Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of ICT and Natural Sciences

Jørgen André Sperre

Segmentation of Knee Joint Using 3D Convolutional Neural Networks

Master's thesis in Simulation and Visualization
Supervisor: Kjell-Inge Gjesdal, Robin Trulssen Bye
June 2020

NTNU
Norwegian University of Science and Technology

Jørgen André Sperre

Segmentation of Knee Joint Using 3D Convolutional Neural Networks

Master's thesis in Simulation and Visualization
Supervisor: Kjell-Inge Gjesdal, Robin Trulssen Bye
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of ICT and Natural Sciences

Abstract

Deep learning techniques have become increasingly popular for medical image segmentation tasks in recent years. This study utilises a 3D convolutional neural network (CNN) called nnU-Net, for the task of automatic semantic segmentation of 13 classes in magnetic resonance (MR) images of the knee joint. Experimentation of various hyper-parameters are used to improve the accuracy of the models, and in the process generate a comparison of the impact from the various hyper-parameters. Models were trained and evaluated on a training dataset consisting of 20 subjects and a validation dataset consisting of 5 subjects, with three different image modalities for each subject. Evaluation of the models found that the nnU-Net architecture was able to accurately segment the knee joint. Hyper-parameter experimentation found that the only improvement was a minor increase in accuracy when adding data augmentation to the model.

Sammendrag

Dyp læring teknikker har hatt økende popularitet for medisin-relaterte segmenterings oppgaver de siste årene. Denne studien bruker ett 3D konvolusjonelt nevralt nettverk (CNN) kalt nnU-Net, for å automatisk semantisk segmentere 13 klasser fra magnetisk resonans (MR) bilder av kneledd. Eksperimentering av forskjellige hyper-parametere er brukt for å forbedre nøyaktigheten til de opptrente modellene, og i prosessen lage en sammenligning av effekten av disse parametrene. Modeller var trent og evaluert på et treningsdatasett som bestod av 20 pasienter, og et valideringsdatasett som bestod av 5 pasienter, med tre forskjellige bilde-modaliteter for hver pasient. Evaluering av modellene fant at nnU-Net arkitekturen var i stand til å lage nøyaktige segmenteringer av kneleddet. Hyper-parameter eksperimenteringen fant at den eneste forbedringen var en liten økning i nøyaktighet, dersom ”data augmentering” ble tatt i bruk.

Preface

This Master's thesis is submitted as a final deliverable work of the Simulation and Visualization Master's program at the Department of ICT and Engineering at Norwegian University of Science and Technology (NTNU) in Ålesund. The work presented in this thesis was carried out in the final semester throughout the spring semester of 2020. The thesis was performed in collaboration with Sunnmøre MR-Klinikk, who supplied the necessary datasets.

The thesis concerns the automatic semantic segmentation of magnetic resonance images of the knee joint. The motivation for choosing this thesis was personal interest in the related field of machine learning and computer vision, and a desire to continue exploring the research field that I was introduced to during the fall semester of 2019.

I would like to thank Sunnmøre MR-Klinikk for providing the data for this thesis, and also further thank Carl Petter Skaar Kulseng for his valuable guidance and support in carrying out the work throughout this thesis. I would also like to thank both Kjell-Inge Gjesdal and Robin Trulssen Bye for their guidance and feedback for writing this thesis. Lastly, I would also like to thank my family for all their continued support and encouragement throughout my studies.

Table of Contents

List of Tables

List of Figures

Acronyms

ACL anterior cruciate ligament 5, 30, 31, 35

AI artificial intelligence xvi, 1, 10, *Glossary:* artificial intelligence

ANN artificial neural network xv, xvi, 5, 10–12, 15, 16, 18, 20, 33, 41, *Glossary:* artificial neural network

API application programming interface 33, *Glossary:* application programming interface

AR augmented reality 2, 35

BGD batch gradient descent 16

CE cross entropy 21, 22, 30

CNN convolutional neural network xi, 3, 5, 9, 18–21, 27, 28, 30, 31, 97, 98, *Glossary:* convolutional neural network

CPU central processing unit xi, 33, 40, 48, 92, *Glossary:* central processing unit

CRF conditional random field 30

DL deep learning 18, *Glossary:* deep learning

DSP digital signal processing 18, *Glossary:* digital signal processing

FCN fully connected neural network 28, 30

FN false negative 24

FP false positive 24

FS fat saturation xi, 7, 8, 34, *Glossary:* fat saturation

GPU graphics processing unit xi, xii, xvii, 18, 33, 40, 48–50, 92, *Glossary:* graphics processing unit

k-NN k-nearest neighbours 30, *Glossary:* k-nearest neighbours

LCL lateral collateral ligament 5

MCL medial collateral ligament 5

ML machine learning xv, xvi, 1–3, 5, 9, 10, 16, 18, 34, 94, *Glossary:* machine learning

MR magnetic resonance 3, 8, 9, 28, 30, 31, 35

MRI magnetic resonance imaging xv, xvi, 2, 3, 5–9, 28, 30, 34, 97, 98, *Glossary:* magnetic resonance imaging

MSE mean square error xvi, 15, *Glossary:* mean square error

NTNU Norges teknisk-naturvitenskapelige universitet 2

OA osteoarthritis 30, 31, *Glossary:* osteoarthritis

OOM out of memory 40, 48, 50, 91, 92

PCL posterior cruciate ligament 5, 30, 31, 35

PD proton density xi, 7, 8, 34, *Glossary:* proton density

PReLU parametric rectified linear unit 15

RAM random access memory xvii, 33, 40, *Glossary:* random access memory

ReLU rectified linear unit xi, 14, 15

RF radio wave frequency 6, 7

ROI region of interest 1, 31

SGD stochastic gradient descent 16

TanH hyperbolic tangent xi, 13, 14

TN true negative 24

TP true positive 24

VR virtual reality 2, 35

VRAM video random access memory 33, 40, *Glossary:* video random access memory

Glossary

application programming interface An API is a set of definitions and protocols that allows a software to access services and resources provided by another software that implements the same API. xiii, 33,

artificial intelligence Computer systems that attempts to imitate human behaviour in order to perform tasks that would normally require human intelligence xiii, 1,

artificial neural network ”Artificial neural networks (ANNs) are statistical models where the mathematical structure reproduces the biological organisation of neural cells simulating the learning dynamics of the brain” [11]. xiii, xv,

central processing unit The CPU is the primary component of a computer that processes instructions. It runs the operating system and applications, constantly receiving input from the user or active software programs. It processes the data and produces output, which may stored by an application or displayed on the screen [12]. xiii,

convolutional neural network Convolutional neural network (CNN) is a specific type of ANN that includes convolutional layers and pooling layers occuring in alternating fashion. CNNs are well suited for image recognition tasks due to sparse connectivity, parameter sharing, subsampling, and local receptive fields rendering them invariant to shifting, scaling, and distortions of input data [13]. xiii,

deep learning Deep learning is a subtopic within ML, consisting of artificial neural networks (ANNs) with a high number of layers, resulting in a deep network. xiii, 18,

digital signal processing Digital signal processing (DSP) is the process of analyzing and modifying a signal to optimize or improve its efficiency or performance. It involves applying various mathematical and computational algorithms to analog and digital signals to produce a signal that’s of higher quality than the original signal [14]. xiii, 18,

fat saturation Fat saturation (FS) MRI images is a technique used to suppress the signal from adipose tissue (fat). xiii,

graphics processing unit A GPU is a processor designed to handle graphics operations. The primary purpose of a GPU is to render 3D graphics. GPUs are effective when performing the same mathematical operations on a large number of data, making them very efficient for AI applications such as training ANNs. xiv,

k-nearest neighbours K-nearest neighbours (k-NN) is a classification algorithm based on similarity measures. An object is classified based on a plurality vote of its neighbours, in which the object is assigned to the class that is most common amongst its neighbours. xiv, 30,

machine learning ”Machine learning (ML) is an application of AI that provides systems the ability to automatically learn and improve from experience without being explicitly programmed” [15]. xiv, 1,

magnetic resonance imaging Magnetic resonance imaging (MRI) is a non-invasive medical imaging method that uses a strong magnetic field and radio waves to generate images of the body [16]. xiv, 2,

mean square error Mean square error is a commonly used loss function for regression problems, the exact formula for MSE is shown in equation 2.6. xiv, 15,

osteoarthritis Osteoarthritis (OA) is the most common form of arthritis, and is casued by the protective cartilage cushions on the ends of bones getting worn down. xiv, 30,

overfitting Overfitting is a problem in ML categorised by the model adjusting itself too closely to the training data [17]. 14

proton density Proton density (PD) MRI images produce contrast by minimizing the impact of and differences [18]. xiv,

radiology Radiology is a medical field that uses various imaging technologies to diagnose and treat patients. 6

random access memory RAM is the amount of memory available to the operating system and applications on a device. RAM is a memory type that has high access speed. xiv, 33,

segmentation Segmentation is a big subfield within digital image processing. It is the process of dividing an image into regions with similar properties, such as colour and texture [17]. xvi, 9, 10

semantic segmentation Semantic segmentation is a specific type of segmentation where each pixel (or voxel for 3D images) is given a class label. 10

software agent In computer science, a software agent is a computer program that acts for a user or another program in a relationship of agency. Agents may be embodied, as when paired with a robot body, or simply as software [19]. 10

T1 T1 relaxation is the time taken for the magnetic vector to return to its resting state. xi, xvi, 7, 31, 34

T2 T2 relaxation is the time needed for the axial spin to return to its resting state. xvi, 7, 31

video random access memory VRAM is a specific type of RAM that is used to store image and video data. The GPU is able to read data from VRAM at significantly higher speeds than standard RAM. xiv, 33,

Chapter 1

Introduction

This chapter will present the background and motivation for this thesis in section 1.1. Additionally, the scope and objectives of the thesis are described in section 1.2 and 1.3. And lastly, the structure of the thesis will be detailed in section 1.4.

1.1 Background & motivation

Medical imaging is one of the most central and important elements in medical practice in this day and age [20]. One of the main applications of medical imaging is to aid in making a diagnosis or to confirm a suspected diagnosis. Medical imaging has also assumed an increasingly important role in surgery, such as by allowing examination of the surgery area beforehand [21]. The rapid increase in both hardware and software capabilities in the last decade has led to an increasing interest in the application of computer vision algorithms in these medical imaging tasks.

One of the most interesting and challenging tasks related to computer vision is segmentation. Segmentation is the process of dividing an image into regions with similar properties. ”Image segmentation is considered the most essential medical imaging process as it extracts the region of interest (ROI) through a semiautomatic or automatic process” [22]. At the moment, most segmentation is performed manually by radiologists. This is a tedious and time-consuming task, which is both mentally and physically straining. This has popularised the adaptation of artificial intelligence (AI) solutions, leading to an ever-increasing trend for machine learning (ML) to be the dominant technique in medical image segmentation [23].

ML methods for segmentation tasks have been utilised for a range of various anatomical datasets, including the brain [24], lungs [25] [26], liver [27], and more. Moreover, a systematic review published in September 2019 [28] compared the performance of deep learning approaches to that of health-care professionals on the detection of disease from medical imaging, and concluded with “deep learning algorithms to have equivalent sensitivity and specificity to health-care professionals.”

Norges teknisk-naturvitenskapelige universitet (NTNU), Sunnmøre MR-Klinikk, and Ålesund Sjukehus are cooperating on a collaboration project, with a final goal of creating a training simulator for surgeons. This goal includes automatic 3D segmented models, with the ability to interact with these models in virtual reality (VR) and augmented reality (AR) with physical and virtual tools. This thesis focuses on the automatic segmentation part of this larger project. This part of the project is a critical component of the project, due to providing the models required for the rest of the project. This segmentation part of the project, and by extension this thesis, will primarily be a collaborative effort with Sunnmøre MR-Klinikk.

1.2 Thesis scope

The scope of this thesis is to automatically segment anatomical regions of interest from magnetic resonance imaging (MRI) images, through the use of ML techniques. The scope is further limited to the training and validation of neural networks for this purpose. The network utilised in this thesis will be the nnU-Net module within Niftynet, and the segmentation will focus on the knee joint. The training aspect of this thesis will include attempts to optimise the hyper-parameters for the neural network, as well as generate comparable results from the various hyper-parameters.

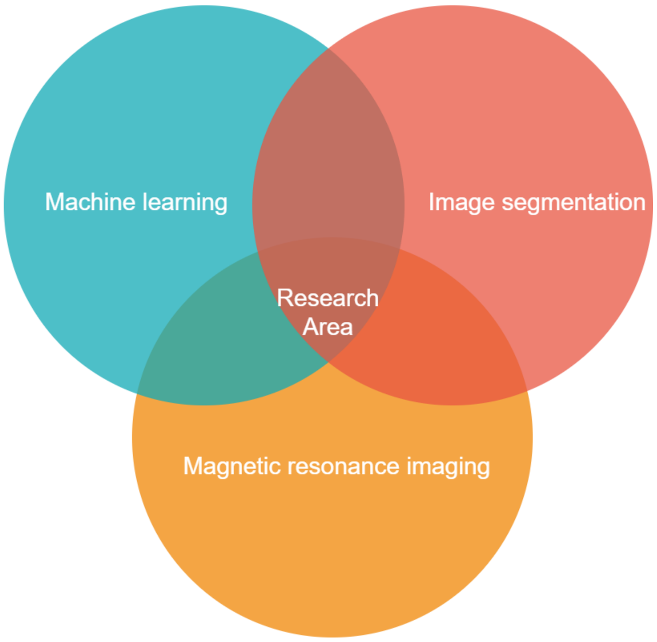


Figure 1.1: Venn diagram of thesis scope

1.3 Goals and objectives

The main goal of this thesis is to train a neural network model that is able to accurately segment new unknown magnetic resonance (MR) images of knees, with sufficiently high accuracy. Furthermore, the experimentation of various hyper-parameters, and their effect on the training and inference processes, will play an important role in the thesis. The thesis is therefore divided into the following two research questions:

Research question 1: *Do the trained neural networks generate a segmentation output of sufficient accuracy?*

The accuracy of the models is determined by a combination of visual inspection and evaluation metrics. Evaluation metrics are usually based on a comparison between the segmented output and a ”ground truth”. This ”ground truth” is generated manually by a human expert, and is therefore subject to both errors and subjectivity. This is why visual inspection in addition to evaluation metrics are essential to evaluate the accuracy.

Research question 2: *What impact do the hyper-parameters have on the training process and inferred segmentation output?*

In the process of achieving sufficiently high accuracy, various hyper-parameters will be subject to experimentation. In order to choose the best parameters, their effect on the network must be both evaluated and understood. Therefore, in an effort to understand the functionality and impact of each hyper-parameter, an overview and comparison of the various experimentation carried out during this thesis will be presented.

1.4 Thesis structure

The thesis is structured as follows:

Chapter 1 - Introduction: Presents the motivation, scope, and objectives of this thesis.

Chapter 2 - Theory: Describes the relevant theory for this thesis. This includes theory about knee anatomy, MRI, segmentation, and ML.

Chapter 3 - Related work: Explores works that are relevant to this thesis. This chapter gives a summary of the state of ML approaches in the medical image segmentation field.

Chapter 4 - Methodology: Presents the methodology used in this thesis. This includes the utilised hardware and software, datasets, and CNN. This chapter also introduces the hyper-parameter optimisation process and describes the baseline configuration of the hyper-parameters.

Chapter 5 - Experiments: Describes the hyper-parameter experimentation that is carried out in this thesis, including reasoning and hypotheses for the performed experimentation.

Chapter 6 - Results: Presents the results of the hyper-parameter experimentation. Results are presented both as visual segmentation masks and evaluation metrics.

Chapter 7 - Discussion: Discusses the results and methodology of the thesis.

Chapter 8 - Conclusion: Contains the conclusion of this thesis. The conclusion answers the previously stated research questions, states the contributions of this thesis, and presents ideas for future work.

Chapter 2

Theory

This chapter presents background theory related to the work carried out in the thesis. First, the anatomy of a knee and MRI are briefly explained. Then the topics of segmentation and deep learning are explored more in-depth.

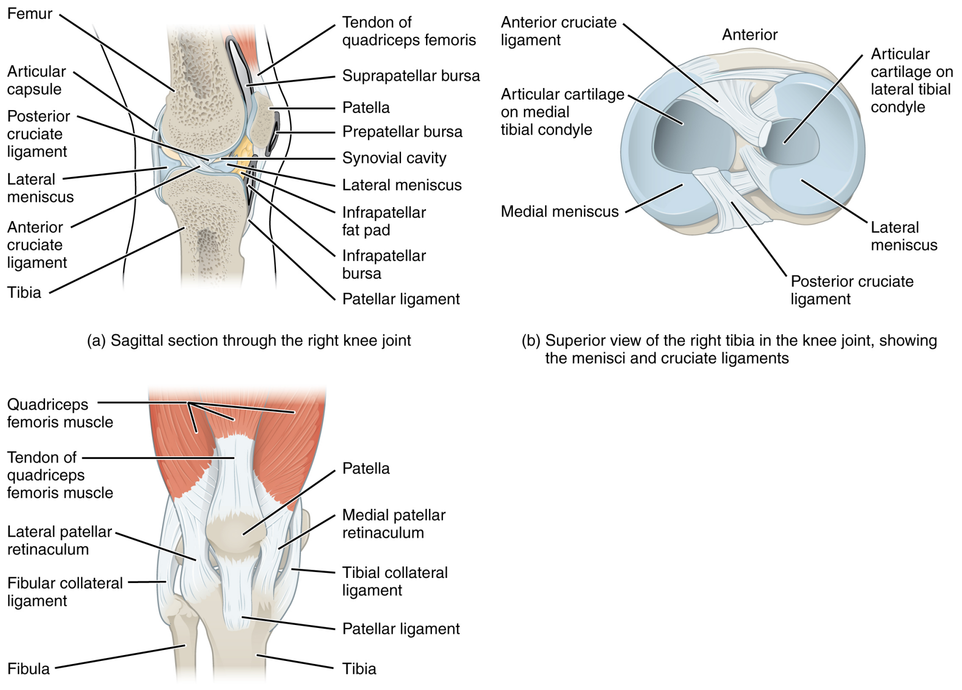
Section 2.1 briefly explains the anatomy of the knee joint. Section 2.2 looks into imaging techniques and different image types for MRI as well as file formats for MRI images. Section 2.3 explores a variety of segmentation methods and explains the specifics of semantic segmentation. Section 2.4 Gives an introduction to ANNs, including structure and important features. Section 2.5 explains how CNNs function, and details why they are effective for semantic segmentation. Section 2.6 explores different loss functions commonly used in semantic segmentation tasks. Section 2.7 explains how data augmentation is done, and what purpose it serves. And lastly, section 2.8 will look into different evaluation metrics for ML and medical imaging tasks.

2.1 Anatomy of the knee

The knee joint connects the thigh and the shin and is one of the largest and most complex joints in the human body. The knee is often viewed as two joints that collectively function as a hinge joint, allowing both flexion and extension as well as small amounts of rotation. The knee consists of four main bones namely the femur (thigh bone), tibia (shin bone), fibula (calf bone), and patella (knee cap).

The leg muscles are connected to the bones with tendons to allow for movement, while ligaments join the bones together while also providing stability to the knee. The anterior cruciate ligament (ACL) and posterior cruciate ligament (PCL) prevents the femur and tibia from sliding backwards or forwards, while the medial collateral ligament (MCL) and lateral collateral ligament (LCL) prevents the femur from sliding side to side. In addition to this, there are also two C-shaped pieces of cartilage called the medial and lateral menisci that act as shock absorbers between the femur and tibia. [29]

In addition to the above-mentioned bones and ligaments which are shown in figure 2.1, the knee and its surrounding region also consist of veins, arteries, muscles, and fat.


Femur
Articular capsule
Posterior cruciate ligament
Lateral meniscus
Anterior cruciate ligament
Tibia
Tendon of quadriceps femoris
Suprapatellar bursa
Patella
Prepatellar bursa
Synovial cavity
Lateral meniscus
Infrapatellar fat pad
Infrapatellar bursa
Patellar ligament
Anterior cruciate ligament
Anterior
Articular cartilage on medial tibial condyle
Medial meniscus
Articular cartilage on lateral tibial condyle
Lateral meniscus
Posterior cruciate ligament
Quadriceps femoris muscle
Tendon of quadriceps femoris muscle
Lateral patellar retinaculum
Fibular collateral ligament
Fibula
Patella
Medial patellar retinaculum
Tibial collateral ligament
Patellar ligament
Tibia

(a) Sagittal section through the right knee joint

(b) Superior view of the right tibia in the knee joint, showing the menisci and cruciate ligaments

(c) Anterior view of right knee

Figure 2.1: Anatomy of the knee [1]

2.2 Magnetic resonance imaging

MRI is one of the most widely used imaging techniques within radiology. The theory behind MRI is well documented in various sources. This section provides a summary of the information presented in [30], which has also been briefly summarised in [17]. A more detailed and illustrated explanation for MRI is presented in [31].

MRI uses the natural magnetic properties of the body to produce images. The hydrogen nucleus (a single proton) is used for imaging purposes because it is found in abundance in water and fat. These hydrogen nuclei have an axial spin, with their axes randomly aligned. When the body is placed in a strong magnetic field, such as an MRI scanner, the protons' axes all align. This uniform alignment creates a magnetic vector oriented along the axis of the MRI scanner.

This magnetic vector is then deflected by adding additional energy (in the form of a radio wave) to the magnetic field. The radio wave frequency (RF) is determined by the sought element (usually hydrogen) and the strength of the magnetic field generated by the MRI scanner. The strength of the magnetic field can be altered electronically from head to toe using a series of gradient electric coils. Thus, by altering the local magnetic field by these small increments, different slices of the body will resonate as different frequencies are applied.

When the RF source is switched off, the magnetic vector returns to its resting state. This causes a signal (in the form of a radio wave) to be emitted from the affected nuclei. It is this resulting radio wave signal which is used to create MRI images. Receiver coils are placed around the body part that is imaged to improve the detection of the emitted signal. The intensity of the received signal is then plotted on a grey scale, and cross-sectional images are generated.

Additionally, there is a difference in how quickly different tissue relax once the RF pulse is switched off. These times are measured in the following two ways. T1 relaxation is the time taken for the magnetic vector to return to its resting state. And T2 relaxation is the time needed for the axial spin to return to its resting state.

”There are no known biological hazards of MRI because, unlike x-ray and computed tomography, MRI uses radiation in the radiofrequency range which is found all around us and does not damage tissue as it passes through.” [30]

2.2.1 Image types

MRI can produce different images depending on the weighting of T1 and T2 relaxation times. Because different tissues have different relaxation times, the weighing can be used to create differences in signal intensities and by extension tissue grey levels. The datasets used in this thesis has three differently weighted images for each patient. These are T1, PD, and FS weighted images.

T1 images

T1 images present the difference in T1 relaxation times. T1 images are useful for identifying fluid filled spaces in the body. Fat appears very bright in these images, while fluid is dark.

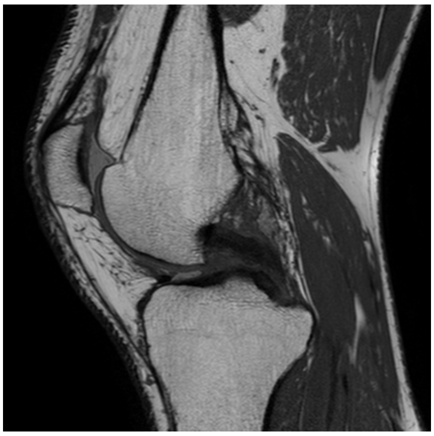


Figure 2.2: Example of an T1 weighted image

PD images

In a PD weighted MR image, it is the tissues with a higher concentration/density of protons (hydrogen nuclei) which produce the strongest signals, and thus appears the brightest. [18]

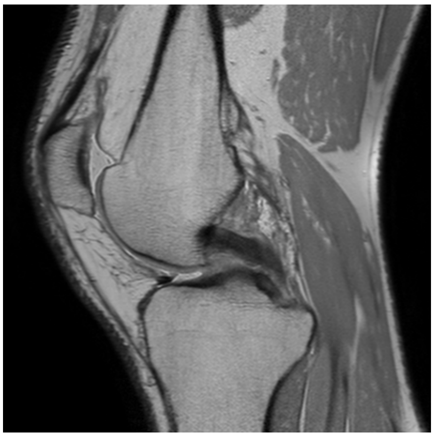


Figure 2.3: Example of an PD weighted image

FS images

FS images are used to suppress the signal from normal adipose tissue. The result is that adipose tissue appears darker, while any other tissue appears brighter by contrast. [32]

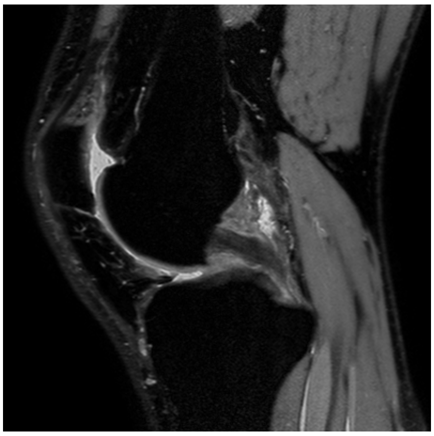


Figure 2.4: Example of an FS weighted image

2.2.2 Image formats

There are a lot of different MRI file formats. The four most commonly used are Analyze, Nifti, Minc, and Dicom. Dicom is designed to standardize the generated images by diagnostic modalities. While the other 3 aim at facilitating and strengthening post-processing analysis. [33]

Nifti

In this thesis, the datasets were supplied as Nifti files. This format can be seen as a revised Analyze format. The notable improvements include updated header information such as rotation and orientation. Nifti also includes support for additional data types, such as unsigned 16-bit. [33]

2.3 Segmentation

Segmentation is a large subfield within digital image processing. Segmentation is a task that aims to divide an image into regions with similar properties, such as colour or texture [34]. Segmentation techniques range from the simple threshold method to the more advanced edge detection and clustering techniques, and also includes various ML algorithms.

Conventional segmentation algorithms often rely on a critical selection of parameters, for instance, to derive an accurate membership function in the case of clustering. This requires a considerable amount of user expertise [35]. These aspects are simply not practical when it comes to more advanced segmentation tasks such as segmenting multiple structures, especially when dealing with complex 3-dimensional structures such as those created by MRI [36] [17].

There are three main difficulties when it comes to segmentation tasks:

- **Noise:** Noise during the data generation process can potentially alter the intensity of either a singular pixel or a group of pixels, resulting in the classification becoming uncertain.

- **Low variety of pixel intensity between classes:** When segmenting multiple classes within the same image, the different classes need to be distinguished somehow. If the variety of pixel intensity between different classes is very low, then they become almost indistinguishable.

- **Class imbalance:** When an image contains classes of varying sizes, the smaller classes are easily ignored during training due to the low impact they have on the overall accuracy of the segmentation task.

The first two of these difficulties are related to the data generation. Some amount of noise is always going to be present during an MRI scan.

The intensity of pixels can have increased variety by utilizing different weightings for the MR image generation. However, if multiple classes are made up of the same or similar tissue, such as tendons and ligaments, then this problem becomes unavoidable. This problem can then only be solved by considering the spatial information of the image, as opposed to strictly the image intensity. This is something that CNNs are especially well suited for, due to their local receptive field which will be discussed more in detail in section 2.5.6.

The class imbalance is also unavoidable during data generation, but can be handled by ANNs through the choice of the loss function, which is briefly mentioned in section 2.4.2 and discussed in detail in section 2.6

Semantic segmentation is the main focus for this thesis, and it is a specific type of segmentation where each pixel (or voxel for 3D images) is given a class label. This task is also often called dense prediction or dense semantic segmentation.

2.4 Artificial neural networks

ANNs is a subfield of AI and ML that is inspired by neuroscience. The goal of ANNs is to replicate the way neurons work in the human brain. ANNs and other ML algorithms are often categorised into four different categories, based on the way they learn:

- **Supervised learning:** Supervised learning is used when we have a dataset of labelled data. Labelled data means that each sample of the dataset also has a corresponding answer that we would like the algorithm to come up with. The algorithm is then able to compare the solution it finds with the label for each data sample, and in that way evaluate how good its solution is.

- **Unsupervised learning:** For unsupervised learning the dataset does not include labels. This means that the algorithm does not know the correct answer for its training data. This approach is most useful for clustering (finding similarities in the data), or anomaly detection.

- **Semi-supervised learning:** This approach is a combination of both supervised and unsupervised learning. This is especially useful for difficult data, when the labelling of a dataset is a very time-consuming task for experts. Another potential benefit is that this allows the algorithm to reach its own conclusions, without introducing potential bias or inaccuracies through manually labelling the data.

- **Reinforcement learning:** Reinforcement learning does not use a training dataset. Instead, the algorithm uses a reward system. The algorithm is trained by an iterative process in which it tries to maximise its cumulative reward. This approach is often used for software agent, for tasks such as path planning, robot motion control, business management, and more.

This thesis will focus on supervised learning algorithms. The dataset that has been supplied for this thesis work consists solely of labelled data, and there is not any unlabelled data available to facilitate semi-supervised learning. This is because the samples in this dataset are unique and generated specifically to be used for training neural networks. What this entails is discussed in more detail in section 4.2.

2.4.1 Structure

ANNs consists of nodes and links, where the nodes act like neurons to propagate values forward to linked nodes when activated. Each node consists of weighted inputs and computes its value as a weighted sum of all its inputs when activated. Additionally, each node typically has a bias that adds a static value to it's propagated value. ANNs consists of an input and an output layer, with any number of hidden layers in between. The input for each layer is the output of the previous layer, where the first layer is directly connected to the input data [17] [37] [38].

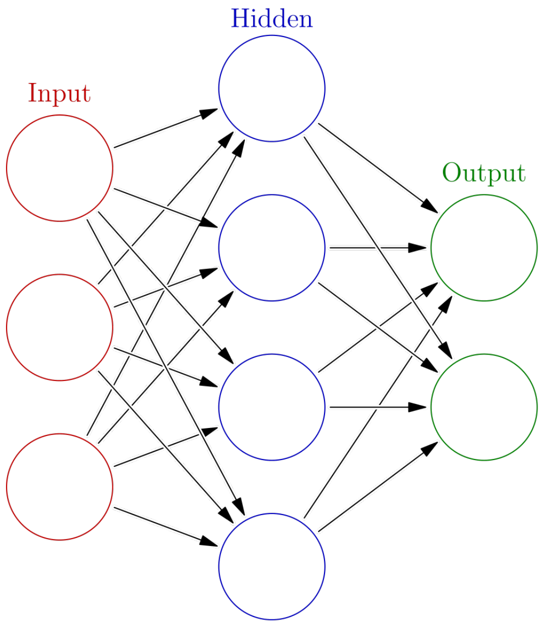


Figure 2.5: Structure of a neural network [2]

As mentioned in [37], the neural network structure with notation and formulas are presented in [39] as the following:

The neural network is composed of neurons connected by directed links. A link from neuron i to neuron j is connected to propagate activation a_i from i to j. Each neuron has an input a_i with an associated weight $w_{i},_{j}$. The weights are numeric values that determine the strength of the connection between neurons. It is also common to add a bias to each node denoted as b.

Equation 2.1 shows the calculation for the weighted input for each node, while equation 2.2 shows the equation with the added bias. Equation 2.3 shows how the output of a node is derived by applying a function to this weighted sum. This applied function is called the activation function, and is discussed more in detail in section 2.4.2.

$$in_j = \sum_{i=0}^{n} w_{i},_{j}\, a_i \tag{2.1}$$

$$in_j = \sum_{i=0}^{n} w_{i},_{j}\, a_i + b \tag{2.2}$$

$$a_j = f(in_j) = f(\sum_{i=0}^{n} w_{i,j}\, a_i) \tag{2.3}$$

The learning process for an ANN is achieved by updating the weights throughout the network. The following sections will give a brief overview of the most important aspects of ANNs.

2.4.2 Activation function

As mentioned previously, the activation function is the function that computes an output for a node based on its weighted input sum. While activation functions could be a binary step or linear, these are not suited for ANNs. Most common activation functions are non-linear, and their main purpose is to provide non-linear properties to the ANN. Without a non-linear activation function, an ANN would function equivalent to a linear regression model. Some of the most commonly used non-linear activation functions are discussed in detail below.

Sigmoid

The sigmoid function, as shown in figure 2.6, has an output between 0 and 1. The main advantage of the sigmoid function is that it normalises the output between 0 and 1. This solves the problem of *exploding gradient*, which is a problem that might occur with linear activation functions. The sigmoid function also gives very clear predictions due to its steep slope between -2 and 2, which results in a tendency for output values to move towards either end of the curve.

There are however some drawbacks to the sigmoid as well. The main problem is the so-called *vanishing gradient*. This problem occurs when reaching very high or low input values. Because of the way the sigmoid flattens out at 0 and 1 quite quickly, we reach a point where changes in the input result in almost no change for the output (For instance, both 10 and 20 as input values would give an output roughly equal to 1). The result of this is that the network could be unable to learn, or simply end up being extremely slow. Another drawback is that it is centered around 0.5.

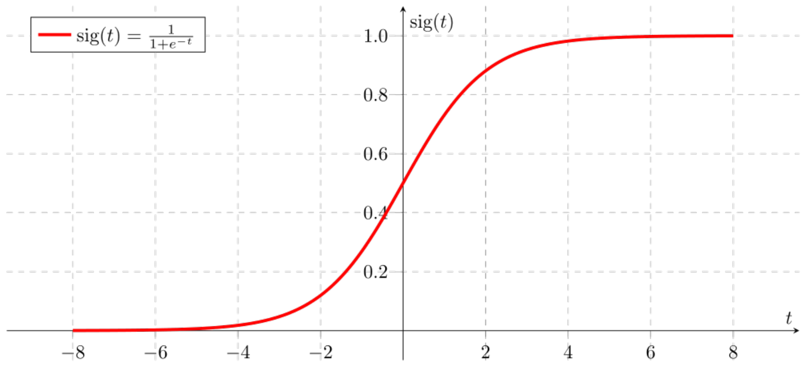


Figure 2.6: Sigmoid activation function

Hyperbolic tangent

The hyperbolic tangent (TanH) activation function, shown in figure 2.7, is also technically a sigmoid function, although it does differ slightly from the standard sigmoid. The only difference is that TanH gives an output ranging from -1 to 1. The benefits of TanH over sigmoid is that it has stronger gradients, as well as being centred around zero. Being centred around zero is beneficial for the same reason that normalising inputs around zero is beneficial. Using a zero-centred activation function results in centring the input for hidden layers throughout the neural network, which makes learning much easier.

The TanH does however still suffer from the same *vanishing gradient* problem as was mentioned for the sigmoid function above. TanH is however still considered to be an improved version of the standard sigmoid function.

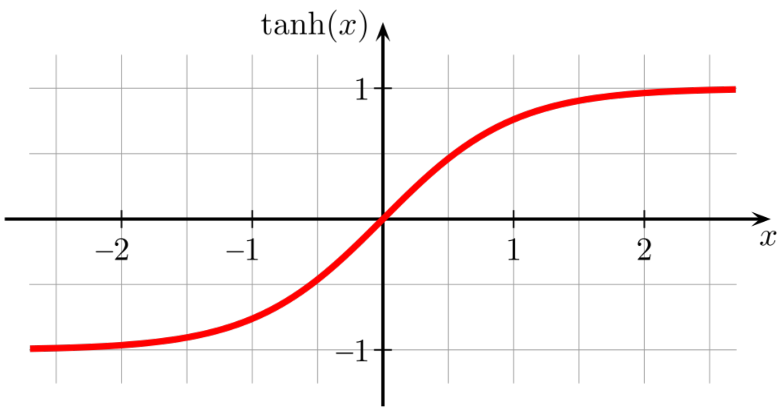


Figure 2.7: TanH activation function [3]

Rectified linear unit

The rectified linear unit (ReLU) activation function, as shown in figure 2.8, generates a linear output for positive input values, while negative values results in zero as the output value. This does make the function non-linear, although it has a range of 0 to ∞. This function is vulnerable to the previously mentioned *exploding gradient* problem, although this is more commonly dealt with by proper learning rates or regularization.

