UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

UNIVERSITAT DE BARCELONA

Master's Degree in Biomedical Engineering

Final Master's Thesis

# Deep learning-based method for the segmentation of intracranial aneurysms in TOF-MRI images

## An automatic approach

**Author**: Arià JAIMEJUAN COMES

**Director**: Alexandre PERERA LLUNA

**Department**: CREB - Systems Engineering, Automatics and Industrial Informatics

Barcelona, June 17, 2023

# Abstract

Unruptured intracranial aneurysms are abnormal bulges or sacs in the walls of cerebral blood vessels, which can potentially lead to severe complications if left untreated. Their diagnosis and treatment heavily rely on accurate segmentation from medical imaging data. Traditional segmentation methods often require extensive manual intervention and expertise, leading to time-consuming and subjective results. The introduction of deep learning techniques offers a promising avenue to overcome these limitations, enabling automated and precise segmentation. This study presents a deep learning-based method for the segmentation of intracranial aneurysms from time-of-flight magnetic resonance imaging. This approach uses a deep convolutional neural network architecture, trained on 2 public dataset containing images from patients with manual segmentations performed by experts. The proposed method achieves state of the art accuracy in this task. Evaluation was conducted on a clinical data set obtaining promising results, with moderate agreement with manual segmentations performed by experts. The proposed method has the potential to improve the accuracy and efficiency of intracranial aneurysm detection and monitoring, which could have a significant impact on healthcare and clinical decision-making.

# Contents

# List of Figures



# List of Tables

## Acronyms

**ACA** Anterior cerebral artery. 7, 8

**CNN** Convolutional neural networks. 10–12

**MCA** Middle cerebral artery. 7

**SAH** Subarachnoid Hemorrhage. 6

**TOF-MRI** Time-of-flight magnetic resonance imaging. 8, 11, 12, 15, 17–20

**UIA** Unruptured Intracranial Aneurysm. 4, 6, 8, 9, 11–13, 15–19

# 1 Introduction

Unruptured Intracranial aneurysms (UIA) are localised dilations of arterial blood vessels within the brain, resulting from weakened vessel walls . Although rather unknown, these abnormalities are more prevalent than previously understood, with approximately 1 in 50 individuals unknowingly harboring a cerebral aneurysm [1]. Although often asymptomatic, among the potential dangers associated with UIAs, subarachnoid hemorrhage (SAH), a severe type of stroke, stands out as a significant concern. SAH occurs when a cerebral aneurysm ruptures, leading to the release of blood into the subarachnoid space surrounding the brain [2]. This condition, although relatively rare in the general population, affecting around 1 in 10,000 individuals, accounts for a substantial number of cases annually, with an estimated 30,000 new cases reported in the United States alone. The mortality rate associated with SAH is alarmingly high, surpassing 40%, with survivors often experiencing neurological and cognitive impairments [3]. Understanding the prevalence, risks, and outcomes of intracranial aneurysms is vital for improving diagnosis, treatment, and patient outcomes in clinical practice.

Accurate measurements play a crucial role in diagnosing UIAs and determining the risk of their growth and rupture [4, 5]. However, detecting and measuring UIAs can present significant challenges in medical imaging. Nonetheless, the field has seen remarkable advancements, particularly in scan resolution, especially at higher field strengths. These advancements have greatly supported radiologists in their efforts to detect UIAs. As a result, the improved capabilities have also led to an increased demand for patient screening. The detection and segmentation of aneurysms continue to pose significant challenges for radiologists, often leading to a bottleneck in the overall diagnostic workflow for patients. This process can be arduous and time-consuming, requiring careful attention to detail and expertise. Automatic methods for the detection and quantification of UIAs hold promise in this regard. However, it is crucial to ensure that these methods do not compromise the accuracy of human observers in detecting and measuring UIAs. One potential approach is automated volumetric segmentation, which would enable 3D quantification of these abnormalities. This could potentially aid in predicting the risk of rupture and improve patient management. Balancing the benefits of automated methods with the expertise of human observers is essential to ensure accurate and reliable results in the detection and assessment of UIAs.

