04 in x- and 0.96 in y-direction.

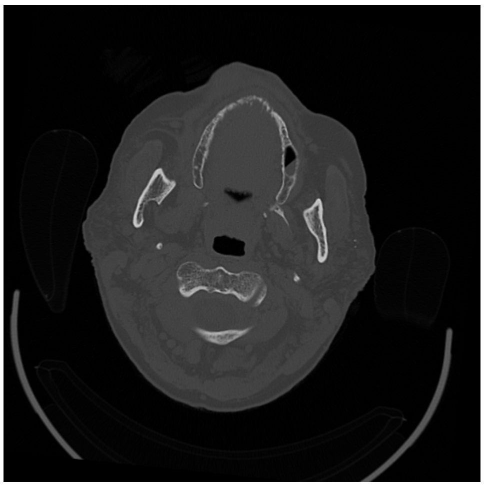


Figure 73: Combined transformations: flipping, rotation $(-6°)$ and scaling (1.03 in x- and y-direction).

Moreover, the added noise types Gaussian as well as Salt and Pepper are shown in the Figures 74 and 75.

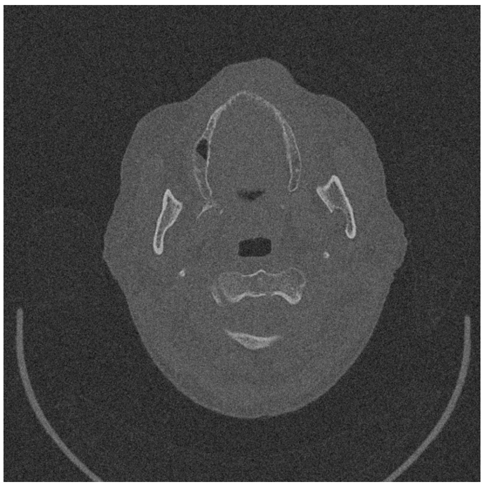


Figure 74: Added Gaussian noise with a mean of zero and a standard deviation of 300.

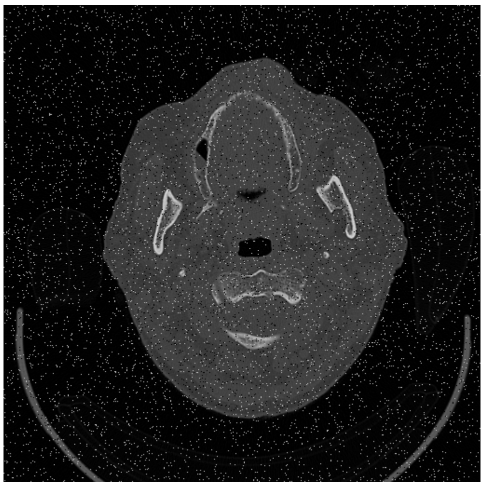


Figure 75: Added Salt and Pepper noise with amplitudes of ±2000 and a density of 0.05.

6.2 Classification Results

The performance of the trained classification CNNs, listed in chapter 5.3.5, was mainly evaluated with the obtained validation and test accuracies as well as with the loss values. Furthermore, the graphical progress of the training accuracy and the loss were investigated.

Table 15 lists an overview of the achieved training and test results of deep learning networks for classification. In Appendix 8.1 is Table 23 visible, which presents these results as percentage values. The validation accuracies and the test accuracies are displayed for the different network topologies, the four different sized training datasets, the various sized convolutional kernels and the two down-sampled image sizes. To remember, dataset one comprises the 1680 original CT images, dataset two the noisy and the original ones (6720 images) and dataset three the affine transformed and also the original ones (13440 images). Dataset four involves both augmentation methods as well as the original images (18480 images). Additionally, the differences of the validation accuracies and the test accuracies are calculated. There is usually a decline of the test accuracy compared to the validation accuracy, with the exception of two values. These two improvements, however, are small. Besides that, the minimal validation accuracy is 0.8333 and the maximum value is one, which appears in eight cases. The minimal test accuracy shows a value of 0.8217 and the highest test accuracy is 0.9883.

Table 15: Results of the trained classification networks. The accuracies of the validation as well as of the artificial generated test dataset are listed below. Furthermore, the differences between these two accuracies are calculated.

Topo-logy	**Dataset No.**	**Conv. Kernel**	**Validation Accuracy**	**Test Acc.**	**Difference of Accuracies**
			Image Size: 50 × 50		
	1	3	0.9333	0.8217	0.1116
3 conv. layers	2	3	0.9250	0.9209	0.0041
	3	3	0.9500	0.9058	0.0442
	4	3	0.9833	0.9578	0.0255
	1	3	0.8333	0.8518	-0.0185
4 conv. layers	2	3	0.9417	0.9354	0.0063
	3	3	0.9786	0.8909	0.0877
	4	3	0.9667	0.9699	-0.0032
	1	3	0.9667	0.8536	0.1131
6 conv. layers	2	3	0.9917	0.9449	0.0468
	3	3	0.9786	0.9210	0.0576
	4	3	0.9944	0.9752	0.0192
	1	5	1.0000	0.8719	0.1281
6 conv. layers	2	5	0.9833	0.9620	0.0213
	3	5	0.9857	0.9643	0.0214
	4	5	1.0000	0.9848	0.0152
	1	7	1.0000	0.8969	0.1031
6 conv. layers	2	7	1.0000	0.9704	0.0296
	3	7	0.9929	0.9443	0.0486
	4	7	**1.0000**	**0.9877**	0.0123
			Image Size: 128 × 128		
	1	7	0.9667	0.9161	0.0506
6 conv. layers	2	7	1.0000	0.9777	0.0223
	3	7	1.0000	0.9701	0.0299
	4	7	**1.0000**	**0.9883**	0.0117

On top of that, Table 16 shows the accuracies, which are averaged over the dataset sizes, and the corresponding standard deviations. It can be noticed that a larger dataset produces a higher validation and test accuracy (except the averaged test accuracy of dataset No. 3). Accordingly, the artificial enlargement of the CT datasets improves the performance of neural networks.

Table 16: Averaged validation and test accuracies of the four different sized training datasets.

Dataset No.	**Mean Valid. Accuracy**	**Std. Dev. Valid.**	**Mean Test Accuracy**	**Std. Dev. Test**
		Image Size: 50×50		
1	0.9467	0.0691	0.8592	0.0277
2	0.9683	0.0330	0.9467	0.0199
3	0.9771	0.0163	0.9253	0.0294
4	0.9889	0.0142	0.9751	0.0120

Additionally, Table 17 presents the validation and test accuracies, which are averaged over the diverse network topologies. Moreover, the calculated standard deviations are listed. It can be concluded that both accuracy values are higher for deeper networks (except the validation accuracy of the net with four convolutional/max-pooling layers). Furthermore, training with a larger convolutional kernel leads to higher accuracy values.

Comparing the averaged values of the networks, which consist of a six convolutional/max-pooling layer topology and a kernel size of seven, shows that the validation accuracies are nearly the same for the two down-sampled image sizes. Although, the averaged test accuracy of the 128×128 images is a bit higher than that of the smaller sized images.

Table 17: Averaged validation and test accuracies of the different network topologies and image sizes.

Topology	**Conv. Kernel**	**Mean Valid. Accuracy**	**Std. Dev. Valid.**	**Mean Test Accuracy**	**Std. Dev. Test**
		Image Size: 50 × 50			
3 conv.	3	0.9479	0.0258	0.9016	0.0575
4 conv.	3	0.9301	0.0663	0.9120	0.0516
6 conv.	3	0.9828	0.0128	0.9237	0.0517
6 conv.	5	0.9923	0.0090	0.9458	0.0503
6 conv.	7	0.9982	0.0036	0.9499	0.0395
		Image Size: 128 × 128			
6 conv.	7	0.9917	0.0167	0.9630	0.0322

In addition to this, the progression of the loss and the training accuracies can be evaluated graphically with the support of TensorFlow's TensorBoard tool. The library TFLearn enables an automatic storage of the metrics during training. Beyond that, the history of the accuracy and also the loss are printed over the training steps in the following figures. Networks, which use the same dataset, were trained with a equal number of iteration steps in order to allow a comparison of the results.

In Figure 76 is the course of the training accuracy of three network configurations, which were trained with the first dataset (1680 images), shown. All displayed nets utilised a convolutional filter size of three and the images were down-sampled to a size of 50 × 50. It is observable that the network with six conv./max-pooling layers (purple) reaches higher accuracy values compared to the shallower networks (orange and cyan). Moreover, a deeper network obtains a higher accuracy after fewer training steps.

Figure 77 shows the corresponding progression of the loss during the training process. It is obvious that the deepest net (purple) has initially a lower loss, while the shallower networks start with higher loss values. Furthermore, the networks with three (orange) and four (cyan) conv./max-pooling layers don't reach such a low loss as the deepest network does.

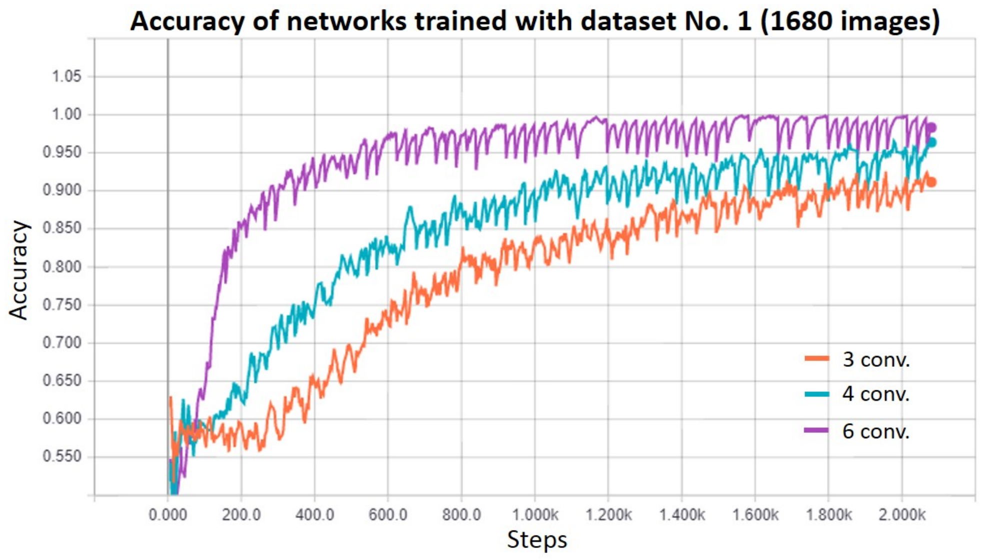


Figure 76: Training accuracy of networks trained with dataset No. 1 (1680 images). The conv. kernel was set to three and the images had a size of 50×50.

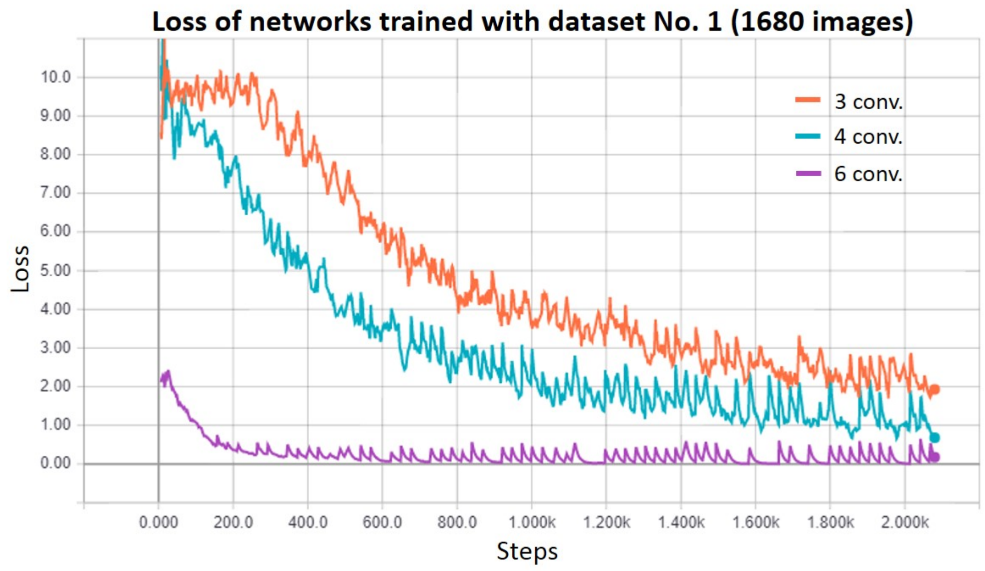


Figure 77: Loss of networks trained with dataset No. 1 (1680 images). The conv. kernel was set to three and the images had a size of 50×50.