One of the advantages of ReLU is that it converges on a solution faster than sigmoid variants, due to its linearity keeping the slope from plateauing. It also does not have the *vanishing gradient* which both sigmoid and TanH suffers from. There is also a level of sparsity when using ReLU. This is due to each node having the possibility of not activating. This is often considered beneficial because we only want meaningful information to be processed as opposed to noise, which results in less overfitting. The calculation for ReLU is also computationally cheap, which together with the sparsity makes it compute significantly faster than the sigmoid variants mentioned above.

The downside of all negative values resulting in zero output, however, is a problem called *dying ReLU*. This problem is categorised by ReLU nodes being considered ”dead” once it gets stuck on the negative side of the function and will always output 0. The reason this happens is because the gradient of the ReLU function becomes zero for the negative range of the function. This makes it unlikely for a node to recover once it falls into the negative side. The problem can often be avoided by using a low learning rate, but there are also some variations of the ReLU function that combats this issue.

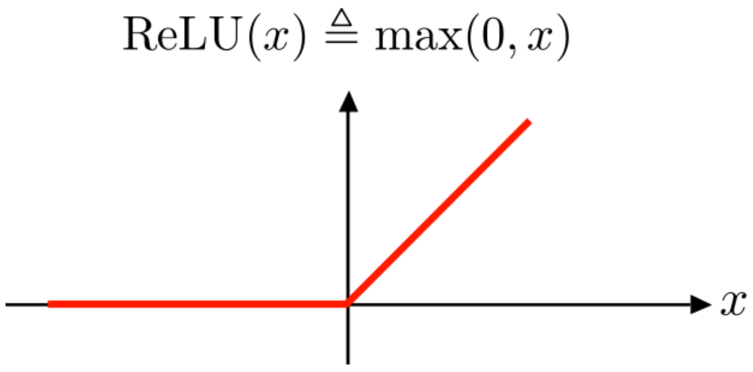


Figure 2.8: ReLU activation function [4]

ReLU variants

As mentioned above, the main drawback to ReLU is the ”dead” nodes caused by the zero output for negative input values. There are two popular alternative variants for ReLU that aims to solve this issue.

The first variant, shown in figure 2.9, is the so-called *leaky ReLU*. This variant has a small slope for negative values, as opposed to the flat line present in the standard ReLU.

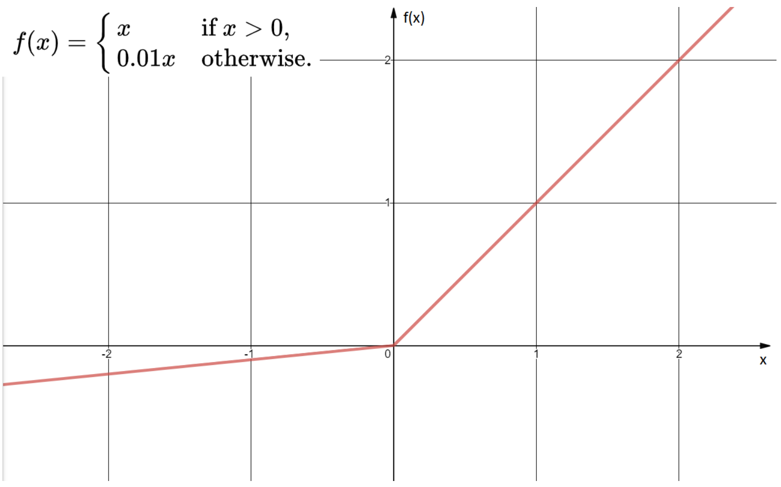


Figure 2.9: Leaky ReLU activation function

The second variant is the parametric rectified linear unit (PReLU) function. This function is almost identical to the leaky ReLU, with the only difference being that the slope coefficient for negative values is represented as a parameter, as opposed to a constant. This parameter is then learned along with all other ANN parameters. Equation 2.4 shows the calculation for the leaky ReLU with a slope coefficient of 0.01, while equation 2.5 shows the calculation for the parametric rectified linear unit (PReLU) where the slope coefficient is denoted as a parameter a.

$$f(x) = max(0.01x, x) \tag{2.4}$$

$$f(x) = max(ax, x) \tag{2.5}$$

2.4.3 Loss function

Another important part of an ANN is the loss function. The purpose of the loss function is to evaluate the output of the network, to measure the accuracy of the model. This is achieved by comparing the output of the model with the ground truth. The exact method or function for this comparison has a lot of different variations, but they all return a measure indicating how incorrect the output of the network is. It is therefore important to choose a loss function that properly correlates with a successful output, as the network will only focus on improving the calculated loss. A commonly used loss function for regression is the mean square error (MSE). The calculation for MSE is shown in equation 2.6, where $E(w)$ is the calculated loss, N is the number of outputs, y_i is the desired output, and $\hat{y_i}$ is the actual output.

$$MSE = E(w) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i}) \tag{2.6}$$

Loss functions that are especially useful for semantic segmentation, and as a result have been explored in this thesis, are discussed in more detail in section 2.6.

2.4.4 Gradient descent

Once the network has calculated its output error through its loss function as mentioned above, the goal is to minimise this error which is often referred to as minimising the loss function. The way an ANN learns and improves is by updating its weights, which is done through a process called backpropagation which is explained in the next section. Gradient descent is used to figure out exactly how the weights should be updated during this backpropagation process, in order to minimise the loss function. Gradient descent uses the derivative of the loss function to find the direction of steepest descent, which determines in which direction weights are updated. How much the weights are updated in the direction determined by gradient descent is decided by a parameter called the learning rate, which is mentioned in more detail in section 2.4.7. The updated weight is then calculated as shown in equation 2.7, where w_{new} is the new weight, w_{old} is the old weight, η is the learning rate, and $E(w)$ is the calculated loss.

$$w_{new} = w_{old} + \eta E(w) \tag{2.7}$$

Several different implementations of gradient descent are used for optimising training. The following list briefly mentions the most commonly used approaches:

- **Batch gradient descent (BGD):** BGD, also often referred to as vanilla gradient descent, is the most basic variation of gradient descent. In BGD, the gradient is calculated based on the entire training dataset. The disadvantage of this implementation is that calculating gradients for the entire dataset for every single update is very slow and inefficient.
- **Stochastic gradient descent (SGD):** SGD, in contrast to BGD, updates the weights for each sample of the training data. This eliminates a lot of redundant computation that is present in BGD, resulting in the network learning at a faster rate. The drawback with this, however, is that weight updates will have a high variance [40].
- **Mini-batch gradient descent:** This approach is a combination of both BGD and SGD, in which weights are updated in batches of n training samples. This reduces the variance of the weight updates, leading to a more stable convergence.

2.4.5 Backpropagation

Backpropagation is a technique for propagating the error backwards from the output and through the network, towards the input layer. This allows the gradient of the error to be calculated in each layer and thus adjust the weight and bias subsequently [41] [42].

2.4.6 Overfitting

Overfitting is a common problem in ML applications and is characterised by the model adjusting itself too closely to the training data, leading to lacking generalisation for the model. This lack of generalisation results in a model that performs very well on the training data and data that happens to be similar to it, while simultaneously performing much worse for any other data.

Two main factors leading to an overfitted model is having more parameters than necessary, and a sparse training dataset. A model with too many parameters is prone to overfitting due to being able to learn too much irrelevant information, such as noise, from the training data. A lacking training dataset can lead to an overfit model by not providing the model with enough information. If the training dataset does not present the model with a variety of data, it will be unable to differentiate between the important and unimportant (e.g. noise) information in the dataset.

It is also worth noting that the opposite of this problem, underfitting, is characterised by a model unable to adjust itself to the training data. This is caused by a model having too few parameters to learn the important information, or a poor training dataset containing too much unimportant information (such as noise) for the model to learn. This is however not as common as overfitting.

The most common technique to avoid overfitting is to discourage the model from becoming too complex, and is called regularization. One of the most common regularization techniques is early stopping, in which a validation dataset is used to test the model during training. If the model starts performing worse for the validation dataset while improving on the training dataset, it is a sign of overfitting.

Another common regularization method is dropout. This functions by disabling some random neurons during each training iteration, while updating the model normally for the remaining neurons. This forces the model to learn a different representation of the data and prevents overfitting.

Lastly, overfitting can be combated by improving the training dataset. A good training dataset is essential for both an accurate model, as well as to avoid overfitting. It is however not always feasible to generate additional training data the normal way, in which case data augmentation is worth considering. Data augmentation can prevent overfitting and improve generalisation of the model and will be discussed more in detail in section 2.7.

2.4.7 Learning rate

The learning rate is briefly mentioned in equation 2.7 in section 2.4.4. This is a variable that determines how large the change in weights should be when they are updated. This is often referred to as the step size, such that for a weight update the gradient determines the direction and the learning rate determines the size of the step in said direction.

The difficulties related to the learning rate comes from a learning rate that is either too low or too high. If a model has a learning rate that is too low, the model will improve slowly or not at all. The likelihood of the model getting stuck in a local optimum solution increases when using a low learning rate, and learning will overall be very slow. With a high learning rate however, the model runs into opposite problems. A high learning rate makes it more difficult for the model to converge on the global optimum, and learning can be more sporadic.

The important part of the learning rate is that it should neither be too low or too high. The exact value, however, is not easily determined, and usually requires some level of trial and error. The learning rate can also be adaptive, in which it varies throughout the training process. This can be beneficial due to a high learning rate quickly converges on the global optimum, without getting stuck in local optimums, while a lower learning rate will allow the model to more accurately fine-tune the weights towards the final solution.

2.4.8 Deep learning

Deep learning (DL) is a subtopic within ML. The unique aspect of DL is that DL emphasises learning through successive layers [43]. DL is typically seen as ANNs consisting of a high number of layers with non-linear activation functions. This layout makes the neural networks more capable of learning complex patterns in data. While the idea of DL is not new, the recent advances in computational technologies, especially in GPUs, has given DL a lot of popularity [44].

2.5 Convolutional Neural Networks

CNNs are inspired by the visual cortex in the brain, and are usually applied to the analysis of visual imagery [45]. The popularity of CNNs comes from their ability to automatically extract important features from images. Additionally, they also have a reduced computational requirement due to their shared weights [46], which is mentioned in more detail in section 2.5.5.

2.5.1 Structure

The structure of CNNs is based on the structure of ANNs, and similarly contain one input layer, any number of hidden layers, and one output layer. In CNNs, the hidden layers contain at least one convolution layer, and usually multiple. The typical architecture of CNNs consists of alternating convolution and pooling layers, as shown in figure 2.10.

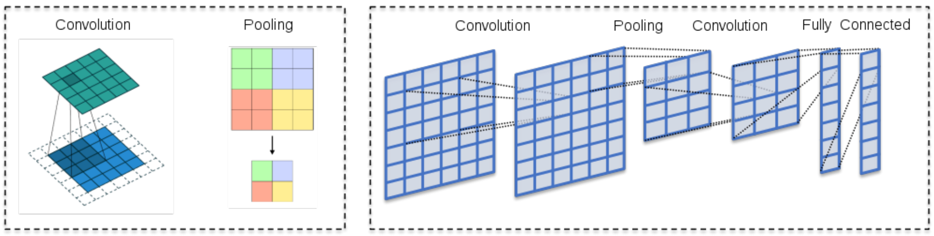


Figure 2.10: CNN architecture with alternating convolution and pooling layers [5]

2.5.2 Convolution layer

The convolutional layers can be considered the feature detection of the CNN. Convolution is a common operation in digital signal processing (DSP), although this is not the same approach utilised in convolution layers. In convolution layers the input data is convoluted with a filter, also referred to as a feature detector. The convolution process of a CNN is a sliding dot product, or cross-correlation, as is explained very elegantly in [47].

The mathematical equation for this operation is presented in equation 2.8. Where f is the input, g is the filter/feature detector, i and j are the indices, m and n are the number of elements in each dimension of the array, and G is the output feature map.

$$G[i,j] = (f * g)[i,j] = \sum_i \sum_j f[i-m, j-n]g[m,n] \tag{2.8}$$

This equation can also be extended for three dimensions, by adding a third dimension o and third index k, as shown in equation 2.9 below.

$$G[i,j,k] = (f * g)[i,j,k] = \sum_i \sum_j \sum_k f[i-m, j-n, k-o]g[m,n,o] \tag{2.9}$$

2.5.3 Pooling

A pooling layer is a form of non-linear down-sampling that is used to reduce the spatial dimensions of a CNN, resulting in reduced data size and fewer parameters. A common approach to CNNs is to include a pooling layer after a series of successive convolution layers, in order to reduce the size of the feature map. As seen in figure 2.10, the pooling reduces the size of the input by calculating a single value from a matrix of the input data. The mathematical operation to calculate this single value usually varies between average and max. The average pooling will calculate the average value of the input matrix, while the max pooling will keep the maximum value present in the matrix. In addition to the size of the pooling filter, the stride determines how far the filter is moved each time. Figure 2.11 shows an example of max pooling with a stride of [2,2].

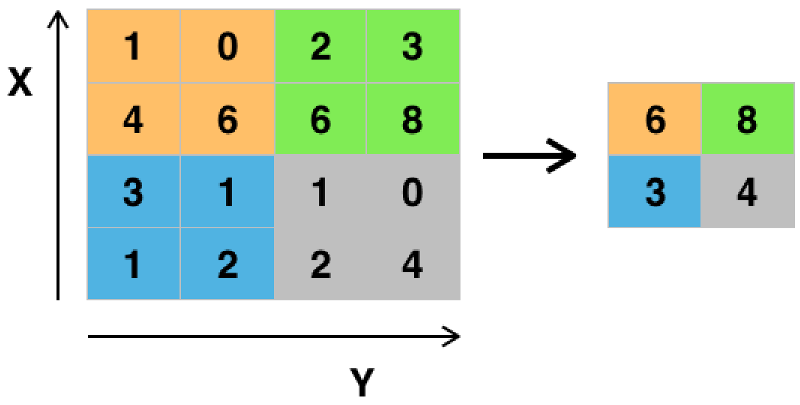


Figure 2.11: Example of max pooling with filter size (2x2) and stride of [2,2] [6]

2.5.4 Fully connected layer

Another common inclusion in a CNN is the fully-connected layer(s). These are typically included to make classification or regression decisions [43]. After the convolution and pooling layers of a CNN, the output is flattened into a single vector before being fed into a fully connected neural network. Any type of neural network can be used for this part of the process, although feed-forward networks are typically used.

2.5.5 Shared weights

Whereas fully connected layers have a unique weight and bias for each of its neurons, the convolutional layers have a feature called shared weights. This comes from the fact that the weights and biases in the convolution layer are shared as a vector, also known as a kernel. These kernels then represent the values of the filter that is used for the convolution operation discussed in section 2.5.2.

Because the convolution process is performed with the same filter over the entire input field, features are detected with indifference to their location in the input. The main benefit of shared weights is, therefore, the fact that the CNN becomes invariant to a translation of the features in the input data. This also has an additional effect of reducing overfitting. Another benefit with shared weights is that the training process of the CNN is faster, due to having fewer parameters to optimise.

2.5.6 Local receptive field

Another drawback of fully connected ANNs comes from the exponentially increasing number of connections when adding additional neurons. This consequently leads to an increase in the number of parameters, resulting in a slower training process. Coupled with the fact that input data in the form of images tend to have large dimensions, in order to conserve as much of the features as possible, the approach with fully connected layers ends up being extremely poor.

There is however no need for layers to be fully connected when the input data is in the form of images. This is because images tend to have a high correlation between adjacent pixels/voxels compared to distant ones. This is taken advantage of in CNNs by having neurons connect to a local region in the previous layer [37]. This local region for the input section of the neuron is referred to as the receptive field of the neuron.

The size of the receptive field of neurons can be increased by stacking multiple convolution layers or by sub-sampling (pooling) [48]. Increasing the size of the receptive field lets the network learn increasingly abstract features. The feature detection of a CNN is therefore relatively basic in the first layers, while later layers are able to detect more complex features.

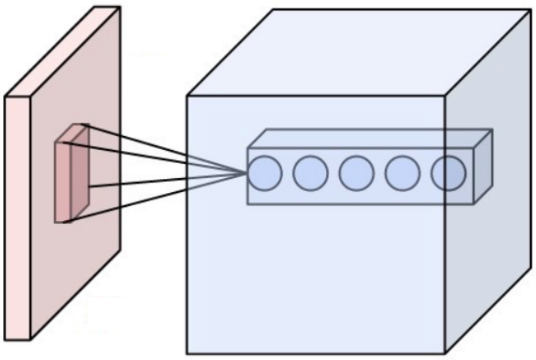


Figure 2.12: Neurons in a convolutional layer (blue), and the corresponding receptive field (red) [7]

2.5.7 Patch-based analysis

As mentioned earlier, input data in the form of images tend to have large dimensions to preserve as much information as possible. Because of this, and the fact that CNNs are relatively computationally costly, CNNs are not applicable for high-resolution images. This is especially true when dealing with three-dimensional images. This is where patch-based analysis is useful.

This approach takes advantage of the shared weights feature of CNNs, which lets them detect features while being invariant to translations. This makes it possible to input the image in the form of smaller patches, which essentially treats the input as a series of smaller images that are pieced back together after the segmentation masks are generated.

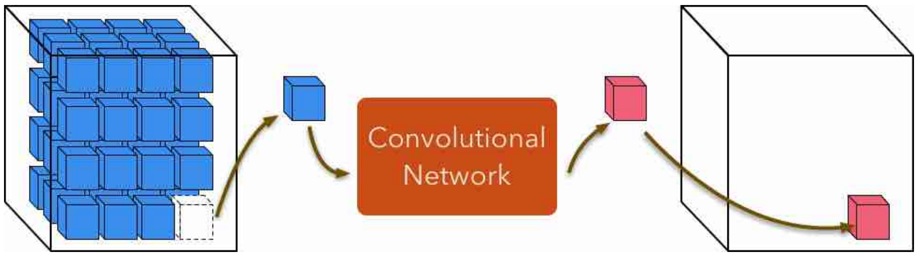


Figure 2.13: Patch-based analysis, as presented in Niftynet documentation [8]

2.6 Semantic segmentation loss functions

The two most common loss functions for semantic segmentation tasks is the pixel-wise cross entropy (CE), and the Dice loss. The Dice loss is first described in section 2.6.1. Next, the CE loss function is detailed in section 2.6.2. And lastly, section 2.6.3 will detail a loss function that combines Dice and CE into a new loss function called DicePlusXEnt, which was proposed in the published paper for nnU-Net [49].

2.6.1 Dice loss

The Dice loss function is based on the Sørensen-Dice coefficient, which is further detailed in section 2.8.2. The Dice loss function was introduced as a novel objective function by Milletari et al. in 2016 for 3D medical image segmentation [50]. The proposed loss function calculates a value between 0 and 1, with the goal of maximising this value. The equation for this Dice loss function is presented in equation 2.10 below, where the sums run over the N voxels, of the predicted binary segmentation volume $p_i \in P$ and the ground truth binary volume $g_i \in G$.

$$D = \frac{2\sum_i^N p_i g_i}{\sum_i^N p_i^2 + \sum_i^N g_i^2} \tag{2.10}$$

Dice loss is a measure of the overlap between the prediction and the ground truth. The main advantage of this approach is that the total size of a class is irrelevant, and only the percentage of correctly predicted pixels is of importance. This works well for class-imbalanced problems. The one drawback of this loss type, however, is that it has a high variance. This is because missing a few pixels in a small object can have the same effect as missing almost the entirety of a large object.

Another important thing to note is that it is generally a good idea to train models by minimising the loss that will be used to evaluate the performance after training. This is another reason that Dice loss is commonly used, due to the popularity of the Dice coefficient evaluation metric discussed later in section 2.8.2.

2.6.2 Cross entropy

CE is another common approach, and is calculated on individual pixels, in contrast to the aforementioned Dice loss. In tasks with multiple classes, the CE is calculated for each class separately and summed together. The equation for calculating the CE for multiple classes is presented in equation 2.11, where y is the ground truth value, $\hat{y}$ is the predicted value, and i, j is the current pixel location.

$$CE = -y_{i,j} \cdot log(\hat{y}_{i,j}) \tag{2.11}$$

While CE avoids the problem that Dice loss faces with regards to disproportional importance of smaller classes, it, in turn, has to deal with the opposite problem of easily ignoring smaller classes in favour of the larger ones.

2.6.3 DicePlusXEnt

An attempt at combining the benefits from both the Dice and CE loss types is to simply combine them, as shown in equation 2.12.

$$L_{total} = L_{dice} + L_{CE} \tag{2.12}$$

This the loss type that was the most promising for the original nnU-Net [49]. One other difference is that for the implementation of this loss type in Niftynet, the Dice loss is calculated slightly different than presented in section 2.6.1. The Niftynet implementation uses the Dice loss calculation presented in equation 2.13 below, where u is the softmax output of the network and v is a one-hot encoding of the ground truth segmentation map. Both u and v have shape $I \times K$ with $i \in I$ being the number of pixels in the training patch/batch and $k \in K$ being the classes.

$$L_{dice} = -\frac{2}{|k|} \sum_{k \in K} \frac{\sum_{i \in I} u_i^k v_i^k}{\sum_{i \in I} u_i^k + \sum_{i \in I} v_i^k} \tag{2.13}$$

2.7 Data augmentation

Data augmentation is, as mentioned previously, one of the methods used to combat overfitting. The idea behind data augmentation is to artificially increase the size of a training dataset. This is achieved by creating new training samples by augmenting samples from the original dataset. The reason this can be beneficial is that the training dataset is expanded to include a variety of conditions that can be expected to appear during testing or validation. These augmentation options can include small changes in scaling, rotation, brightness, and contrast to name a few. The goal is that these additions to the training dataset will result in the model being invariant with regards to these conditions, such that a small rotation as a result of imprecise data generation does not negatively affect the model accuracy.

Data augmentation is however not a straightforward procedure, and it requires some level of expertise to be able to choose the best augmentation options. It is important to not increase the amount of irrelevant data. For the case of supervised learning, the augmented data samples will retain the same label as the original data sample we augmented. If for example a model is trained to determine which direction a car is facing, then it would be a bad idea to include rotation in our data augmentation as we would suddenly have samples with wrong labels. It is therefore essential to understand the dataset well to be able to choose augmentation options that are able to generate samples with plausible conditions.

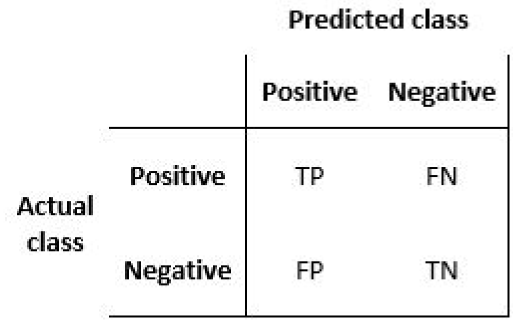
2.7.1 Elastic deformation

Elastic deformation is a data augmentation technique that warps the original image using a displacement field. The utilised approach to generate this warping varies, but the elastic deformation within Niftynet is based on the approach by Milletari et al. in [50]. This approach is described as: ”During every training iteration, we fed as input to the network randomly deformed versions of the training images by using a dense deformation field obtained through a $2 \times 2 \times 2$ grid of control-points and B-spline interpolation.”

2.8 Evaluation metrics

2.8.1 Machine learning evaluation metrics

A very common way to represent prediction results from a classification problem is by using a confusion matrix, as shown in figure 2.14. The confusion matrix is a summary of the number of correct and incorrect predictions. The terms of the confusion matrix are often used as the foundation for more advanced evaluation metrics, which are discussed in more detail further below.



		Predicted class	
		Positive	Negative
Actual class	Positive	TP	FN
	Negative	FP	TN

Figure 2.14: Confusion matrix [9]

The confusion matrix terms are defined in the following list:

- **True positive (TP):** Actual class is positive, and is correctly predicted to be positive.
- **True negative (TN):** Actual class is negative, and is correctly predicted to be negative.
- **False positive (FP):** Actual class is negative, and is incorrectly predicted to be positive.
- **False negative (FN):** Actual class is positive, and is incorrectly predicted to be negative.

Pixel accuracy

One of the simplest ways to evaluate a model is by using the pixel accuracy. This is a simple metric for the percentage of correctly predicted pixels. It is an especially bad indication of performance when dealing with class imbalance, due to only considering the number of correctly predicted pixels. The calculation for this metric is shown in equation 2.14 below.

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.14}$$

2.8.2 Evaluation metrics for medical imaging

When it comes to medical image segmentation tasks, two main evaluation metrics are commonly used, called the Dice coefficient and the Jaccard index [51]. These are both metrics that evaluate the overlap (union) between two samples. These are therefore used in situations where a ground truth is available to compare against the predicted output.

Jaccard index

The Jaccard index [52], also known as ”intersection over union”, is a combined measure of the similarity as well as the diversity of sample sets. The Jaccard index is defined as the size of the intersection divided by the size of the union of the sample sets, as presented in equation 2.15

$$J(A,B) = \frac{(A \cap B)}{A \cup B} = \frac{(A \cap B)}{|A| + |B| - |A \cap B|} \tag{2.15}$$

Figure 2.15 below shows a visual indication of the Jaccard index, as the area of overlap divided by the area of union between two samples.

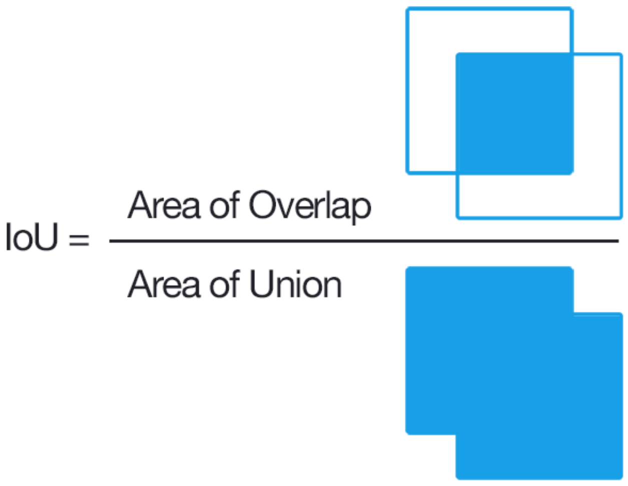


Figure 2.15: Visual representation of the Jaccard index (IoU) [10]

Dice coefficient

The (Sørensen-)Dice coefficient [53][54], often referred to as the ”similarity coefficient” or F1 score is one of the commonly used evaluation metrics for comparing two samples. The calculation of the Dice coefficient is presented in equation 2.16 below, where A and B are the two sets being compared, and $|A|$ and $|B|$ is the number of elements in each set.

$$D(A,B) = \frac{2(A \cap B)}{|A| + |B|} \tag{2.16}$$

Due to the Dice coefficient and Jaccard index being very similar, they can easily be used to represent each other, as shown in equation 2.17

$$J = \frac{2(D)}{2 - D} \quad \text{and} \quad D = \frac{2J}{J + 1} \tag{2.17}$$

Chapter 3

Related work

This chapter will look into some of the previously published works that relate to this project. Section 3.1 presents traditional rule-based techniques for medical image segmentation. Section 3.2 will discuss the current state of deep learning applications in the medical imaging field. And section 3.3 looks into attempts at segmenting knee joint anatomy through the use of CNNs.

3.1 Traditional rule-based segmentation techniques for medical image segmentation

Medical images are one of the most complex images to segment. Not only are anatomical structures typically variable in appearance, but they also have a high level of complexity. Coupled with varying sizes for objects of interest and diverse image modalities, the resulting task is not easily solved. There are simply no general segmentation technique or universal feature set that can accurately segment any medical image. There are however some popular segmentation techniques, as is well explained in [55], which are outlined below.

The basis for rule-based segmentation techniques is an assumption that image features over a specific region follow a set of heuristic rules. The most simple and straightforward approach for this is the thresholding, in which features are defined only based on pixel intensity. The benefits of thresholding are its fast computation speed, although the results are rarely sufficient. The simplest thresholding method only divides the image into two regions, the object of interest and the background, and accomplishes this by labelling a pixel as the object of interest if the pixel intensity exceeds a set value, otherwise, it is labelled as the background. The threshold can be either fixed to a value, or adaptive. Common ways to set the value are mostly statistical analysis of either the entire image or local areas around the currently evaluated pixel.

The main downside of thresholding, however, lies in its simple nature. Any overlap in intensity between the object of interest and the background will result in an inaccurate segmentation. Furthermore, a common requirement for medical image segmentation is that objects should be connected to a region. One approach that solves this is the region-growing approach, also called region merging. This approach first selects a few initial seeds, then the seeds grow by including any neighbouring pixels that comply with a set of criterion. These criteria are predefined to specify the required properties of the regions. As a result, these segmentation results rely heavily on these criteria and the initial selection of seeds. [56] [57]

Another region-based approach is the region split-and-merge. In this approach, the image is initialised as a set of regions, and subsequently split and merged according to a set of rules. Just as with the region-growing, the split-and-merge is also highly dependant on the initialisation. Split-and-merge has however been successfully applied to several problems, including large brain lesions [58], cavity deletion [59], retinal blood vessels [60], and pulmonary nodules [61]. Region-based approaches are also often used as semi-automatic segmentation tools to serve as a foundation for manual segmentations [62].

3.2 Deep learning applications in medical image segmentation

In 2014, Mengqiao et. al [63] introduced one of the first 3D CNN models to segment brain tumor MR images. This model was a 22-layer deep CNN. This idea was followed by Kamnitsas et. al [64] in 2015, where an 11-layer deep, double pathway, 3D CNN was developed for segmenting brain lesions in MR images. The resulting improvements were well explained by Hesamian et. al [65] as the following: ”There were two parallel pathways with the same size of the receptive field, and the second pathway received the patches from a subsampled representation of the image. This allowed processing of greater areas around the voxel, which benefited the entire system with multi-scale context. This modification along with using a smaller kernel size of 3×3 has produced better accuracy (an average Dice coefficient of 0.66). On top of that, a lower processing time (3 min for a 3D scan with four modalities) compared to its original design has been achieved.”

In 2015, Ronneberger et. al [66] proposed a CNN architecture for segmenting biomedical images, called U-Net. This architecture built on the fully connected neural network (FCN) architecture that was proposed by Long et. al [67] in 2014. The architecture consisted of a contracting path to capture context and a symmetric expanding path that enabled precise localisation. This proposed method won the ”ISBI cell tracking challenge 2015”, for segmentation of neuronal structures in electron microscopic stacks.

Also in 2015, Zhang et. al [68] proved that the performance of MRI image segmentation was significantly improved when using multi-modality input images for 2D CNNs.

In 2016, Milletari et. al [50] proposed an approach to 3D image segmentation based on a volumetric FCN, with a similar architecture to the U-Net mentioned above. The proposed solution, named V-Net, was developed for segmenting MRI images depicting prostate. The model was evaluated on the ”PROMISE 2012” dataset, in which it achieved a Dice score of 0.869. This score was just shy of the best-reported score of 0.879, which was achieved by Vincent et. al [69], by a method based on active appearance models.

3.2.1 U-Net variants

Due to the versatility and performance of the U-Net architecture proposed by Ronneberger et. al [66] for segmenting biomedical images, multiple improvements have been built as extensions of this architecture.

The 3D U-Net was proposed by Çiçek et. al [70] in 2016. This proposed network took the 2D convolution, 2D up-convolution, and 2D pooling layers that were present in the original 2D U-Net architecture, and replaced them with their 3D equivalents, namely 3D convolutions, 3D transposed convolutions, and 3D pooling layers. Another improvement included doubling the number of kernels before max pooling in both the contracting path and the expanding path. This change eliminated the bottleneck as suggested by Szegedy et. al in [71]. The proposed network was able to achieve a Jaccard index of 0.863 in segmentation of the Xenopus kidney, in which a 2D U-Net (which segmented the data slice by slice) had a Jaccard index of 0.796.

In 2018, Oktay et. al [72] proposed an attention gate U-Net model, similar to the approach proposed by Jetley et. al [73]. The functionality of these attention gates was to act as filters, filtering out noise and irrelevant information from skip connections. When compared to the standard U-Net model, the performance saw an increase of 2-3%.

In 2018, Zhou et. al [74] proposed a nested U-Net variant, called ”UNet++”. This proposed variant redesigned the skip pathways of the original U-Net, and as a result, transformed the connectivity of the encoder and decoder parts. When compared to the standard U-Net, this proposed variant achieved an average increase in the Jaccard index of 3.9, when applied to nuclei segmentation in the microscopy images, liver segmentation in abdominal CT scans, and polyp segmentation in colonoscopy videos.

Also in 2018, the U-Net variant utilised in this thesis, called nnU-Net (”no-new-Net”), was proposed by Isensee et. al [49]. The idea behind this variant was that the original U-Net ”comprises several degrees of freedom regarding the exact architecture, preprocessing, training and inference”. nnU-Net was therefore designed as a self-adapting network, such that the design of the network was dependant on the input data. ”For each task, the nnU-Net automatically runs a five-fold cross-validation for three different automatically configures[sic] U-Net models and the model (or ensemble) with the highest mean foreground dice score is chosen for final submission”. The proposed network was submitted to the ”Medical Segmentation Decathlon” in 2018 [75], in which it received first place when tested on 10 different segmentation tasks.

In 2019, Ibtehaz et. al [76] proposed a variant where the standard convolutional layers of the U-Net architecture was replaced by a customised block called ”Multires block”. This ”Multires block” was inspired by the ”Inception block” previously proposed by Szegedy et. al [77] in 2014. This proposed architecture was tested on five different datasets and saw improvements of 0.62%, 1.14%, 2.63%, 5.07%, and 10.15% over the standard U-Net.

3.3 Convolutional neural networks for segmentation of knee joint anatomy

In 2007, Folkesson et. al [78] presented a multi-class classification method, that combined two binary k-nearest neighbours (k-NN) classifiers. The binary classifiers were divided such that one was used to find the tibial medial cartilage, whilst the other found the femoral medial cartilage. This resulted in the segmentation of three classes, the two aforementioned ones and the background. This method was tested on 114 unseen scans and achieved a mean Dice score of 0.8135 for the tibial, and 0.77 for the femoral.

In 2013, Prasoon et. al [79] proposed a voxel classification system based on integrating three 2D CNNs, each having a one-to-one association with the xy, yz and zx planes of a 3D image, respectively. This approach was applied to the segmentation of the tibial cartilage in low field knee MRI scans. This method was a binary classification to segment the tibial cartilage, and was tested on the same 114 unseen scans as Folkesson et. al [78] as mentioned above. The achieved mean Dice coefficient was reported as 0.8249 for the tibial cartilage.

In 2017, Antony et. al [80] utilised a FCN to quantify the severity of osteoarthritis (OA). This approach used a weighted ratio of categorical CE and mean-squared loss as its loss function. This approach was trained and tested both separately and combined on both the OAI dataset containing 3146 training images and 1300 test images and the MOST dataset containing 2,020 training images and 900 test images. The resulting Jaccard indexes were 0.83, 0.81, and 0.83 respectively.

In 2018, Zhou et. al [81] proposed an extensive segmentation pipeline by combining a semantic segmentation CNN, 3D fully-connected conditional random field (CRF), and 3D simplex deformable modelling. The method was evaluated on 3D fast spin-echo (3D-FSE) MR image datasets, consisting of 20 subjects. The samples consisted of 13 unique classes, namely the background, femur, femoral cartilage, tibia, tibial cartilage, patella, patellar cartilage, meniscus, tendons, muscle, joint effusion, fat pad, and other non-specified tissues. All musculoskeletal tissues after the full process had a mean Dice coefficient above 0.7.