# 2 Background and Motivation

## 2.1 Intracranial Aneurysm Clinical Background

The cerebral vasculature encompasses a network of blood vessels, veins, and arteries that supply the brain with blood. Serving as the primary blood supply to the brain, the carotid arteries provide nourishment to the majority of the cerebrum and vertebral circulation. The circulation is divided into anterior and posterior components, with the internal carotid arteries supplying the anterior circulation through the anterior cerebral artery (ACA) and middle cerebral artery (MCA). The ACA supplies blood to the frontal parietal and a portion of the occipital lobe, while the MCA, being the largest branch, supplies blood to the cerebrum. The posterior circulation is supported by the vertebral arteries, which form the basilar artery, nourishing the cerebellum and posterior meningeal arteries [6].

The convergence of these various arteries at the base of the brain forms the Circle of Willis, a ring-like structure comprised of the anterior, middle, posterior, and communicating arteries, providing collateral circulation and the ability to create a bypass in the event of branch occlusion (see Figure 1). The internal carotid and vertebral arteries divide into smaller arteries and arterioles. These vessels travel along the surface of the brain before entering the brain tissue to supply blood to specific regions of the cerebral cortex. Morphological variations in the Circle of Willis are common [7] and can introduce flow disruptions that may increase the risk of aneurysm formation [8].

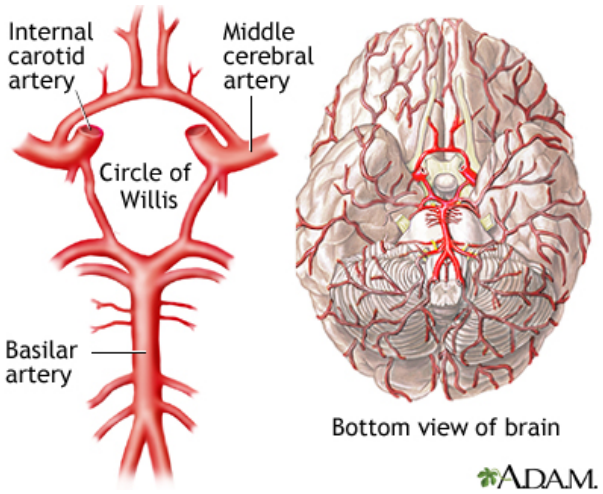



Figure 1: Schematic Representation of the Circle of Willis [9].

Cerebral aneurysms are a cerebrovascular pathology characterised by abnormal expansions that occur within the blood vessel's wall. These anomalies arise due to degeneration and weakness in the elastic cap of the vessel wall. Aneurysms can result in intracranial bleeding, leading to a hemorrhagic stroke and other complications with a significant risk of morbidity and mortality [10]. Its classification can be based on various factors, including size, shape, location, vessel type, and associated conditions. Small aneurysms have a diameter of less than 10mm, while large aneurysms measure above 10mm, and giant aneurysms exceed 25mm. The most common shape observed is saccular, whereas fusiform aneurysms are less prevalent (see Figure 2). Aneurysms are frequently found in the Circle of Willis and the anterior circulation.

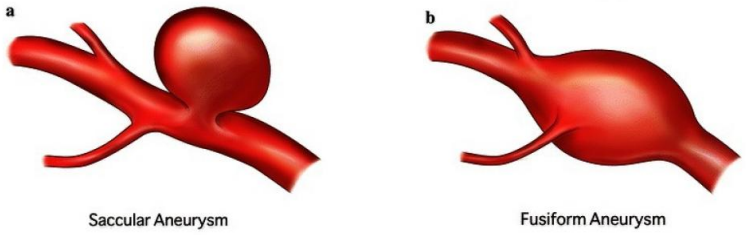



Figure 2: Aneurysm's shape drawing [11].

The formation of cerebral aneurysms is associated with several risk factors, including age, sex (more common in women), smoking, alcohol habits, familial background [12], and underlying conditions such as atherosclerosis [13]. The development of aneurysms has a strong correlation with biological inflammatory pathways that result in changes in local blood flow, mechanical properties of the vessel wall, and biochemical mediators [14]. Although often asymptomatic, ruptured aneurysms can lead to severe headaches, neurological damage, and other symptoms. Diagnosis commonly includes the application of imaging methods like computed tomography angiography (CTA), magnetic resonance angiography (MRA), or angiography (XA). Since these conditions are often asymptomatic, they are frequently detected during routine check-ups.