For comparison, the Figures 78 and 79 also illustrate the training history of the metrics of the three network configurations analysed priorly. However, the training datasets cover 13440 images. Again, the deeper the networks, the earlier approximate both metrics towards optimal values. Nevertheless, even shallower topologies reach nearly an accuracy of one and a loss of zero in the case of a bigger dataset.

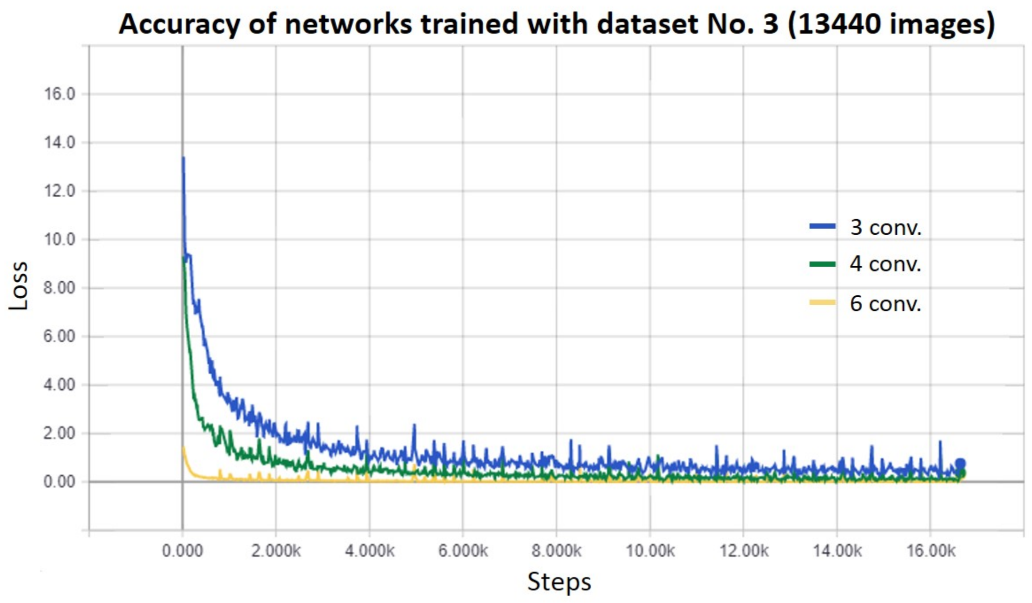


Figure 78: Training accuracy of networks trained with dataset No. 3 (13440 images). The conv. kernel was set to three and the images had a size of 50×50.

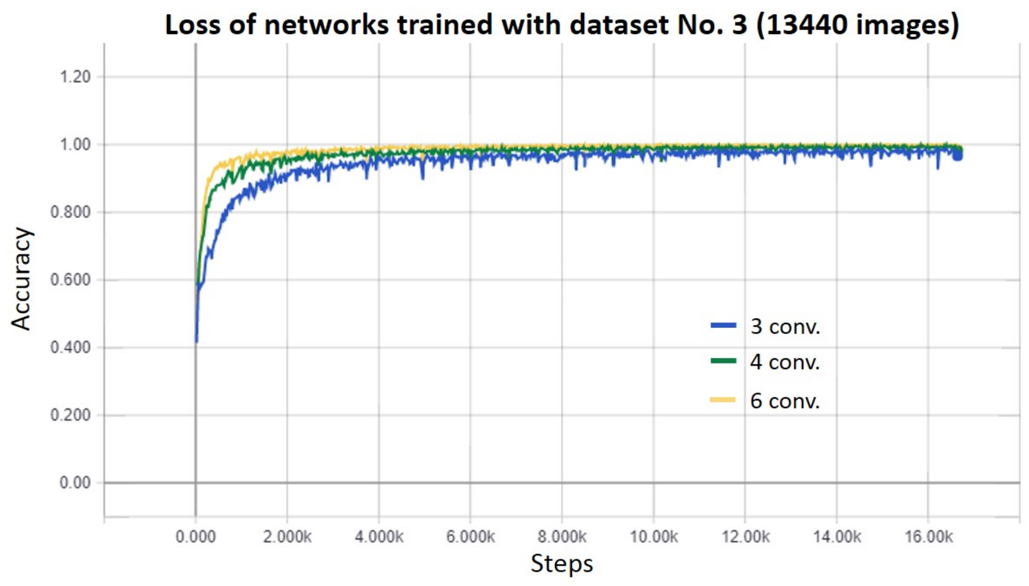


Figure 79: Loss of networks trained with dataset No. 3 (13440 images). The conv. kernel was set to three and the images had a size of 50×50.

For reasons of clarity, the diagrams of the networks, which were trained with dataset No. 2 (6720 images) and No. 4 (18480 images), can be found in the Appendix section 8.1. These two augmented datasets produce similar results as the examples before.

Apart from that, the following two Figures 80 and 81 display the training accuracy and loss of five networks, which were trained with dataset No. 4 (18480 images). The networks with the topology including three conv./max-pooling (orange) and four conv./max-pooling layers (cyan) exhibit worse progressions of the metrics compared to the nets with six conv./max-pooling layers. Thereby, the networks with the higher filter size of seven (green and yellow) produce the best results. These two curves show nearly the same behaviour, whereby one network is trained with 50×50 sized images and the other one is trained with 128×128 sized images. Both correspond to the two networks, which achieved the best results for the validation and test accuracies (bold values in the Tables 15 and 23).

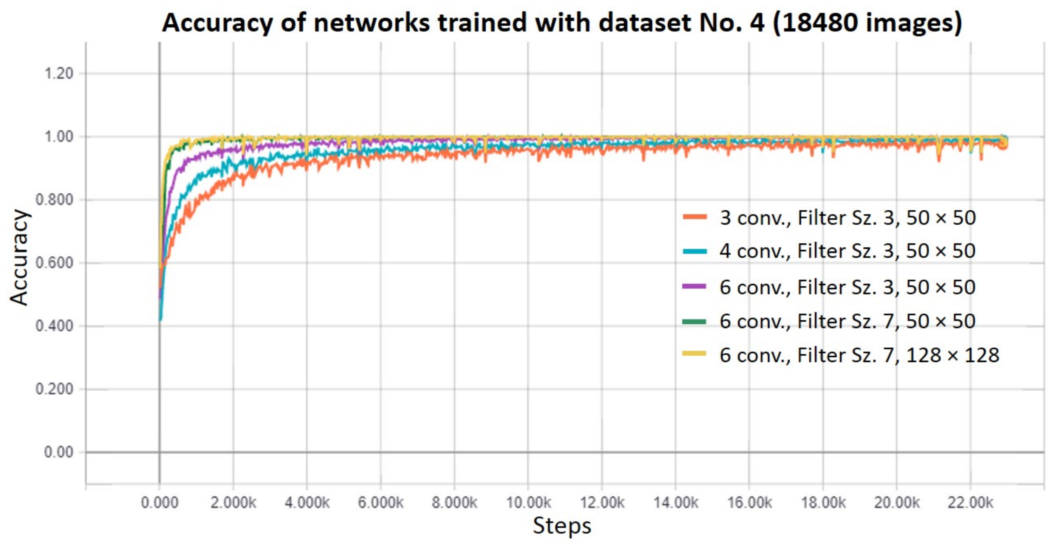


Figure 80: Training accuracy of five networks trained with dataset No. 4 (18480 images). The networks were trained with different filter sizes and a diverse image matrix.

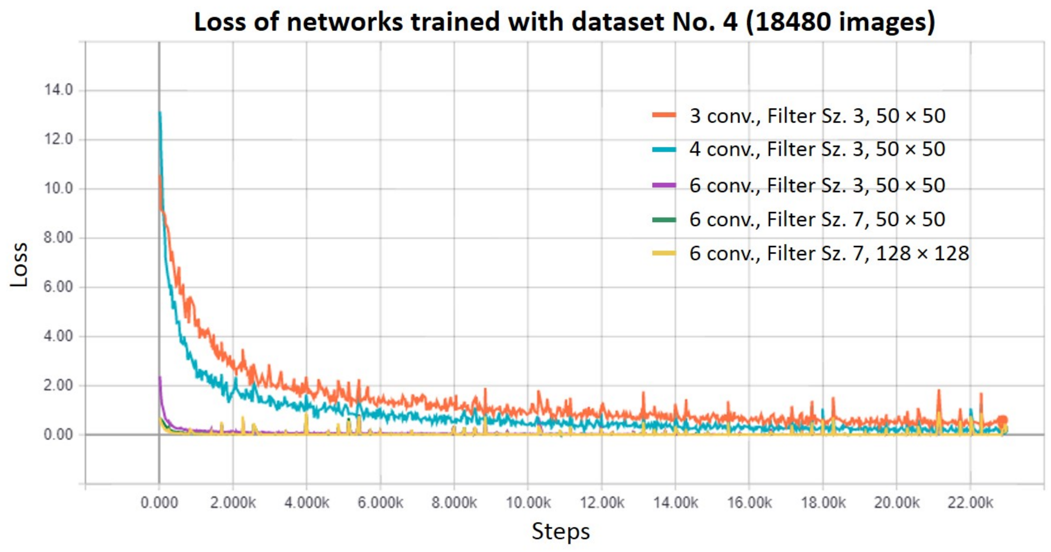
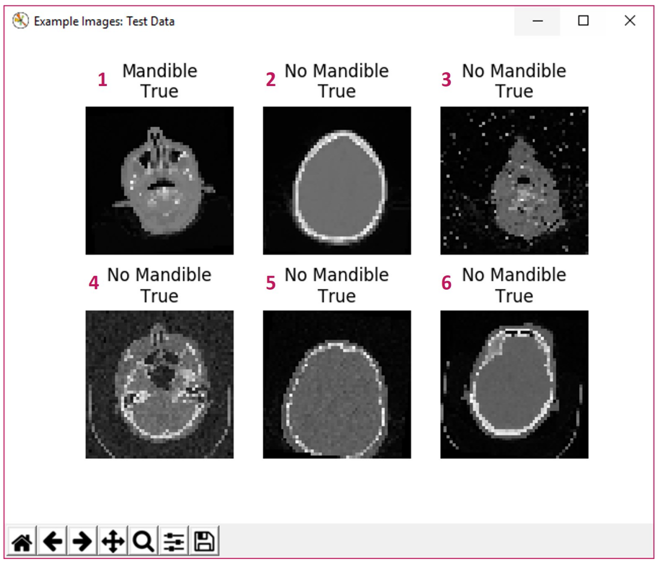


Figure 81: Loss of five networks trained with dataset No. 4 (18480 images). The networks were trained with different filter sizes and a diverse image matrix.

On top of that, Figure 82 depicts some classified images of the test dataset. The classification was achieved with the network comprising six conv./max-pooling layers and a filter size of seven, whilst the training images had a size of 50 × 50 (green curves in the Figures 80 and 81). The numerical data next to the image displays the class-prediction vectors (probabilities of the classes) of the tested images.



Tested Images with a matrix size of **50 × 50**

```
1 [  1.60847267e-05   9.99983907e-01]
2 [ 1.   0.]
3 [  1.00000000e+00   3.68111439e-14]
4 [  1.00000000e+00   4.98811305e-11]
5 [  1.00000000e+00   1.07357921e-32]
6 [ 1.   0.]
```

Figure 82: Test images (50 × 50) and their predicted classes. The network had six conv./max-pooling layers and a filter size of seven.

To conclude, the two neural models with the topology of six conv./max-pooling layers, the filter size of seven and the largest training dataset deliver the best validation and test accuracies for both image sizes (bold values in Tables 15 and 23). The test accuracy of the dataset with the 128 × 128 images is a bit better. However, training the network with the 50 × 50 sized images took 19 minutes and 29 seconds, whilst training with the larger sized images took 42 minutes and 38 seconds. Not only the training time, but also the time required for testing was prolonged for the larger sized images: Testing the images with a size of 50 × 50 lasted about 33 seconds, whereas testing the 128 × 128 sized ones took about 1 minute and 20 seconds.