In February 2019, Ambellan et. al [82] presented a method for automatic segmentation of knee bones and cartilage from MRI images. This approach combined 3D Statistical Shape Models and 2D as well as 3D CNNs. This approach was trained and tested on three different datasets, namely the SKI10, OAI Imorphics, and OAI ZIB datasets, containing 150, 88, and 507 images respectively. The results were summarised for the OAI ZIB dataset, in which the model achieved a Dice coefficients of 98.5, 98.5, 89.9, and 85.6, for the femoral bone, tibial bone, femoral cartilage, and tibial cartilage respectively.

In June 2019, Homlong [37] utilised a CNN U-Net for the semantic segmentation of the bones, the ACL, and the PCL of the knee joint. That project was also performed in collaboration with Sunnmøre MR-Klinikk. In that project, the dataset consisted of samples from 17 difference knees (with 10 being used for training), with three image modalities for each, and the dimensions of the images were $275 \times 400 \times 400$. The resulting performance for segmenting these 4 classes was reported with a Dice score of 0.99314 ± 0.00173, and a Jaccard index of 0.98638 ± 0.00341.

This current thesis can thereby be seen as a continuation of the work presented by

Homlong [37], seeing as the main differences are increased image dimensions to $400 \times 400 \times 400$, training dataset doubled from 10 to 20 subjects, the number of classes increased from 4 to 13, and the utilised CNN is different.

In September 2019, Byra et. al [83] developed a deep learning-based method for knee menisci segmentation in 3D ultrashort echo time (UTE) cones MR imaging, and to automatically determine MR relaxation times, namely the T1, T1p , and T2 parameters. This approach was utilised for assessing knee OA. The dataset consisted of 61 samples manually segmented by radiologists. Transfer learning was applied to develop 2D attention U-Net CNNs for the menisci segmentation based on each radiologist's ROIs separately. This method was a binary segmentation with the menisci as the sole ROI. The two models that were developed achieved Dice scores of 0.860 and 0.833.

In October 2019, Pettersen [84] utilised a U-Net inspired CNN, called "MartiNet", to segment the bones, theACL, and the PCL of the knee joint. This project was carried out alongside the work by Homlong [37] mentioned earlier. They were both a collaboration with Sunnmøre MR-Klinikk, and they both utilised the same datasets as a result. The resulting segmentations were of similar accuracy to those achieved by Homlong, with a pixel accuracy (not Dice score) reported as 99.60%. "As further work, this CNN could detect more labels. The rate of learning in this CNN was fast and it had an accuracy of better than 95 % after only a few iterations. This shows that there is space for more complex problems and it is possible to add more labels to the segmentation". Thus, this current thesis performs this task of further work that was suggested by Pettersen.

In February 2020, Chen [85] proposed a deep 3D CNN to segment the knee bone in a resampled image volume to enlarge the contextual information and incorporating prior shape constraint. Additionally, in order to restore the bone segmentation back to the original resolution, a restoration network was also proposed. The cartilage was segmented using a conventional U-Net-like network. The ultimate results were the combination of the bone and cartilage outcomes through post-processing. The solution was assessed by using the dataset from "Grand Challenge SKI10". The proposed method achieved a Dice score of 0.98, 0.98, 0.89, and 0.88, for the femur bone, tibia bone, femur cartilage, and tibia cartilage respectively.

Chapter 4

Methodology

This chapter will describe the methodology for this thesis. The utilised hardware and software is presented in section 4.1. Section 4.2 details the dataset for this thesis, including the generation and formats of the data. The specifics of the applied ANNs are detailed in section 4.3. And lastly, the methodology related to hyper-parameter optimisation is presented in section 4.4.

4.1 Hardware & Software

4.1.1 Hardware

The hardware consisted of a single computer with two identical GPUs, where one of those GPUs was reserved for this thesis. Thus, the computer components utilised in this thesis, are presented in the following list:

- **GPU:** NVIDIA GeForce RTX 2080 Ti, 11 GB video random access memory (VRAM).
- **CPU:** AMD Ryzen Threadripper 2950X 16-core Processor, 3.50GHz.
- **random access memory (RAM):** 64GB, 2667 MHz.

4.1.2 Software

The software utilised as a part of this thesis is provided in the following list, including a short description for their function:

- **Niftynet:** [86] [87] Niftynet is an open-source platform implemented based on TensorFlow application programming interface (API) for deep learning in the medical imaging domain. Niftynet was used to create, customise, train, and evaluate neural networks during this thesis.
- **six:** [88] Compatibility library for Python 2 and 3. Prerequisite for Niftynet.

- **NiBabel:** [89] NiBabel is a Python library used to read and write some common neuroimaging file formats. Prerequisite for Niftynet.
- **SciPy:** [90] SciPy is a collection of open-source Python-based software for mathematics, science, and engineering. Prerequisite for Niftynet.
- **NumPy:** [91] NumPy is the fundamental package for scientific computing with Python. It adds support for multi-dimensional arrays and efficient computation of these arrays. Prerequisite for Niftynet.
- **Pandas:** [92] [93] Pandas is an open-source Python library that adds functionality for data analysis and manipulation. Prerequisite for Niftynet.
- **Pillow:** [94] Pillow is a Python imaging library, that adds image processing capabilities. Prerequisite for Niftynet.
- **Blinker:** [95] Blinker is a Python library that provides object-to-object and broadcast signalling for Python objects. Prerequisite for Niftynet.
- **TensorFlow:** [96] TensorFlow is an end-to-end open source platform for ML. TensorFlow is the ML platform that Niftynet is built upon. Prerequisite for Niftynet.
- **CUDA:** [97] ”CUDA Toolkit provides a development environment for creating high performance GPU-accelerated applications”. Prerequisite for Niftynet.
- **ITK-SNAP:** [98] ITK-SNAP is a software application used to segment structures in 3D medical images. This software was also used to view and export segmentation results in this thesis.
- **Excel:** [99] Excel was used to import evaluation score csv files and visualise them as tables.

4.2 Data

The data for this thesis was provided by Sunnmøre MR-Klinikk. The data was generated by scanning volunteers. The sampling of human (knee) data did not contain any identifying or sensitive information about the volunteers, and was approved by the Regional Committee for Medical and Health Research Ethics (REK nr. 61225). Because this thesis was only a part of a larger ongoing project, the data was provided incrementally throughout the thesis work. More specifically, the validation dataset was not available until later parts of the thesis, leading to an inability to properly test/evaluate the trained models at the earlier stages of the thesis. This was somewhat alleviated by training models during the earlier parts of the thesis, and delaying the evaluation until a dataset was available.

The data was generated by MRI, which is explained in more detail in section 2.2. The data was generated in three different image weightings for each patient, namely as T1, PD, and FS weighted images. The images were all aligned to the same axis orientations and with identical dimensions. The dimensions of the images were $400 \times 400 \times 400$, resulting in a total of 64 million voxels per image. The voxel dimensions were [0.4mm, 0.4mm, 0.4mm]. These voxel dimensions are much smaller than standard MR images, resulting in a higher resolution. The reason the images were generated with a higher than normal resolution was that they were generated specifically for the task of an automated segmentation, in which the segmented output would be used for simulation and AR/VR purposes.

The training dataset consisted of 20 patients, with 3 differently weighted images for each, whilst the validation dataset contained 5 patients, also with 3 differently weighted images for each. The segmented ground truth masks for both the training and validation datasets were manually segmented by Sunnmøre MR-Klinikk. The segmented classes are presented in the following list:

1. Bone (medulla)
2. PCL
3. ACL
4. Muscle
5. Cartilage
6. Bone (cortex)
7. Arteries
8. Collateral ligaments
9. Tendons
10. Meniscus
11. Fat
12. Veins

The segmented ground truth was generated similarly, but with some differences due to the state of the on-going project when they were generated. When the validation dataset was generated, the ground truth had been updated to include more detailed segmentation of the tendons and veins (class 9 and 12). Due to models already being trained on the older training dataset, in which these classes were labelled differently in the ground truth, the evaluation score is not accurate for these two classes. The evaluation score for class 9 is still included in this thesis, as the differences for this class were not too large, whilst class 12 has been ignored in the evaluation scores due to this large difference rendering it obsolete.

The total amount of data utilised for this thesis is then summarised to 3 differently weighted images and one ground truth segmentation mask, for each patient. This adds up to 25x4 = 100 Nifti files.

4.3 Convolutional neural network implementation

This thesis focused on optimising the nnU-Net implementation of a 3D U-Net, as described in [49]. This choice was largely because the network achieved good results in medical decathlon 2018 (which is a somewhat similar dataset) [75]. The main difference from the standard 3D U-Net is that the input size is equal to the output size, due to using padded convolutions. Furthermore, the leaky ReLu activation functions offer non-linearity. The number of filters before upsampling is reduced. Normalisation is implemented as instance normalisation as opposed to batch. Fits a spatial window 128x128x128 with a batch size of 2 on one TitanX GPU for training. And lastly, no learned upsampling, resulting in linear resizing. [100]

4.4 Hyper-parameter optimization

One major drawback when using Niftynet is that it does not include support for hyper-parameter optimization. At the same time, however, it is worth noting that hyper-parameter optimization does increase the computational requirement in the short term. It is therefore not necessarily feasible for the current task presented in this thesis due to the sheer size of the data, which already pushes the training time to multiple days at a minimum for a single model. Thus, any optimization methods (which has to train models for each desirable parameter combination, and then evaluate it) would extend an already long training process.

Furthermore, the validation dataset was generated gradually throughout the thesis. This would have led to either inconsistency between models caused by different validation datasets, or a lacking validation dataset if the provisional dataset available at the start of the thesis would be kept for the entire duration. A final alternative would be for the training dataset to have been reduced in order to increase the size of the validation dataset. Neither of these 3 options would have been ideal.

The conclusion is then that hyper-parameters had to be optimized manually, where the training of multiple parameter combinations would be training throughout the thesis, and the evaluation of the models would be delayed until the validation dataset was completed.

One last thing to mention is that the choice of parameter changes was an iterative process. Some changes were chosen almost at random to simply test the impact of a parameter. Other changes were chosen after evaluating trained models on the partial validation dataset to determine the best parameters for continued training. As a quick example, two models would be trained with similar parameters except for one of them including data normalisation. Then the trained models would be evaluated. If the model without normalisation had a significantly better evaluation score, then the subsequent models would also be trained without normalisation. Evaluation of trained models on the partially complete validation set was therefore used to determine which hyper-parameters to train subsequent models with.

One caveat, however, is that certain parameters can potentially work well together with certain other parameters. There is also the clear uncertainties and potentially inaccurate evaluations when using a small or only partially complete validation dataset. Therefore, no parameter was simply discarded or set to a fixed value for all subsequent models. The evaluation of the partial validation dataset was only used as a rough indication of the impact from a given parameter.

The specific choice of hyper-parameters experimented with for the training is detailed in the following chapter.

4.5 Baseline model configuration

The following list presents the parameter configuration that was used as a baseline. Any parameter that is not mentioned to be a specific value for a certain experiment, is thereby kept at this baseline value. The loss type is not included in this baseline, but rather specified for each model (e.g. ”Dice baseline” or ”DicePlusXEnt baseline” model).

- **Resolution (pixel dimension):** (0.4,0.4,0.4)
- **Spatial window size:** (128,128,128) for all training, (224,224,224) for inference results presented after sections 6.1 and 6.2.
- **Normalisation:** False
- **Learning rate:** 0.0001
- **Data augmentation** False
- **Window sampler:** Uniform
- **Batch size:** 1
- **Queue length:** 5
- **Samples per volume:** 1
- **Activation function:** Leakyrelu

Chapter 5

Experiments

This chapter will present the various hyper-parameters that were experimented with within this thesis. The chapter will detail the functionality of the hyper-parameters, as well as establish the reasoning and hypotheses behind the conducted experimentation.

5.1 Resolution (pixel dimension)

To change the resolution of the input data, the data can be resampled. In Niftynet, this option is set by the parameter ”pixdim”. This parameter sets the desired voxel dimensions for the input image, such that input data is resampled to the desired voxel dimensions before being fed into the network. As mentioned in section 4.2, the data for this thesis was generated with voxel sizes of 0.4mm. Therefore, if the resolution of the image were to be halved along each axis (resampled from [400,400,400] to [200,200,200]), then the ”pixdim” parameter would be set to 0.8mm.

The hypothesis around experimentation with this parameter was based around an issue referred to as ”false positive volume”. This volume initially appeared on all attempts to train a nnU-Net model, and changes to various other parameters had minimal impact. The shape of this volume seemed to be constrained to a cube with similar size to the spatial window size, leading to a suspicion that this parameter was related to the issue. But because of the memory limitations for the spatial window size mentioned in the next section, the only alternative was to decrease the size of the input data.

Thus, experimentation with this parameter was used to investigate the correlation between the size of the input data and the spatial window. More specifically, the experimentation aimed to check whether or not the difference between these two sizes was the cause of this ”false positive volume”. This issue is explained in more detail in section 6.1.

5.2 Spatial window size

The spatial window size defines the spatial size of the input of a convolutional network [8]. The spatial window size is an array of three integers [x,y,z], that specify the size of the input window used in the patch-based analysis, as detailed in section 2.5.7. There is therefore a hard requirement for this parameter to not exceed the size of the images in the dataset. Furthermore, it is generally desirable to keep this parameter as large as possible within hardware limitations. This is because there is usually a lot of global information that can be learned over the abstraction of multiple layers that can be lost with smaller patches.

The only requirement for this parameter is, as mentioned above, that it can not exceed the size of the data samples in the dataset (including padding size). But in addition to this, there are also constraints depending on which network module is used. For the no_new_net module in Niftynet, the window size is required to be divisible on 16 due to the network having max-pooling 4 times.

The maximum value for the spatial window size also depends on the available hardware. More specifically, the memory consumption is heavily impacted by this parameter, and the available memory is therefore likely the deciding factor for this parameter. It is also worth noting that this available memory depends on where the training is performed. When training is set to run on the GPU, the associated VRAM will be utilised. And similarly, by running the training on the CPU, the installed RAM will be used. This means that models trained on the CPU can use a higher spatial window size if the available amount of RAM is sufficient. However, using the GPU is in almost all cases preferred due to a significant difference in computational time.

5.2.1 Training

The original idea for this parameter was that it should be as high as possible for best results, practically meaning as high as possible without experiencing out of memory (OOM) errors. The reasoning behind this is that one would ideally input the entire image into the model in order to preserve the most information possible, but the hardware would limit the possible size of the input. This then led to the assumption that a larger input size for the model would carry more information, leading to a potentially more accurate model.

5.2.2 Inference

As mentioned above this was originally set to the same value as the one used for training. This was with the assumption that any maximum value for training would also be the maximum value for inference due to both operations using the exact same model and input data size.

5.2.3 Additional hypothesis

A secondary hypothesis was developed at a later point when resuming nnU-Net experimentation, after a brief period of investigating different network modules. Due to the ”false positive volume” issue mentioned above having a cubed size seemingly equal to the spatial window size, it was assumed to be correlated in some way. The goal of this experimentation was then to investigate whether or not this issue was correlated to this parameter, and if so for which values of the parameter it occurs.

5.3 Loss type

Loss type is the name used in Niftynet configuration files for the type of loss functions. The loss function was discussed generally in section 2.4.3, and in detail for segmentation tasks in section 2.6, but is briefly summarised as the function responsible for estimating the accuracy of a model during its training process.

The hypothesis for experimentation with this parameter was that DicePlusXEnt would perform the best, based on results published in the original paper for nnU-Net [49]. It is however worth noting that the reported results were found by using a completely different dataset, both in terms of entities, image modality, image geometry, and dataset size.

As detailed in section 2.6, experimentation with loss types was mainly limited to Dice and DicePlusXEnt.

5.4 Normalisation

Input data normalisation is often considered to be strictly beneficial and an important addition to include when training ANNs. The importance and benefits achieved through normalisation have been shown for a large variety of ANN applications [101] [102] [103].

Some of the reasons normalisation is important is that when transforming the data such that the mean value is close to zero, any data which is exclusively positive or negative will end up with both positive and negative values after normalisation. Another benefit is from scaling the data down to a standard range, which prevents issues caused by varying input value scales. Both of these aspects are important for the training process, especially when considering the activation function designs discussed in section 2.4.2. The cumulative result of this is in theory to make the data much easier for the ANN to learn from. Normalisation in Niftynet is applied in the form of histogram standardisation, as described in [104].

The hypothesis for this parameter was that it would speed up the training time, due to making the data more computationally efficient. Secondly, it was assumed that normalisation would increase the accuracy of a model, as seen in other applications in the medical image segmentation field such as in [105].

5.5 Learning rate

The learning rate is explained in more detail in section 2.4.7 but can be summarised as the step size used to update the model weights.

The hypothesis for the learning rate was that it would mainly impact training time. The learning rate was assumed to not directly impact the performance of the final trained model, but rather have an impact on the time taken to reach said, final model. For this reason, the learning rate was not tested extensively, and experimentation was in essence a process of finding a good learning rate value, that was used for training subsequent models throughout the thesis. The initial learning rate value applied for the nnU-Net models simply followed the one used in the original paper presented in [49], being a value of 0.0001. This learning rate was considered to be on the low side of commonly used values, and most attempts, therefore, included increasing the value, although some attempts were also made with a lower learning rate.

5.6 Data Augmentation

Data augmentation is explained in more detail in section 2.7. The specific options available in a Niftynet configuration is detailed in [106]. When testing the potential benefits for data augmentation in this thesis, the following options were applied:

- **Rotation angle:** This randomly rotates each axis a random number of degrees based on the provided interval, and was set to [-10, 10].
- **Scaling percentage:** This randomly scales each axis based on the provided range, and was also set to [-10, 10].
- **Elastic deformation:** This parameter is discussed in detail in section 2.7.1. Attempts at data augmentation in this thesis did include this parameter as it has been shown to increase accuracy in similar problems previously, such as in [107] [108] [109].

Due to the choice of augmentation parameters being a vital part of the success when it comes to data augmentation, and the fact that each attempt would require a significant amount of training time, the amount of data augmentation experimentation was limited.

The hypothesis for this parameter was that due to the low number of samples in the training dataset, the addition of data augmentation would improve the generalisation of the model. At the same time, the training time will increase by quite a lot, and may therefore not be worth adding as it would reduce the overall number of trainable models, and thus limit the amount of experimentation for other parameters.

Chapter 6

Results

This chapter will present the results from the experimentation detailed in the previous chapter. The evaluation metrics used to compare and evaluate the results are explained in section 2.8.2. 3D images of the segmented outputs are set to hide the fat (label 11) to avoid obscuring all other labels from view, except for sections 6.1 and 6.2 as the segmented output is not interesting for these sections. The evaluation scores also do not include label 12, due to the large difference between the ground truth segmentation of this class in the training and validation datasets. Label 9 also suffers a slightly lower score due to this, but not enough to invalidate it.

The evaluation metrics used to present the accuracy of the models, detailed in section 2.8.2, are the dice coefficient and Jaccard index as well as the standard deviation for each of them. At the beginning of this chapter, the evaluation scores are not relevant, and therefore not shown. This is because this chapter will first focus on figuring out a solution to an issue referred to as the ”false positive volume”, which is detailed and shown in section 6.1.

6.1 Resolution (pixel dimension)

The ”false positive volume” issue can be described as a volume of false-positive predictions, contained within a cubed area. This problem initially occurred for all models of nnU-Net that were trained, regardless of changes to other parameters. For a visual example, the following three figures will showcase different occurrences of this issue. Figure 6.1 shows the worst occurrence of the issue, figure 6.2 shows a normal occurrence of the issue, and figure 6.3 shows the best occurrence of the issue.

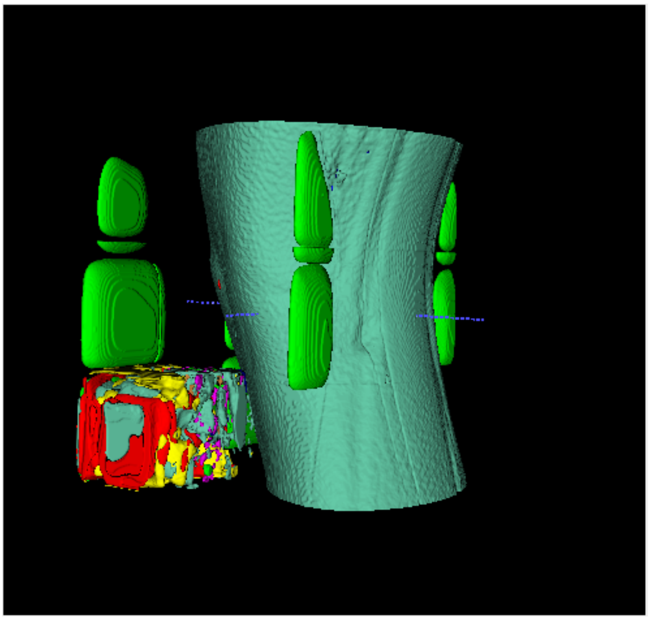


Figure 6.1: Worst example of the ”false positive volume” issue

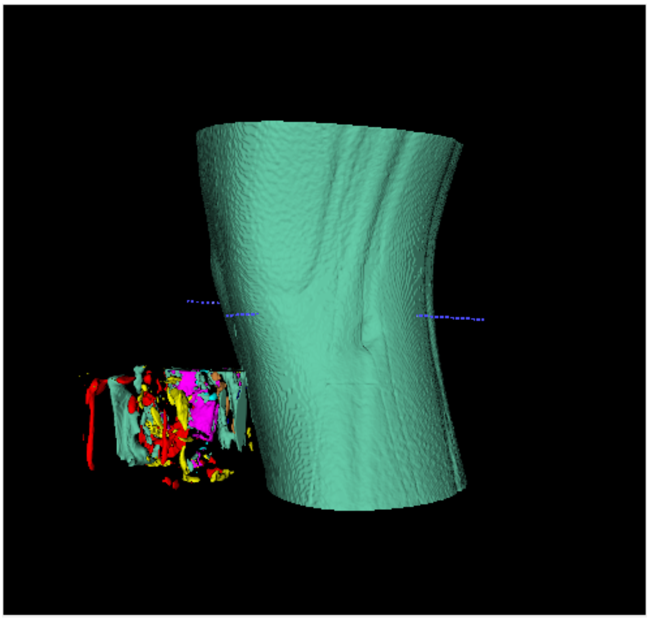


Figure 6.2: Normal example of the ”false positive volume” issue

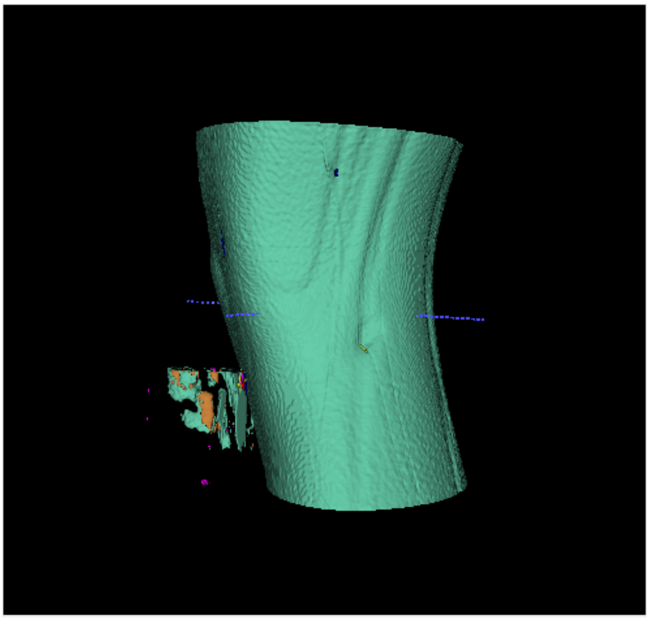


Figure 6.3: Best example of the ”false positive volume” issue

When attempting to resolve this issue by changing the resolution, it was the configuration for the best example that was used as a baseline. The pixel dimension was tested in the following order 1.0, 0.5, 0.7, and 0.6. Figure 6.3 shows the result for the unchanged pixel dimension of 0.4. The following series of figures shows the result from the various tested pixel dimensions, sorted in ascending order.

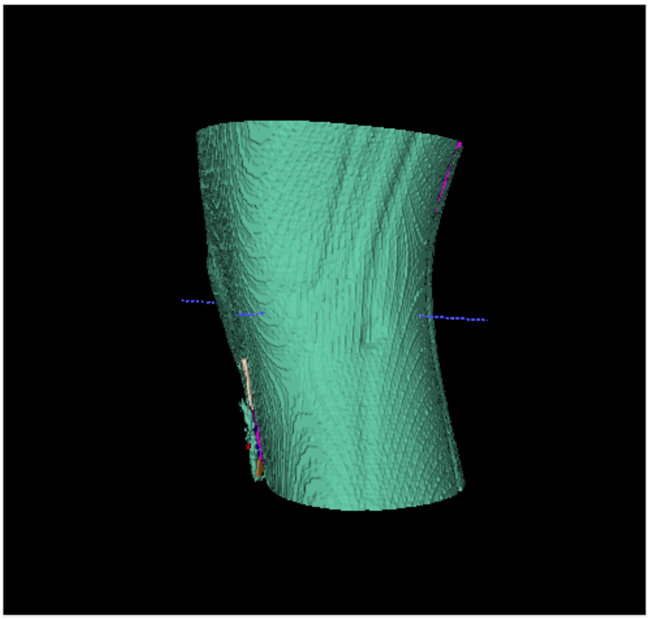


Figure 6.4: Pixel dimension 0.5mm

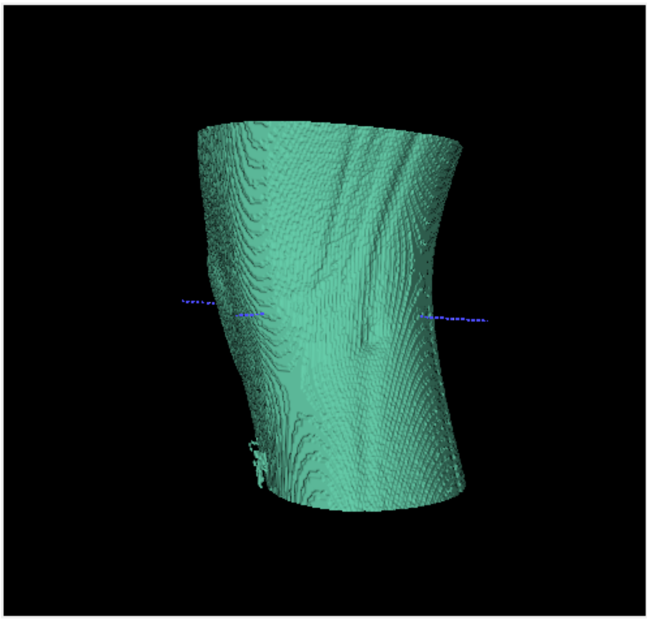


Figure 6.5: Pixel dimension 0.6mm

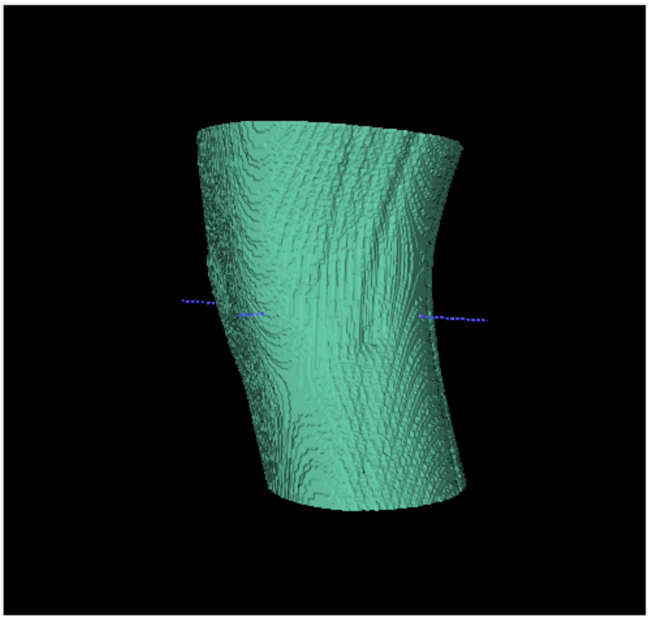


Figure 6.6: Pixel dimension 0.7mm

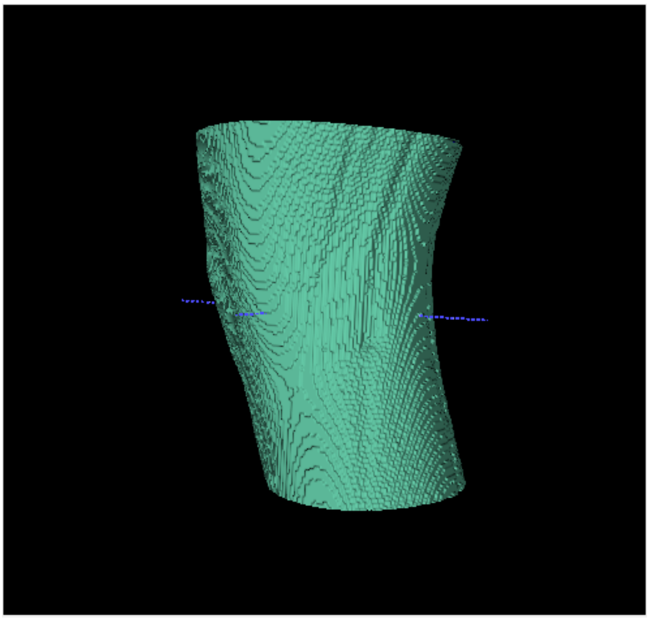


Figure 6.7: Pixel dimension 1.0mm

6.2 Spatial window size

The output inference for models with varying spatial window sizes is presented below. The first attempt at increasing this parameter was to set it to the maximum possible value of (400,400,400). This resulted in OOM errors for attempts at using both GPU and CPU. The next attempts were made at spatial window size (256,256,256). This once again resulted in an OOM error message when running on the GPU. It did however manage to run on the CPU. The downside of running on the CPU was very apparent, as the inference required hours as opposed to minutes to complete. It did however complete the inference, resulting in the output shown in figure 6.8.

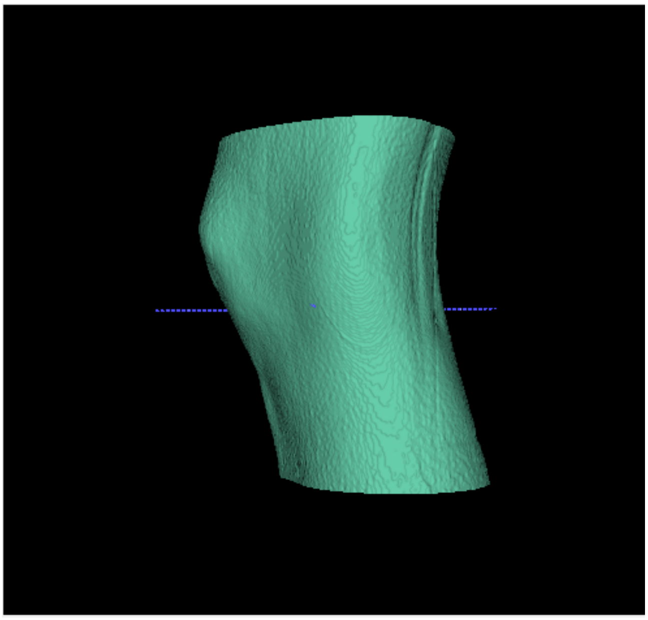


Figure 6.8: spatial window size (256,256,256) on CPU

The next part was to figure out if the GPU would be able to run an inference with a high enough spatial window to achieve similar results. The next attempt for the GPU inference was set to use a spatial window size of (192,192,192). Subsequent test increased the spatial window size by 16, as that is the constraint for this parameter in nnU-Net. The following figures present the results from this incremental increase in the spatial window size.

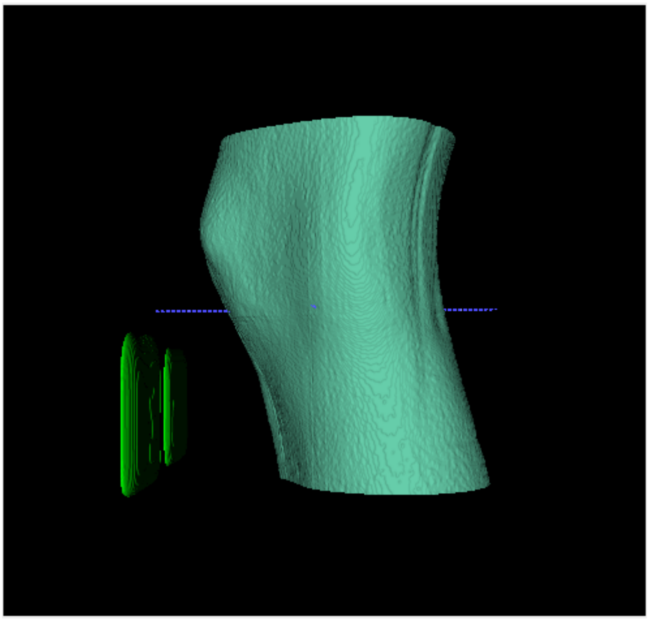


Figure 6.9: spatial window size (192,192,192) on GPU

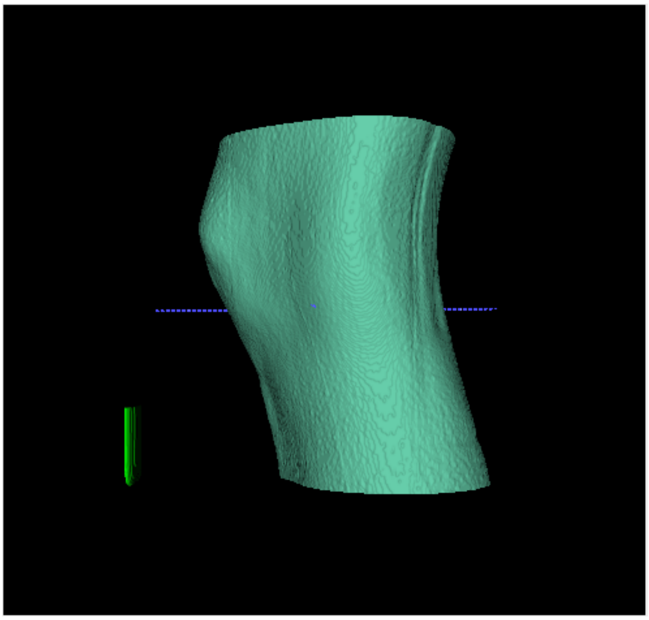


Figure 6.10: spatial window size (208,208,208) on GPU

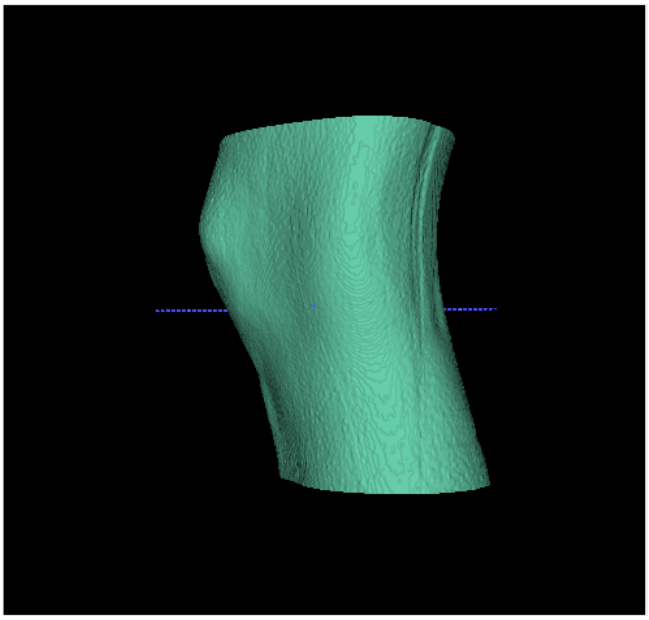
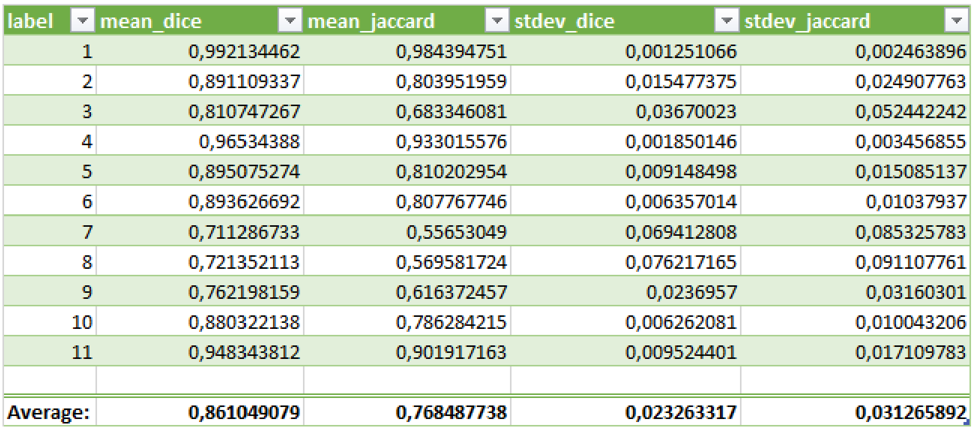


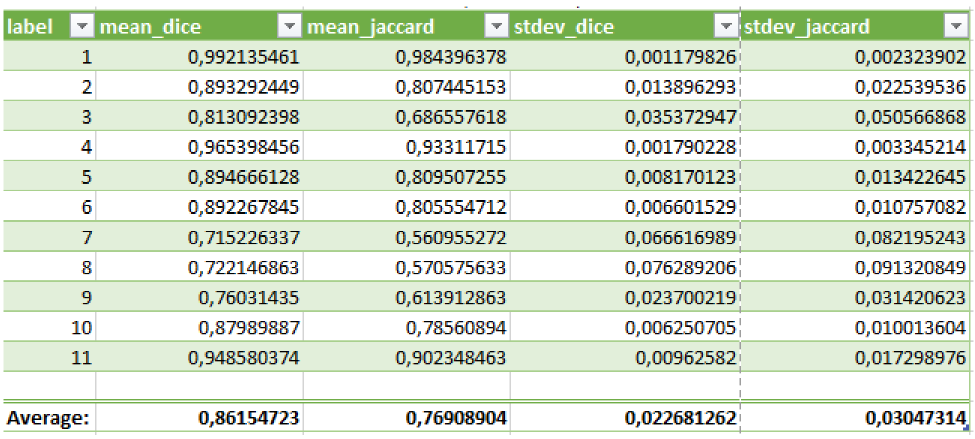
Figure 6.11: spatial window size (224,224,224) on GPU

Attempts to increase the parameter even further ran into OOM errors. The only possible increase was that one of the three axes could be increased, resulting in a spatial window size of (240,224,224). The evaluation score for spatial window size (224,224,224) is shown in figure 6.2 below, while figure 6.1 shows the score for spatial window size (240,224,224).