Currently, there are two courses of action: monitoring or intervention. Cerebral aneurysms can be managed through non-surgical methods such as observation and medication. For small, asymptomatic aneurysms with a low rupture risk, close monitoring through regular imaging may be recommended to assess any potential growth, as it plays a crucial role in determining the risk of rupture [15]. Regular monitoring enables informed decisions regarding the need for treatment [16]. Contrast-enhanced computed tomography angiography (CTA) and non-contrast 3D time-of-flight magnetic resonance angiography (ACA) are the most common imaging techniques used for monitoring UIAs. TOF-MRI, in particular, is well-suited for routine follow-up

imaging due to its non-invasive nature and lack of contrast agent or radiation requirement [17]. Alternatively, medications like calcium channel blockers and statins can help control underlying risk factors such as hypertension and high cholesterol [18].

Surgical treatment options involve more direct intervention. Clipping, a traditional surgical technique, involves placing a metal clip at the aneurysm's base to halt blood flow and prevent rupture. This procedure requires a craniotomy, which involves opening the skull. Alternatively, endovascular coiling is a minimally invasive procedure where platinum coils are inserted into the aneurysm using a catheter. This promotes blood clotting and seals off the weakened area [19]. Flow diversion is a newer technique using a stent-like device placed across the aneurysm's neck to redirect blood flow away from it, reducing the risk of rupture [20]. Similarly, flow disruption with intrasacular devices like WEB combine coiling and flow diversion [21] and modify the blood flow inducing intra-aneurysmal thrombosis.

## 2.2 Traditional Segmentation Techniques

Historically, manual segmentation of intracranial aneurysms has been performed by radiologists using imaging modalities such as computed tomography (CT) and magnetic resonance imaging (MRI). While these methods can provide valuable insights, they are labor-intensive, time-consuming, and prone to inter- and intra-observer variability. Traditional computational approaches, including thresholding, region-growing, and active contour models, have been employed to automate segmentation to some extent. However, these methods often struggle with accurately capturing the complex shape and positional variability of aneurysms.

Some more complex methods specific to the task can also be found in literature. Semi-automatic approaches involve defining the aneurysm's neck( where it connects to the parent vessel) before performing the segmentation process [22] while other methods utilise the shape of the aneurysm, such as blobness filters or shape analysis of vessel segmentations, to aid in detection [23, 24].

However, UIA segmentation has proven to be a challenging task, and there has been limited success in this area. The complexity arises from their diverse locations, their varying positions relative to blood vessels, small size, and wide variation in shape and configuration.

## 2.3 Deep Learning in Medical Image Segmentation

In recent years, deep learning techniques, specifically convolutional neural networks (CNNs), have revolutionized medical image segmentation tasks. CNNs excel at learning hierarchical representations from large-scale data, enabling them to capture intricate patterns and features in medical images. Several studies have explored the application of CNNs for the segmentation of various anatomical structures. These deep learning-based methods have demonstrated improved accuracy, robustness, and efficiency compared to traditional approaches.

Two important breakthroughs in the field where U-Net and nnU-Net models for addressing the unique challenges and requirements of biomedical imaging. U-Net [25], introduced by Ronneberger et al. in 2015, presented a novel architecture that revolutionised the segmentation task. It introduced a U-shaped network design consisting of an encoder-decoder structure, enabling the model to capture both local and global contextual information effectively. The skip connections between the encoder and decoder layers facilitated the integration of fine-grained details, enhancing the segmentation accuracy (see Figure 3). U-Net quickly gained popularity due to its simplicity, efficiency, and impressive performance across various medical imaging modalities.

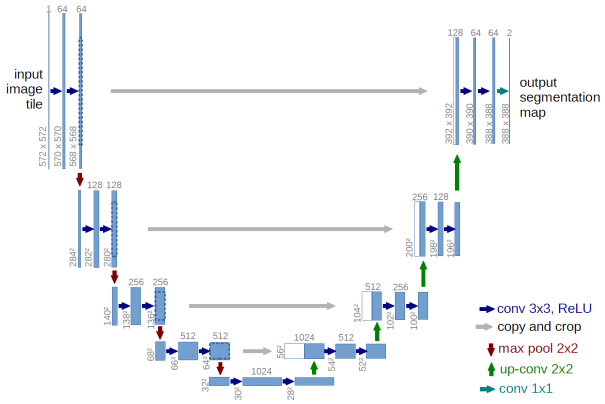



Figure 3: U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations. [25].