As a consequence, the network, which was trained with the smaller sized images and the shorter training and testing time duration, was used for the further segmentation task.

6.3 Segmentation Results

The performance of the segmentation networks was evaluated with the mean intersections over unions, which were calculated with the TF-Slim library after finishing the network training, as well as with the Dice coefficients and the Hausdorff distances, which were computed with the MeVisLab implementations for a patient's dataset. Furthermore, the training progress was analysed with the loss function visualised in TensorBoard.

All datasets were manually segmented by two physicians, whereby the ground truths of doctor A were utilised for training the networks of this thesis. Nonetheless, the inter-observer variability was appreciated between the two available manual segmentations. Therefore, the contours of doctor A were supposed to be the ground truths, whereas the segmentations of physician B were assumed to be the algorithmic results. Table 18 displays the calculations of the Dice coefficients and the Hausdorff distances between the two manual segmentations evaluated separately for each patient. It is obvious that a down-sampling of the masks leads to a deterioration of the Dice scores and the Hausdorff distances.

It has to be kept in mind that the Hausdorff values diminish, because the down-sampling of the image matrix leads also to a down-sizing of the depicted objects and thus, the distances exhibit smaller values. A further issue is that the information about absolute voxel sizes (listed in Table 2) is lost after down-sampling the images within the Python implementations. The loss of these dimensions invokes that all predicted segmentations have voxels with a shape of $1 \times 1 \times 1$, whereby the units, for instance millimeters, are not defined. The values of Hausdorff distances of this section always refer to these new, unitless voxel sizes.

In addition to this, the inter-observer Dice coefficients and Hausdorff distances are averaged over the patients, the standard deviations are listed as well as the extrema are stated in subsequent table.

Table 18: Inter-observer variability of the manual segmentations produced by doctor A (ground truth) and doctor B (algorithmic segmentation).

Patient	**DSC**	**Hausdorff**	**DSC**	**Hausdorff**
	256 × 256		512 × 512	
1	0.9168	3.0000	0.9433	4.2426
2	0.8874	4.1231	0.9176	7.0000
3	0.8960	3.0000	0.9266	5.4772
4	0.9149	3.1623	0.9469	6.3246
5	0.9008	2.4495	0.9368	4.1231
6	0.9125	3.1623	0.9447	6.4031
7	0.9183	5.0990	0.9412	6.4031
8	0.9125	3.0000	0.9424	3.3166
9	0.8943	3.7417	0.9253	7.2801
10	0.9095	2.8284	0.9373	4.5826
Mean	**0.9063**	**3.3566**	**0.9362**	**5.5153**
Std. Dev.	0.0108	0.7696	0.0098	1.3668
Min.	0.8874	2.4495	0.9176	3.3166
Max.	0.9183	5.0990	0.9469	7.2801

On top of that, the mean intersection over union was calculated for each dataset after accomplished training. Table 19 lists these values for the four different training datasets. To remember, dataset I and II encompass the original available CT slices, whilst the other two training sets (III and IV) involve augmented images. Moreover, two datasets (I and III) were down-sampled to a size of 256 × 256 for training the networks.

Besides that, Table 19 displays also the mean IoU of the test dataset, which consists of geometric transformed and noisy images that were not utilised for training. The test images were also down-sampled if they were applied to segmentation networks, which were trained with images consisting of a size of 256 × 256.

The influence of the various deep architecures (FCN-32s, FCN-16s and FCN-8s) is clearly recognisable from the values of the table below. The integration of max-pooling layers into the upsampling step improves the results. Hence. the more max-pooling layers are involved, the better is the mean IoU of the whole dataset. Furthermore, the computed metrics of the down-sampled datasets are distinctly worse compared to a training with the original sized images. Nevertheless, a comparison of the datasets, which comprise the original available images, and the datasets, which involve artificial generated images, shows that the mean IoU improves slightly for the augmented training sets with the exception of the FCN-32s model trained with the datasets I and III. Apart from that, the testing mean IoU is expectedly a bit worse for each network configuration.

Table 19: Mean intersection over union evaluated after accomplished training. The mean IoU was computed for the training datasets and the test dataset. Furthermore, all utilised CT slices and masks have a quadratic shape. However, there is only one numeral listed for the resolution of the images.

Training Datasets	**Image Size**	**FCN Type**	**Training mean IoU**	**Testing mean IoU**
I (702 images)	256	FCN-32s	0.6246	0.5748
	256	FCN-16s	0.7588	0.6784
	256	FCN-8s	0.8558	0.7671
II (702 images)	512	FCN-32s	0.7677	0.6853
	512	FCN-16s	0.8817	0.8415
	512	FCN-8s	0.9148	0.8776
III (4212 images)	256	FCN-32s	0.6102	0.5976
	256	FCN-16s	0.7622	0.7461
	256	FCN-8s	0.8511	0.8291
IV (4212 images)	512	FCN-32s	0.7955	0.7835
	512	FCN-16s	0.8927	0.8791
	512	FCN-8s	0.9243	0.9132

Beyond that, the original datasets were provided separately to the combined classification and segmentation networks (see chapter 5.4.6). Consequently, an evaluation per patient was feasible. Therefore, the whole dataset of one patient was supplied to the best performing classification network. Ideally, all slices were correctly classified. Afterwards, the images, that were identified to show the lower jawbone, were delivered to the FCN networks in order to produce algorithmic segmentations. These predictions were automatically stored as PNG-files to permit a re-build of a 3D image stack and subsequently to evaluate the Dice scores and the Hausdorff distances in MeVisLab. It can be imagined that the analysis per patient produces a huge amount of numerical results. To keep the overview, there are only the results of the segmentation network, which generated the best evaluation metrics, listed in Table 20. The FCN-8s, which was trained with dataset IV, emerged to offer the best performance.

Table 20 also informs about correct classified slices. All images of the patients were identified accurately with the exception of patient nine. The classification network forecasted that 92 slices instead of 91 show the lower jawbone. Thus, the slice, that was wrongly predicted, was also used as an input for the segmentation networks.

Additionally, the table below outlines the averaged segmentation metrics, the standard deviations and the extrema. The mean Dice coefficient shows a value of 0.9203, which is a bit lower than the inter-observer variability of 0.9362. By contrast, the averaged Hausdorff distance exhibits a higher value of 7.3221 compared to the inter-observer variability of 5.5153. As a result, the FCN-8s model produces for the original CT slices poorer results than the inter-observer variability. However, the decline is actually slight.

Table 20: Segmentation metrics of the FCN-8s model, which was trained with dataset No. IV and analysed per patient. The original CT slices were provided as an input to the trained models.

FCN-8s: Trained with dataset IV

Patient	**Dice**	**Hausdorff**	**mean IoU**	**Correct Classified**
1	0.9368	5.0000	0.9401	✓
2	0.9008	6.7823	0.9092	✓
3	0.9119	9.4340	0.9185	✓
4	0.9243	10.4403	0.9290	✓
5	0.9103	10.7238	0.9171	✓
6	0.9223	6.0828	0.9275	✓
7	0.9387	5.3852	0.9415	✓
8	0.9304	5.3852	0.9345	✓
9	0.8996	8.6023	0.9082	92 instead of 91
10	0.9277	5.3852	0.9321	✓
Mean	**0.9203**	**7.3221**	**0.9258**	
Std. Dev.	0.0140	2.2575	0.0120	
Min.	0.8996	5.0000	0.9082	
Max.	0.9387	10.7238	0.9415	

For the sake of completeness, screenshots of the Microsoft Excel tables, which comprise the segmentation metrics of all networks, are displayed in the Appendix section 8.2. Thereby, the original CT slices were provided as an input to the networks, while the results are evaluated separately per patient.

Nevertheless, a summary of the averaged Dice scores and standard deviations of the referred Appendix 8.2 is shown in Table 21. Hence, the averaged metrics are displayed for all trained FCNs. Again, the values of this table approve that the FCN-8s model delivers better results than the FCN-16s and the FCN-32s networks. Moreover, down-sampling has a negative effect on the Dice scores. Despite that, the more images are utilised for training, the better are the segmentation metrics with the exception of training the FCN-32s model with the datasets I and III. Another aspect is that the standard deviations most commonly decrease for the FCN-16s and the FCN-8s networks.

Table 21: Dice coefficients computed for the original CT slices and averaged over the patients. The averaged Dice scores are displayed for all network configurations as well as the standard deviations are listed. Furthermore, all utilised CT slices and masks have a quadratic shape. However, there is only one numeral listed for the resolution of the images.

Training Datasets	**Image Size**	**FCN Type**	**Mean Dice Score**	**Std. Dev. Dice Score**
I (702 images)	256	FCN-32s	0.3974	0.0686
	256	FCN-16s	0.6755	0.0501
	256	FCN-8s	0.8293	0.0239
II (702 images)	512	FCN-32s	0.6940	0.0519
	512	FCN-16s	0.8634	0.0214
	512	FCN-8s	0.9048	0.0151
III (4212 images)	256	FCN-32s	0.3631	0.0551
	256	FCN-16s	0.6835	0.0553
	256	FCN-8s	0.8306	0.0249
IV (4212 images)	512	FCN-32s	0.7398	0.0493
	512	FCN-16s	0.8824	0.0206
	512	FCN-8s	0.9203	0.0140

On top of that, the classification and segmentation networks with the best performance were tested with artificially generated images, while the metrics were analysed per patient. However, not all images of the test dataset were applied, but only the slices, which were deformed with a simultaneous application of flipping, rotation (+6°) and symmetric scaling (0.97 in x- and y-direction). The other test images were not deployed, because the evaluation per patient was not automatically feasible in MeVisLab.

Table 22 lists the mean intersections over unions, the Dice coefficients and the Hausdorff distances of the augmented slices. The segmentation metrics were averaged as well as the standard deviations and the minima and maxima are displayed. A comparison of the averaged metrics of the test images with those of the original images (Table 20) outlines that the results delivered by the test datasets are expectedly a bit poorer. Following this, the averaged test mean IoU and Dice score decrease, whereas the averaged Hausdorff distance of the test dataset shows a higher value. Moreover, it has to be noticed that the slices were not correctly classified in eight cases. However, only the images, which were predicted to show the lower jawbone, were provided to the segmentation networks.

Table 22: Segmentation metrics of testing the FCN-8s model, which was initially trained with dataset IV. Augmented patient CT slices were provided as an input to the trained model, whilst the evaluation is conduced separately for each patient.

Testing of the FCN-8s model (trained with dataset IV)

Patient	**Dice**	**Hausdorff**	**mean IoU**	**Correct Classified**
1	0.8922	16.7332	0.9331	81 instead of 90
2	0.8710	9.6954	0.8943	88 instead of 92
3	0.8983	10.2956	0.9028	✓
4	0.9097	12.3693	0.9158	✓
5	0.9025	11.4455	0.9140	98 instead of 97
6	0.9103	23.9374	0.9214	65 instead of 66
7	0.9122	8.3666	0.9299	70 instead of 73
8	0.9016	12.4097	0.9288	44 instead of 49
9	0.8624	9.2195	0.8924	88 instead of 91
10	0.9035	24.6779	0.9237	65 instead of 66
Mean	**0.8964**	**13.9150**	**0.9156**	
Std. Dev.	0.0169	5.9509	0.0147	
Min.	0.8624	8.3666	0.8924	
Max.	0.9122	24.6779	0.9331	

In addition to this, Figure 83 shows the progress of the loss function during training the FCN-32s, the FCN-16s and the FCN-8s networks with dataset No. IV. There is only one curve visible, as the subsequent models are initialised with trained weights of the previous one. The FCN-32s training lasted till iteration step 42120, FCN-16s training was finished with step 84240 and the FCN-8s was trained until the final iteration step 126360. Hence, the TensorBoard visualisation was generated within one illustration. Moreover, TensorBoard offers the opportunity to smooth curves if a lot of variations occur. The actual loss values are illustrated with the bright orange curve, whereas the smoothed curve is depicted with the dark orange colour. It is remarkable that the first segmentation model (FCN-32s) exhibits higher loss values than the following networks. Towards the end of training, there are smaller variations of the loss function occurring and a general lower loss is achieved despite to the beginning. Additionally, there are peaks visible if training of a succeeding network is launched. The reason for this is that there are new, un-trained upsampling layers introduced with the subsequent networks.