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,992134462	0,984394751	0,001251066	0,002463896
2	0,891109337	0,803951959	0,015477375	0,024907763
3	0,810747267	0,683346081	0,03670023	0,052442242
4	0,96534388	0,933015576	0,001850146	0,003456855
5	0,895075274	0,810202954	0,009148498	0,015085137
6	0,893626692	0,807767746	0,006357014	0,01037937
7	0,711286733	0,55653049	0,069412808	0,085325783
8	0,721352113	0,569581724	0,076217165	0,091107761
9	0,762198159	0,616372457	0,0236957	0,03160301
10	0,880322138	0,786284215	0,006262081	0,010043206
11	0,948343812	0,901917163	0,009524401	0,017109783
Average:	**0,861049079**	**0,768487738**	**0,023263317**	**0,031265892**

Table 6.1: spatial window size (224,224,224)



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,992135461	0,984396378	0,001179826	0,002323902
2	0,893292449	0,807445153	0,013896293	0,022539536
3	0,813092398	0,686557618	0,035372947	0,050566868
4	0,965398456	0,93311715	0,001790228	0,003345214
5	0,894666128	0,809507255	0,008170123	0,013422645
6	0,892267845	0,805554712	0,006601529	0,010757082
7	0,715226337	0,560955272	0,066616989	0,082195243
8	0,722146863	0,570575633	0,076289206	0,091320849
9	0,76031435	0,613912863	0,023700219	0,031420623
10	0,87989887	0,78560894	0,006250705	0,010013604
11	0,948580374	0,902348463	0,00962582	0,017298976
Average:	**0,86154723**	**0,76908904**	**0,022681262**	**0,03047314**

Table 6.2: spatial window size (240,224,224)

6.3 Loss type

While different loss types were used when comparing the parameters in the remaining section of this chapter, those are presented in the corresponding sections later in this chapter. This section will cover the results from where the learning rate is the only deviation from the baseline configuration detailed in section 4.5. The results for each loss type are displayed at various iteration counts because the number of iterations required to reach a satisfactory evaluation score is also relevant.

6.3.1 Dice loss type

Figures 6.12 - 6.16 and tables 6.3 - 6.7 shows the segmented output and evaluation score for the baseline model with Dice loss type, at intervals of 10000 iterations.

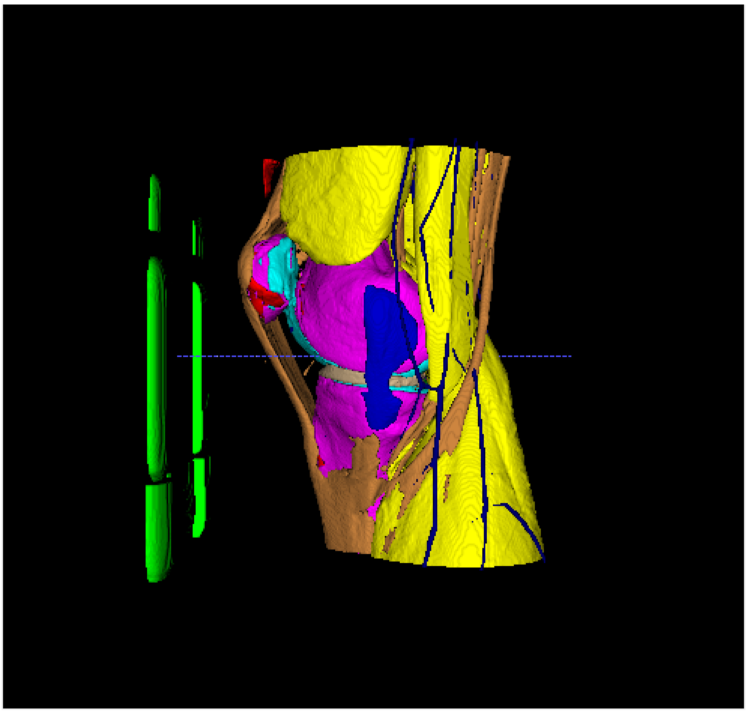
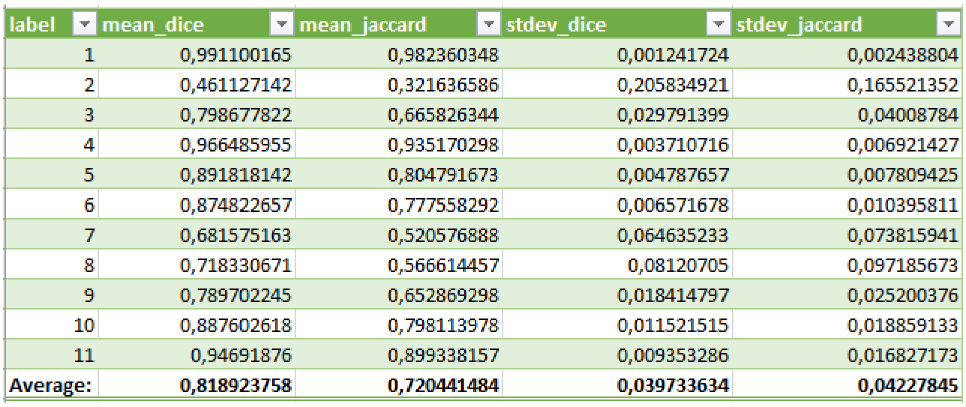


Figure 6.12: Dice baseline at 10000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,991100165	0,982360348	0,001241724	0,002438804
2	0,461127142	0,321636586	0,205834921	0,165521352
3	0,798677822	0,665826344	0,029791399	0,04008784
4	0,966485955	0,935170298	0,003710716	0,006921427
5	0,891818142	0,804791673	0,004787657	0,007809425
6	0,874822657	0,777558292	0,006571678	0,010395811
7	0,681575163	0,520576888	0,064635233	0,073815941
8	0,718330671	0,566614457	0,08120705	0,097185673
9	0,789702245	0,652869298	0,018414797	0,025200376
10	0,887602618	0,798113978	0,011521515	0,018859133
11	0,94691876	0,899338157	0,009353286	0,016827173
Average:	**0,818923758**	**0,720441484**	**0,039733634**	**0,04227845**

Table 6.3: Dice baseline at 10000 iterations

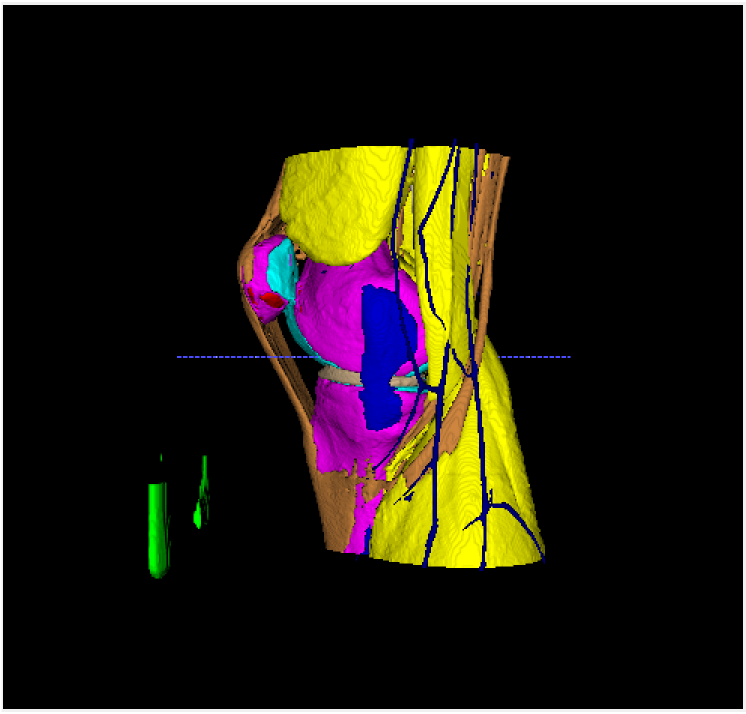
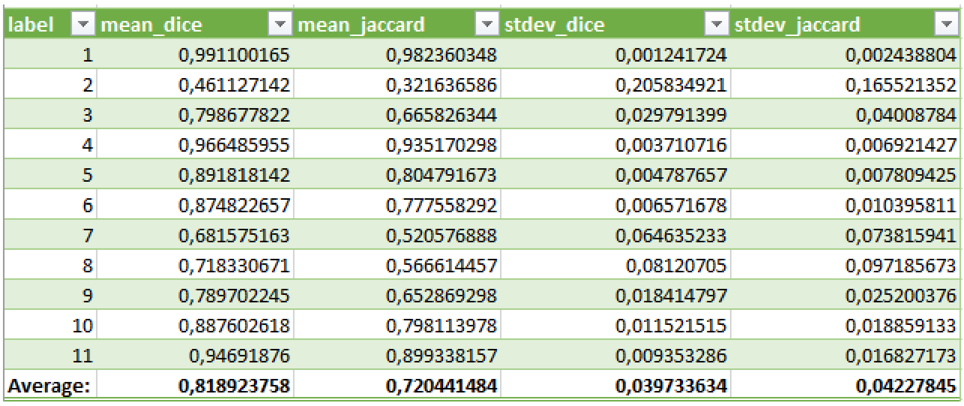


Figure 6.13: Dice baseline at 20000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,991100165	0,982360348	0,001241724	0,002438804
2	0,461127142	0,321636586	0,205834921	0,165521352
3	0,798677822	0,665826344	0,029791399	0,04008784
4	0,966485955	0,935170298	0,003710716	0,006921427
5	0,891818142	0,804791673	0,004787657	0,007809425
6	0,874822657	0,777558292	0,006571678	0,010395811
7	0,681575163	0,520576888	0,064635233	0,073815941
8	0,718330671	0,566614457	0,08120705	0,097185673
9	0,789702245	0,652869298	0,018414797	0,025200376
10	0,887602618	0,798113978	0,011521515	0,018859133
11	0,94691876	0,899338157	0,009353286	0,016827173
Average:	**0,818923758**	**0,720441484**	**0,039733634**	**0,04227845**

Table 6.4: Dice baseline at 20000 iterations

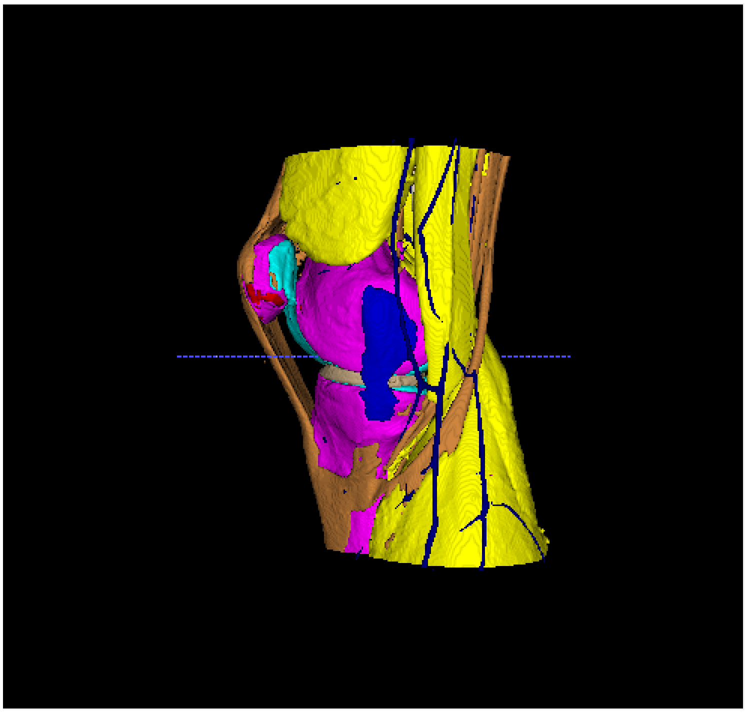
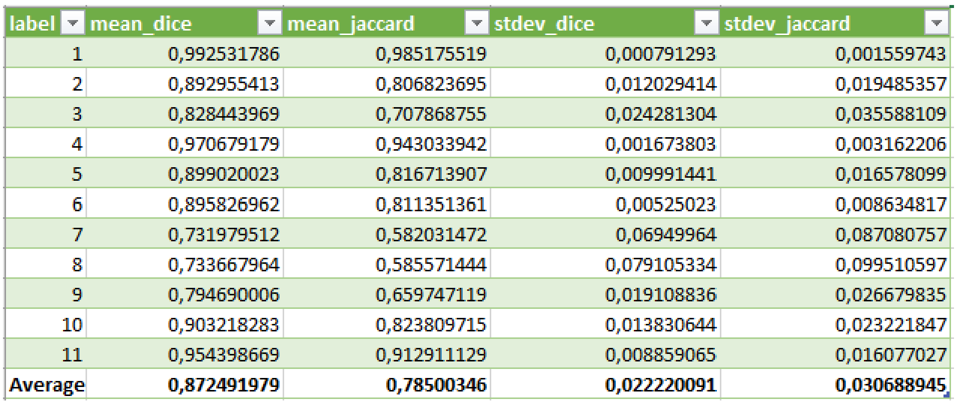


Figure 6.14: Dice baseline at 30000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,992531786	0,985175519	0,000791293	0,001559743
2	0,892955413	0,806823695	0,012029414	0,019485357
3	0,828443969	0,707868755	0,024281304	0,035588109
4	0,970679179	0,943033942	0,001673803	0,003162206
5	0,899020023	0,816713907	0,009991441	0,016578099
6	0,895826962	0,811351361	0,00525023	0,008634817
7	0,731979512	0,582031472	0,06949964	0,087080757
8	0,733667964	0,585571444	0,079105334	0,099510597
9	0,794690006	0,659747119	0,019108836	0,026679835
10	0,903218283	0,823809715	0,013830644	0,023221847
11	0,954398669	0,912911129	0,008859065	0,016077027
Average	**0,872491979**	**0,78500346**	**0,022220091**	**0,030688945**

Table 6.5: Dice baseline at 30000 iterations

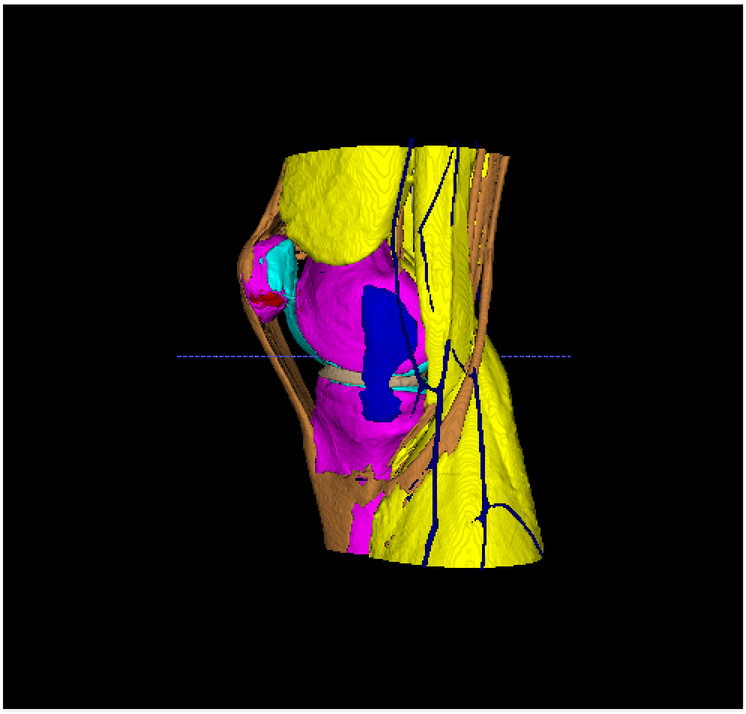
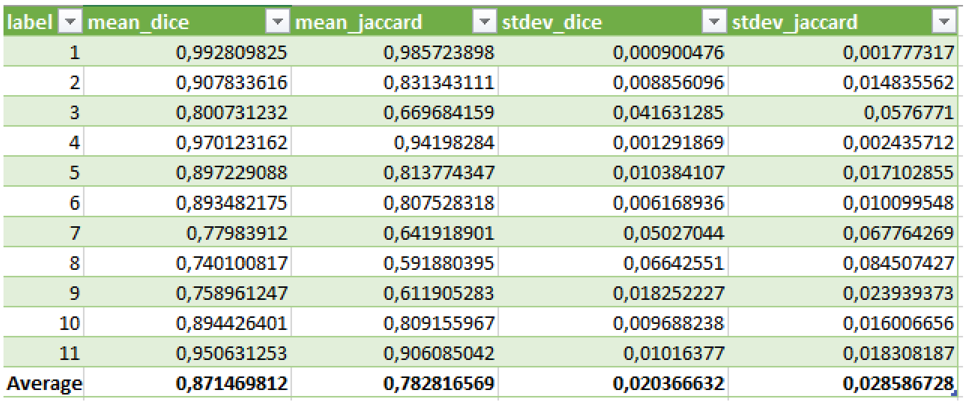


Figure 6.15: Dice baseline at 40000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,992809825	0,985723898	0,000900476	0,001777317
2	0,907833616	0,831343111	0,008856096	0,014835562
3	0,800731232	0,669684159	0,041631285	0,0576771
4	0,970123162	0,94198284	0,001291869	0,002435712
5	0,897229088	0,813774347	0,010384107	0,017102855
6	0,893482175	0,807528318	0,006168936	0,010099548
7	0,77983912	0,641918901	0,05027044	0,067764269
8	0,740100817	0,591880395	0,06642551	0,084507427
9	0,758961247	0,611905283	0,018252227	0,023939373
10	0,894426401	0,809155967	0,009688238	0,016006656
11	0,950631253	0,906085042	0,01016377	0,018308187
Average	**0,871469812**	**0,782816569**	**0,020366632**	**0,028586728**

Table 6.6: Dice baseline at 40000 iterations

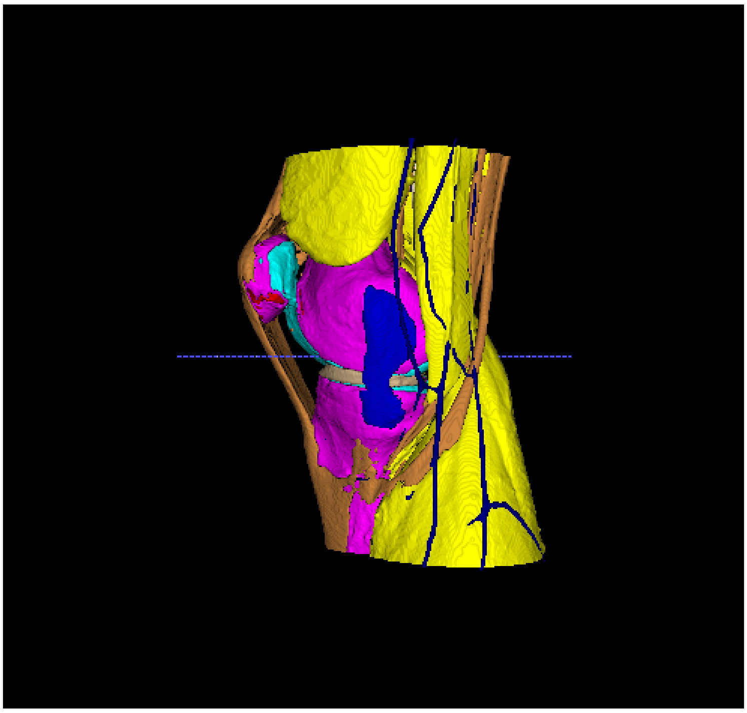
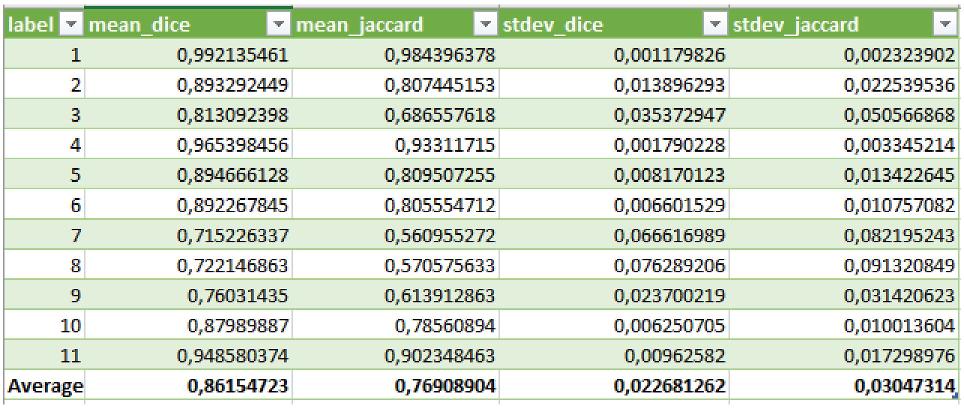


Figure 6.16: Dice baseline at 50000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,992135461	0,984396378	0,001179826	0,002323902
2	0,893292449	0,807445153	0,013896293	0,022539536
3	0,813092398	0,686557618	0,035372947	0,050566868
4	0,965398456	0,93311715	0,001790228	0,003345214
5	0,894666128	0,809507255	0,008170123	0,013422645
6	0,892267845	0,805554712	0,006601529	0,010757082
7	0,715226337	0,560955272	0,066616989	0,082195243
8	0,722146863	0,570575633	0,076289206	0,091320849
9	0,76031435	0,613912863	0,023700219	0,031420623
10	0,87989887	0,78560894	0,006250705	0,010013604
11	0,948580374	0,902348463	0,00962582	0,017298976
Average	**0,86154723**	**0,76908904**	**0,022681262**	**0,03047314**

Table 6.7: Dice baseline at 50000 iterations

6.3.2 DicePlusXEnt loss type

Figures 6.17 - 6.21 and tables 6.8 - 6.12 shows the segmented output and evaluation score for the baseline model with DicePlusXEnt loss type, at intervals of 10000 iterations.

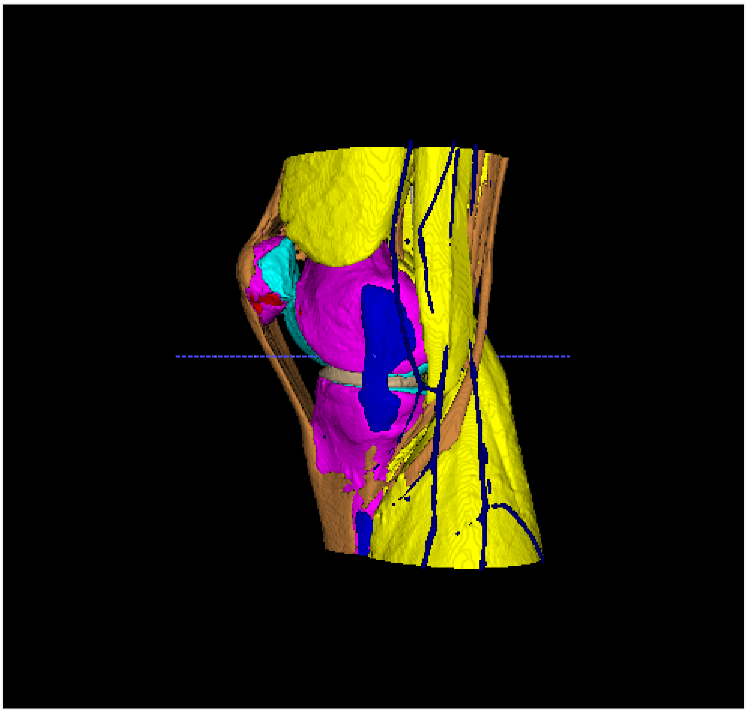
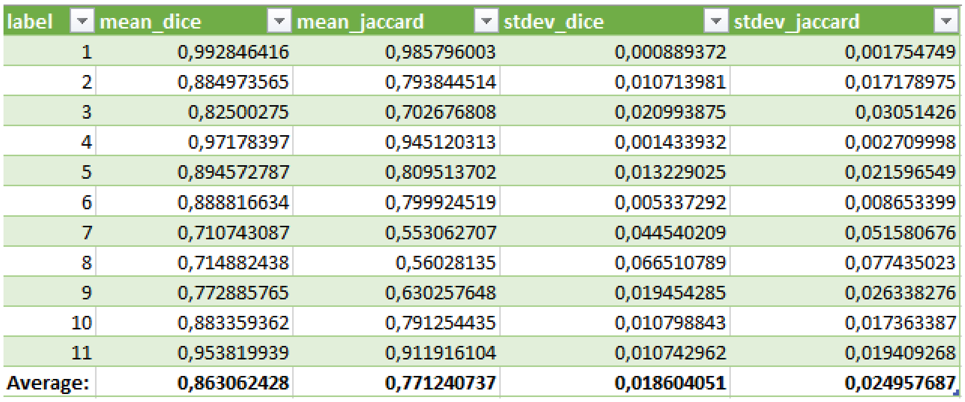


Figure 6.17: DicePlusXEnt baseline at 10000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,992846416	0,985796003	0,000889372	0,001754749
2	0,884973565	0,793844514	0,010713981	0,017178975
3	0,82500275	0,702676808	0,020993875	0,03051426
4	0,97178397	0,945120313	0,001433932	0,002709998
5	0,894572787	0,809513702	0,013229025	0,021596549
6	0,888816634	0,799924519	0,005337292	0,008653399
7	0,710743087	0,553062707	0,044540209	0,051580676
8	0,714882438	0,56028135	0,066510789	0,077435023
9	0,772885765	0,630257648	0,019454285	0,026338276
10	0,883359362	0,791254435	0,010798843	0,017363387
11	0,953819939	0,911916104	0,010742962	0,019409268
Average:	**0,863062428**	**0,771240737**	**0,018604051**	**0,024957687**

Table 6.8: DicePlusXEnt baseline at 10000 iterations

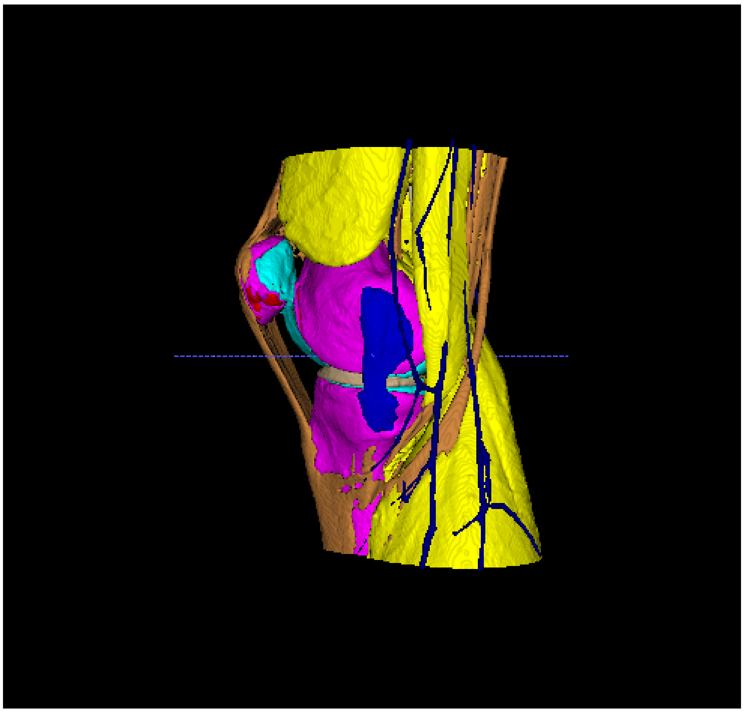
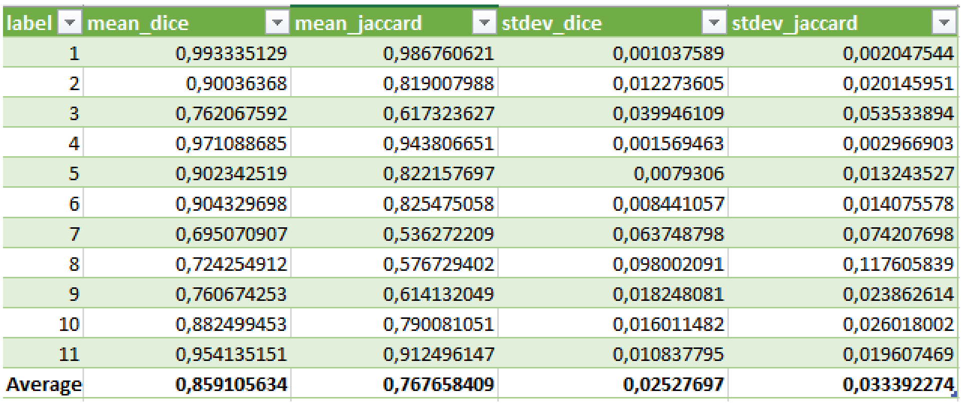


Figure 6.18: DicePlusXEnt baseline at 20000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,993335129	0,986760621	0,001037589	0,002047544
2	0,90036368	0,819007988	0,012273605	0,020145951
3	0,762067592	0,617323627	0,039946109	0,053533894
4	0,971088685	0,943806651	0,001569463	0,002966903
5	0,902342519	0,822157697	0,0079306	0,013243527
6	0,904329698	0,825475058	0,008441057	0,014075578
7	0,695070907	0,536272209	0,063748798	0,074207698
8	0,724254912	0,576729402	0,098002091	0,117605839
9	0,760674253	0,614132049	0,018248081	0,023862614
10	0,882499453	0,790081051	0,016011482	0,026018002
11	0,954135151	0,912496147	0,010837795	0,019607469
Average	**0,859105634**	**0,767658409**	**0,02527697**	**0,033392274**

Table 6.9: DicePlusXEnt baseline at 20000 iterations

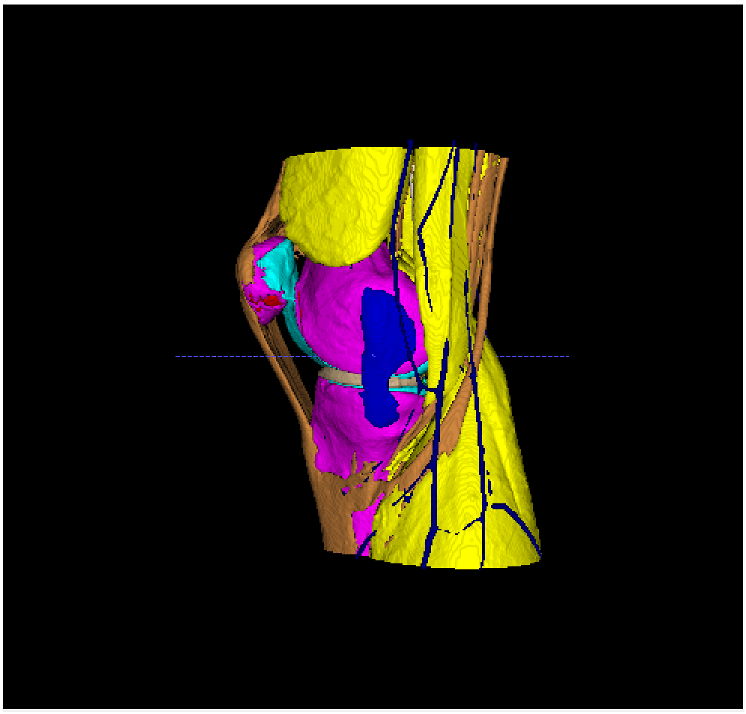
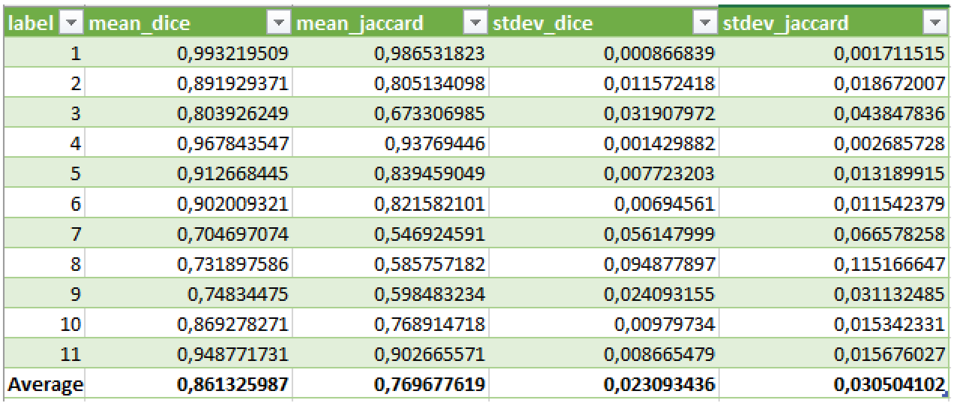