Building upon the success of U-Net, nnU-Net [26], proposed by Isensee et al. in 2021, introduced several advancements to further improve the performance and usability of medical image segmentation. nnU-Net focused on addressing the challenges of limited annotated data, as acquiring large-scale labelled datasets in the medical domain is often challenging. It introduced a framework that combined U-Net architecture with an efficient training pipeline and data aug-

mentation strategies.

Both U-Net and nnU-Net have revolutionized medical image segmentation by significantly improving the accuracy and efficiency of the process. These architectures have been successfully applied to various medical imaging tasks, such as organ segmentation, tumor detection, and lesion delineation. The advancements provided by U-Net and nnU-Net have paved the way for automated and reliable medical image analysis, enabling faster diagnosis, treatment planning, and research advancements. Their widespread adoption and ongoing research in this area continue to shape the field, facilitating advancements in precision medicine and enhancing patient care.

## 2.4 Intracranial Aneurysms automatic segmentation: Literature Overview

The application of deep learning in UIA segmentation holds promising perspectives for advancements in the field. However, the success of deep learning models heavily relies on the availability of large amounts of labelled data for training. Unfortunately, in the context of aneurysm segmentation, obtaining such annotated data can be a laborious and challenging process, requiring manual identification and delineation of aneurysm regions in medical images. The task of manual annotation is time-consuming and demands expertise, adding complexity to the development and training of deep learning models for accurate aneurysm segmentation.

Because of this, most of the work done in the field revolves around the The ADAM (Aneurysm Detection and Analysis) Segmentation Challenge (`https://adam.isi.uu.nl/`). The datset consist of 93 cases (TOF-MRI and one structural MR image) containing at least one untreated, unruptured intracranial aneurysm. This dataset has played a significant role in advancing the field of intracranial aneurysm segmentation by providing annotated data, addressing one of the key challenges in developing accurate and robust segmentation algorithms: the need for large-scale, high-quality labelled datasets.

Through the ADAM Segmentation Challenge, the field has witnessed the emergence of state-of-the-art approaches that push the boundaries of segmentation accuracy and efficiency. A thorough summary of the results for all contestants can be found in [27]. The different approaches employed various neural network architectures and techniques to improve segmentation accuracy. Some methods utilized 2D CNNs with unique input branches and concatenation strategies, while others employed 3D CNNs based on U-Net architectures. Different loss functions, such as the generalized dice loss and boundary loss, were used to optimize the segmentation process.

Ensemble methods, including averaging or majority voting of multiple models, were employed to enhance the final segmentation results. Input modalities varied, including raw TOF signal, blood vessel segmentation, maximum intensity projections (MIPs), and aligned structural images. The best model was a 3D fully-convolutional neural network based on nnUNet architecture for TOF-MRI segmentation. The models were trained using five-fold cross-validation and employed two different loss functions: Dice loss and cross entropy, as well as Dice loss with topK loss [28]. These loss functions were chosen for their robustness in handling highly imbalanced segmentation tasks [29]. During prediction, an ensemble approach was used by combining the outputs of the five models with the best performance, presenting a mean Dice score of 0.41 on the ADAM test set.

Similar articles show results for a cohort of 140 IA cases with cranial computed tomography angiography (CTA) images [30] by using 3 different CNN architectures: 3D UNet [31], VNet [32] and 3D Res-UNet . The models underwent training for 500 epochs using the Adam optimizer with an initial learning rate of 0.0001, and using the the Dice loss. The best performing model was the 3D-UNet architecture with a mean Dice coefficient of 0.818. Another article [33] yielded similar results for a cohort of 253 patients with 294 aneurysms using a 3D CNN based on DeepMedic [34].

In general, most approches use CNN models with variations of the U-Net architecture for volumetric images with different image modalities presenting different levels of accuracy, with the CTA based models yilding the best results.