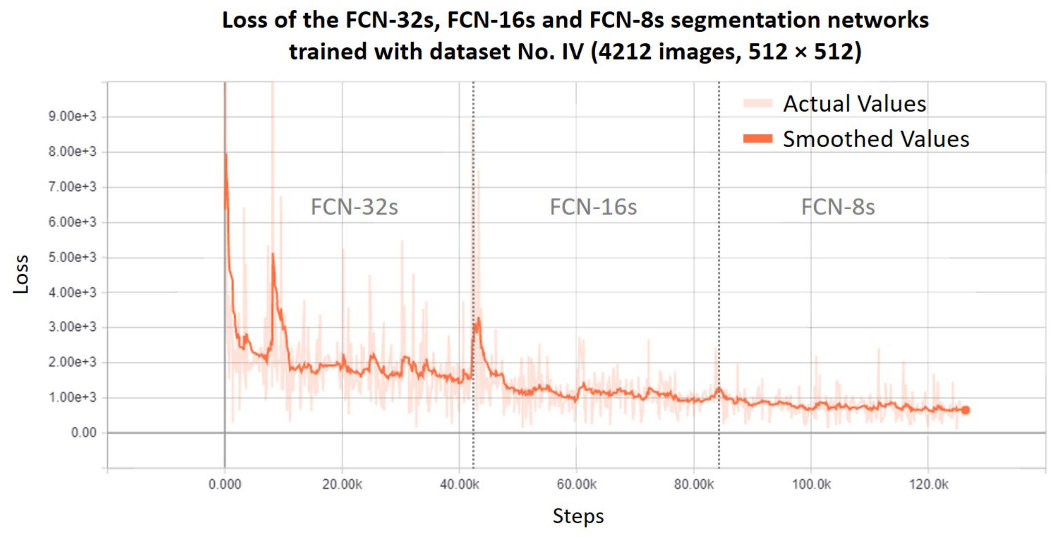


Figure 83: Progress of the loss function during training the FCN-32s (1^{st}), the FCN-16s (2^{nd}) and the FCN-8s (3^{rd}) models with image dataset IV.

Beyond that, Figure 84 depicts exemplary segmentations, which were predicted with the FCN-32s, the FCN-16s and the FCN-8s models. The networks, which were utilised for the forecast, were trained with dataset III (down-sampled images). Moreover, the initial CT image (slice 30 of patient one) and the ground truth are displayed. Not only the evaluation metrics of preceding tables indicate an improvement of the segmentation results with the involvement of max-pooling layers, but also the depicted visual results support this aspect. The predicted mandible of the FCN-32s net seems to be awkward, whilst the lower jawbone of the FCN-16s has already an arched shape. The result of the FCN-8s model is the smoothest one.

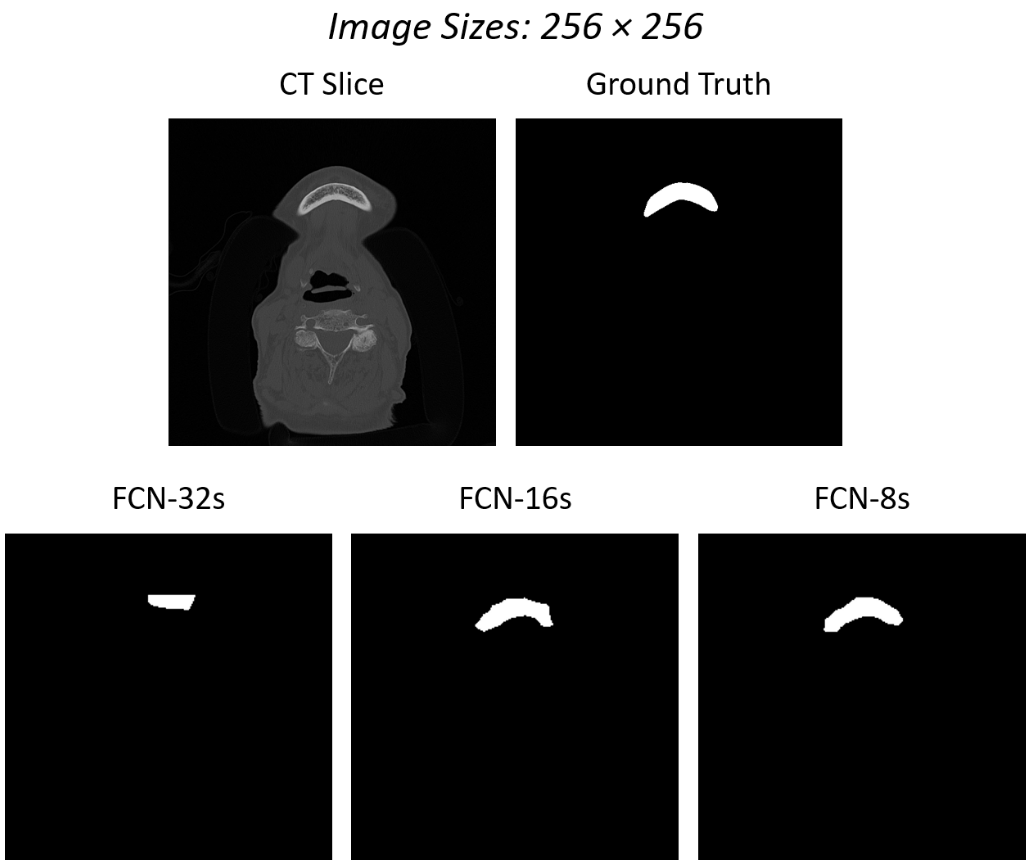


Figure 84: Comparison of a CT slice (256 × 256), its ground truth and the predicted segmentations. The segmentations were forecasted with the FCN-32s, the FCN-16s and the FCN-8s models, which were trained with dataset III.

Despite to the previous examples, Figure 85 shows forecasted segmentations of slice 48 of patient one, whereby the networks were used for prediction, which were trained with dataset IV (original image sizes). The generated segmentations are not that jerky as the down-sampled ones. Furthermore, the lower jaw, which is forecasted with the FCN-8s model, achieves the most similar shape as the the ground truth.

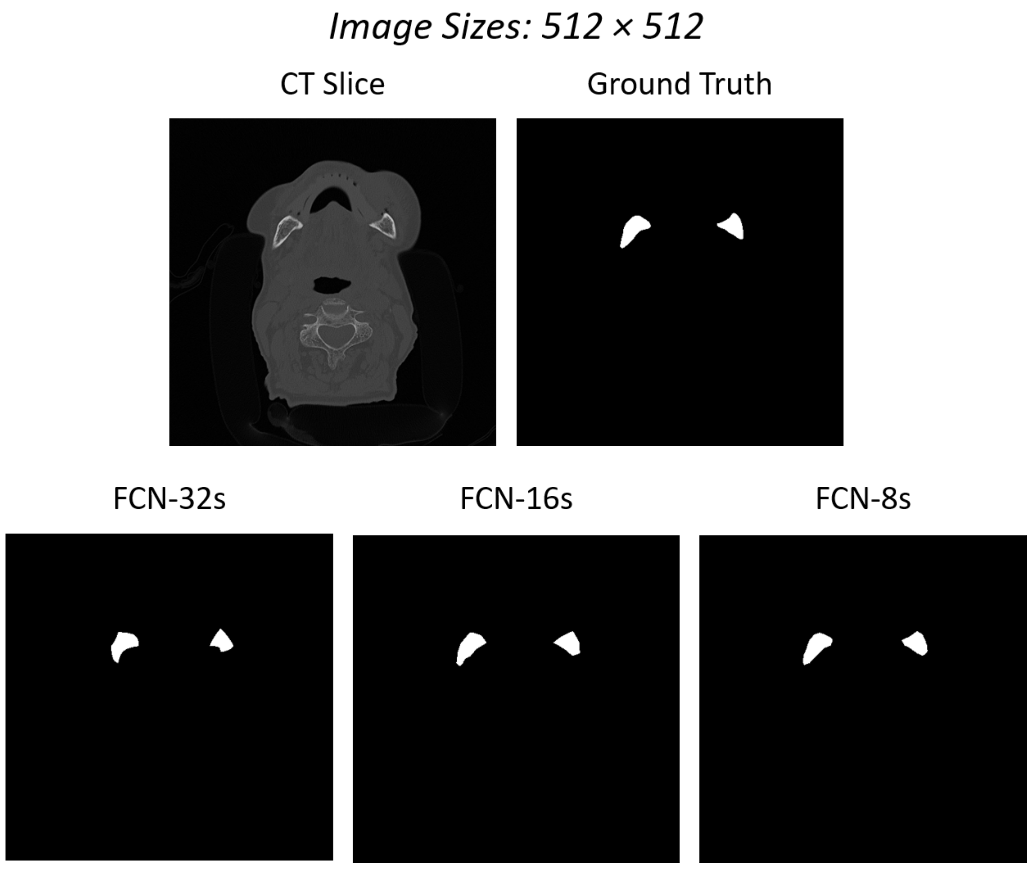


Figure 85: Comparison of a CT slice (512×512), its ground truth and the predicted segmentations. The segmentations were forecasted with the FCN-32s, the FCN-16s and the FCN-8s models, which were trained with dataset IV.

On top of that, Figure 86 displays predicted probability maps of three different input images. Therefore, the CT slices 38, 58 and 78 of patient one were tested with the FCN-32s, the FCN-16s and the FCN-8s models, which were initially trained with dataset IV. The displayed color maps refer to the probability of an occurring mandible. Hence, yellow coloured voxels indicate that they are more likely part of the lower jawbone, whereas blue coloured ones imply the background. It is remarkable that the probabilities are clearer with the usage of max-pooling layers. Moreover, the ground truths are displayed.

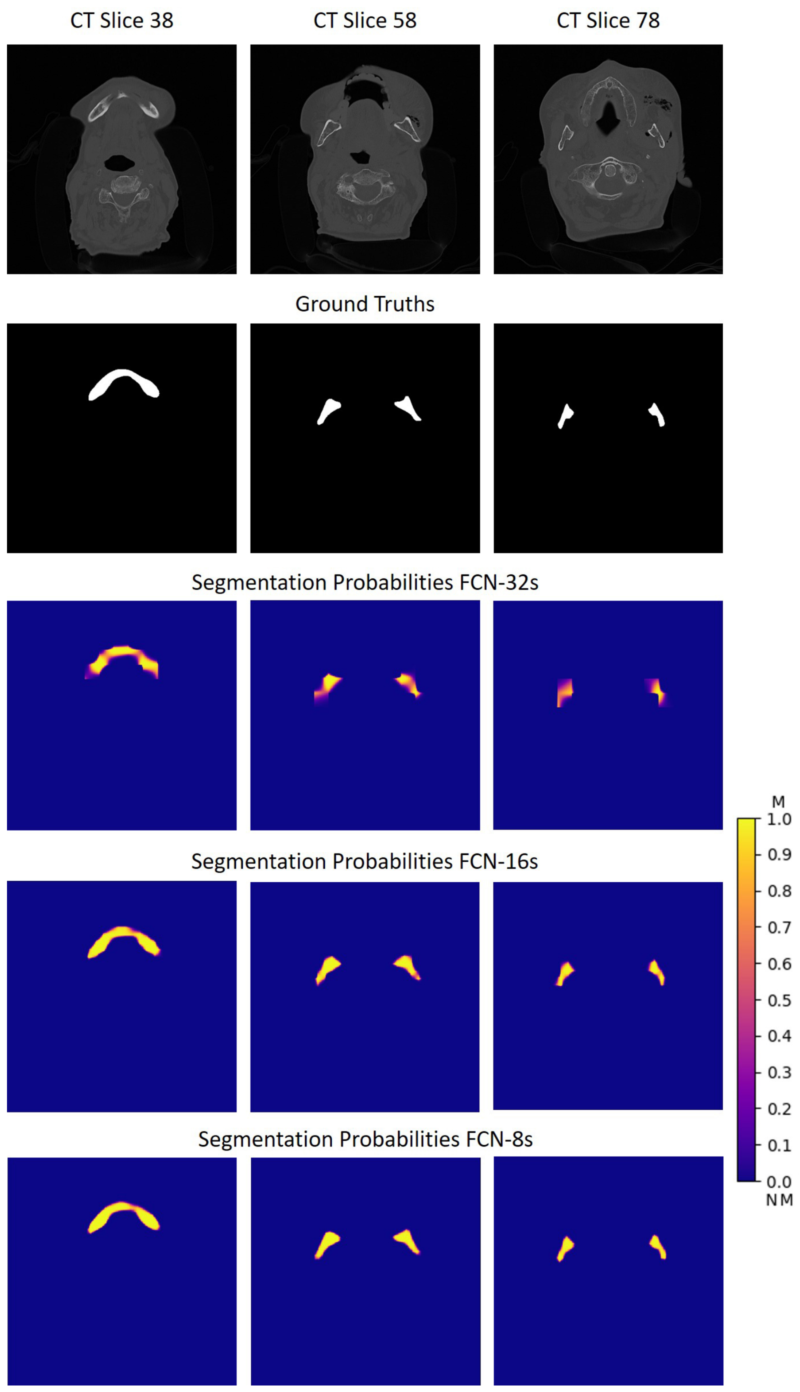

CT Slice 38
CT Slice 58
CT Slice 78
Ground Truths
Segmentation Probabilities FCN-32s
Segmentation Probabilities FCN-16s
Segmentation Probabilities FCN-8s
M
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
NM

Figure 86: Depiction of three CT slices, their ground truths and the predicted probability maps. The maps were forecasted with the networks trained with dataset IV. The brighter the voxels, the more likely they are part of the mandible (M), whilst the blue color implies that there is probably no mandible (NM) appearing.