Figure 6.19: DicePlusXEnt baseline at 30000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,993219509	0,986531823	0,000866839	0,001711515
2	0,891929371	0,805134098	0,011572418	0,018672007
3	0,803926249	0,673306985	0,031907972	0,043847836
4	0,967843547	0,93769446	0,001429882	0,002685728
5	0,912668445	0,839459049	0,007723203	0,013189915
6	0,902009321	0,821582101	0,00694561	0,011542379
7	0,704697074	0,546924591	0,056147999	0,066578258
8	0,731897586	0,585757182	0,094877897	0,115166647
9	0,74834475	0,598483234	0,024093155	0,031132485
10	0,869278271	0,768914718	0,00979734	0,015342331
11	0,948771731	0,902665571	0,008665479	0,015676027
Average	**0,861325987**	**0,769677619**	**0,023093436**	**0,030504102**

Table 6.10: DicePlusXEnt baseline at 30000 iterations

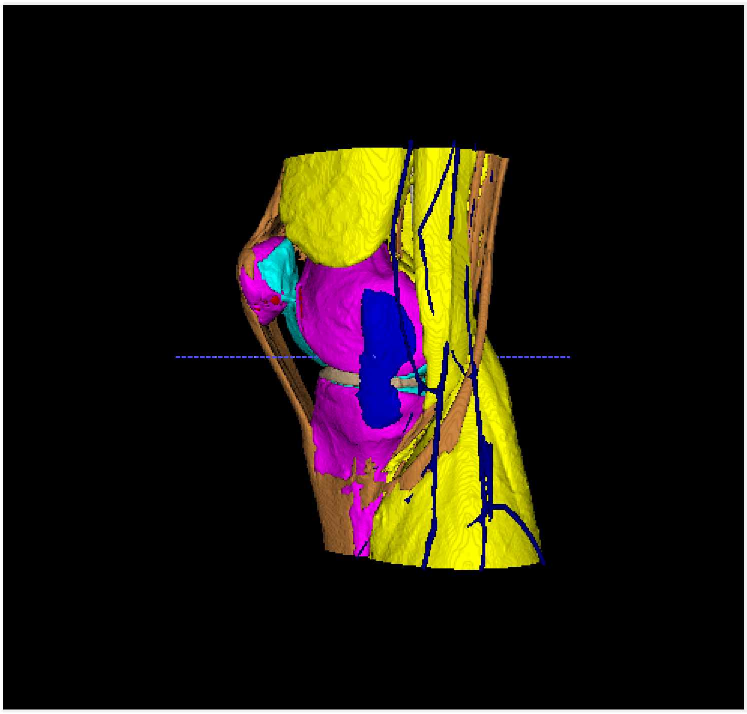
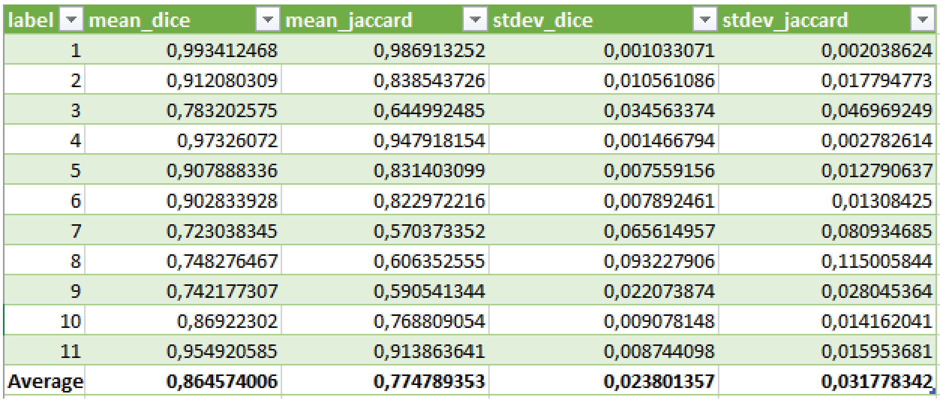


Figure 6.20: DicePlusXEnt baseline at 40000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,993412468	0,986913252	0,001033071	0,002038624
2	0,912080309	0,838543726	0,010561086	0,017794773
3	0,783202575	0,644992485	0,034563374	0,046969249
4	0,97326072	0,947918154	0,001466794	0,002782614
5	0,907888336	0,831403099	0,007559156	0,012790637
6	0,902833928	0,822972216	0,007892461	0,01308425
7	0,723038345	0,570373352	0,065614957	0,080934685
8	0,748276467	0,606352555	0,093227906	0,115005844
9	0,742177307	0,590541344	0,022073874	0,028045364
10	0,86922302	0,768809054	0,009078148	0,014162041
11	0,954920585	0,913863641	0,008744098	0,015953681
Average	**0,864574006**	**0,774789353**	**0,023801357**	**0,031778342**

Table 6.11: DicePlusXEnt baseline at 40000 iterations

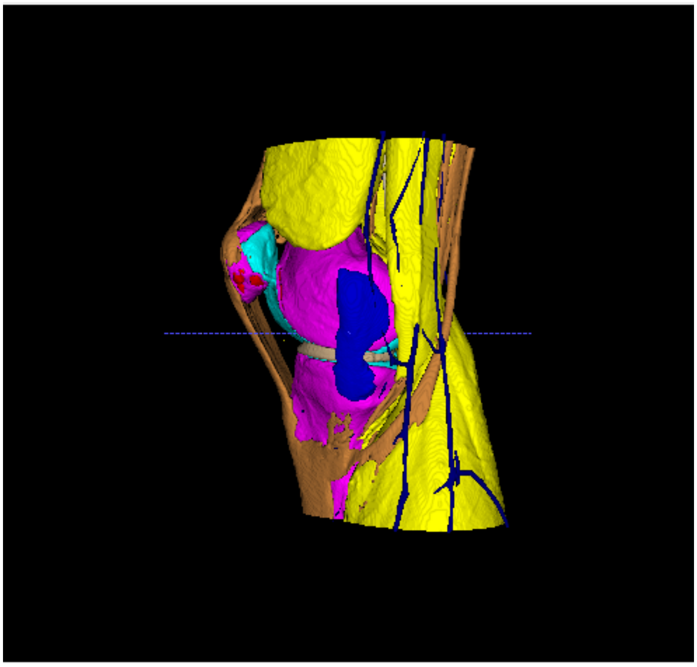
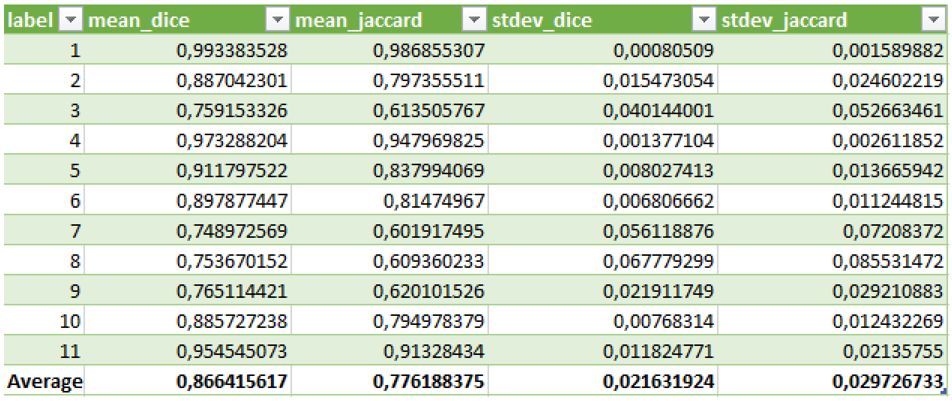


Figure 6.21: DicePlusXEnt baseline at 50000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,993383528	0,986855307	0,00080509	0,001589882
2	0,887042301	0,797355511	0,015473054	0,024602219
3	0,759153326	0,613505767	0,040144001	0,052663461
4	0,973288204	0,947969825	0,001377104	0,002611852
5	0,911797522	0,837994069	0,008027413	0,013665942
6	0,897877447	0,81474967	0,006806662	0,011244815
7	0,748972569	0,601917495	0,056118876	0,07208372
8	0,753670152	0,609360233	0,067779299	0,085531472
9	0,765114421	0,620101526	0,021911749	0,029210883
10	0,885727238	0,794978379	0,00768314	0,012432269
11	0,954545073	0,91328434	0,011824771	0,02135755
Average	**0,866415617**	**0,776188375**	**0,021631924**	**0,029726733**

Table 6.12: DicePlusXEnt baseline at 50000 iterations

6.4 Normalisation

This section provides the results for models with normalisation. These are split into categories based on which model configuration was used in addition to the normalisation. Also worth noting was that the iteration time was reduced to around 28% after adding normalisation (400 vs 1400 iterations in the same time frame). Because of this, the total number of iterations is lower for the models shown in this section.

6.4.1 Dice baseline with normalisation

Figures 6.22 - 6.25 and tables 6.13 - 6.16 shows the segmented output and evaluation score for the model with normalisation and Dice loss type, first at 5000 iterations, and then every 5000 iterations starting at 7500.

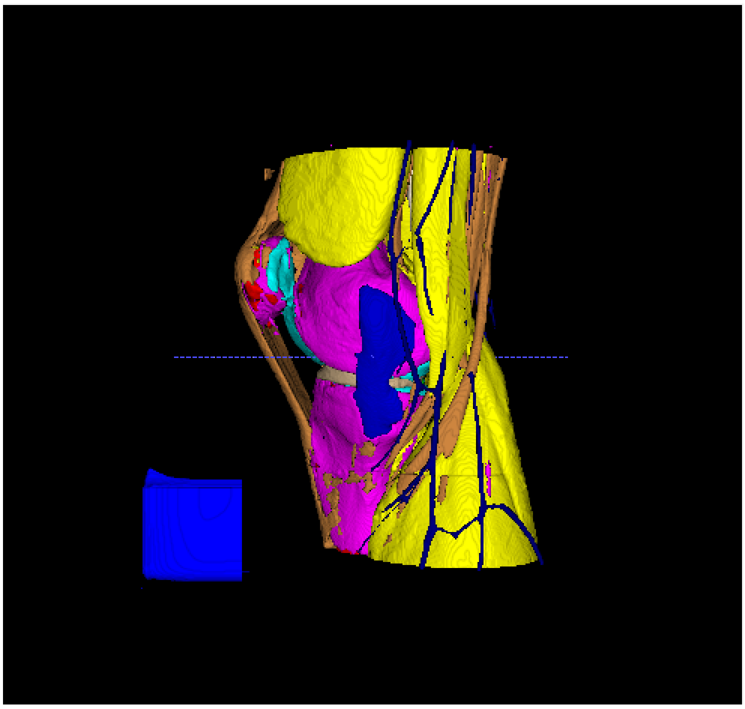
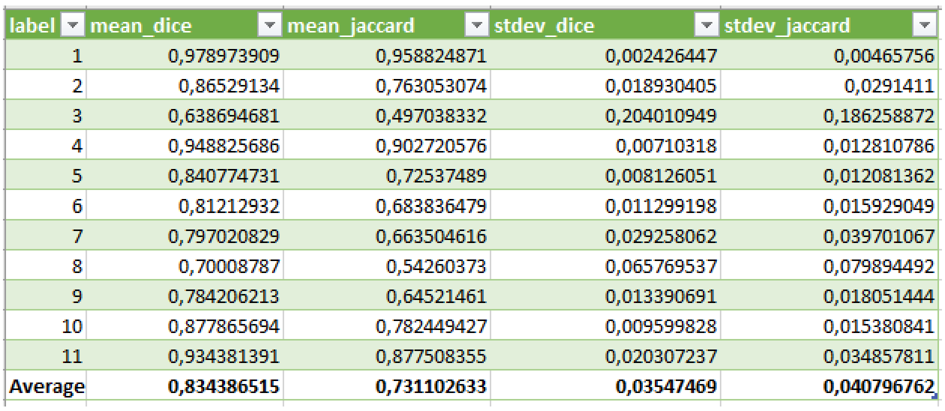


Figure 6.22: Dice baseline with normalisation at 5000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,978973909	0,958824871	0,002426447	0,00465756
2	0,86529134	0,763053074	0,018930405	0,0291411
3	0,638694681	0,497038332	0,204010949	0,186258872
4	0,948825686	0,902720576	0,00710318	0,012810786
5	0,840774731	0,72537489	0,008126051	0,012081362
6	0,81212932	0,683836479	0,011299198	0,015929049
7	0,797020829	0,663504616	0,029258062	0,039701067
8	0,70008787	0,54260373	0,065769537	0,079894492
9	0,784206213	0,64521461	0,013390691	0,018051444
10	0,877865694	0,782449427	0,009599828	0,015380841
11	0,934381391	0,877508355	0,020307237	0,034857811
Average	**0,834386515**	**0,731102633**	**0,03547469**	**0,040796762**

Table 6.13: Dice baseline with normalisation at 5000 iterations

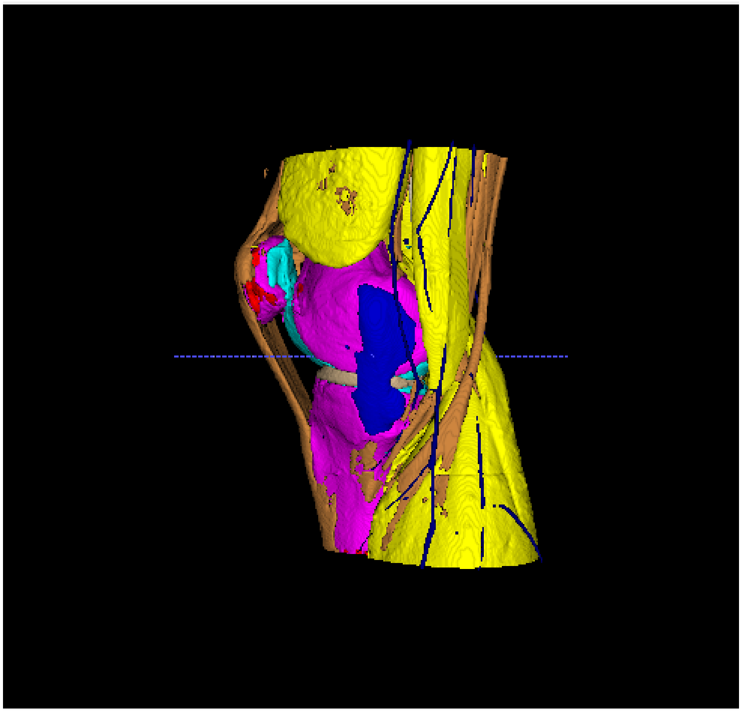


Figure 6.23: Dice baseline with normalisation at 7500 iterations

label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,982288669	0,965203399	0,002249741	0,004341635
2	0,885921254	0,795510757	0,014614249	0,023291596
3	0,834907496	0,716961486	0,016908704	0,024779599
4	0,948371427	0,901990174	0,010224676	0,018310371
5	0,871014438	0,771742911	0,013167512	0,020670884
6	0,83108437	0,711089596	0,008989126	0,01327997
7	0,735518025	0,585023821	0,059202019	0,071569603
8	0,754552562	0,608810661	0,053494635	0,069050999
9	0,774157923	0,631741197	0,013859125	0,018546501
10	0,887624161	0,798426338	0,018084724	0,029088457
11	0,939662809	0,886608352	0,0158786	0,027775145
Average	**0,858645739**	**0,761191699**	**0,020606646**	**0,029154978**

Table 6.14: Dice baseline with normalisation at 7500 iterations

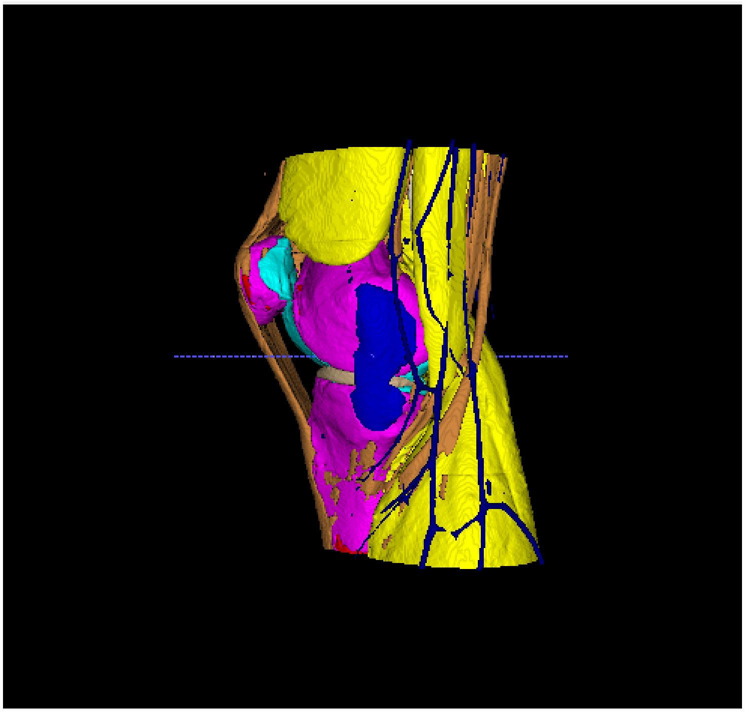
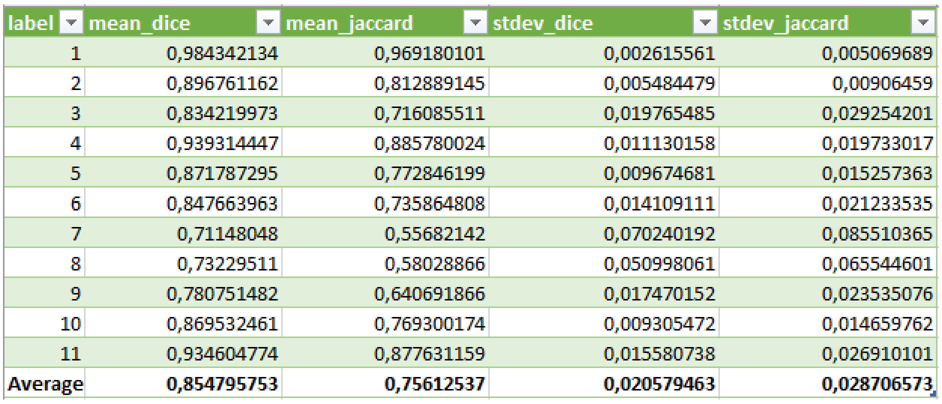


Figure 6.24: Dice baseline with normalisation at 12500 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,984342134	0,969180101	0,002615561	0,005069689
2	0,896761162	0,812889145	0,005484479	0,00906459
3	0,834219973	0,716085511	0,019765485	0,029254201
4	0,939314447	0,885780024	0,011130158	0,019733017
5	0,871787295	0,772846199	0,009674681	0,015257363
6	0,847663963	0,735864808	0,014109111	0,021233535
7	0,71148048	0,55682142	0,070240192	0,085510365
8	0,73229511	0,58028866	0,050998061	0,065544601
9	0,780751482	0,640691866	0,017470152	0,023535076
10	0,869532461	0,769300174	0,009305472	0,014659762
11	0,934604774	0,877631159	0,015580738	0,026910101
Average	**0,854795753**	**0,75612537**	**0,020579463**	**0,028706573**

Table 6.15: Dice baseline with normalisation at 12500 iterations

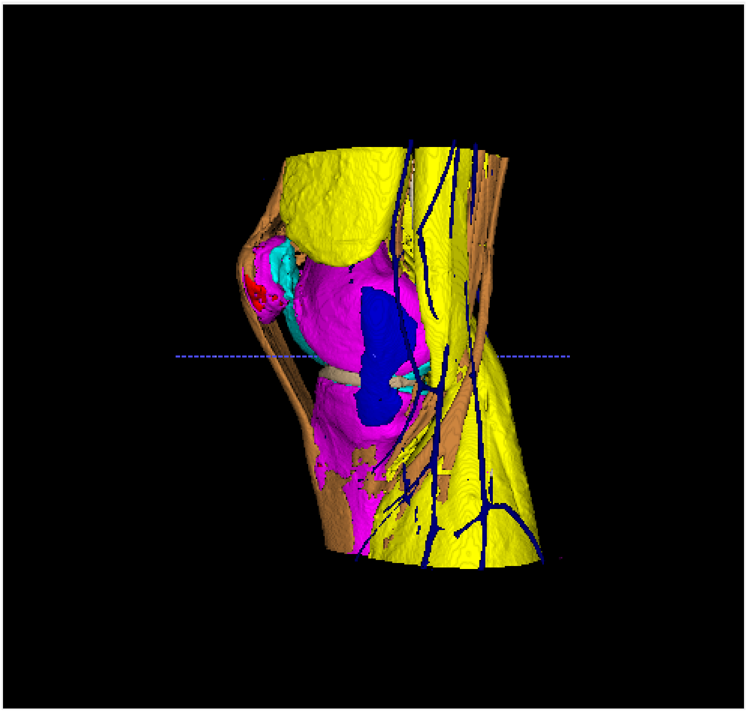
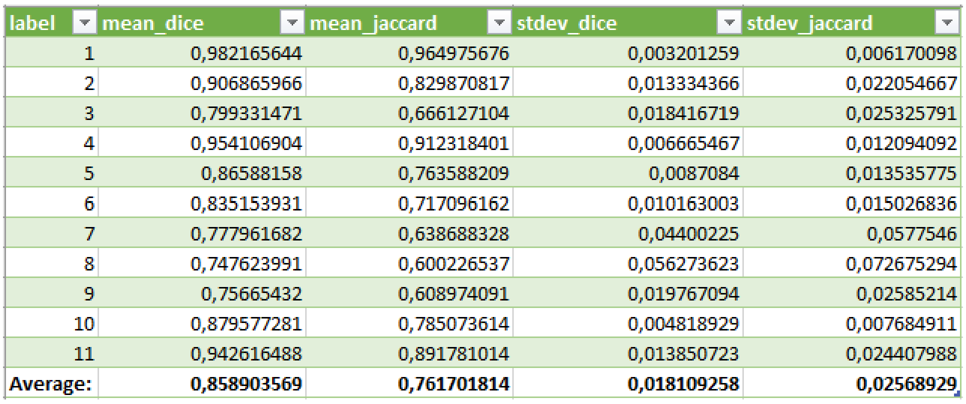


Figure 6.25: Dice baseline with normalisation at 17500 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,982165644	0,964975676	0,003201259	0,006170098
2	0,906865966	0,829870817	0,013334366	0,022054667
3	0,799331471	0,666127104	0,018416719	0,025325791
4	0,954106904	0,912318401	0,006665467	0,012094092
5	0,86588158	0,763588209	0,0087084	0,013535775
6	0,835153931	0,717096162	0,010163003	0,015026836
7	0,777961682	0,638688328	0,04400225	0,0577546
8	0,747623991	0,600226537	0,056273623	0,072675294
9	0,75665432	0,608974091	0,019767094	0,02585214
10	0,879577281	0,785073614	0,004818929	0,007684911
11	0,942616488	0,891781014	0,013850723	0,024407988
Average:	**0,858903569**	**0,761701814**	**0,018109258**	**0,02568929**

Table 6.16: Dice baseline with normalisation at 17500 iterations

6.4.2 DicePlusXEnt baseline with normalisation

Figures 6.26 - 6.28 and tables 6.17 - 6.19 shows the segmented output and evaluation score for the model with normalisation and DicePlusXEnt loss type every 5000 iterations.

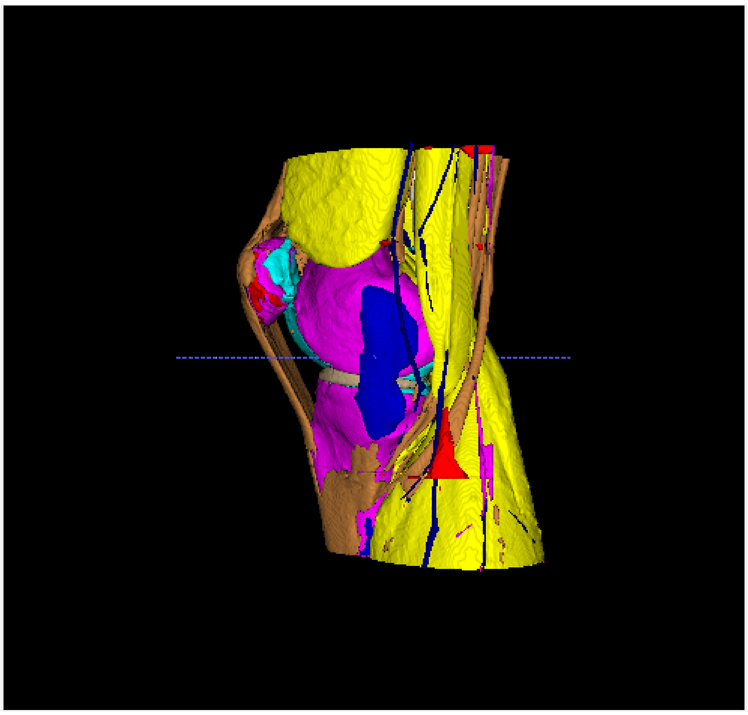
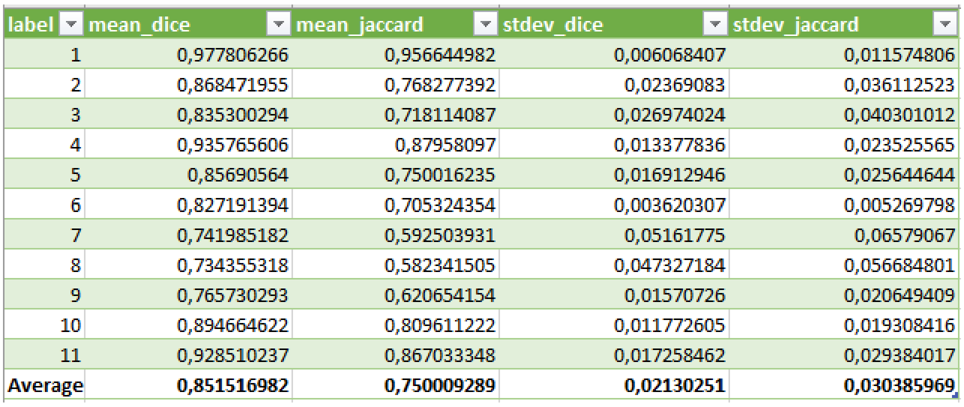


Figure 6.26: Dice baseline with normalisation at 5000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,977806266	0,956644982	0,006068407	0,011574806
2	0,868471955	0,768277392	0,02369083	0,036112523
3	0,835300294	0,718114087	0,026974024	0,040301012
4	0,935765606	0,87958097	0,013377836	0,023525565
5	0,85690564	0,750016235	0,016912946	0,025644644
6	0,827191394	0,705324354	0,003620307	0,005269798
7	0,741985182	0,592503931	0,05161775	0,06579067
8	0,734355318	0,582341505	0,047327184	0,056684801
9	0,765730293	0,620654154	0,01570726	0,020649409
10	0,894664622	0,809611222	0,011772605	0,019308416
11	0,928510237	0,867033348	0,017258462	0,029384017
Average	**0,851516982**	**0,750009289**	**0,02130251**	**0,030385969**

Table 6.17: DicePlusXEnt baseline with normalisation at 5000 iterations

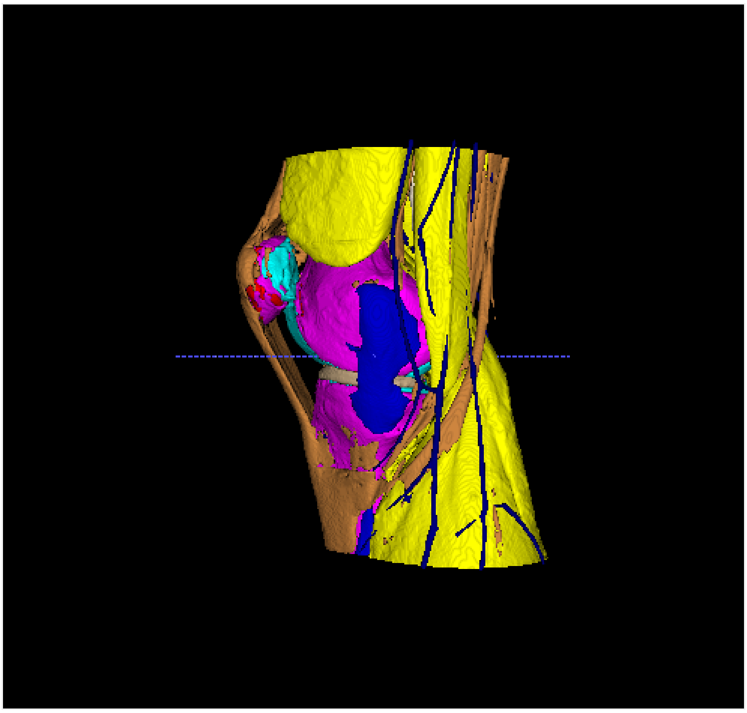
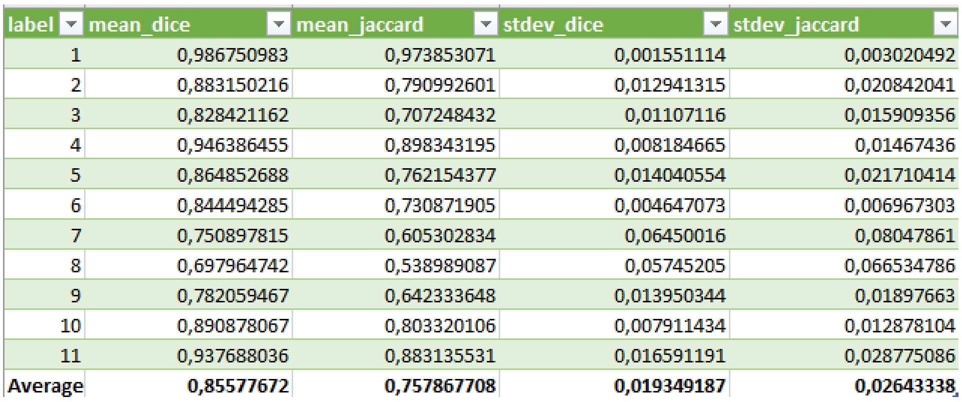


Figure 6.27: Dice baseline with normalisation at 10000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,986750983	0,973853071	0,001551114	0,003020492
2	0,883150216	0,790992601	0,012941315	0,020842041
3	0,828421162	0,707248432	0,01107116	0,015909356
4	0,946386455	0,898343195	0,008184665	0,01467436
5	0,864852688	0,762154377	0,014040554	0,021710414
6	0,844494285	0,730871905	0,004647073	0,006967303
7	0,750897815	0,605302834	0,06450016	0,08047861
8	0,697964742	0,538989087	0,05745205	0,066534786
9	0,782059467	0,642333648	0,013950344	0,01897663
10	0,890878067	0,803320106	0,007911434	0,012878104
11	0,937688036	0,883135531	0,016591191	0,028775086
Average	**0,85577672**	**0,757867708**	**0,019349187**	**0,02643338**

Table 6.18: DicePlusXEnt baseline with normalisation at 10000 iterations

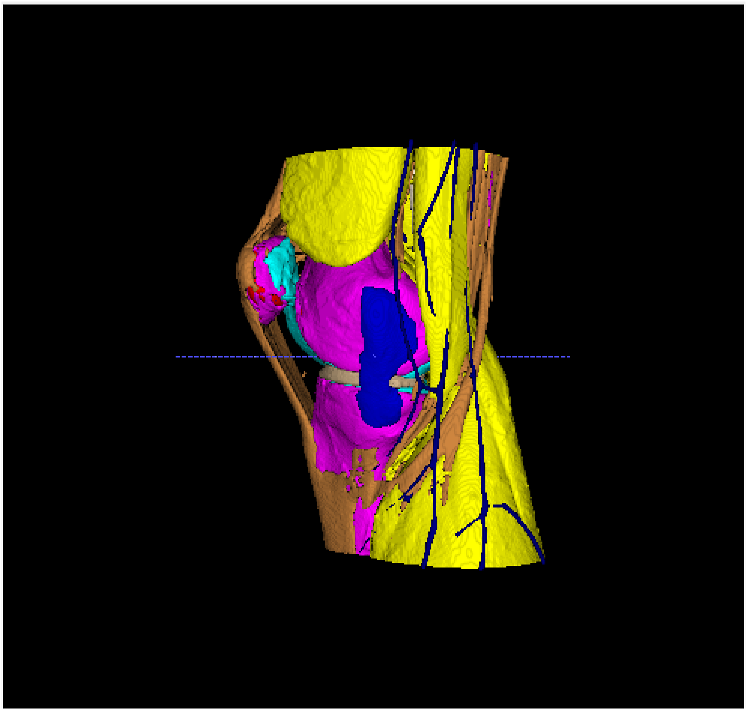
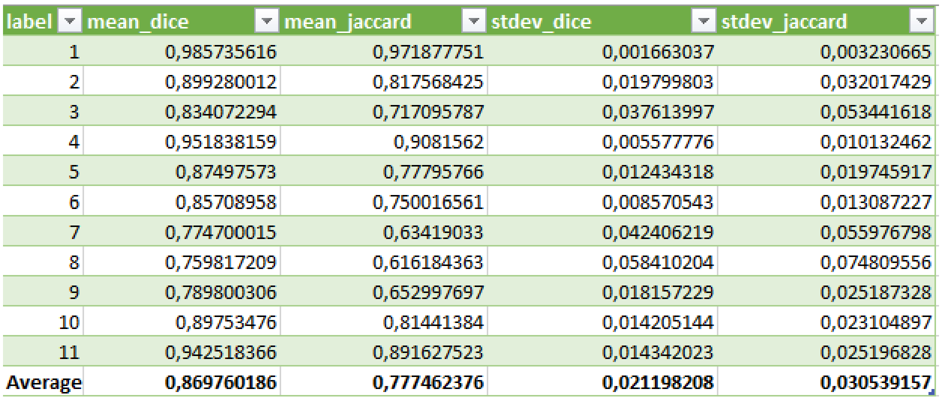


Figure 6.28: Dice baseline with normalisation at 15000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,985735616	0,971877751	0,001663037	0,003230665
2	0,899280012	0,817568425	0,019799803	0,032017429
3	0,834072294	0,717095787	0,037613997	0,053441618
4	0,951838159	0,9081562	0,005577776	0,010132462
5	0,87497573	0,77795766	0,012434318	0,019745917
6	0,85708958	0,750016561	0,008570543	0,013087227
7	0,774700015	0,63419033	0,042406219	0,055976798
8	0,759817209	0,616184363	0,058410204	0,074809556
9	0,789800306	0,652997697	0,018157229	0,025187328
10	0,89753476	0,81441384	0,014205144	0,023104897
11	0,942518366	0,891627523	0,014342023	0,025196828
Average	**0,869760186**	**0,777462376**	**0,021198208**	**0,030539157**

Table 6.19: DicePlusXEnt baseline with normalisation at 15000 iterations

6.4.3 DicePlusXEnt with 0.001 learning rate and normalisation

Figures 6.29 and 6.30, and tables 6.20 and 6.21 shows the segmented output and evaluation score for the model with normalisation, DicePlusXEnt loss type, and learning rate of 0.001, at 10000 and 15000 iterations respectively.