## 2.5 Challenges in Intracranial Aneurysm Segmentation

Despite the progress made in deep learning-based segmentation, several challenges persist. One key challenge is the limited availability of annotated data, as acquiring reliable ground truth segmentations can be difficult and time-consuming and imply working with hospitals and the medical sector. Furthermore, the inherent variability in aneurysm shape, size, and appearance poses additional difficulties for accurate segmentation. Addressing these challenges requires the development of novel methodologies that can effectively handle limited data and handle variations in aneurysm morphology.

## 2.6 Objectives

The objective of this master's thesis is to develop a novel deep learning method for the automatic segmentation of UIAs. The aim is to leverage the power of deep learning algorithms

to accurately and efficiently identify and delineate aneurysm regions in medical images without the need for manual intervention. By automating the segmentation process, this approach has the potential to streamline the diagnosis and treatment planning of intracranial aneurysms, ultimately improving patient care and outcomes.

# 3 Methods

After conducting a thorough review of the current state-of-the-art research on the problem under consideration, it was concluded that adopting a modular solution approach would be advantageous due to the problem's complexity. In order to effectively address the challenges associated with the problem, the decision was made to divide it into two separate and more manageable tasks: **aneurysm localization** and **aneurysm segmentation**. By addressing each task independently, it becomes feasible to tackle the challenges separately and develop tailored solutions that are more precise and specific to each task.

Aneurysm localization involves identifying the presence and approximate location of aneurysms within medical images, providing a foundation for subsequent analysis. Aneurysm segmentation, on the other hand, aims to precisely delineate the boundaries of the identified aneurysms, enabling accurate measurement and analysis.

The proposed algorithm comprises two distinct deep learning models. The first model focuses on extracting the precise location information of the aneurysm from the input image. The output of this model, which captures the aneurysm location, will be passed on to the second model. The second model, in conjunction with the input images, will process the information and generate a binary segmentation mask. A schematic representation of the full algorithm is shown in figure 4.

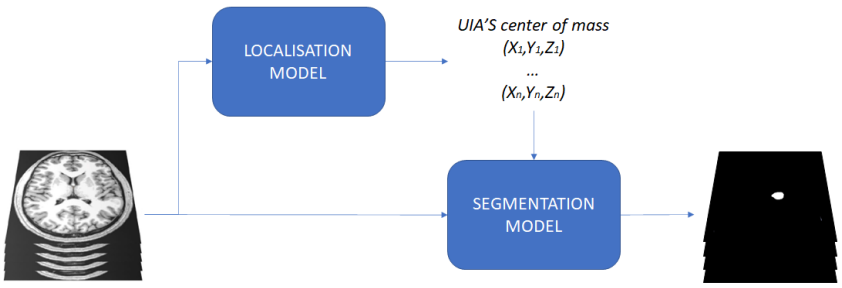



Figure 4: Schematic representation of the sub-modules of the UIA segmentation algorithm.

To create these models, there are two main approaches to consider: utilizing off-the-shelf architectures or building a model from scratch. Off-the-shelf architectures are pre-existing,

ready-to-use neural network architectures developed by third parties. They provide the advantage of requiring minimal coding since the program handles processes such as data loading, training, and validation. This means that users can leverage the architecture without the need for extensive coding or implementation. The off-the-shelf architecture streamlines the model creation process, allowing users to focus more on the specific application or problem they are addressing, rather than spending time on the intricacies of coding the entire model from scratch.

On the other hand, creating a model from scratch involves designing and training a custom architecture tailored to the specific requirements of the problem at hand. This approach offers flexibility in model design and allows to incorporate domain-specific knowledge and insights into the architecture. Developing a custom model requires a deep understanding of the problem domain, extensive experimentation, and careful parameter tuning.

To effectively address the two tasks, distinct architectures were utilized. For the aneurysm localization task, a suitable approach involves leveraging an off-the shelf architecture, given its compelxity. Off-the-shelf architectures are developed by experts and subjected to rigorous testing and optimization processes, offering the advantage of delivering good performance across a wide range of tasks and datasets. This proven performance makes them particularly valuable when there are limitations in terms of resources or time. By leveraging these pre-existing architectures, one can bypass the need to develop and fine-tune a new model from scratch, saving significant time and computational resources. Additionally, the expertise and thorough testing that go into these architectures ensure that they have already been optimized to perform well in various scenarios. This reliability and efficiency make off-the-shelf architectures an attractive choice, especially when there is a need for quick and effective solutions.