7 Discussion and Future Outlook

To bring up to mind, a MeVisLab network and a macro module were generated to process and enlarge the head-neck CT datasets during this thesis. Moreover, the ultimate objective was to implement deep networks, which permit an automatic segmentation of the mandible. Therefore, classification networks were trained in order to distinguish whether a slice comprises the lower jawbone or not and consequently, segmentation networks computed the algorithmic demarcations within these slices. All networks were trained and tested with images exported by the MeVisLab realisations.

To summarise the implementations in MeVisLab, the "Save Slices for Deep Learning" network as well as the *SaveAsSingleSlices* macro module were generated, which support the conversion of the ground truth contours into binary masks and the automatic storage of the image slices as separate files. The provided graphical user interface enables a comfortable adjustment of the exportation settings. Moreover, the user is able to define the parameters for a synthetic image generation. If the default augmentation parameters are applied, 18480 CT slices are exported compared to the initial available 1680 images. Although, it would be even possible to create a higher number of augmented images, as the exportation parameters are adjusted depending on the user's intention. However, it is not meaningful to overcome the problem of too few training data with the exportation of millions of synthetic images, since the basic CT slices are still the same.

The specific values of the augmentation parameters (Tables 4, 11, 13 and 14) were chosen according to physiological reasonable ranges. A human may turn the head a view degrees to the left or right during standard CT scans, but a rotation angle in the range of up to 180° won't be feasible. Moreover, the scaling parameters simulated small size variations. The mirroring of anatomical structures was only allowed over the y-axis, as it is not customary to position a patient face down in a CT scanner. Apart form that, the noise parameters were chosen in such a way that the anatomical structures were still identifiable.

In addition to this, the generated module's user interface enables that all parameters are stated before the exportation is launched with the *Start Export* button. Hence, the original as well as the synthetic images are generated with one exportation step. Besides that, the CT slices are imported as NRRD-files into the MeVisLab software. Consequently, if any other medical data, for instance MR images, are stored as NRRD-files, they can be processed with the implemented network and macro module, too.

Nevertheless, it has to be noticed that each dataset must be imported separately into the MeVisLab environment. This was not a problem during this thesis, as there were only ten different datasets present. However, if there might be more datasets available in the future, it is advised to enhance the module network with the opportunity of an automatic loading and processing of several files. A further improvement might be an extension of the MeVisLab network with other augmentation methods, for instance elastic deformations. Despite that, it is important to mention that the more images are available for training, the more memory capacity is required to store the CT slices locally. This was not a challenge during this work, but it might be advised to augment the images online for huge datasets. Online augmentation implies that the synthetic slices are not stored locally, but the original images are rather deformed during the training process.

All in all, the MeVisLab implementations emerged to be absolutely helpful to process the CT datasets in order to permit a training of the classification and segmentation networks of this master thesis.

On top of that, the main difference between the created classification and segmentation models was - apart from the network topologies - that the classification nets were trained from scratch with the available data, whilst the segmentation models were built on a pre-trained network. This pre-trained VGG-16 model was initially trained to classify images of the ImageNet database. Nevertheless, both approaches delivered satisfying results. The classification is a “simplier” task, as it is only decided if the whole image shows the mandible or not. The segmentation task, however, requires a determination of every voxel of an image.

In addition to this, the metrics of the classification task (Table 15), but also of the segmentation task (Tables 19 and 22) show that test datasets produce in general poorer results as the training datasets. A trained model is, of course, adapted to the training images. Nevertheless, the goal of network training is to achieve also good results for new and unseen data.

It is important to emphasize that the test data of this work was not completely independent, because the test images were generated from the original ones. The reason for this circumstance is that it was complete impossible to acquire new test images during the establishment of this thesis. New CT datasets must display an unbroken mandible as well as the patients should not have any teeth. A fulfillment of both requirements is rather unlikely and thus, such images are difficult to be found by the doctors in the hospital's image databases. Furthermore, the physicians cannot spend a lot of time on searching for these special kinds of images during the clinical routine. Moreover, it is also time-consuming to produce the manual segmentations. Apart from that, new CT datasets could not be acquired from test persons on grounds of radiation exposure. As a result, it is definitely advised to accomplish further deep learning investigations with medical databases, that contain a larger amount of images.

Nonetheless, the produced results are surprisingly acceptable in consideration of the minor dataset size. Take, for example, the averaged Dice coefficient of the test dataset of Table 22, which shows a value of 0.8964. In contrast, the inter-observer variability is 0.9362. It is not meaningful to achieve Dice coefficients of one. A value of one implies in fact a perfect predicted segmentation, but even though two different physicians don't produce in general exact the same manual segmentations. Hence, the results of the Dice scores are quite satisfying. Moreover, the Hausdorff distance exhibits an averaged value of 13.9150, which is poorer than the inter-observer variability of 5.5153. To remember, those values refer to the unitless voxel dimensions. Thus, there are no units declared. Apart from that, the mean IoU was not calculated for the inter-observer variability, as this metric was computed after training the neural networks.

A probable reason for the achieved proficient metrics might be that the mandible is considerably noticeable in CT images. The bone has a higher attenuation ability of the X-Rays compared to the surrounding soft tissue (Figure 10). Hence, the mandible can be easily demarcated from the CT slices. Furthermore, a lower jawbone has a similar shape towards different patients.

These aspects were also clarified by B. Ibragimov and L. Xing in their publication [24]. They trained segmentation networks for the demarcation of different anatomical structures of the head and neck region, but the automatic mandible prediction produced the best mean Dice score of 89.5%.

Beyond that, not only the numerical results, but also the depictions of the algorithmic segmentations in the Figures 84, 85 and 86 are gratifying. During the evaluation of the results, it was realised that down-sampling of the images reduces the training time of the segmentation networks, but the results (visual predictions and metrics) are distinctively worse compared to the original sized images. In general, training a network exhibits a long time duration, but once it is trained efficiently, testing new images is achieved quickly.

On the whole, the most essential problem, which must be solved for additional deep learning implementations in medicine, is the lack of available images. If there are databases utilised, which comprise a huge amount of images, it must be kept in mind that the ground truths must be created manually for supervised training. A trick to overcome this problem may be the utilisation of overlaid images (e. g. registration of nuclear medical images on CT or MR images). For instance, cancerous tissue might be segmented in CT slices, whereby the nuclear medical information corresponds to the ground truths, as the tracers accumulate in tumours. If the problems of the lack of available data are resolved, more detailed investigations may be feasible in the field of network architectures or parallel GPU training.

To conclude, the implemented networks of this thesis were an explanatory step for the application of deep models in the medical domain, but for a usage in clinical routine a training and also a testing with a lot of more images is essential.

8 Appendix

The Appendix lists additional achieved results of the classification and segmentation tasks for the sake of completeness.

8.1 Additional Classification Results

Figure 87 illustrates the progress of the accuracy during training three different classification networks. The convolutional filter size exhibits a value of three and the net was trained with dataset No. 2 including 6720 CT slices. Figure 88 shows the associated loss function. Moreover, the metrics of three networks trained with dataset No. 4 (18480 images) are displayed in the Figures 89 and 90. Again, the filter size is three and the images are down-sampled to a size of 50 × 50.

Besides that, Figure 91 shows classified examples of the network including six conv./max-pooling layers, a convolutional filter size of seven and the images were down-sampled to a size of 128 × 128.

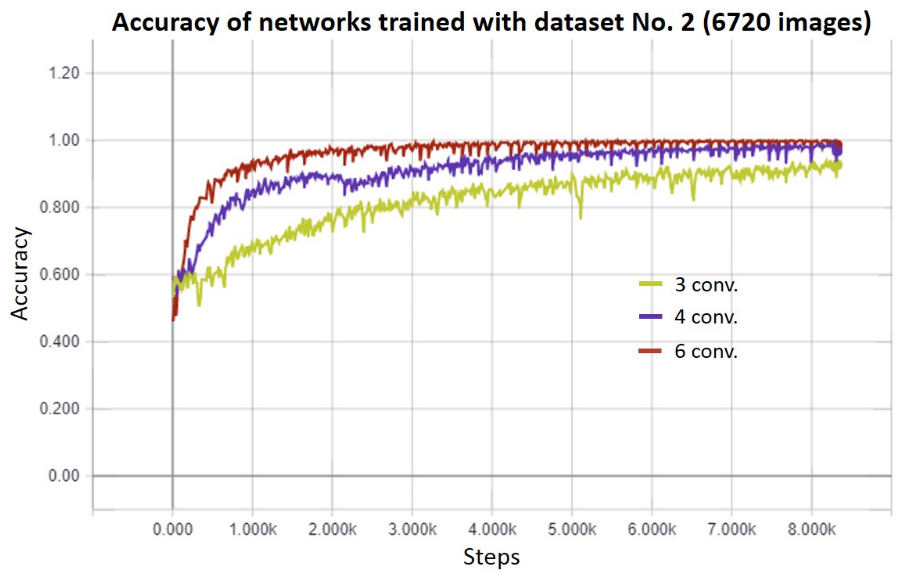


Figure 87: Training accuracy of networks trained with dataset No. 2 (6720 images). The conv. kernel was set to three and the images had a size of 50 × 50.

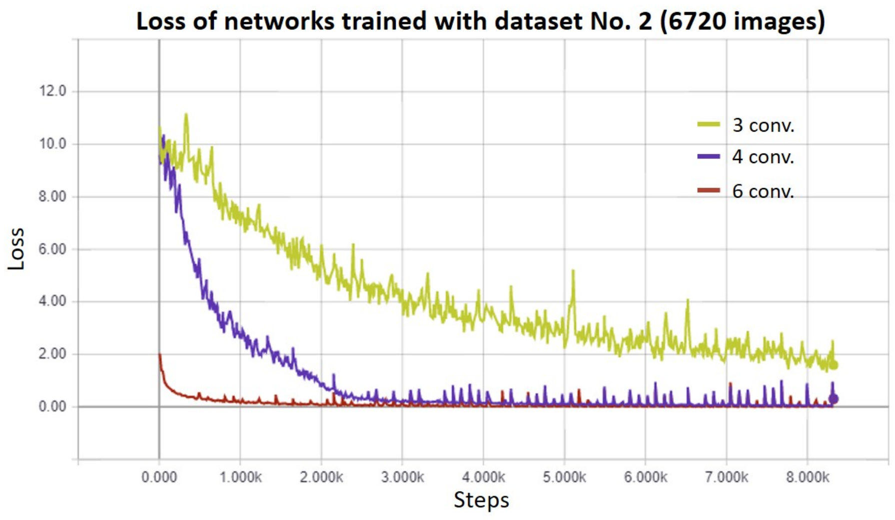


Figure 88: Loss of networks trained with dataset No. 2 (6720 images). The conv. kernel was set to three and the images had a size of 50×50.