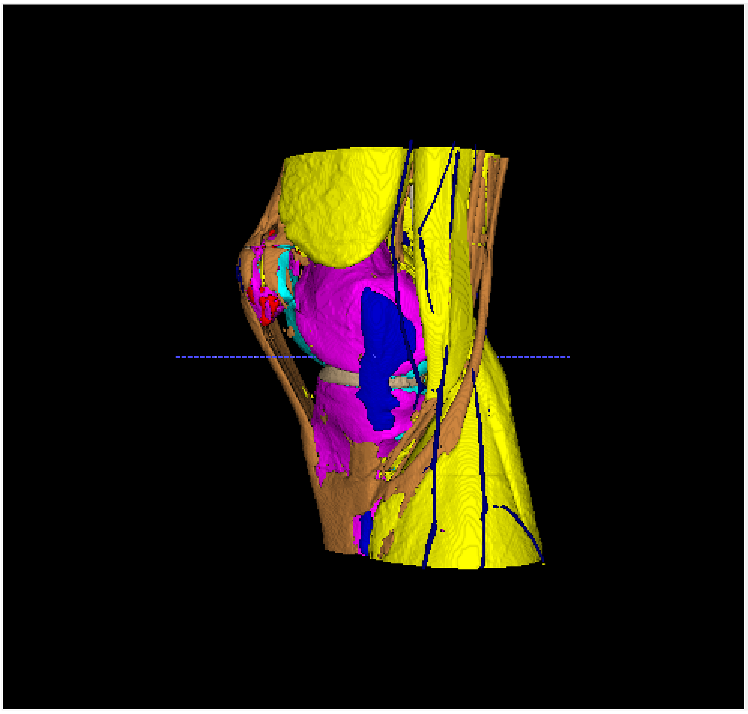
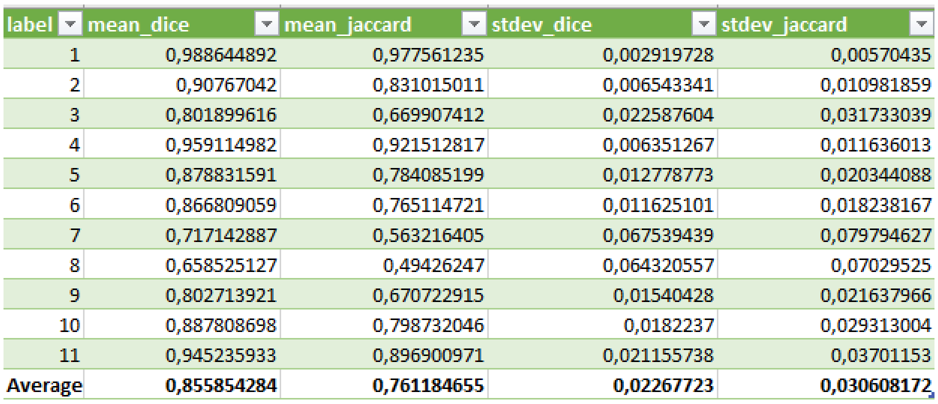


Figure 6.29: Dice baseline with 0.001 learning rate and normalisation at 10000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,988644892	0,977561235	0,002919728	0,00570435
2	0,90767042	0,831015011	0,006543341	0,010981859
3	0,801899616	0,669907412	0,022587604	0,031733039
4	0,959114982	0,921512817	0,006351267	0,011636013
5	0,878831591	0,784085199	0,012778773	0,020344088
6	0,866809059	0,765114721	0,011625101	0,018238167
7	0,717142887	0,563216405	0,067539439	0,079794627
8	0,658525127	0,49426247	0,064320557	0,07029525
9	0,802713921	0,670722915	0,01540428	0,021637966
10	0,887808698	0,798732046	0,0182237	0,029313004
11	0,945235933	0,896900971	0,021155738	0,03701153
Average	**0,855854284**	**0,761184655**	**0,02267723**	**0,030608172**

Table 6.20: DicePlusXEnt baseline with 0.001 learning rate and normalisation at 10000 iterations

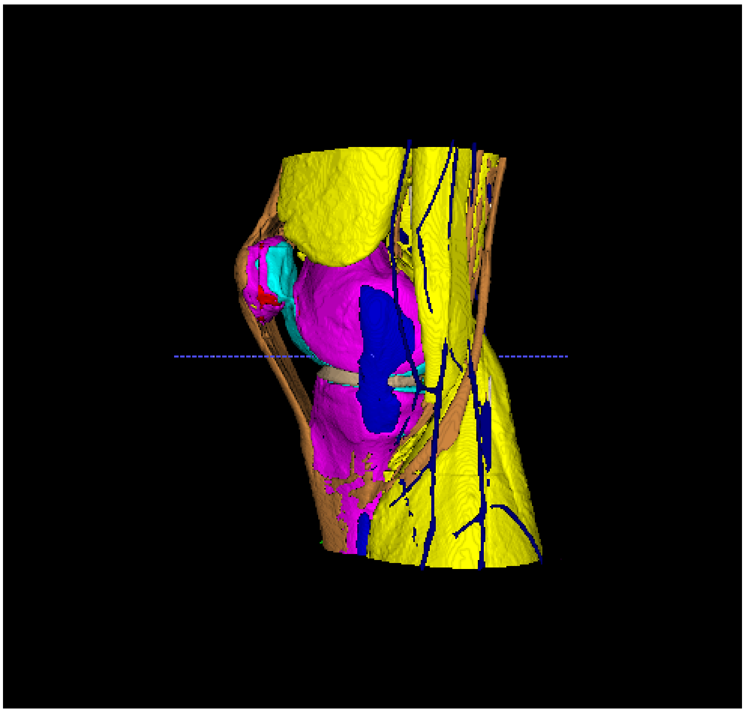
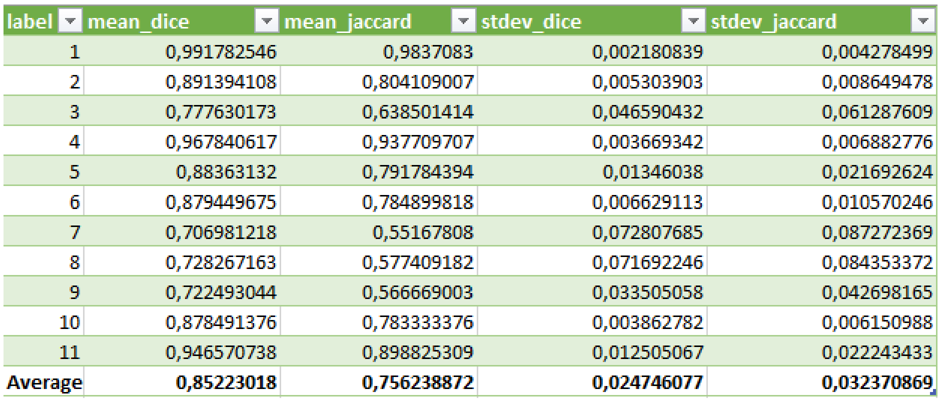


Figure 6.30: Dice baseline with 0.001 learning rate and normalisation at 15000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,991782546	0,9837083	0,002180839	0,004278499
2	0,891394108	0,804109007	0,005303903	0,008649478
3	0,777630173	0,638501414	0,046590432	0,061287609
4	0,967840617	0,937709707	0,003669342	0,006882776
5	0,88363132	0,791784394	0,01346038	0,021692624
6	0,879449675	0,784899818	0,006629113	0,010570246
7	0,706981218	0,55167808	0,072807685	0,087272369
8	0,728267163	0,577409182	0,071692246	0,084353372
9	0,722493044	0,566669003	0,033505058	0,042698165
10	0,878491376	0,783333376	0,003862782	0,006150988
11	0,946570738	0,898825309	0,012505067	0,022243433
Average	**0,85223018**	**0,756238872**	**0,024746077**	**0,032370869**

Table 6.21: DicePlusXEnt baseline with 0.001 learning rate and normalisation at 15000 iterations

6.5 Learning rate

When experimenting with the learning rate, the spatial window size parameter had not been increased for inference as detailed in section 6.2. And because of this, the Dice-PlusXEnt loss type was seen as superior to the normal Dice loss, because of the severity of the ”false positive volume”. This can be seen in figure 6.1 and figure 6.2, showing the results of Dice and DicePlusXEnt respectively. This is the reason for the following section regarding learning rate experimentation were performed with the DicePlusXEnt loss type.

The baseline learning rate for DicePlusXEnt has already been presented in section 6.3.2, and will therefore not be reiterated in this section. The following sections will present the results from varying learning rates in ascending order, starting at 0.00001 in section 6.5.1, then 0.001 in section 6.5.2, and lastly 0.01 in section 6.5.3

6.5.1 Learning rate 0.00001

Figures 6.31 - 6.34 and tables 6.22 - 6.25 shows the segmented output and evaluation score for the model with DicePlusXEnt loss type and 0.00001 learning rate, every 10000 iterations.

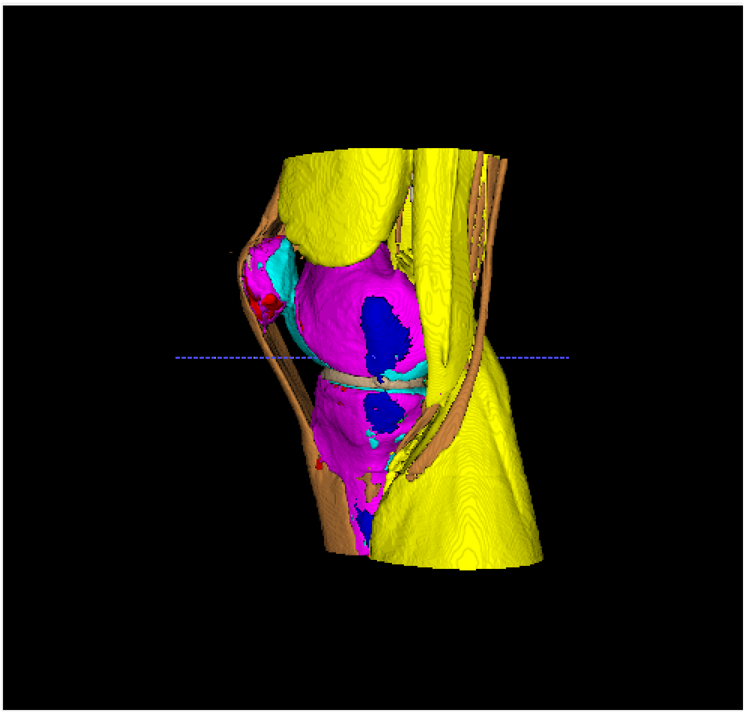
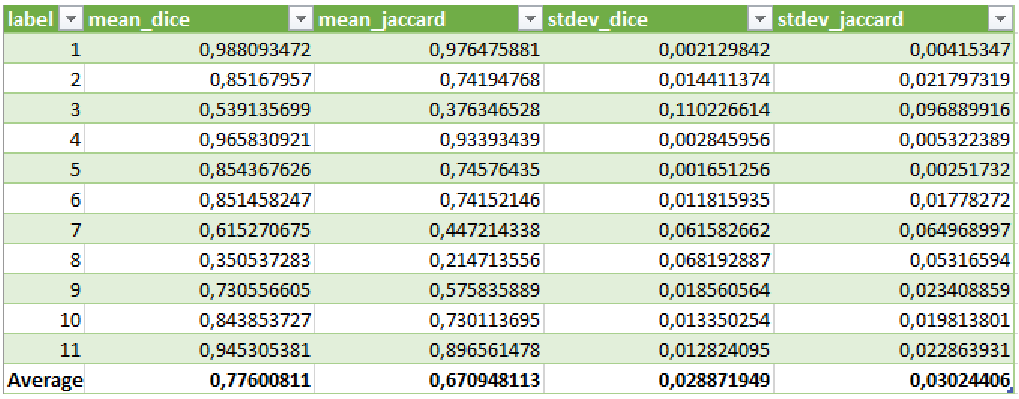


Figure 6.31: DicePlusXEnt baseline with 0.00001 learning rate at 10000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,988093472	0,976475881	0,002129842	0,00415347
2	0,85167957	0,74194768	0,014411374	0,021797319
3	0,539135699	0,376346528	0,110226614	0,096889916
4	0,965830921	0,93393439	0,002845956	0,005322389
5	0,854367626	0,74576435	0,001651256	0,00251732
6	0,851458247	0,74152146	0,011815935	0,01778272
7	0,615270675	0,447214338	0,061582662	0,064968997
8	0,350537283	0,214713556	0,068192887	0,05316594
9	0,730556605	0,575835889	0,018560564	0,023408859
10	0,843853727	0,730113695	0,013350254	0,019813801
11	0,945305381	0,896561478	0,012824095	0,022863931
Average	**0,77600811**	**0,670948113**	**0,028871949**	**0,03024406**

Table 6.22: DicePlusXEnt baseline with 0.00001 learning rate at 10000 iterations

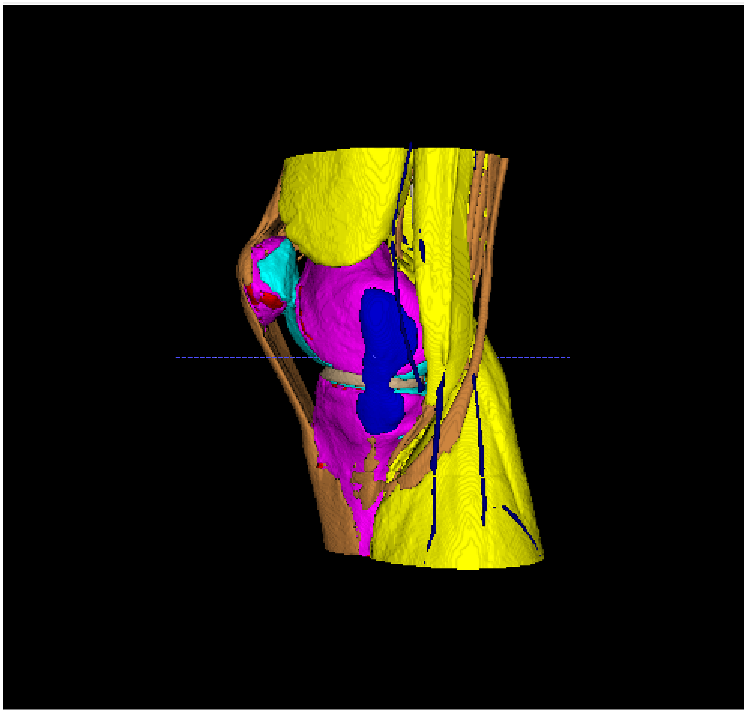
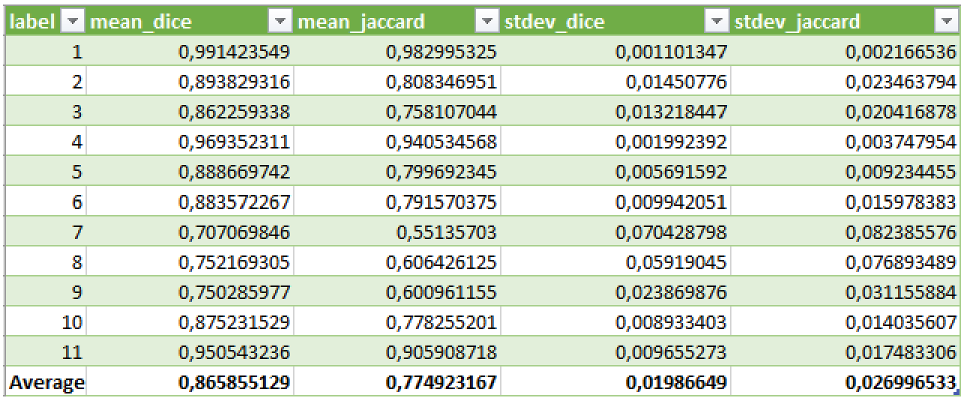


Figure 6.32: DicePlusXEnt baseline with 0.00001 learning rate at 20000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,991423549	0,982995325	0,001101347	0,002166536
2	0,893829316	0,808346951	0,01450776	0,023463794
3	0,862259338	0,758107044	0,013218447	0,020416878
4	0,969352311	0,940534568	0,001992392	0,003747954
5	0,888669742	0,799692345	0,005691592	0,009234455
6	0,883572267	0,791570375	0,009942051	0,015978383
7	0,707069846	0,55135703	0,070428798	0,082385576
8	0,752169305	0,606426125	0,05919045	0,076893489
9	0,750285977	0,600961155	0,023869876	0,031155884
10	0,875231529	0,778255201	0,008933403	0,014035607
11	0,950543236	0,905908718	0,009655273	0,017483306
Average	**0,865855129**	**0,774923167**	**0,01986649**	**0,026996533**

Table 6.23: DicePlusXEnt baseline with 0.00001 learning rate at 20000 iterations

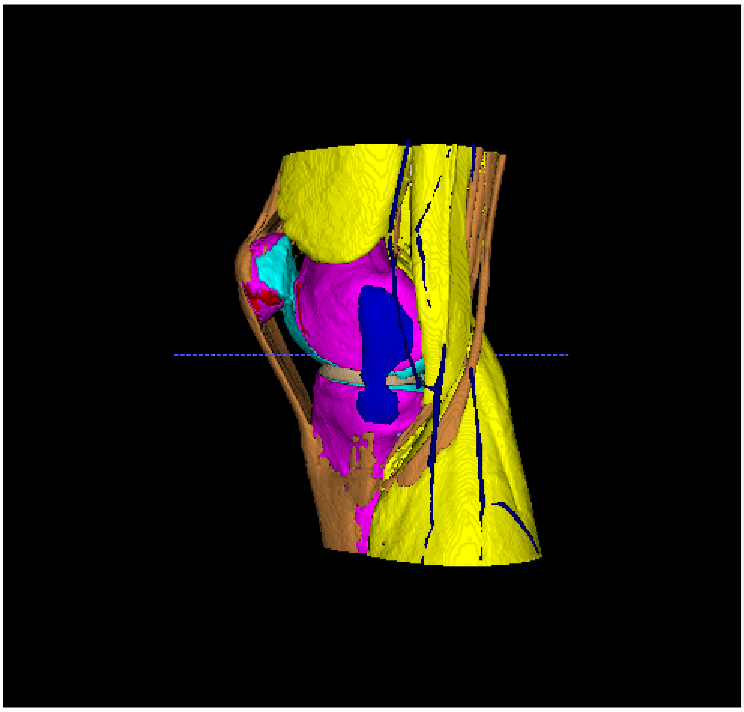
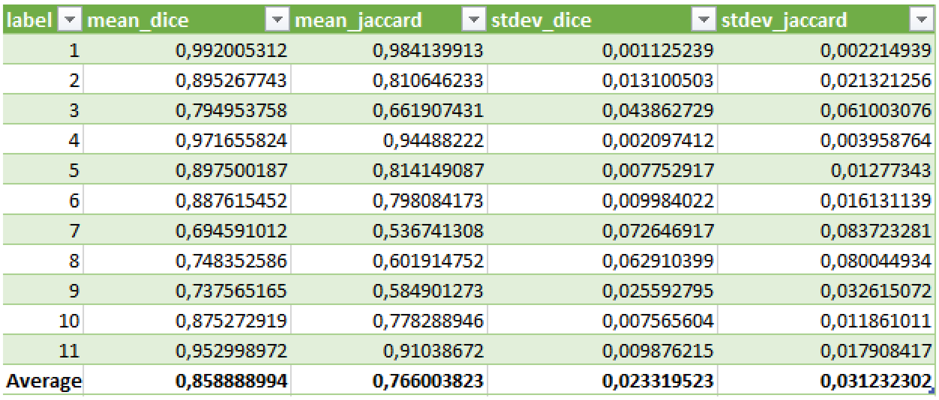


Figure 6.33: DicePlusXEnt baseline with 0.00001 learning rate at 30000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,992005312	0,984139913	0,001125239	0,002214939
2	0,895267743	0,810646233	0,013100503	0,021321256
3	0,794953758	0,661907431	0,043862729	0,061003076
4	0,971655824	0,94488222	0,002097412	0,003958764
5	0,897500187	0,814149087	0,007752917	0,01277343
6	0,887615452	0,798084173	0,009984022	0,016131139
7	0,694591012	0,536741308	0,072646917	0,083723281
8	0,748352586	0,601914752	0,062910399	0,080044934
9	0,737565165	0,584901273	0,025592795	0,032615072
10	0,875272919	0,778288946	0,007565604	0,011861011
11	0,952998972	0,91038672	0,009876215	0,017908417
Average	**0,858888994**	**0,766003823**	**0,023319523**	**0,031232302**

Table 6.24: DicePlusXEnt baseline with 0.00001 learning rate at 30000 iterations

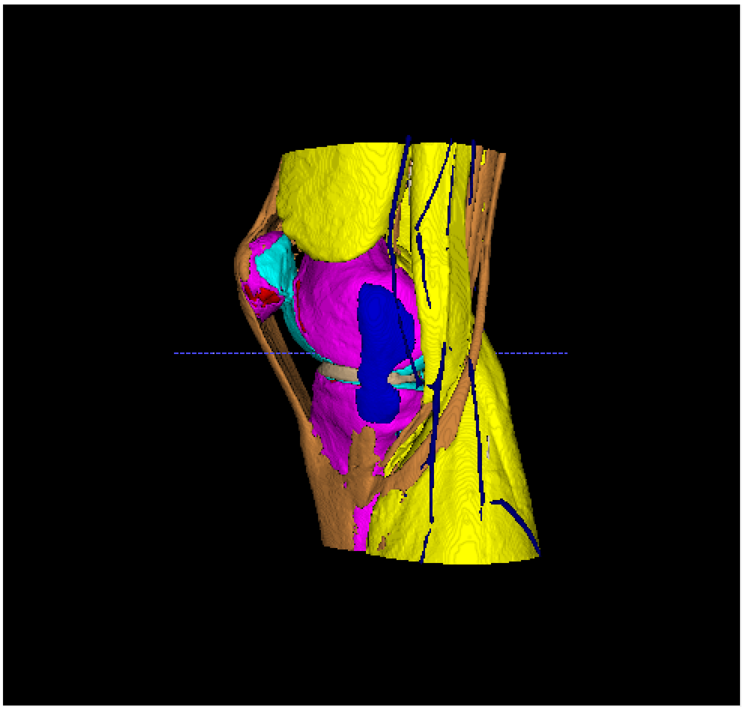
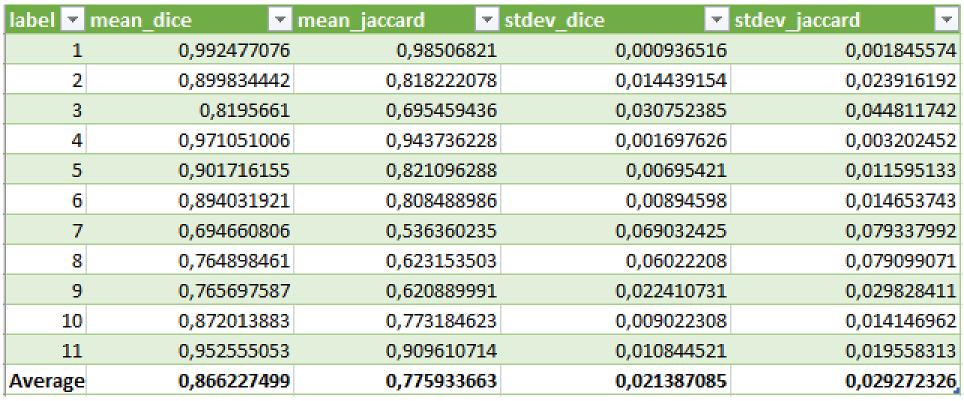


Figure 6.34: DicePlusXEnt baseline with 0.00001 learning rate at 40000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,992477076	0,98506821	0,000936516	0,001845574
2	0,899834442	0,818222078	0,014439154	0,023916192
3	0,8195661	0,695459436	0,030752385	0,044811742
4	0,971051006	0,943736228	0,001697626	0,003202452
5	0,901716155	0,821096288	0,00695421	0,011595133
6	0,894031921	0,808488986	0,00894598	0,014653743
7	0,694660806	0,536360235	0,069032425	0,079337992
8	0,764898461	0,623153503	0,06022208	0,079099071
9	0,765697587	0,620889991	0,022410731	0,029828411
10	0,872013883	0,773184623	0,009022308	0,014146962
11	0,952555053	0,909610714	0,010844521	0,019558313
Average	**0,866227499**	**0,775933663**	**0,021387085**	**0,029272326**

Table 6.25: DicePlusXEnt baseline with 0.00001 learning rate at 40000 iterations

6.5.2 Learning rate 0.001

Figures 6.35 - 6.37 and tables 6.26 - 6.28 shows the segmented output and evaluation score for the model with DicePlusXEnt loss type and 0.001 learning rate, every 15000 iterations.

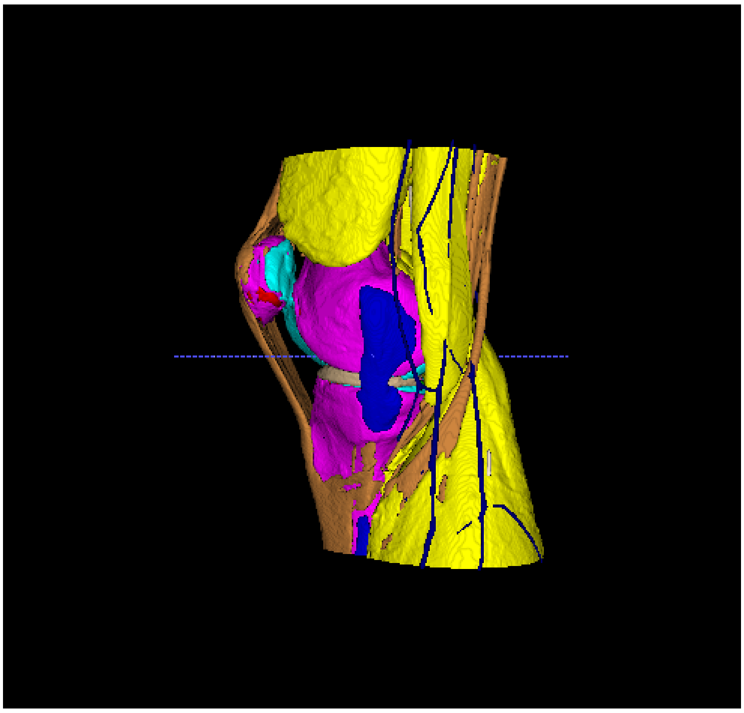
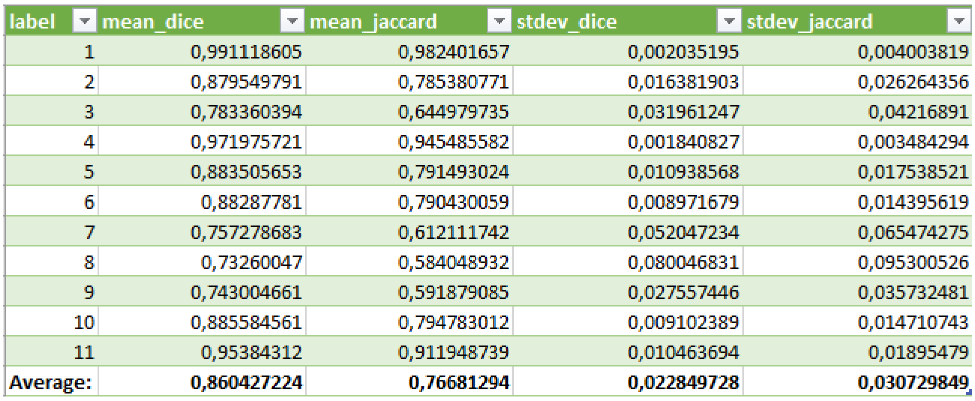


Figure 6.35: DicePlusXEnt baseline with 0.001 learning rate at 15000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,991118605	0,982401657	0,002035195	0,004003819
2	0,879549791	0,785380771	0,016381903	0,026264356
3	0,783360394	0,644979735	0,031961247	0,04216891
4	0,971975721	0,945485582	0,001840827	0,003484294
5	0,883505653	0,791493024	0,010938568	0,017538521
6	0,88287781	0,790430059	0,008971679	0,014395619
7	0,757278683	0,612111742	0,052047234	0,065474275
8	0,73260047	0,584048932	0,080046831	0,095300526
9	0,743004661	0,591879085	0,027557446	0,035732481
10	0,885584561	0,794783012	0,009102389	0,014710743
11	0,95384312	0,911948739	0,010463694	0,01895479
Average:	**0,860427224**	**0,76681294**	**0,022849728**	**0,030729849**

Table 6.26: DicePlusXEnt baseline with 0.001 learning rate at 15000 iterations

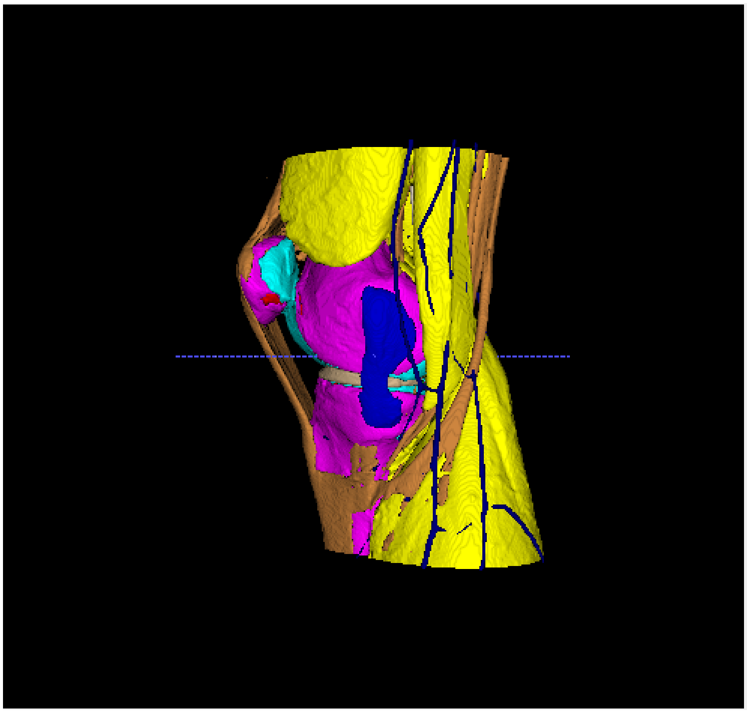
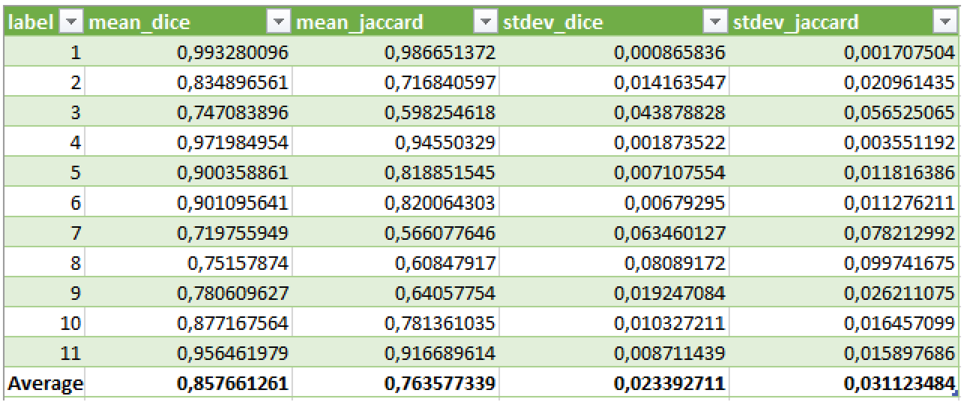


Figure 6.36: DicePlusXEnt baseline with 0.001 learning rate at 30000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,993280096	0,986651372	0,000865836	0,001707504
2	0,834896561	0,716840597	0,014163547	0,020961435
3	0,747083896	0,598254618	0,043878828	0,056525065
4	0,971984954	0,94550329	0,001873522	0,003551192
5	0,900358861	0,818851545	0,007107554	0,011816386
6	0,901095641	0,820064303	0,00679295	0,011276211
7	0,719755949	0,566077646	0,063460127	0,078212992
8	0,75157874	0,60847917	0,08089172	0,099741675
9	0,780609627	0,64057754	0,019247084	0,026211075
10	0,877167564	0,781361035	0,010327211	0,016457099
11	0,956461979	0,916689614	0,008711439	0,015897686
Average	**0,857661261**	**0,763577339**	**0,023392711**	**0,031123484**

Table 6.27: DicePlusXEnt baseline with 0.001 learning rate at 30000 iterations

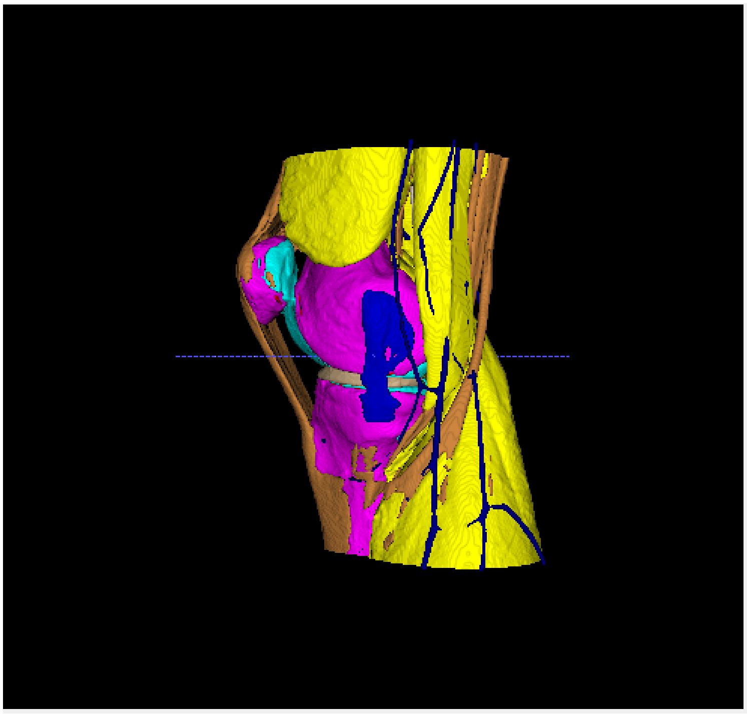
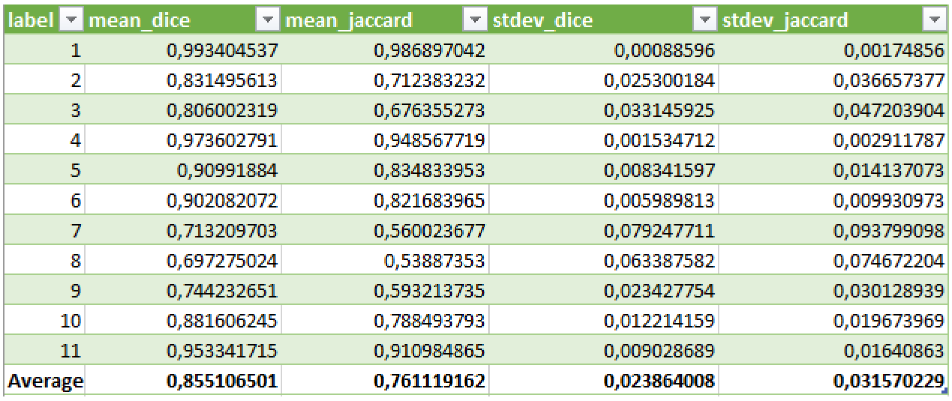


Figure 6.37: DicePlusXEnt baseline with 0.001 learning rate at 45000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,993404537	0,986897042	0,00088596	0,00174856
2	0,831495613	0,712383232	0,025300184	0,036657377
3	0,806002319	0,676355273	0,033145925	0,047203904
4	0,973602791	0,948567719	0,001534712	0,002911787
5	0,90991884	0,834833953	0,008341597	0,014137073
6	0,902082072	0,821683965	0,005989813	0,009930973
7	0,713209703	0,560023677	0,079247711	0,093799098
8	0,697275024	0,53887353	0,063387582	0,074672204
9	0,744232651	0,593213735	0,023427754	0,030128939
10	0,881606245	0,788493793	0,012214159	0,019673969
11	0,953341715	0,910984865	0,009028689	0,01640863
Average	**0,855106501**	**0,761119162**	**0,023864008**	**0,031570229**

Table 6.28: DicePlusXEnt baseline with 0.001 learning rate at 45000 iterations

6.5.3 Learning rate 0.01

Figures 6.38 - 6.43 and tables 6.29 - 6.34 shows the segmented output and evaluation score for the model with DicePlusXEnt loss type and 0.01 learning rate, every 10000 iterations.