Conversely, for aneurysm segmentation, a custom model will be trained from scratch to address the specific challenges of accurately delineating aneurysm boundaries. This approach allows for tailored design choices that cater to the intricacies of the task. The model will be trained using crops centered around the aneurysm, as the localization information will be obtained from the localisation model. By training the segmentation model from scratch, domain-specific knowledge can be integrated, and the model's parameters can be optimized to achieve superior segmentation performance.

The decision to employ different approaches is aimed at gaining a better understanding of each approach's implications and implementation. This comprehensive analysis allows for

a thorough examination of the strengths, limitations, and considerations associated with off-the-self and custom architectures, contributing valuable insights to the field of medical image analysis.

## 3.1 Data Collection & Preprocesing

Two separate image datasets were utilized in this study, both consisting of TOF-MRI images.

### 3.1.1 ADAM: Aneurysm Detection And seMentation Challenge

The Aneurysm Detection and Segmentation Challenge 2020 (`https://adam.isi.uu.nl/`) was held by the MICCAI (Medical Image Computing and Computer Assisted Intervention, `http://www.miccai.org/`) Society in 2020. The challenge consisted on two distinct tasks: Task 1 focused on the automatic detection of UIAs on TOF-MRI, while Task 2 aimed to develop a method capable of automatically segmenting UIAs on TOF-MRI. The goal of the challenge was to find a method that performs optimally for either one of both tasks.

The dataset comprised a total of 254 brain TOF-MRA scans with 282 untreated unruptured intracranial aneurysms:

- **Training dataset**: Public for everyone upon registration to the challenge. Consisting of 113 cases, including 93 cases with at least one untreated UIA (35 baseline and follow-up cases from the same subject*, as well as 23 cases from unique subjects), and 20 cases of subjects without UIAs.
- **Test dataset**: Available when submitting a model for the challenge. Consisting of 141 cases, with 115 cases containing at least one untreated UIA (43 baseline and follow-up cases from the same subject*, along with 29 cases from unique subjects), and 26 cases of subjects without UIAs.

*Follow-up scan where taken 6 months apart.

Each case consisted of a TOF-MRI image and a structural image, including T1-weighted, T2-weighted, or FLAIR sequences (see figure 6). The MRIs were performed at UMC Utrecht, the Netherlands, using various Philips scanners with field strengths of 1, 1.5, or 3T.

The acquired TOF-MRIs had voxel spacing ranging from 0.195 to 1.04 mm in-plane and slice thickness ranging from 0.4 to 0.7 mm. Due to the clinical nature of the data and its collection from multiple studies conducted over a span of 2001 to 2019, there was no standardized

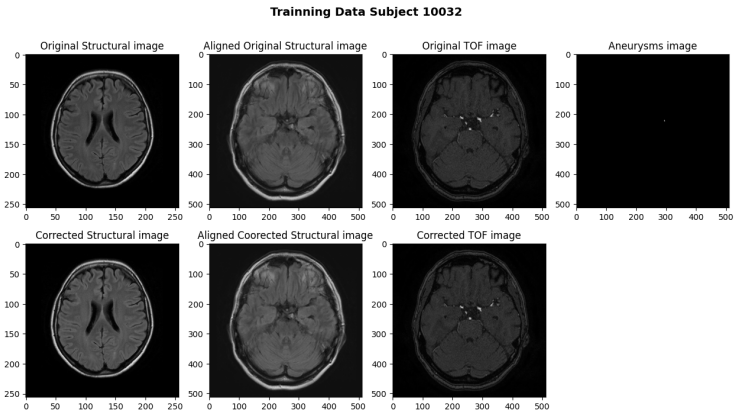



Figure 5: Plot for an arbitray slice for the different images provided for subject 10032

acquisition protocol. The median patient age for subjects with UIAs was 55 years (range 24-75 years) with 75% of subjects being female, whilst, the median patient age for subjects without UIAs was 41 years (range 19-61 years) with 65% of subjects being female.

The UIAs exhibited variability in size, with a median maximum diameter of 3.6 mm and a range from 1.0 to 15.9 mm. Approximately 25% of the scans contained multiple UIAs, and 28% of the scans involved treated UIAs, either coiled or clipped (59 cases). Figure 6 shows the aneurysm size distribution only for patients in the training data set, with a mean diameter size of 4.06 mm.