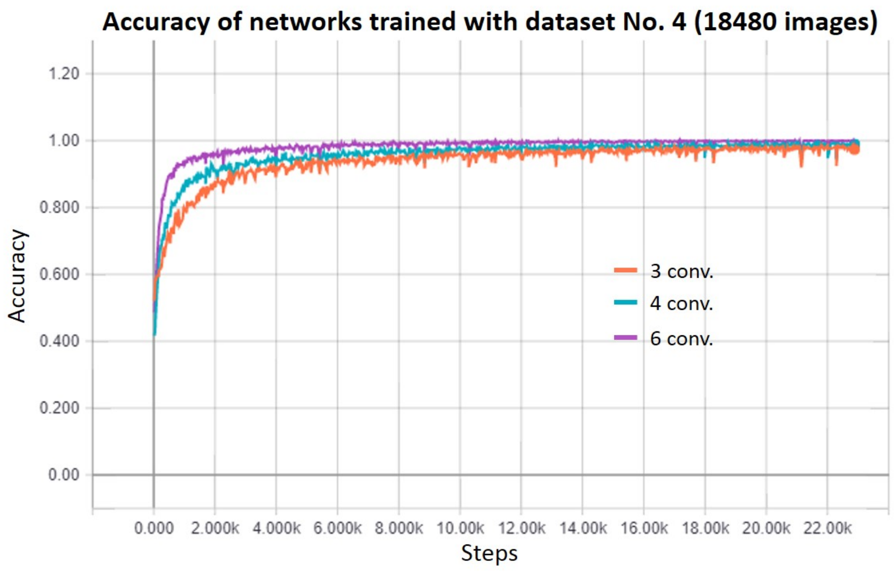


Figure 89: Training accuracy of networks trained with dataset No. 4 (18480 images). The conv. kernel was set to three and the images had a size of 50×50.

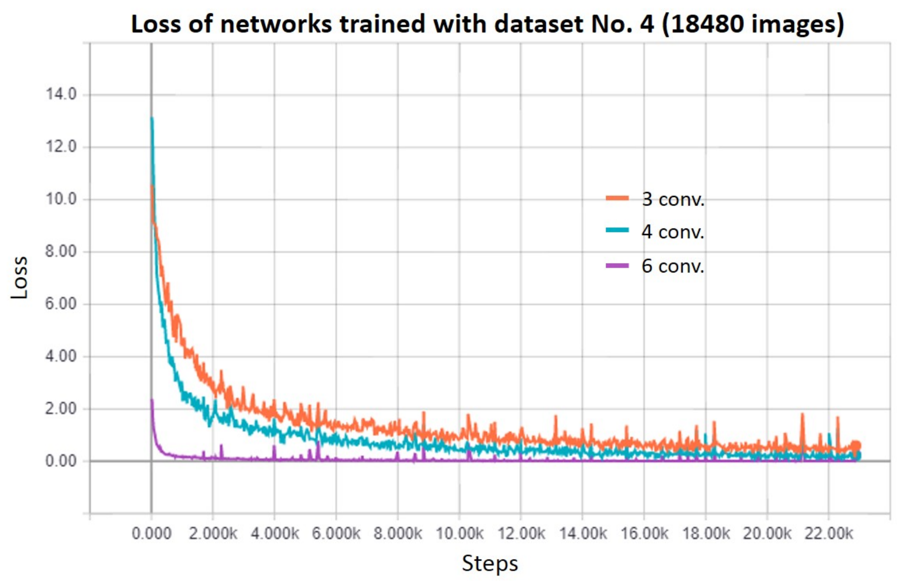
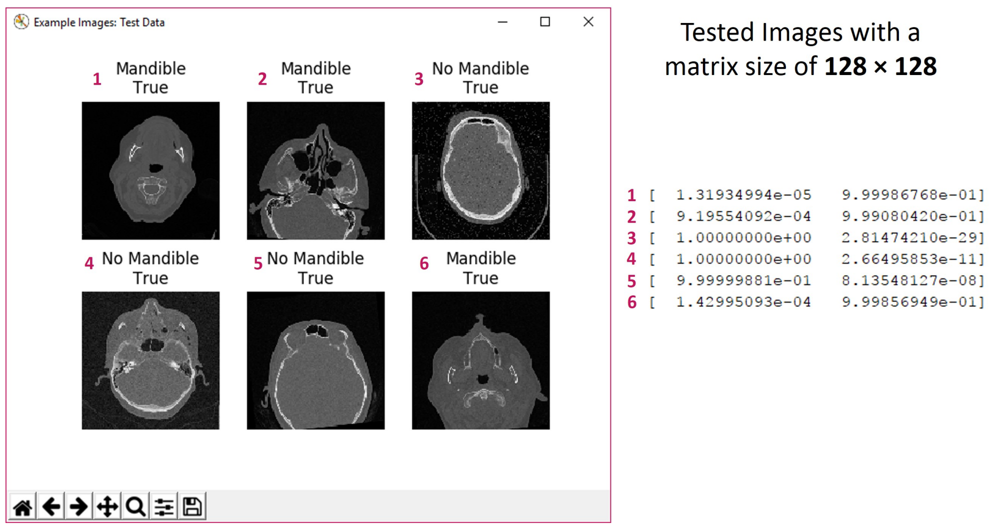


Figure 90: Loss of networks trained with dataset No. 4 (18480 images). The conv. kernel was set to three and the images had a size of 50 × 50.



```
1 [   1.31934994e-05    9.99986768e-01]
2 [   9.19554092e-04    9.99080420e-01]
3 [   1.00000000e+00    2.81474210e-29]
4 [   1.00000000e+00    2.66495853e-11]
5 [   9.99999881e-01    8.13548127e-08]
6 [   1.42995093e-04    9.99856949e-01]
```

Figure 91: Test images (128 × 128) and their predicted classes. The network had six conv./max-pooling layers and a filter size of seven.

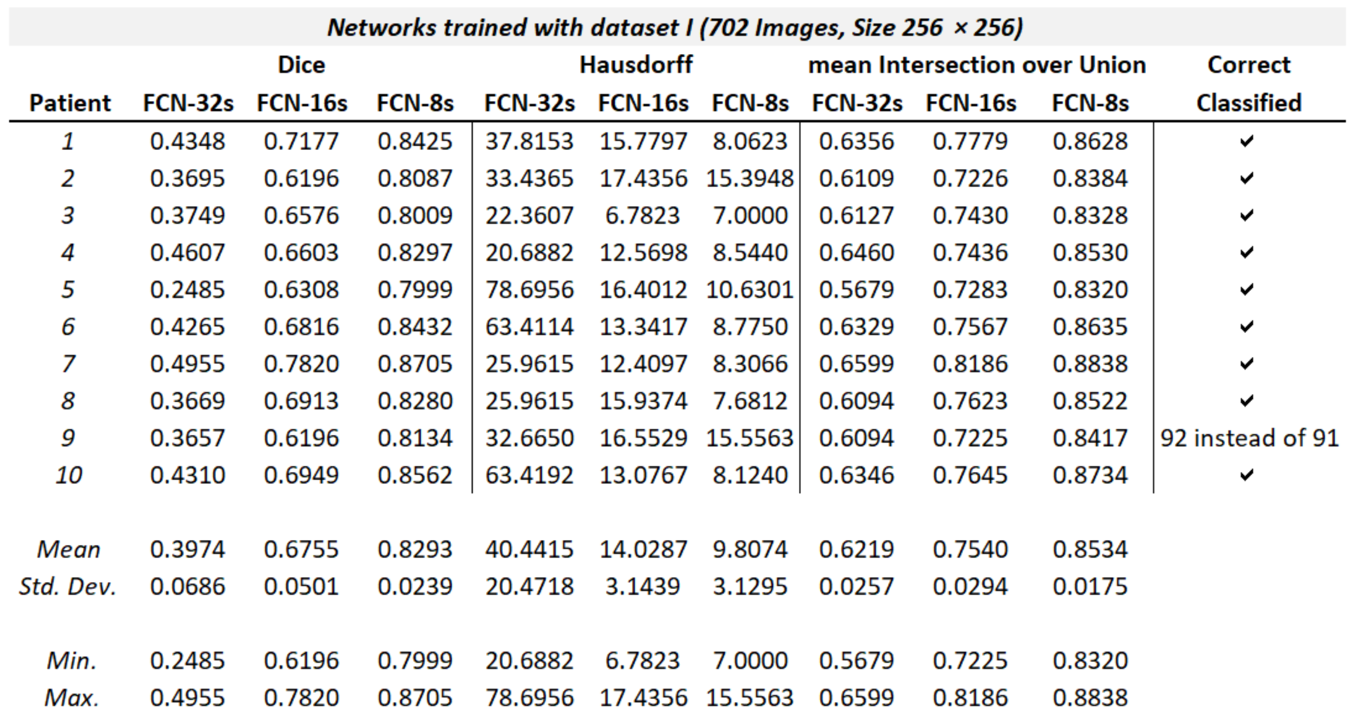
Table 23 lists the achieved classification accuracies, which can be found in Table 15 in chapter 6.2, as percentage values.

Table 23: Results of the trained classification networks in percent. The accuracies of the validation as well as of the artificial generated test dataset are listed below. Furthermore, the differences between these two accuracies are calculated.

Topo-logy	**Dataset No.**	**Conv. Kernel**	**Validation Accuracy %**	**Test Acc. %**	**Difference of Accuracies %**
			Image Size: 50×50		
3 conv. layers	1	3	93.33	82.17	11.16
	2	3	92.50	92.09	0.41
	3	3	95.00	90.58	4.42
	4	3	98.33	95.78	2.55
4 conv. layers	1	3	83.33	85.18	-1.85
	2	3	94.17	93.54	0.63
	3	3	97.86	89.09	8.77
	4	3	96.67	96.99	-0.32
6 conv. layers	1	3	96.67	85.36	11.31
	2	3	99.17	94.49	4.68
	3	3	97.86	92.10	5.76
	4	3	99.44	97.52	1.92
6 conv. layers	1	5	100.00	87.19	12.81
	2	5	98.33	96.20	2.13
	3	5	98.57	96.43	2.14
	4	5	100.00	98.48	1.52
6 conv. layers	1	7	100.00	89.69	10.31
	2	7	100.00	97.04	2.96
	3	7	99.29	94.43	4.86
	4	7	**100.00**	**98.77**	1.23
			Image Size: 128×128		
6 conv. layers	1	7	96.67	91.61	5.06
	2	7	100.00	97.77	2.23
	3	7	100.00	97.01	2.99
	4	7	**100.00**	**98.83**	1.17

8.2 Additional Segmentation Results

The subsequent Figures 92, 93, 94 and 95 display screenshots of the metrics produced by all available segmentation networks, whereby the results are evaluated per patient. The original CT slices were provided as the test images. Furthermore, the metrics are averaged over the patients as well as the standard deviations and the extrema are stated. Beyond that, the amount of correctly classified CT slices is listed.