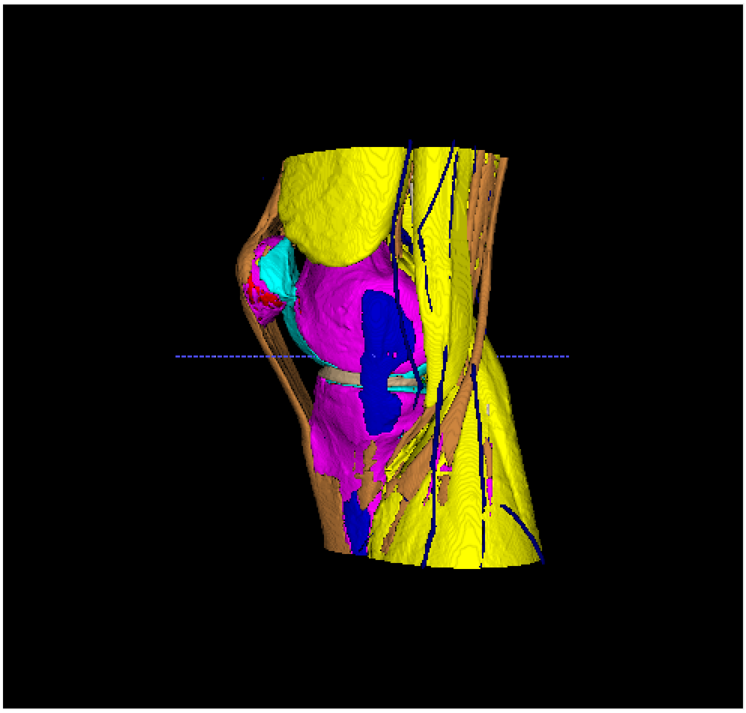
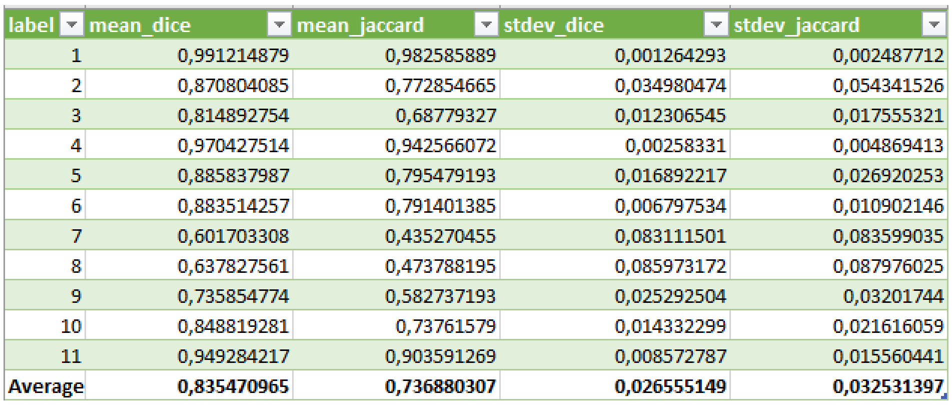


Figure 6.38: DicePlusXEnt baseline with 0.01 learning rate at 10000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,991214879	0,982585889	0,001264293	0,002487712
2	0,870804085	0,772854665	0,034980474	0,054341526
3	0,814892754	0,68779327	0,012306545	0,017555321
4	0,970427514	0,942566072	0,00258331	0,004869413
5	0,885837987	0,795479193	0,016892217	0,026920253
6	0,883514257	0,791401385	0,006797534	0,010902146
7	0,601703308	0,435270455	0,083111501	0,083599035
8	0,637827561	0,473788195	0,085973172	0,087976025
9	0,735854774	0,582737193	0,025292504	0,03201744
10	0,848819281	0,73761579	0,014332299	0,021616059
11	0,949284217	0,903591269	0,008572787	0,015560441
Average	**0,835470965**	**0,736880307**	**0,026555149**	**0,032531397**

Table 6.29: DicePlusXEnt baseline with 0.001 learning rate at 10000 iterations

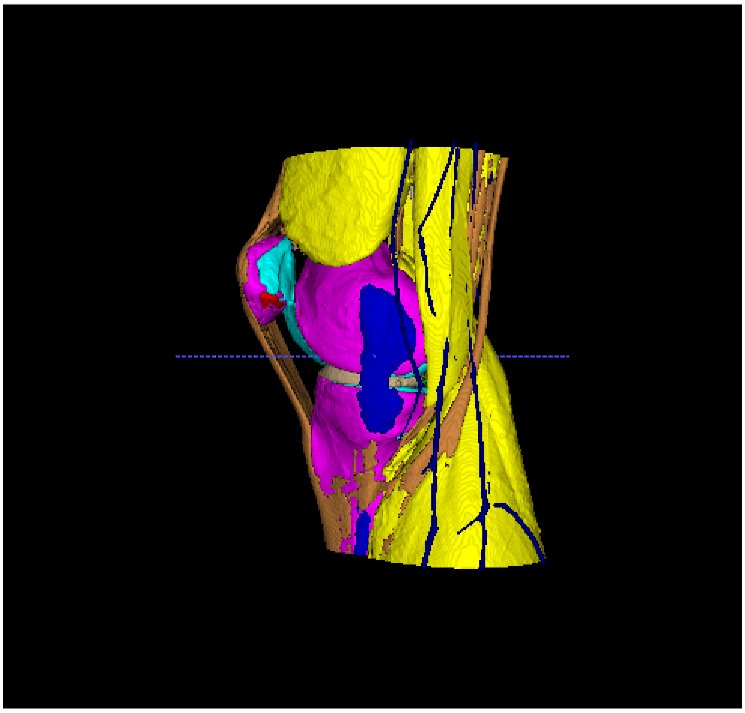
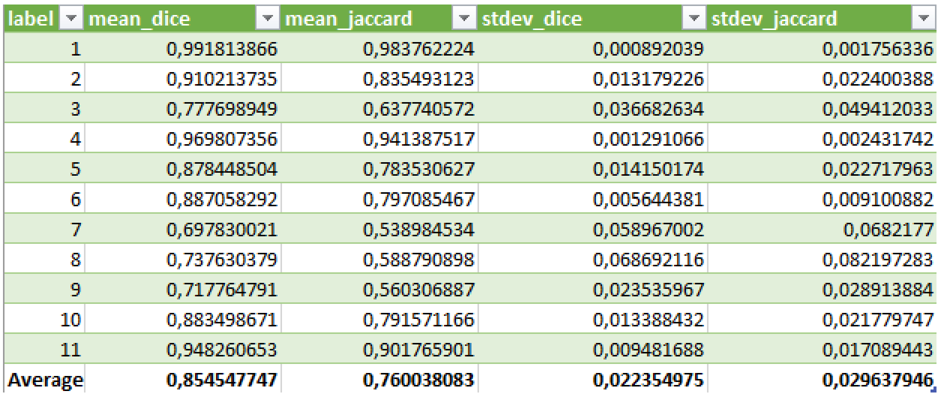


Figure 6.39: DicePlusXEnt baseline with 0.01 learning rate at 20000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,991813866	0,983762224	0,00892039	0,001756336
2	0,910213735	0,835493123	0,013179226	0,022400388
3	0,777698949	0,637740572	0,036682634	0,049412033
4	0,969807356	0,941387517	0,001291066	0,002431742
5	0,878448504	0,783530627	0,014150174	0,022717963
6	0,887058292	0,797085467	0,005644381	0,009100882
7	0,697830021	0,538984534	0,058967002	0,0682177
8	0,737630379	0,588790898	0,068692116	0,082197283
9	0,717764791	0,560306887	0,023535967	0,028913884
10	0,883498671	0,791571166	0,013388432	0,021779747
11	0,948260653	0,901765901	0,009481688	0,017089443
Average	**0,854547747**	**0,760038083**	**0,022354975**	**0,029637946**

Table 6.30: DicePlusXEnt baseline with 0.001 learning rate at 20000 iterations

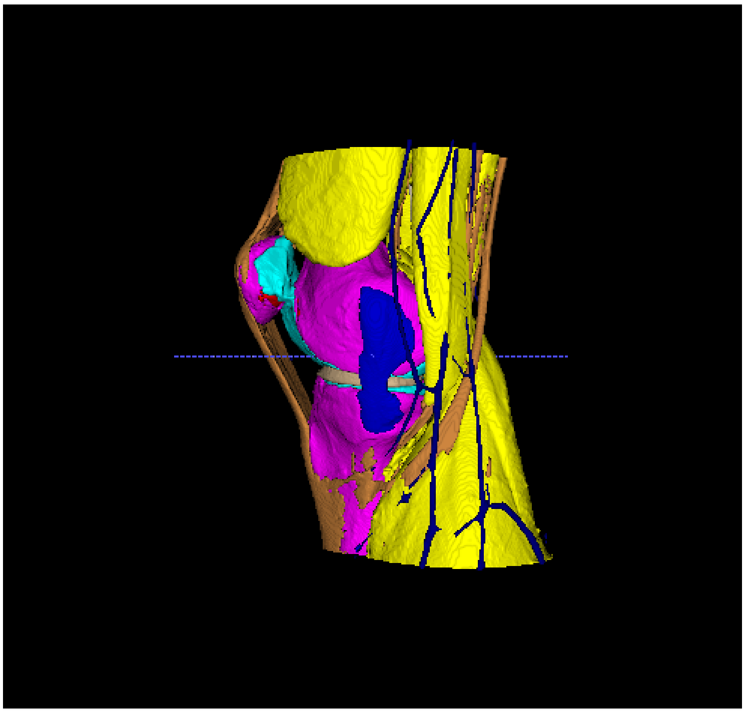
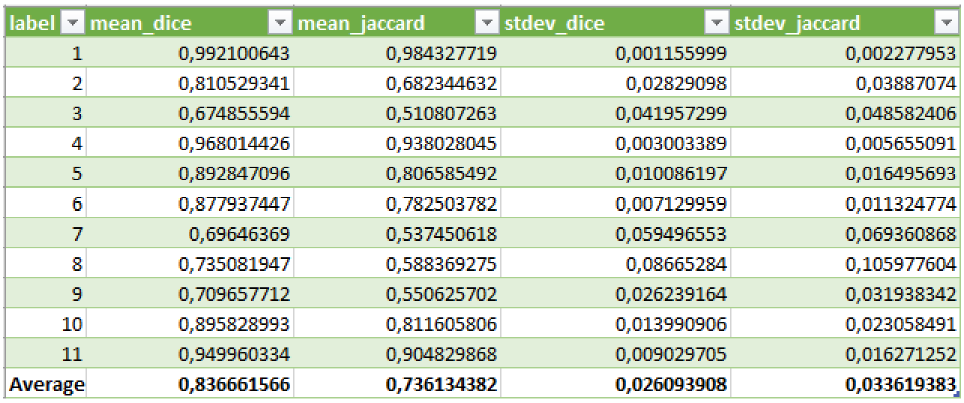


Figure 6.40: DicePlusXEnt baseline with 0.01 learning rate at 30000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,992100643	0,984327719	0,001155999	0,002277953
2	0,810529341	0,682344632	0,02829098	0,03887074
3	0,674855594	0,510807263	0,041957299	0,048582406
4	0,968014426	0,938028045	0,003003389	0,005655091
5	0,892847096	0,806585492	0,010086197	0,016495693
6	0,877937447	0,782503782	0,007129959	0,011324774
7	0,69646369	0,537450618	0,059496553	0,069360868
8	0,735081947	0,588369275	0,08665284	0,105977604
9	0,709657712	0,550625702	0,026239164	0,031938342
10	0,895828993	0,811605806	0,013990906	0,023058491
11	0,949960334	0,904829868	0,009029705	0,016271252
Average	**0,836661566**	**0,736134382**	**0,026093908**	**0,033619383**

Table 6.31: DicePlusXEnt baseline with 0.001 learning rate at 30000 iterations

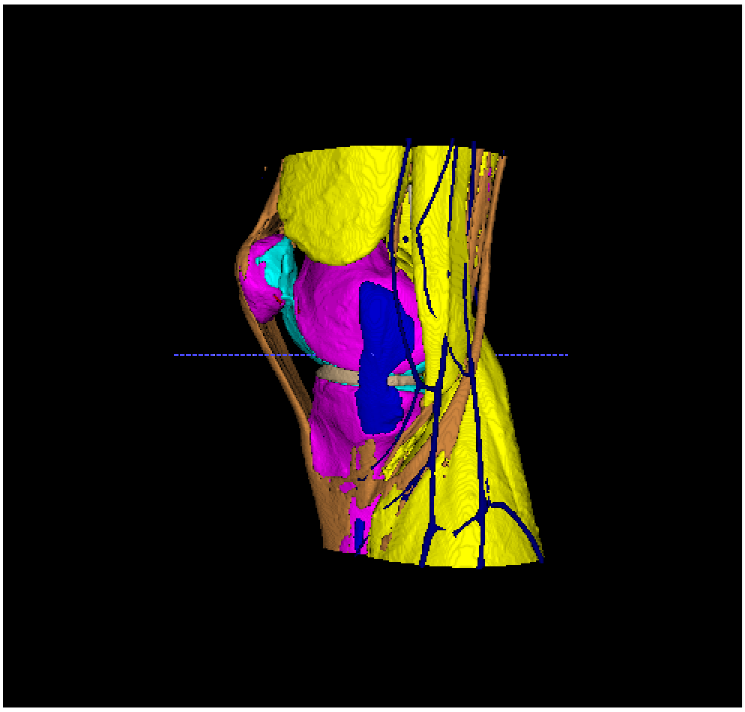
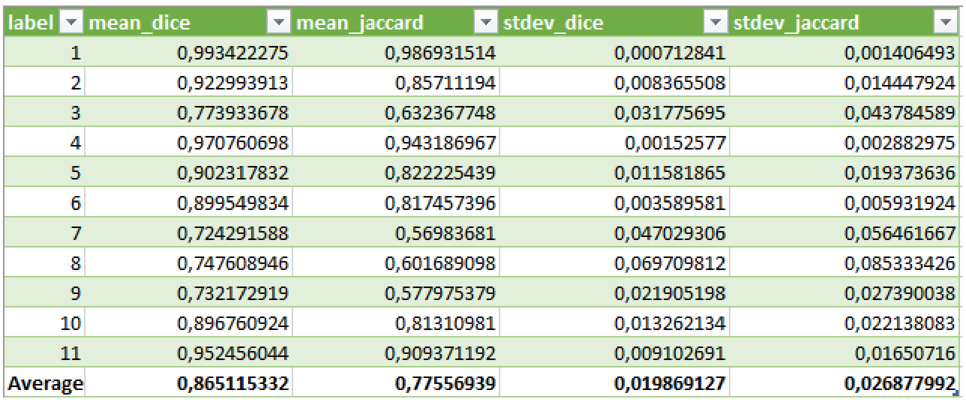


Figure 6.41: DicePlusXEnt baseline with 0.01 learning rate at 40000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,993422275	0,986931514	0,000712841	0,001406493
2	0,922993913	0,85711194	0,008365508	0,014447924
3	0,773933678	0,632367748	0,031775695	0,043784589
4	0,970760698	0,943186967	0,00152577	0,002882975
5	0,902317832	0,822225439	0,011581865	0,019373636
6	0,899549834	0,817457396	0,003589581	0,005931924
7	0,724291588	0,56983681	0,047029306	0,056461667
8	0,747608946	0,601689098	0,069709812	0,085333426
9	0,732172919	0,577975379	0,021905198	0,027390038
10	0,896760924	0,81310981	0,013262134	0,022138083
11	0,952456044	0,909371192	0,009102691	0,01650716
Average	**0,865115332**	**0,77556939**	**0,019869127**	**0,026877992**

Table 6.32: DicePlusXEnt baseline with 0.001 learning rate at 40000 iterations

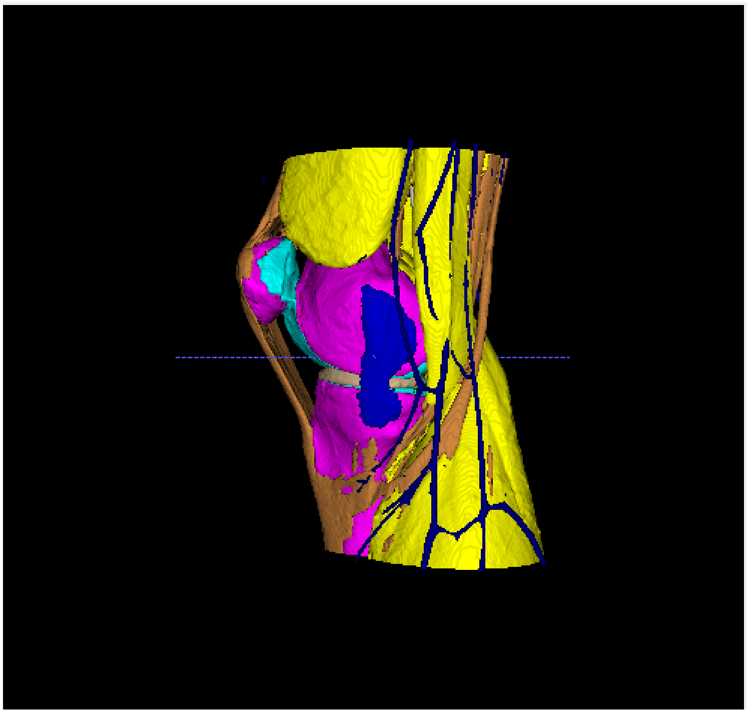
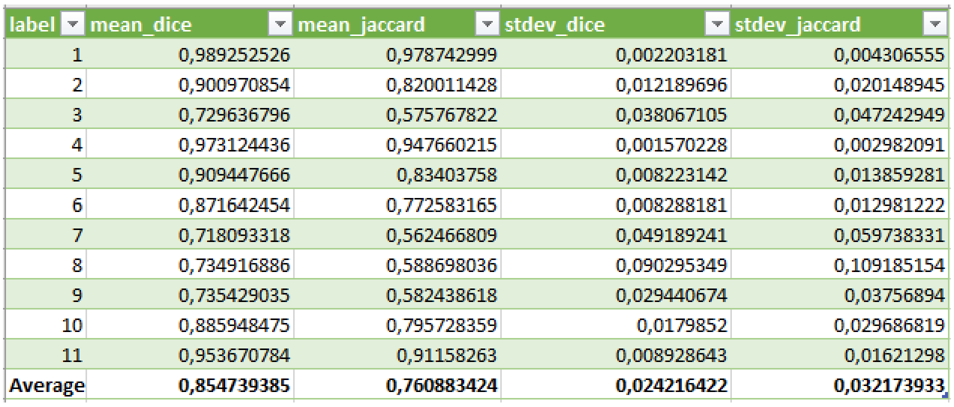


Figure 6.42: DicePlusXEnt baseline with 0.01 learning rate at 50000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,989252526	0,978742999	0,002203181	0,004306555
2	0,900970854	0,820011428	0,012189696	0,020148945
3	0,729636796	0,575767822	0,038067105	0,047242949
4	0,973124436	0,947660215	0,001570228	0,002982091
5	0,909447666	0,83403758	0,008223142	0,013859281
6	0,871642454	0,772583165	0,008288181	0,012981222
7	0,718093318	0,562466809	0,049189241	0,059738331
8	0,734916886	0,588698036	0,090295349	0,109185154
9	0,735429035	0,582438618	0,029440674	0,03756894
10	0,885948475	0,795728359	0,0179852	0,029686819
11	0,953670784	0,91158263	0,008928643	0,01621298
Average	**0,854739385**	**0,760883424**	**0,024216422**	**0,032173933**

Table 6.33: DicePlusXEnt baseline with 0.001 learning rate at 50000 iterations

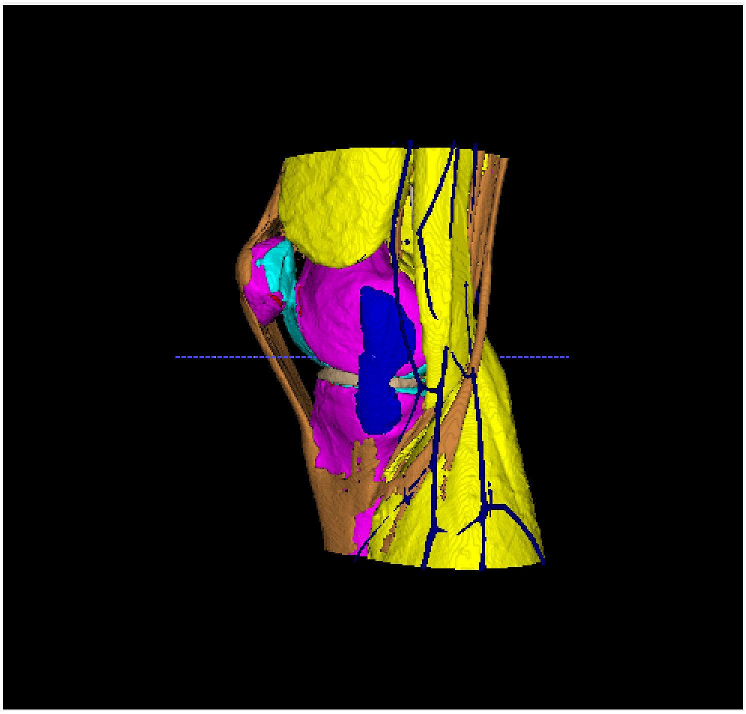
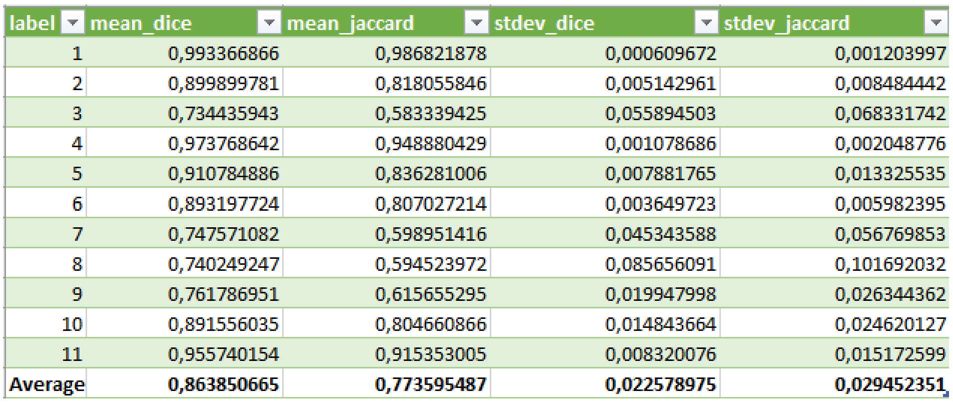


Figure 6.43: DicePlusXEnt baseline with 0.01 learning rate at 60000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,993366866	0,986821878	0,000609672	0,001203997
2	0,899899781	0,818055846	0,005142961	0,008484442
3	0,734435943	0,583339425	0,055894503	0,068331742
4	0,973768642	0,948880429	0,001078686	0,002048776
5	0,910784886	0,836281006	0,007881765	0,013325535
6	0,893197724	0,807027214	0,003649723	0,005982395
7	0,747571082	0,598951416	0,045343588	0,056769853
8	0,740249247	0,594523972	0,085656091	0,101692032
9	0,761786951	0,615655295	0,019947998	0,026344362
10	0,891556035	0,804660866	0,014843664	0,024620127
11	0,955740154	0,915353005	0,008320076	0,015172599
Average	**0,863850665**	**0,773595487**	**0,022578975**	**0,029452351**

Table 6.34: DicePlusXEnt baseline with 0.001 learning rate at 60000 iterations

6.6 Data Augmentation

Figures 6.44 - 6.47 and tables 6.35 - 6.38 shows the segmented output and evaluation score for the baseline DicePlusXEnt model with data augmentation as specified in section 5.6, every 5000 iterations.

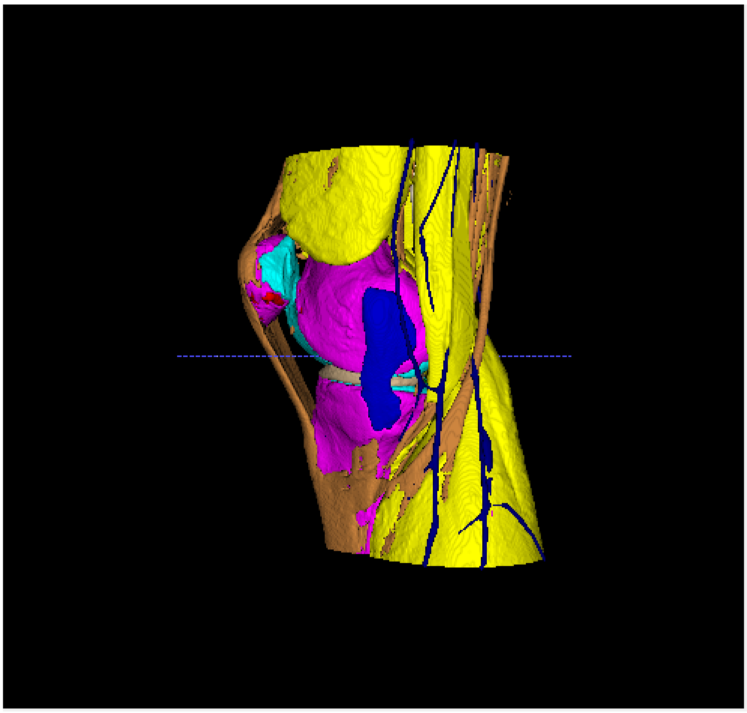
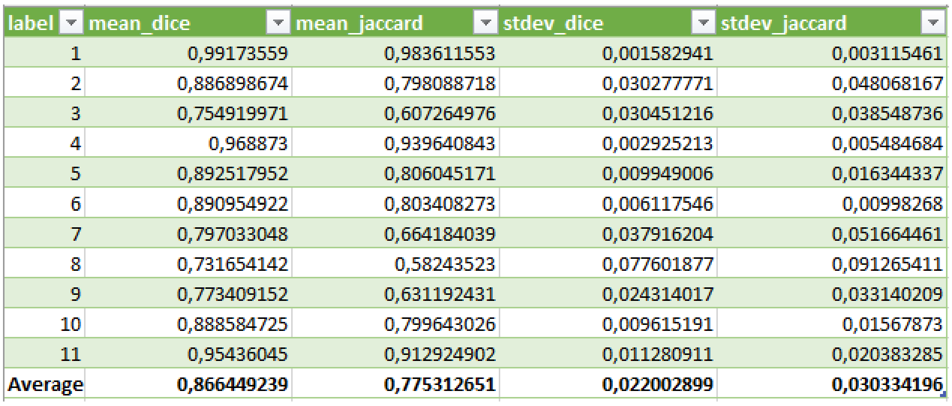


Figure 6.44: DicePlusXEnt baseline with augmentation at 5000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,99173559	0,983611553	0,001582941	0,003115461
2	0,886898674	0,798088718	0,030277771	0,048068167
3	0,754919971	0,607264976	0,030451216	0,038548736
4	0,968873	0,939640843	0,002925213	0,005484684
5	0,892517952	0,806045171	0,009949006	0,016344337
6	0,890954922	0,803408273	0,006117546	0,00998268
7	0,797033048	0,664184039	0,037916204	0,051664461
8	0,731654142	0,58243523	0,077601877	0,091265411
9	0,773409152	0,631192431	0,024314017	0,033140209
10	0,888584725	0,799643026	0,009615191	0,01567873
11	0,95436045	0,912924902	0,011280911	0,020383285
Average	**0,866449239**	**0,775312651**	**0,022002899**	**0,030334196**

Table 6.35: DicePlusXEnt baseline with augmentation at 5000 iterations

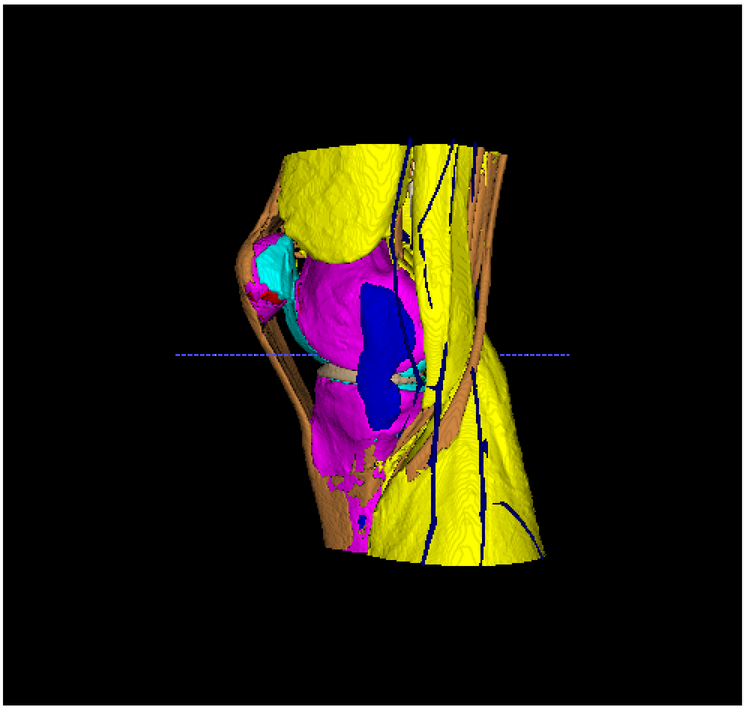
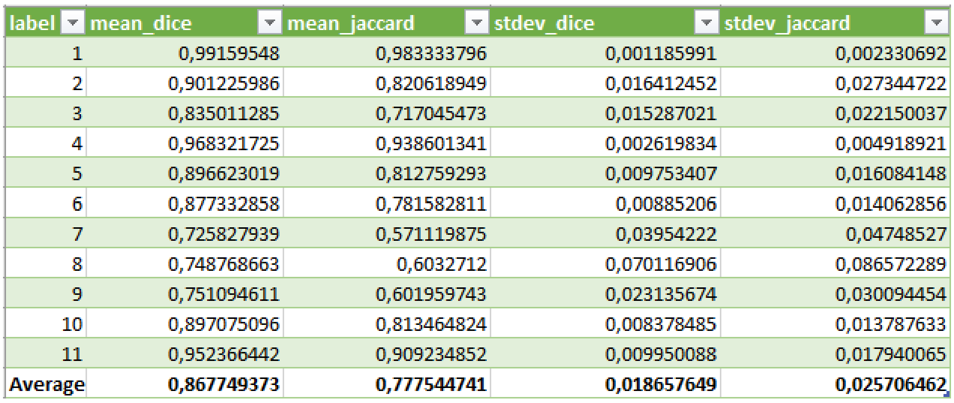


Figure 6.45: DicePlusXEnt baseline with augmentation at 10000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,99159548	0,983333796	0,001185991	0,002330692
2	0,901225986	0,820618949	0,016412452	0,027344722
3	0,835011285	0,717045473	0,015287021	0,022150037
4	0,968321725	0,938601341	0,002619834	0,004918921
5	0,896623019	0,812759293	0,009753407	0,016084148
6	0,877332858	0,781582811	0,00885206	0,014062856
7	0,725827939	0,571119875	0,03954222	0,04748527
8	0,748768663	0,6032712	0,070116906	0,086572289
9	0,751094611	0,601959743	0,023135674	0,030094454
10	0,897075096	0,813464824	0,008378485	0,013787633
11	0,952366442	0,909234852	0,009950088	0,017940065
Average	**0,867749373**	**0,777544741**	**0,018657649**	**0,025706462**

Table 6.36: DicePlusXEnt baseline with augmentation at 10000 iterations

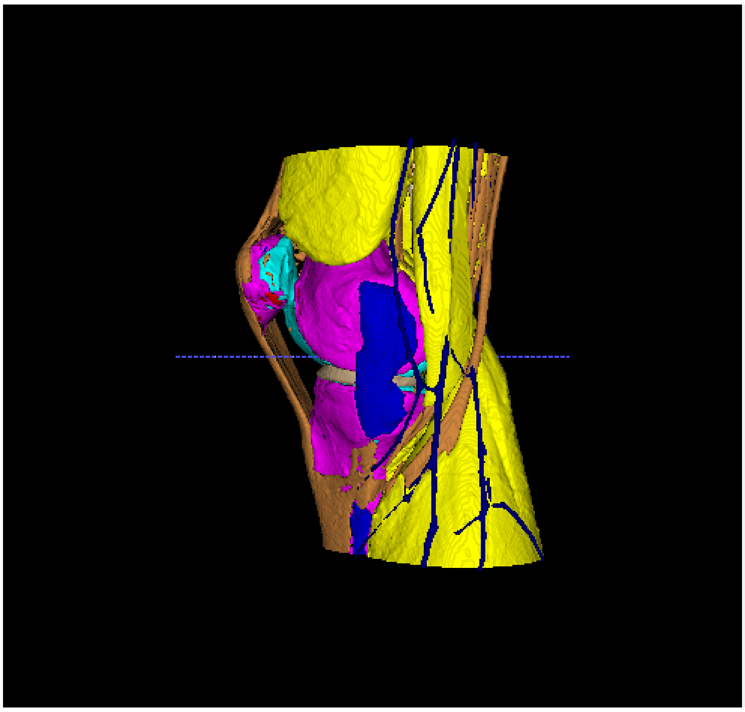


Figure 6.46: DicePlusXEnt baseline with augmentation at 15000 iterations

label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,99123387	0,982623251	0,001272506	0,002502406
2	0,897995351	0,815264097	0,01615564	0,026584999
3	0,757464665	0,610569795	0,030388479	0,039166758
4	0,971138584	0,943902326	0,001796101	0,003396035
5	0,892887474	0,806557332	0,006168839	0,010091411
6	0,888752513	0,799823198	0,005504956	0,008902033
7	0,726015841	0,575729433	0,079157584	0,094304893
8	0,72468144	0,573024329	0,072050123	0,084859384
9	0,746665198	0,596611713	0,028923082	0,037657111
10	0,892700723	0,806460631	0,013276771	0,022034977
11	0,953609327	0,911459416	0,008566052	0,015562523
Average	**0,858467726**	**0,765638684**	**0,023932739**	**0,031369321**

Table 6.37: DicePlusXEnt baseline with augmentation at 15000 iterations

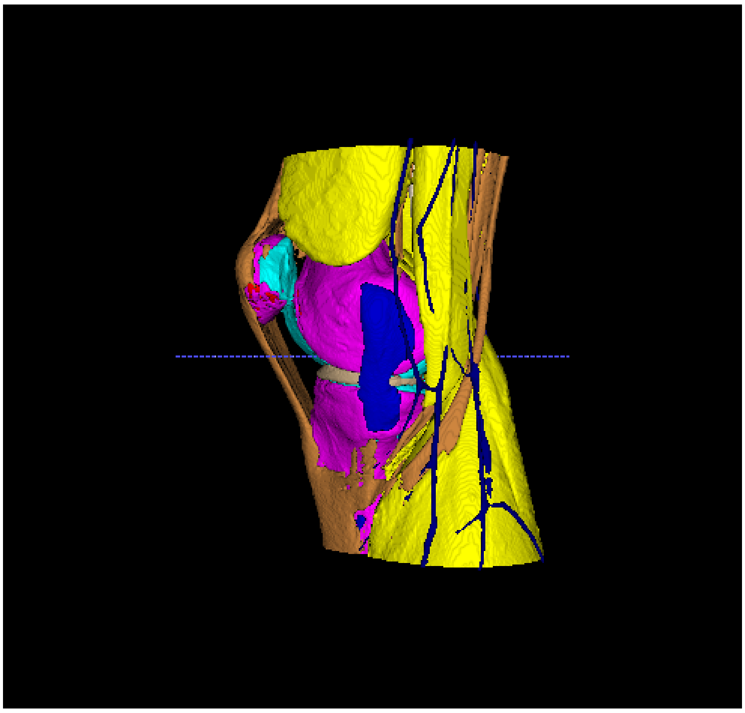
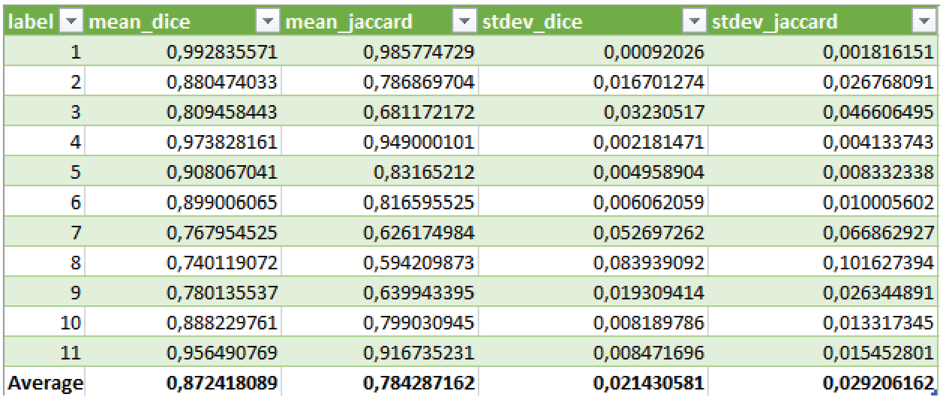


Figure 6.47: DicePlusXEnt baseline with augmentation at 20000 iterations



label	mean_dice	mean_jaccard	stdev_dice	stdev_jaccard
1	0,992835571	0,985774729	0,00092026	0,001816151
2	0,880474033	0,786869704	0,016701274	0,026768091
3	0,809458443	0,681172172	0,03230517	0,046606495
4	0,973828161	0,949000101	0,002181471	0,004133743
5	0,908067041	0,83165212	0,004958904	0,008332338
6	0,899006065	0,816595525	0,006062059	0,010005602
7	0,767954525	0,626174984	0,052697262	0,066862927
8	0,740119072	0,594209873	0,083939092	0,101627394
9	0,780135537	0,639943395	0,019309414	0,026344891
10	0,888229761	0,799030945	0,008189786	0,013317345
11	0,956490769	0,916735231	0,008471696	0,015452801
Average	**0,872418089**	**0,784287162**	**0,021430581**	**0,029206162**

Table 6.38: DicePlusXEnt baseline with augmentation at 20000 iterations

Chapter 7

Discussion

This chapter contains a discussion and evaluation of the results and methodology presented in the previous chapters.