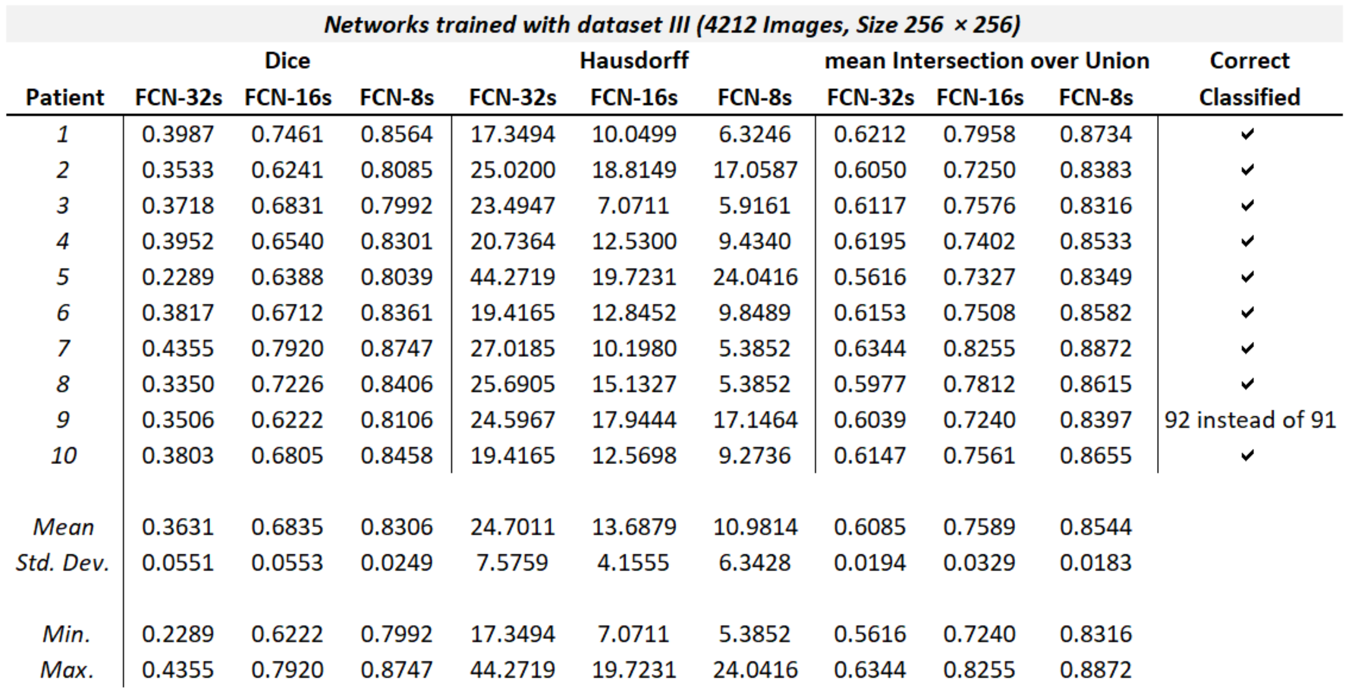
Networks trained with dataset I (702 Images, Size 256 × 256)										
	Dice			Hausdorff			mean Intersection over Union			Correct
Patient	FCN-32s	FCN-16s	FCN-8s	FCN-32s	FCN-16s	FCN-8s	FCN-32s	FCN-16s	FCN-8s	Classified
1	0.4348	0.7177	0.8425	37.8153	15.7797	8.0623	0.6356	0.7779	0.8628	✓
2	0.3695	0.6196	0.8087	33.4365	17.4356	15.3948	0.6109	0.7226	0.8384	✓
3	0.3749	0.6576	0.8009	22.3607	6.7823	7.0000	0.6127	0.7430	0.8328	✓
4	0.4607	0.6603	0.8297	20.6882	12.5698	8.5440	0.6460	0.7436	0.8530	✓
5	0.2485	0.6308	0.7999	78.6956	16.4012	10.6301	0.5679	0.7283	0.8320	✓
6	0.4265	0.6816	0.8432	63.4114	13.3417	8.7750	0.6329	0.7567	0.8635	✓
7	0.4955	0.7820	0.8705	25.9615	12.4097	8.3066	0.6599	0.8186	0.8838	✓
8	0.3669	0.6913	0.8280	25.9615	15.9374	7.6812	0.6094	0.7623	0.8522	✓
9	0.3657	0.6196	0.8134	32.6650	16.5529	15.5563	0.6094	0.7225	0.8417	92 instead of 91
10	0.4310	0.6949	0.8562	63.4192	13.0767	8.1240	0.6346	0.7645	0.8734	✓
Mean	0.3974	0.6755	0.8293	40.4415	14.0287	9.8074	0.6219	0.7540	0.8534	
Std. Dev.	0.0686	0.0501	0.0239	20.4718	3.1439	3.1295	0.0257	0.0294	0.0175	
Min.	0.2485	0.6196	0.7999	20.6882	6.7823	7.0000	0.5679	0.7225	0.8320	
Max.	0.4955	0.7820	0.8705	78.6956	17.4356	15.5563	0.6599	0.8186	0.8838	

Figure 92: Overview of the segmentation metrics produced by the networks trained with dataset I. The original CT slices were provided for testing, whilst the evaluation was accomplished separately for each patient.

Networks trained with dataset II (702 Images, Size 512 × 512)

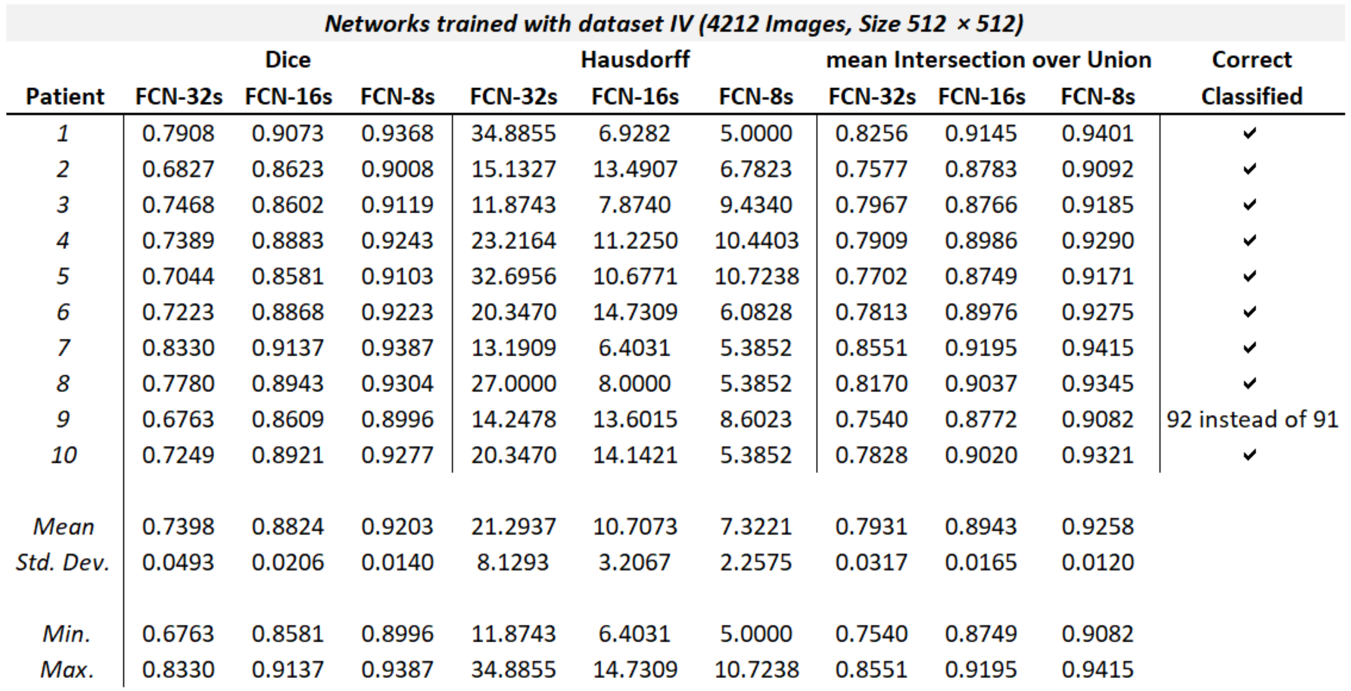
	Dice			Hausdorff			mean Intersection over Union			Correct
Patient	**FCN-32s**	**FCN-16s**	**FCN-8s**	**FCN-32s**	**FCN-16s**	**FCN-8s**	**FCN-32s**	**FCN-16s**	**FCN-8s**	**Classified**
1	0.7540	0.8850	0.9184	49.1325	7.0000	7.3485	0.8009	0.8960	0.9240	✓
2	0.6213	0.8396	0.8860	29.2233	15.9374	53.5724	0.7237	0.8610	0.8971	✓
3	0.7033	0.8410	0.8849	105.1330	96.3379	98.7168	0.7696	0.8619	0.8961	✓
4	0.6760	0.8697	0.9069	22.4944	86.3192	10.3441	0.7528	0.8836	0.9140	✓
5	0.6624	0.8442	0.9010	35.4119	14.8661	13.0767	0.7458	0.8643	0.9093	✓
6	0.6950	0.8671	0.9086	21.1187	12.1244	6.1644	0.7647	0.8819	0.9157	✓
7	0.7677	0.8991	0.9281	95.6556	7.6158	6.0000	0.8091	0.9073	0.9321	✓
8	0.7380	0.8722	0.9087	28.0891	9.2195	6.4807	0.7909	0.8859	0.9158	✓
9	0.6144	0.8385	0.8869	28.0891	16.0312	53.7215	0.7200	0.8601	0.8978	92 instead of 91
10	0.7075	0.8774	0.9187	17.7200	11.8743	5.1962	0.7721	0.8900	0.9243	✓
Mean	0.6940	0.8634	0.9048	43.2068	27.7326	26.0621	0.7650	0.8792	0.9126	
Std. Dev.	0.0519	0.0214	0.0151	31.4478	33.7524	31.9438	0.0303	0.0166	0.0125	
Min.	0.6144	0.8385	0.8849	17.7200	7.0000	5.1962	0.7200	0.8601	0.8961	
Max.	0.7677	0.8991	0.9281	105.1330	96.3379	98.7168	0.8091	0.9073	0.9321	

Figure 93: Overview of the segmentation metrics produced by the networks trained with dataset II. The original CT slices were provided for testing, whilst the evaluation was accomplished separately for each patient.



	Networks trained with dataset III (4212 Images, Size 256 × 256)									
	Dice			Hausdorff			mean Intersection over Union			Correct
Patient	FCN-32s	FCN-16s	FCN-8s	FCN-32s	FCN-16s	FCN-8s	FCN-32s	FCN-16s	FCN-8s	Classified
1	0.3987	0.7461	0.8564	17.3494	10.0499	6.3246	0.6212	0.7958	0.8734	✓
2	0.3533	0.6241	0.8085	25.0200	18.8149	17.0587	0.6050	0.7250	0.8383	✓
3	0.3718	0.6831	0.7992	23.4947	7.0711	5.9161	0.6117	0.7576	0.8316	✓
4	0.3952	0.6540	0.8301	20.7364	12.5300	9.4340	0.6195	0.7402	0.8533	✓
5	0.2289	0.6388	0.8039	44.2719	19.7231	24.0416	0.5616	0.7327	0.8349	✓
6	0.3817	0.6712	0.8361	19.4165	12.8452	9.8489	0.6153	0.7508	0.8582	✓
7	0.4355	0.7920	0.8747	27.0185	10.1980	5.3852	0.6344	0.8255	0.8872	✓
8	0.3350	0.7226	0.8406	25.6905	15.1327	5.3852	0.5977	0.7812	0.8615	✓
9	0.3506	0.6222	0.8106	24.5967	17.9444	17.1464	0.6039	0.7240	0.8397	92 instead of 91
10	0.3803	0.6805	0.8458	19.4165	12.5698	9.2736	0.6147	0.7561	0.8655	✓
Mean	0.3631	0.6835	0.8306	24.7011	13.6879	10.9814	0.6085	0.7589	0.8544	
Std. Dev.	0.0551	0.0553	0.0249	7.5759	4.1555	6.3428	0.0194	0.0329	0.0183	
Min.	0.2289	0.6222	0.7992	17.3494	7.0711	5.3852	0.5616	0.7240	0.8316	
Max.	0.4355	0.7920	0.8747	44.2719	19.7231	24.0416	0.6344	0.8255	0.8872	

Figure 94: Overview of the segmentation metrics produced by the networks trained with dataset III. The original CT slices were provided for testing, whilst the evaluation was accomplished separately for each patient.



Networks trained with dataset IV (4212 Images, Size 512 × 512)										
	Dice			**Hausdorff**			**mean Intersection over Union**			**Correct**
Patient	**FCN-32s**	**FCN-16s**	**FCN-8s**	**FCN-32s**	**FCN-16s**	**FCN-8s**	**FCN-32s**	**FCN-16s**	**FCN-8s**	**Classified**
1	0.7908	0.9073	0.9368	34.8855	6.9282	5.0000	0.8256	0.9145	0.9401	✓
2	0.6827	0.8623	0.9008	15.1327	13.4907	6.7823	0.7577	0.8783	0.9092	✓
3	0.7468	0.8602	0.9119	11.8743	7.8740	9.4340	0.7967	0.8766	0.9185	✓
4	0.7389	0.8883	0.9243	23.2164	11.2250	10.4403	0.7909	0.8986	0.9290	✓
5	0.7044	0.8581	0.9103	32.6956	10.6771	10.7238	0.7702	0.8749	0.9171	✓
6	0.7223	0.8868	0.9223	20.3470	14.7309	6.0828	0.7813	0.8976	0.9275	✓
7	0.8330	0.9137	0.9387	13.1909	6.4031	5.3852	0.8551	0.9195	0.9415	✓
8	0.7780	0.8943	0.9304	27.0000	8.0000	5.3852	0.8170	0.9037	0.9345	✓
9	0.6763	0.8609	0.8996	14.2478	13.6015	8.6023	0.7540	0.8772	0.9082	92 instead of 91
10	0.7249	0.8921	0.9277	20.3470	14.1421	5.3852	0.7828	0.9020	0.9321	✓
Mean	0.7398	0.8824	0.9203	21.2937	10.7073	7.3221	0.7931	0.8943	0.9258	
Std. Dev.	0.0493	0.0206	0.0140	8.1293	3.2067	2.2575	0.0317	0.0165	0.0120	
Min.	0.6763	0.8581	0.8996	11.8743	6.4031	5.0000	0.7540	0.8749	0.9082	
Max.	0.8330	0.9137	0.9387	34.8855	14.7309	10.7238	0.8551	0.9195	0.9415	

Figure 95: Overview of the segmentation metrics produced by the networks trained with dataset IV. The original CT slices were provided for testing, whilst the evaluation was accomplished separately for each patient.

References

[1] T. Aach and O. Dössel. *Bildgebung durch Projektionsröntgen. Biomedizinische Technik - Medizinische Bildgebung*, pages 9 – 58. Walter de Gruyter, 2014. 7, 8

[2] M. Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR*, abs/1603.04467, 2016. 30, 31

[3] MeVis Medical Solutions AG. Getting Started. Available: `http://mevislabdownloads.mevis.de/docs/current/MeVisLab/Resources/Documentation/Publish/SDK/GettingStarted/index.html`, October 2016. (Last access 17.04.2017). 12

[4] MeVis Medical Solutions AG. MeVisLab Definition Language (MDL) Reference. Available: `http://mevislabdownloads.mevis.de/docs/current/MeVisLab/Resources/Documentation/Publish/SDK/MDLReference/index.html`, October 2016. (Last access 26.04.2017). 41

[5] MeVis Medical Solutions AG. MeVisLab Reference Manual. Available: `http://mevislabdownloads.mevis.de/docs/current/MeVisLab/Resources/Documentation/Publish/SDK/MeVisLabManual/index.html`, October 2016. (Last access 17.04.2017). 12

[6] MeVis Medical Solutions AG. MeVisLab Scripting Reference. Available: `http://mevislabdownloads.mevis.de/docs/current/MeVisLab/Resources/Documentation/Publish/SDK/ScriptingReference/index.html`, October 2016. (Last access 26.04.2017). 41

[7] I. N. Bankman and J. Rogowska. *Handbook of Medical Image Processing and Analysis*, pages 71 – 90. Academic Press - Elsevier, 2009. 17

[8] T. M. Buzug. *Einführung in die Computertomographie*, pages 1 – 10, 44 – 52. Springer Verlag, 2004. 7

[9] T. M. Buzug and T. Flohr. *Computertomographie. Biomedizinische Technik - Medizinische Bildgebung*, pages 59 – 111. Walter de Gruyter, 2014. 7, 8, 10, 11, 12

[10] K. Chatfield et al. Return of the Devil in the Details: Delving Deep into Convolutional Nets. *CoRR*, abs/1405.3531, 2014. 32

[11] P. F. Christ et al. *Automatic Liver and Lesion Segmentation in CT Using Cascaded Fully Convolutional Neural Networks and 3D Conditional Random Fields*, pages 415 – 423. Springer International Publishing, 2016. 16

[12] V. Christlein and T. Würfl. *Deep Learning Tutorial - Basics*. BVM-Workshop, Heidelberg, March 2017. 23, 24, 28

[13] J. Deng et al. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248 – 255, June 2009. 20, 68

[14] T. M. Deserno. *Medizinische Bildverarbeitung. Medizintechnik*, pages 825 – 846. Springer Verlag, 2011. 17

[15] O. Dössel. *Bildgebende Verfahren in der Medizin*, pages 1 – 69, 107 – 145. Springer Verlag, 2000. 9, 10, 11, 12

[16] J. Egger et al. Integration of the OpenIGTLink Network Protocol for Image-Guided Therapy with the Medical Platform MeVisLab. *CoRR*, abs/1309.1863, 2013. 12

[17] J. Egger et al. HTC Vive MeVisLab integration via OpenVR for medical applications. *PLOS ONE*, 12(3), 03 2017. 12

[18] J. Fanghänel et al. *Waldeyer - Anatomie des Menschen*, pages 177 – 354. Walter de Gruyter, 2003. 3, 4

[19] Siemens Healthcare GmbH. SOMATOM Definition Edge. Available: `https://static.healthcare.siemens.com/siemens_hwem-hwem_ssxa_websites-context-root/wcm/idc/groups/public/@global/@imaging/@ct/documents/download/mdaw/mtyx/~edisp/somatom_definition_edge_brochure-00024835.pdf`, 2016. (Last access 06.05.2017). 9

[20] C. Gsaxner et al. Exploit 18F-FDG Enhanced Urinary Bladder in PET Data for Deep Learning Ground Truth Generation in CT Scans. *SPIE Medical Imaging*, 2017. Accepted. 41

[21] S. Guadarrama and N. Silberman. TensorFlow-Slim. Available: `https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim`, 2017. (Last access 27.09.2017). 19, 31, 68

[22] H. Handels. *Medizinische Bildverarbeitung: Bildanalyse, Mustererkennung und Visualisierung für die computergestützte ärztliche Diagnostik und Therapie*, pages 63, 95 – 156, 220 – 249. Vieweg + Teubner Verlag, 2009. 17, 18, 21, 22

[23] A. W. Harley. An Interactive Node-Link Visualization of Convolutional Neural Networks. In *ISVC*, pages 867 – 877, 2015. 26

[24] B. Ibragimov and L. Xing. Segmentation of organs-at-risks in head and neck CT images using convolutional neural networks. *Medical Physics*, 44(2):547 – 557, 2017. 7, 16, 18, 19, 33, 39, 54, 110

[25] Google Inc. Google Trends. Available: `https://trends.google.at/trends/explore?q=Deep%20Learning`, 2017. (Last access 23.07.2017). 1

[26] Google Inc. TensorFlow. An open-source software library for Machine Intelligence. Available: `https://www.tensorflow.org/`, 2017. (Last access 05.10.2017). 30, 68, 75

[27] Kaggle Inc. kaggle. Available: `https://www.kaggle.com/`, 2016. (Last access 04.07.2017). 20

[28] Wikimedia Foundation Inc. Wikipedia Die freie Enzyklopädie - Gefilterte Rückprojektion. Available: `https://de.wikipedia.org/wiki/Gefilterte_R%C3%BCckprojektion`, 2016. (Last access 15.06.2017). 40

[29] Wikimedia Foundation Inc. Wikipedia Die freie Enzyklopädie - Röntgen. Available: `https://de.wikipedia.org/wiki/R%C3%B6ntgen`, 2017. (Last access 24.05.2017). 8

[30] B. Jähne. *Digitale Bildverarbeitung*, pages 541 – 554. Springer Verlag, 2012. 16

[31] N. Jones. Computer science: The learning machines. *Nature*, 505(7482):146 – 148, January 2014. 1, 63

[32] U. Karrenberg. *Neuronale Netze. Signale - Prozesse - Systeme*, pages 443 – 476. Springer Vieweg, 2012. 21, 60

[33] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural*

Information Processing Systems 25, pages 1097 – 1105. Curran Associates, Inc., 2012. 32

[34] R. Kruse et al. *Computational Intelligence.* Springer Vieweg, 2015. 20

[35] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436 – 444, May 2015. 20, 21

[36] T. Lehmann et al. *Bildverarbeitung für die Medizin*, pages 359 – 394. Springer Verlag, 1997. 16

[37] M. Lenzen. *Natürliche und künstliche Intelligenz*, page 88. Campus Verlag, 2002. 20

[38] J. Long, E. Shelhamer, and T. Darrell. Fully Convolutional Networks for Semantic Segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015. 34, 35, 68, 72, 79

[39] Fraunhofer MEVIS. Fraunhofer MEVIS Forum. Available: `https://forum.mevis.fraunhofer.de/index.php`, 2016. (Last access 19.04.2017). 45

[40] Microsoft. CNTK. Available: `https://github.com/Microsoft/CNTK`, September 2017. (Last access 14.09.2017). 30

[41] Microsoft. The Microsoft Cognitive Toolkit. Available: `https://docs.microsoft.com/en-us/cognitive-toolkit/`, July 2017. (Last access 14.09.2017). 30

[42] P. Mildenberger, M. Eichelberg, and E. Martin. Introduction to the DICOM standard. *European Radiology*, 12(4):920 – 927, April 2002. 39

[43] A. Nischwitz et al. *Computergrafik und Bildverarbeitung*, pages 458 – 482. Vieweg + Teubner, 2011. 22, 63

[44] National Library of Medicine Insight Segmentation and Registration Toolkit (ITK). itk::HausdorffDistanceImageFilter. Available: `https://itk.org/Doxygen/html/classitk_1_1HausdorffDistanceImageFilter.html`, 2017. (Last access 28.09.2017). 19

[45] D. Pakhomov. TF Image Segmentation: Image Segmentation framework. Available: `https://github.com/warmspringwinds/tf-image-segmentation`, May 2017. (Last access 30.09.2017). 29, 68, 75

[46] D. Pakhomov et al. Deep Residual Learning for Instrument Segmentation in Robotic Surgery. *arXiv preprint arXiv:1703.08580*, 2017. 29, 68

[47] B. Pfarrkirchner. Data Generation. Available: `https://github.com/birgitPf/Data_Generation`, October 2017. (Last access 17.10.2017). 41

[48] B. Pfarrkirchner. Lower Jawbone Data Generation For Deep Learning Tools. Masterproject, June 2017. 12, 41

[49] B. Pfarrkirchner et al. Lower Jawbone Data Generation for Deep Learning Tools under MeVisLab. *SPIE Medical Imaging*, 2017. Accepted. 41

[50] PythonProgramming. Classifying Cats vs Dogs with a Convolutional Neural Network on Kaggle. Available: `https://pythonprogramming.net/convolutional-neural-network-kats-vs-dogs-machine-learning-tutorial/`, August 2017. (Last access 21.08.2017). 57

[51] BAIR (Berkeley Artificial Intelligence Research). Caffe. Available: `http://caffe.berkeleyvision.org/`, 2017. (Last access 14.09.2017). 30

[52] J. Schmidhuber. Deep Learning in Neural Networks: An Overview. *CoRR*, abs/1404.7828, 2014. 20

[53] M. Schünke, E. Schulte, and U. Schumacher. *PROMETHEUS Kopf, Hals und Neuroanatomie*, pages 1 – 60. Georg Thieme Verlag, 2009. 3, 4, 6

[54] P. Y. Simard, D. Steinkraus, and J. C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.*, pages 958 – 963, August 2003. 32

[55] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*, abs/1409.1556, 2014. 31, 35

[56] S. Srinivas et al. *An Introduction to Deep Convolutional Neural Nets for Computer Vision. Deep Learning for Medical Image Analysis*, pages 25 – 52. Academic Press - Elsevier, 2017. 20, 26, 28, 29, 30

[57] H. Suk. *An Introduction to Neural Networks and Deep Learning. Deep Learning for Medical Image Analysis*, pages 3 – 24. Academic Press - Elsevier, 2017. 20, 21, 25, 26, 28, 59

[58] TFlearn. TFLearn: Deep learning library featuring a higher-level API for TensorFlow. Available: `http://tflearn.org/`, August 2017. (Last access 29.09.2017). 31, 57

[59] J. Walde. *Design Künstlicher Neuronaler Netze*, pages 46 – 47. Deutscher Universitäts-Verlag, 2005. 62

[60] T. Würfl. *Deep Learning Tutorial - Weakly- and Unsupervised Deep Learning*. BVM-Workshop, Heidelberg, March 2017. 1

[61] J. Yangqing et al. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675 – 678. ACM, 2014. 30

[62] D. Yu et al. An Introduction to Computational Networks and the Computational Network Toolkit. Technical report, October 2014. 30