7.1 Resolution (pixel dimension)

As mentioned in section 5.1, the goal with resolution experimentation was to combat the ”false positive volume” detailed in section 6.1. As shown in the results presented in section 6.1, the issue was initially present with the original pixel dimension of 0.4mm. When resampling the image with a pixel dimension of 0.5mm, this issue improved but remained visible. Similarly, when increasing the pixel dimension to 0.6mm, the issue was reduced even further, while remaining slightly visible. Once the pixel dimension was increased to 0.7mm, the problem was completely resolved. The issue was also not visible for the 1.0mm pixel dimension model.

These results confirm the hypothesis presented in section 5.1, by proving that the ”false positive volume” issue can be resolved by resampling the image to a lower resolution. The conclusion of these results, however, are not completely clear. These results only show that downsampling the resolution of the input image resolved this issue. It is not clear whether this is a result of reducing the size difference between the spatial window and the input image, or if it is simply due to the reduced input image size alone. Having that said, however, this experimentation did serve its purpose by proving that the large size of the input data was in some way correlated to this ”false positive volume” issue, and thereby facilitated the spatial window size experimentation discussed in the following section.

7.2 Spatial window size

As mentioned in section 4.5, the baseline value for this parameter was set to (128,128,128) when training the models. Attempts to increase this resulted in OOM errors, effectively establishing this value as the maximum size with the available hardware. Attempts to lower this size were not carried out, both due to time constraints as well as a lack of belief in good results. And as mentioned in section 5.2.2, the original assumption was that this would also be the maximum spatial window size for inference. But due to the results found in section 7.1 above, further experimentation with increasing this parameter for inference was carried out.

The initial test for this hypothesis, as shown in figure 6.8, was to run inference on the CPU with a spatial window size of (256,256,256). As seen in the result, this completely resolved the ”false positive volume” issue. This attempt was however extremely slow, taking hours as opposed to minutes, making it unfeasible to perform for all of the remaining trained models. This led to attempts to increase the spatial window size while running inference on the GPU.

As shown in section 6.2, the ”false positive volume” is still present with a spatial window size of (192,192,192). It does however only appear as a thin slice, much less prominent than when the size was set to (128,128,128). The next attempt increased the spatial window size to the next supported cubed value of (208,208,208), and, as shown in figure 6.10, this almost manages to get rid of the issue. It does appear as if the ”false positive volume” is, in fact, the exact same as the previous figure, and that increasing the spatial window size only results in less of the volume showing up. And then finally as seen in figure 6.11, with a spatial window size of (224,224,224), the ”false positive volume” issue is completely gone. Attempts to increase this parameter even further resulted in OOM errors when running on the GPU. Thus, the highest possible spatial window size for inference with the available hardware coincidentally ended up being the lowest required spatial window size to avoid the ”false positive volume” issue.

This ”false positive volume” issue appeared in the inference output for every single model with lower spatial window size, regardless of any variations in other parameters. And after increasing the spatial window size for inference to (224,224,224), the issue disappeared completely from all the models. Therefore, this issue was found not to be related to the training of the models but only correlated to the inference part.

Based on results from resolution and spatial window size experiments, it is safe to conclude that the cause of the ”false positive volume” issue was the size difference between the spatial window and input image. This is based on the fact that both decreasing the input image size, and increasing the spatial window size, resolved the issue. It is also worth noting that the issue is also only present for the inference, and the training does not seem to have a significant impact on the issue.

This ”false positive volume” issue appears to be unique, as no other instances have been found online. One possible reason could be the high resolution of the dataset images, being $400 \times 400 \times 400$. It seems likely that the problem appears due to a high difference between the original size of the dataset and the spatial window size used in inference. This would mean that the requirement for the spatial window size to avoid this problem is correlated to the size of input images. This is however just a hypothesis and has not been investigated in this thesis.

7.3 Loss type

The first loss type considered is the standard Dice Coefficient. As seen in figures 6.12 and 6.13, the ”false positive volume” still appears initially, but is completely gone in figure 6.14 at 30000 iterations. Furthermore, by comparing the evaluation scores in table 6.5 at 30000 iterations, with tables 6.6 and 6.7 at 40000 and 50000 iterations respectively, the accuracy of the model decreases with a higher number of iterations. The best model for the Dice baseline is therefore reached after only 30000 iterations.

On the other hand, the DicePlusXEnt model does not have the ”false positive volume” issue present after 10000 iterations. It also starts of with a similar trend to the standard Dice, in which table 6.8 at 10000 iterations performs better than both 20000 iterations in table 6.9 and 30000 iterations in table 6.9. This trend does, however, turn around, and the evaluation scores for both 40000 and 50000 iterations, in tables 6.11 and 6.12, are better than the score achieved at 10000 iterations. The best model for the DicePlusXEnt baseline is therefore reached after 50000 iterations and performs slightly worse than the standard Dice loss type.

These results show that with a baseline configuration of the network, as presented in section 4.5, the dice loss results in a slightly higher evaluation score, represented as both the Dice Coefficient and the Jaccard Index. This is in contrast to the results from the originally published nnU-Net paper [49], in which the DicePlusXEnt loss type was superior. The difference between the two loss types were however minimal. It is also possible that this difference is only due to the specific choice of baseline parameters, and the loss type will therefore also need to be considered for the remaining sections of this chapter.

7.4 Normalisation

As seen in figure 6.22, the ”false positive volume” was also initially for the Dice loss type when including normalisation. But this did once again resolve with an increase in iterations. As seen in tables 6.23, 6.24, and 6.25, the evaluation score for normalisation remains stable around 0.85-0.86.

In contrast, the DicePlusXEnt loss type model with normalisation shown had a slow and steady increase in evaluation score from 5000 to 15000 iterations, as shown in tables 6.17, 6.18, and 6.19. This model was performed slightly worse than the baseline Dice model. Additionally, an attempt at increasing the learning rate for this model from 0.0001 to 0.001, presented in tables 6.29 and 6.30, did not achieve a comparable score.

These findings are similar to those presented in the original nnU-Net paper [49], showing that the DicePlusXEnt loss type performs better than the standard Dice when normalisation is enabled. However, the evaluation score for this model did not surpass the baseline Dice model discussed in the previous section. While this could happen if given enough iterations of the normalisation model, this was not attempted in this thesis due to the increased iteration time experienced when adding normalisation.

While normalisation supposedly speeds up learning, this was not found to be the case in this thesis. The time for each iteration increased by over 300%, and models did not perform better when trained for the same amount of time with normalisation as the models without. Some possible reasons would be the specific normalisation configuration, such as foreground normalisation. It is also possible that since normalisation increases the sampling time for each iteration, it would benefit from increasing the number of samples per volume. These are however only speculations and has not been tested in this thesis.

7.5 Learning rate

When looking at the learning rate, the decreased learning rate of 0.00001 presented in figure 6.31 to 6.34 shows promising results. As expected with a low learning rate, the model takes a while to reach a good solution, having large jumps in accuracy every 10000 iterations. Surprisingly, the model performance did not gradually increase, but rather decreased for 30000 iterations before improving again at 40000 iterations. Both models at 20000 and 40000 iterations had similar evaluation scores, but neither succeeded the baseline learning rate of 0.0001 that was discussed earlier in section 7.3.

The increased learning rate of 0.001, did as expected reach a moderately good score of 0.86 after only 15000 iterations as shown in table 6.35. It did however not manage to improve on this, and the evaluation score only went downhill after this as presented in tables 6.36 and 6.37.

When increasing the learning rate even further to 0.01, the model also approached a good score quite quickly. One thing to note, however, is that the evaluation score keeps going up and down in the range of 0.836-0.865. This is expected with such a high learning rate, as the model keeps overstepping the optimum. The best model which was reached after 40000 iterations, shown in table 6.41, is almost identical to the one achieved by the baseline learning rate.

Based on these results, it appears that the learning rate does not have a significant impact on the best-achieved score for a model, but rather on the number of iterations to achieve it. A high learning rate also results in a somewhat unstable final model, in which the accuracy of the final model is to a certain degree randomly decided by when the model is stopped. This is a common problem for ML tasks, and is combated by early stopping. This is however not a function that is included in Niftynet, and could not be utilised for this thesis. These results do however emphasize the importance of early stopping when it comes to ML tasks.

7.6 Data Augmentation

While the DicePlusXEnt model with data augmentation did not quite outperform the baseline Dice model, it was extremely close to it. Moreover, it managed to outperform the baseline DicePlusXEnt model slightly. Similar to normalisation, however, it did have the drawback increasing the iteration time by quite a lot. So while the augmentation did improve results slightly, it did require extensive computation time to do so, and the resulting improvement was very slight.

One potential reason for the low level of improvement could be the high quality of the data generation. The data is generated in precisely the same manner in an effort to reduce the variance of the data. This would reduce the importance of data augmentation, due to the low level of variance between the training and test datasets. Another aspect is that the data generation only focuses on healthy knees, which would also contribute to overall reduced variance. Another potential reason for these results could be due to the chosen augmentation options. While the options were carefully considered, as detailed in section 5.6, it is still possible that these options resulted in unrealistic augmentations.

Chapter 8 Conclusion

This thesis aimed to utilise CNNs for semantic segmentation of MRI images of the knee joint. More specifically, the experimentation in this thesis was performed using the nnU-Net module within Niftynet. Based on the experiments and corresponding results presented in this thesis, it can be concluded that this approach has a large potential to provide accurate segmentation masks.

Several experiments with various hyper-parameters have been carried out, and their resulting segmentation masks compared. This has provided insight and a better understanding of the role that hyper-parameters have in the training process, and their impact on the resulting segmentation accuracy for medical image segmentation tasks.

Next, the two research questions presented in the beginning of the thesis are reiterated and answered below.

Research question 1: *Do the trained neural networks generate a segmentation output of sufficient accuracy?*

The segmentation masks generated by the trained neural networks were found to be sufficiently accurate. The accuracy varies between the different segmented classes, although this is to be expected. Overall, the results achieved a sufficiently high accuracy, both in terms of evaluation metrics and visual inspection, and thereby establish the potential of CNNs for the automatic semantic segmentation of knee joint.

Research question 2: *What impact do the hyper-parameters have on the training process and inferred segmentation output?*

The experiments carried out in this thesis consisted of a range of varying hyper-parameters. The impact from these hyper-parameters were discussed in detail in the previous chapter. The baseline model was found to provide the best results. One addition that improved results was the data augmentation. While this did not in fact end up being the best model, it did perform better than a completely equal model without augmentation. This is likely because of the low size of the training dataset, such that artificially increasing this size, by adding variance to the data, results in an improved generalisation for the model.

8.1 Contribution

The contributions of this thesis comes from establishing the potential for CNNs to automatically segment 13 classes from high-resolution 3D MRI images. More specifically, this thesis proved the efficacy of the nnU-Net architecture for this specific task. This thesis also presented comparable results from various trained models, and a discussion regarding the impact each hyper-parameter had on the model.

8.2 Future work

The work in this thesis has further established the potential for the application of CNNs for automatic segmentation of medical images, specifically the nnU-Net for knee joint MRI images. There is, however, room for improvements. The following list contains some possible ideas and aspects of this thesis that can be improved as future work:

- **Improving the choice of hyper-parameters:** The parameters experimented with in this thesis was not a full list of all available hyper-parameters. It is quite possible that any parameter that was not tested in this thesis, such as the window sampler and activation function, would have the potential of improving the model substantially. Furthermore, it would be interesting to see whether or not the model would improve even further with a higher spatial window size, as this was not possible to test with the available hardware.

- **Different CNN architectures:** This thesis was confined to the nnU-Net architecture provided by Niftynet. It would be interesting to see how different CNN architectures would compare to the results in this thesis.

- **Different anatomical structures:** The work in this thesis was limited to the knee joint. The plan for the collaboration project is however to expand the efforts to different anatomical structures, such as the shoulder joint. It would be interesting to see if the best choice of hyper-parameters would remain the same when segmenting different anatomical structures and whether the observed impact of the various hyper-parameters would differ.

- **Transfer learning:** As an addition to the previous point, it would be interesting to see whether or not transfer learning would be a good approach when transitioning to different anatomical structures. Due to most tissues having similar composition regardless of its location, it is very plausible that transfer learning would return good results, especially if the new anatomical structure is another joint.

Bibliography

[1] College O. ”917 Knee Joint”. Illustration from Anatomy Physiology, Connexions Web site. https://creativecommons.org/licenses/by/3.0/legalcode; 2013. Visited on 2020-05-09. Available from: https://commons.wikimedia.org/wiki/File:917_Knee_Joint.jpg.

[2] ca G. Colored neural network, https://creativecommons.org/licenses/by-sa/3.0/legalcode;. Visited on 2020-05-10. Available from: https://commons.wikimedia.org/wiki/File:Colored_neural_network.svg.

[3] Geek3. Hyperbolic Tangent, https://creativecommons.org/licenses/by-sa/3.0/legalcode;. Visited on 2020-05-11. Available from: https://commons.wikimedia.org/wiki/File:Hyperbolic_Tangent.svg.

[4] Renanar2. ReLU and Nonnegative Soft Thresholding Functions, https://creativecommons.org/licenses/by-sa/4.0/legalcode;. Visited on 2020-05-11. Available from: https://commons.wikimedia.org/wiki/File:ReLU_and_Nonnegative_Soft_Thresholding_Functions.svg.

[5] Maier A. ”ConvolutionAndPooling”, https://creativecommons.org/licenses/by/3.0/legalcode; 2019. Visited on 2020-05-26. Available from: https://commons.wikimedia.org/wiki/File:ConvolutionAndPooling.svg.

[6] Aphex34. ”Max pooling”, https://creativecommons.org/licenses/by-sa/4.0/legalcode; 2015. Visited on 2020-05-26. Available from: https://commons.wikimedia.org/wiki/File:Max_pooling.png.

[7] Aphex34. ”Conv layer”, https://creativecommons.org/licenses/by-sa/4.0/legalcode; 2015. Visited on 2020-05-28. Available from: https://commons.wikimedia.org/wiki/File:Conv_layer.png.

[8] Niftynet-documentation. Patch-based analysis;. Visited on 2020-05-14. Available from: https://niftynet.readthedocs.io/en/dev/window_sizes.html.

[9] Errachete. ”Binary confusion matrix”, https://creativecommons.org/licenses/by-sa/4.0/legalcode; 2019. Visited on 2020-05-28. Available from: https://commons.wikimedia.org/wiki/File:Binary_confusion_matrix.jpg.

[10] Rosebrock A. ”Intersection over Union - visual equation”, https://creativecommons.org/licenses/by-sa/4.0/legalcode; 2016. Visited on 2020-05-28. Available from: https://commons.wikimedia.org/wiki/File:Intersection_over_Union_-_visual_equation.png.

[11] Bertolaccini L, Solli P, Pardolesi A, Pasini A. An overview of the use of artificial neural networks in lung cancer research. Journal of Thoracic Disease. 2017 04;9.

[12] Per Christensson T. CPU Definition; 2014. Visited on 2020-05-14. Available from: https://techterms.com/definition/cpu.

[13] Mahmood A, Bennamoun M, An S, Sohel F, Boussaid F, Hovey R, et al. Chapter 21 - Deep Learning for Coral Classification. In: Samui P, Sekhar S, Balas VE, editors. Handbook of Neural Computation. Academic Press; 2017. p. 383 – 401. Available from: http://www.sciencedirect.com/science/article/pii/B9780128113189000211.

[14] Techopedia. Digital Signal Processing (DSP); 2013. Visited on 2020-05-26. Available from: https://www.techopedia.com/definition/2360/digital-signal-processing-dsp.

[15] Expert System Team. What is Machine Learning? A definition; 2017. Visited on 2020-03-18. Available from: https://expertsystem.com/machine-learning-definition/.

[16] Lewis T. What is an MRI (Magnetic Resonance Imaging)?;. Visited on 2020-05-09. Available from: https://www.livescience.com/39074-what-is-an-mri.html.

[17] Sperre J. A Review of Deep Learning Approaches for Medical Image Segmentation. Unpublished. 2019;.

[18] R George, J Dela Cruz, R Singh, Rajapandian Ilangovan. Proton density (PD) image characteristics;. Visited on 2020-05-10. Available from: https://mrimaster.com/characterise%20image%20pd.html.

[19] Nwana HS. Software agents: an overview. The Knowledge Engineering Review. 1996;11(3):205–244.

[20] Kitchener P. Importance of Medical Imaging;. Visited on 2020-05-18. Available from: https://www.xray.com.au/importance-of-medical-imaging/.

[21] Elson D, Yang GZ. In: Athanasiou T, Debas H, Darzi A, editors. The Principles and Role of Medical Imaging in Surgery. Berlin, Heidelberg: Springer Berlin Heidelberg; 2010. p. 529–543. Available from: https://doi.org/10.1007/978-3-540-71915-1_39.

[22] Guo Y, Ashour AS. 11 - Neutrosophic sets in dermoscopic medical image segmentation. In: Guo Y, Ashour AS, editors. Neutrosophic Set in Medical Image Analysis. Academic Press; 2019. p. 229 – 243. Available from: http://www.sciencedirect.com/science/article/pii/B9780128181485000114.

[23] Cai H, Verma R, Ou Y, Lee S, Melhem ER, Davatzikos C. PROBABILISTIC SEGMENTATION OF BRAIN TUMORS BASED ON MULTI-MODALITY MAGNETIC RESONANCE IMAGES. In: 2007 4th IEEE International Symposium on Biomedical Imaging: From Nano to Macro; 2007. p. 600–603.

[24] Fiaz M, Ali K, Rehman A, Gul MJ, Jung SK. Brain MRI Segmentation using Rule-Based Hybrid Approach; 2019.

[25] El-Baz A, Elnakib A, Abou-El-Ghar M, Gimel'farb G, Falk R, Farag A. Automatic Detection of 2D and 3D Lung Nodules in Chest Spiral CT Scans. International journal of biomedical imaging. 2013 02;2013:517632.

[26] Mansoor A, Bagci U, Foster B, Xu Z, Papadakis G, Folio L, et al. Segmentation and Image Analysis of Abnormal Lungs at CT: Current Approaches, Challenges, and Future Trends. Radiographics : a review publication of the Radiological Society of North America, Inc. 2015 07;35:1056–76.

[27] Zheng Z, Zhang X, Xu H, Liang W, Zheng S, Shi Y. A Unified Level Set Framework Combining Hybrid Algorithms for Liver and Liver Tumor Segmentation in CT Images. BioMed Research International. 2018 08;2018:1–26.

[28] Liu X, Faes L, Kale A, Wagner S, Fu D, Bruynseels A, et al. A comparison of deep learning performance against health-care professionals in detecting diseases from medical imaging: a systematic review and meta-analysis. The Lancet Digital Health. 2019 09;1.

[29] Hoffman M. Picture of the Knee, Human Anatomy;. Visited on 2020-05-09. Available from: https://www.webmd.com/pain-management/knee-pain/picture-of-the-knee#1.

[30] Berger A. Magnetic resonance imaging. BMJ. 2002;324(7328):35. Available from: https://www.bmj.com/content/324/7328/35.

[31] Tilakaratna P. How Magnetic Resonance Imaging works explained simply.;. Visited on 2020-05-10. Available from: `https://www.howequipmentworks.com/mri_basics/`.

[32] R George, J Dela Cruz, R Singh, Rajapandian Ilangovan. T1 SE/T1 TSE/T1 FSE Fat saturated;. Visited on 2020-05-10. Available from: `https://mrimaster.com/characterise%20image%20t1%20fat%20sat.html`.

[33] Larobina M, Murino L. Medical Image File Formats. Journal of digital imaging. 2013 12;.

[34] Sharma N, Aggarwal L. Automated medical image segmentation techniques. Journal of medical physics / Association of Medical Physicists of India. 2010 04;35:3–14.

[35] Lakare S. 3D Segmentation Techniques for Medical Volumes. 2000 01;.

[36] Kaur D, Kaur Y. Various Image Segmentation Techniques: A Review; 2014. .

[37] Homlong EG. Computer-Aided Diagnostics: Segmentation of Knee Joint Anatomy Using Deep Learning Techniques;. Visited on 2020-05-10. Available from: `http://hdl.handle.net/11250/2621247`.

[38] Gu J, Wang Z, Kuen J, Ma L, Shahroudy A, Shuai B, et al. Recent Advances in Convolutional Neural Networks. Pattern Recognition. 2015 12;.

[39] Brewka G. Artificial intelligence—a modern approach by Stuart Russell and Peter Norvig, Prentice Hall. Series in Artificial Intelligence, Englewood Cliffs, NJ. The Knowledge Engineering Review. 1996;11(1):728.

[40] Ruder S. An overview of gradient descent optimization algorithms;. Visited on 2020-05-12. Available from: `https://ruder.io/optimizing-gradient-descent/index.html#gradientdescentvariants`.

[41] Marsland S. Machine Learning: An Algorithmic Perspective, Second Edition. 2nd ed. Chapman Hall/CRC; 2014.

[42] Gribbestad M. Prognostics and Health Management for Air Compressors Based on Deep Learning Techniques;. Visited on 2020-05-12. Available from: `http://hdl.handle.net/11250/2621757`.

[43] Chollet F. Deep Learning with Python. 1st ed. USA: Manning Publications Co.; 2017.

[44] Chen X, Lin X. Big Data Deep Learning: Challenges and Perspectives. IEEE Access. 2014;2:514–525.

[45] Valueva MV, Nagornov NN, Lyakhov PA, Valuev GV, Chervyakov NI. Application of the residue number system to reduce hardware costs of the convolutional neural network implementation. Mathematics and Computers in Simulation. 2020;177:232 – 243. Available from: `http://www.sciencedirect.com/science/article/pii/S0378475420301580`.

[46] LeCun Y, Bengio Y, Hinton G. Deep Learning. Nature. 2015 05;521:436–44.

[47] IceCream Labs. 3x3 convolution filters — A popular choice; 2018. Visited on 2020-05-26. Available from: `https://medium.com/@icecreamlabs/3x3-convolution-filters-a-popular-choice-75ab1c8b4da8`.

[48] Luo W, Li Y, Urtasun R, Zemel R. Understanding the Effective Receptive Field in Deep Convolutional Neural Networks; 2017.

[49] Isensee F, Petersen J, Klein A, Zimmerer D, Jaeger P, Kohl S, et al. nnU-Net: Self-adapting Framework for U-Net-Based Medical Image Segmentation. 2018 09;.

[50] Milletari F, Navab N, Ahmadi SA. V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation; 2016.

[51] Bertels J, Eelbode T, Berman M, Vandermeulen D, Maes F, Bisschops R, et al.. Optimizing the Dice Score and Jaccard Index for Medical Image Segmentation: Theory Practice; 2019.

[52] Jaccard P. ”Étude comparative de la distribution florale dans une portion des Alpes et des Jura”. Bulletin de la Société vaudoise des sciences naturelles; 1901.

[53] Sørensen T. ”A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons”. Kongelige Danske Videnskabernes Selskab; 1948.

[54] Dice LR. Measures of the Amount of Ecologic Association Between Species. Ecology. 1945;26(3):297–302. Available from: `http://www.jstor.org/stable/1932409`.

[55] Elnakib A, Gimel’farb G, Suri J, El-Baz A. In: Medical Image Segmentation: A Brief Survey; 2011. p. 1–39.

[56] Robert M Haralick LGS. Image segmentation techniques; 1984.

[57] Zucker SW. Region growing: Childhood and adolescence. Computer Graphics and Image Processing. 1976;5(3):382 – 399. Available from: `http://www.sciencedirect.com/science/article/pii/S0146664X76800147`.

[58] Hojjatoleslami SA, Kruggel F. Segmentation of large brain lesions. IEEE Transactions on Medical Imaging. 2001;20(7):666–669.

[59] Wan SY, Higgins W. Symmetric region growing. IEEE transactions on image processing : a publication of the IEEE Signal Processing Society. 2003 02;12:1007–15.

[60] Mendonça AM, Campilho AJC. Segmentation of retinal blood vessels by combining the detection of centerlines and morphological reconstruction. IEEE Transactions on Medical Imaging. 2006;25:1200–1213.

[61] Mukhopadhyay S. A Segmentation Framework of Pulmonary Nodules in Lung CT Images. Journal of digital imaging. 2015 06;29.

[62] Justice RK, Stokely EM, Strobel JS, D REIM, Smith WM. Medical image segmentation using 3D seeded region growing. In: Hanson KM, editor. Medical Imaging 1997: Image Processing. vol. 3034. International Society for Optics and Photonics. SPIE; 1997. p. 900 – 910. Available from: `https://doi.org/10.1117/12.274179`.

[63] Mengqiao W, Jie Y, Yilei C, Hao W. The multimodal brain tumor image segmentation based on convolutional neural networks. In: 2017 2nd IEEE International Conference on Computational Intelligence and Applications (ICCIA); 2017. p. 336–339.

[64] Kamnitsas K, Chen L, Ledig C, Rueckert D, Glocker B. Multiscale 3d convolutional neural networks for lesion segmentation in brain MRI. Proc MICCAI Ischemic Stroke Lesion Segmentation Challenge. 2015 01;.

[65] Hesamian MH, Jia W, He X, Kennedy P. Deep Learning Techniques for Medical Image Segmentation: Achievements and Challenges. Journal of Digital Imaging. 2019 05;32.

[66] Ronneberger O, Fischer P, Brox T. U-Net: Convolutional Networks for Biomedical Image Segmentation; 2015.

[67] Long J, Shelhamer E, Darrell T. Fully Convolutional Networks for Semantic Segmentation; 2014.

[68] Zhang W, Li R, Deng H, Wang L, Lin W, Ji S, et al. Deep convolutional neural networks for multi-modality isointense infant brain image segmentation. NeuroImage. 2015;108:214 – 224. Available from: `http://www.sciencedirect.com/science/article/pii/S1053811914010660`.

[69] Vincent G, Guillard G, Bowes M. Fully Automatic Segmentation of the Prostate using Active Appearance Models; 2012. .

[70] Özgün Çiçek, Abdulkadir A, Lienkamp SS, Brox T, Ronneberger O. 3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation; 2016.

[71] Szegedy C, Vanhoucke V, Ioffe S, Shlens J, Wojna Z. Rethinking the Inception Architecture for Computer Vision; 2015.

[72] Oktay O, Schlemper J, Folgoc LL, Lee M, Heinrich M, Misawa K, et al.. Attention U-Net: Learning Where to Look for the Pancreas; 2018.

[73] Jetley S, Lord NA, Lee N, Torr PHS. Learn To Pay Attention; 2018.

[74] Zhou Z, Siddiquee MMR, Tajbakhsh N, Liang J. UNet++: A Nested U-Net Architecture for Medical Image Segmentation; 2018.

[75] Decathlon MS. Generalisable 3D Semantic Segmentation;. Visited on 2020-05-16. Available from: `http://medicaldecathlon.com/results.html`.

[76] Ibtehaz N, Rahman MS. MultiResUNet: Rethinking the U-Net architecture for multimodal biomedical image segmentation. Neural Networks. 2020 Jan;121:74–87. Available from: `http://dx.doi.org/10.1016/j.neunet.2019.08.025`.

[77] Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, et al.. Going Deeper with Convolutions; 2014.

[78] Folkesson J, Dam E, Olsen O, Pettersen P, Christiansen C. Segmenting Articular Cartilage Automatically Using a Voxel Classification Approach. Medical Imaging, IEEE Transactions on. 2007 02;26:106 – 115.

[79] Prasoon A, Petersen K, Igel C, Lauze F, Dam E, Nielsen M. Deep Feature Learning for Knee Cartilage Segmentation Using a Triplanar Convolutional Neural Network. vol. 16; 2013. p. 246–53.

[80] Antony J, McGuinness K, Moran K, O'Connor NE. Automatic Detection of Knee Joints and Quantification of Knee Osteoarthritis Severity using Convolutional Neural Networks; 2017.

[81] Zhou Z, Zhao G, Kijowski R, Liu F. Deep Convolutional Neural Network for Segmentation of Knee Joint Anatomy. Magnetic Resonance in Medicine. 2018 03;80.

[82] Ambellan F, Tack A, Ehlke M, Zachow S. Automated segmentation of knee bone and cartilage combining statistical shape knowledge and convolutional neural networks: Data from the Osteoarthritis Initiative. Medical Image Analysis. 2019;52:109 – 118. Available from: `http://www.sciencedirect.com/science/article/pii/S1361841518304882`.

[83] Byra M, Wu M, Zhang X, Jang H, Ma Y, Chang E, et al. Knee menisci segmentation and relaxometry of 3D ultrashort echo time cones MR imaging using attention U-Net with transfer learning. Magnetic Resonance in Medicine. 2019 09;83.

[84] Pettersen M. Segmentation of MR Images Using CNN;. Visited on 2020-05-30. Available from: `https://hdl.handle.net/11250/2650400`.

[85] Chen H. Automatic segmentation and motion analysis of the knee joint based on MRI and 4DCT images. University of Twente. Netherlands; 2020.

[86] Gibson E, Li W, Sudre C, Fidon L, Shakir DI, Wang G, et al. NiftyNet: a deep-learning platform for medical imaging; 2018. Available from: https://www.sciencedirect.com/science/article/pii/S0169260717311823.

[87] Eli Gibson WL, et al.. Niftynet;. Visited on 2020-05-16. Available from: https://niftynet.io/.

[88] Peterson B. Six: Python 2 and 3 Compatibility Library;. Visited on 2020-05-16. Available from: https://six.readthedocs.io/.

[89] Matthew Brett MHMACBCPM Chris Markiewicz, Cheng C. NiBabel Access a cacophony of neuro-imaging file formats;. Visited on 2020-05-16. Available from: https://nipy.org/nibabel/.

[90] Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods. 2020;17:261–272.

[91] van der Walt S, Colbert SC, Varoquaux G. The NumPy Array: A Structure for Efficient Numerical Computation. Computing in Science Engineering. 2011;13(2):22–30.

[92] pandas development team T. pandas-dev/pandas: Pandas. Zenodo; 2020. Available from: https://doi.org/10.5281/zenodo.3509134.

[93] Wes McKinney. Data Structures for Statistical Computing in Python. In: Stéfan van der Walt, Jarrod Millman, editors. Proceedings of the 9th Python in Science Conference; 2010. p. 56 – 61.

[94] Clark A, et al.. Python Imaging Library;. Visited on 2020-05-16. Available from: https://python-pillow.org/#.

[95] Kirtand J. Fast, simple object-to-object and broadcast signaling;. Visited on 2020-05-16. Available from: https://pythonhosted.org/blinker/.

[96] Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, et al.. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems; 2015. Software available from tensorflow.org. Available from: https://www.tensorflow.org/.

[97] NVIDIA. CUDA Toolkit;. Visited on 2020-05-16. Available from: https://developer.nvidia.com/cuda-toolkit.

[98] Yushkevich PA, Piven J, Cody Hazlett H, Gimpel Smith R, Ho S, Gee JC, et al. User-Guided 3D Active Contour Segmentation of Anatomical Structures: Significantly Improved Efficiency and Reliability. Neuroimage. 2006;31(3):1116–1128.

[99] Microsoft Corporation. Microsoft Excel;. Available from: https://office.microsoft.com/excel.

[100] Niftynet-documentation. niftynet.network.no_new_net module;. Visited on 2020-05-16. Available from: https://niftynet.readthedocs.io/en/dev/niftynet.network.no_new_net.html.

[101] Sola J, Sevilla J. Importance of input data normalization for the application of neural networks to complex industrial problems. Nuclear Science, IEEE Transactions on. 1997 07;44:1464 – 1468.

[102] Sola J, Sevilla J. Importance of input data normalization for the application of neural networks to complex industrial problems. IEEE Transactions on Nuclear Science. 1997;44(3):1464–1468.

[103] Hoffer E, Banner R, Golan I, Soudry D. Norm matters: efficient and accurate normalization schemes in deep networks; 2018.

[104] Nyul LG, Udupa JK, Xuan Zhang. New variants of a method of MRI scale standardization. IEEE Transactions on Medical Imaging. 2000;19(2):143–150.

[105] Zhou XY, Yang GZ. Normalization in Training U-Net for 2D Biomedical Semantic Segmentation; 2018.

[106] Niftynet-documentation. Data augmentation during training;. Visited on 2020-05-15. Available from: https://niftynet.readthedocs.io/en/dev/config_spec.html#data-augmentation-during-training.

[107] Simard PY, Steinkraus D, Platt JC. Best practices for convolutional neural networks applied to visual document analysis. In: Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.; 2003. p. 958–963.

[108] Andersson E, Berglund R. Evaluation of Data Augmentation of MR Images for Deep Learning; 2018. .

[109] Ronneberger O, Fischer P, Brox T. U-Net: Convolutional Networks for Biomedical Image Segmentation; 2015.

Jørgen André Sperre

Segmentation of Knee Joint Using 3D Convolutional Neural Networks

NTNU

Norwegian University of
Science and Technology