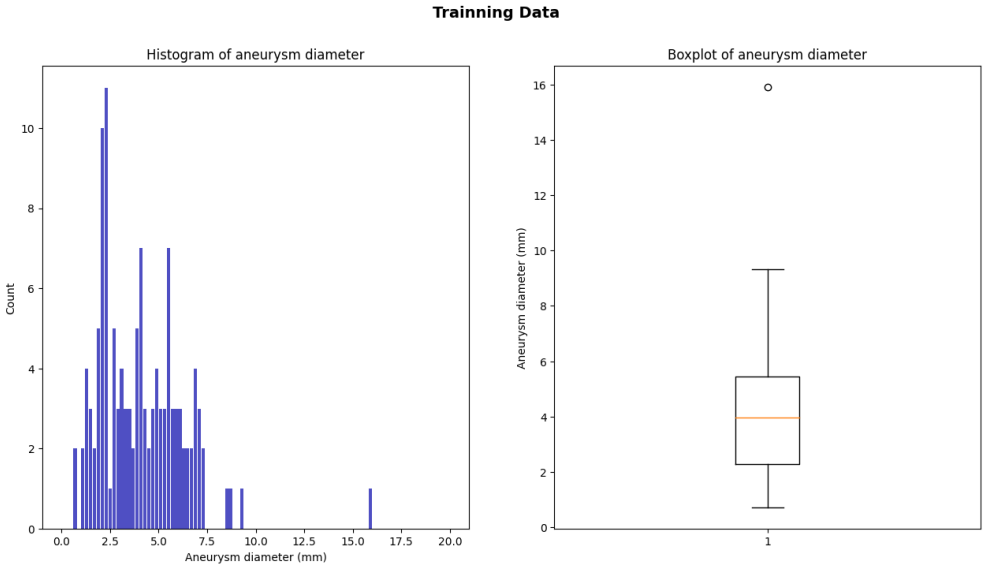



Figure 6: Histogram and box-plot of aneurysm's diameter for the ADAM training dataset

The dataset used in the ADAM Challenge was comprehensive and diverse, encompassing

different clinical protocols and scanner variations over a significant time period. This realism reflects the challenges faced in real-world scenarios and provides a robust foundation for evaluating and advancing automatic detection and segmentation methods for UIAs on TOF-MRI.

The protocol for aneurysm annotation was developed by an interventional neuro-radiologist, with more than 10 years of experience in the field. On each axial slice of the TOF-MRI a contour around the outline of the aneurysm was drawn. The annotations were always drawn to be from the level of the neck to the dome of the aneurysm. The neck corresponds to the opening of the aneurysm from the parent vessel. The dome is the furthest part of the aneurysm from the parent vessel. None of the parent vessel was included in the annotation. During annotation, radiologists had access to the structural image and a radiologist report made at the time of the scan. Reports indicated the rough location and size of the aneurysm. This consensus annotation was used to produce the binary masks as the official ground truth data set.The contours were converted to binary masks, including all voxels with at least half of their volume within the manually drawn contour. These masks were dilated by 1 pixel in-plane (with a 3x3x1 kernel). In case of overlap between labels 1 and 2, label 1 was assigned.

All subject had been anonymised. For every subject the images as well as some additional information were provided:

- a text file with the 3D voxel coordinates of the centre of mass of the aneurysms and the maximum radius of the aneurysm
- a label image with labels:
  - 0 = Background
  - 1 = Untreated, unruptured aneurysm
  - 2 = Treated aneurysms or artefacts resulting from treated aneurysms

Table 1 provides a brief description of each file. The structural image (T1, T2 or FLAIR) where aligned to the TOF image using the image registration software elastix [35].

In this project, only the training dataset from the ADAM Challenge was utilized. As no model was submitted to the challenge, the test dataset was not available for evaluation. Therefore, the focus of this project was solely on leveraging the training dataset to develop and evaluate the method for automatic detection and segmentation of UIAs.

| File | Details |
|---|---|
| /orig/TOF.nii.gz | The original TOF-MRI image. This image is used to manually delineate the aneurysms. |
| /orig/struct.nii.gz | The original structural image. |
| /orig/reg_struct_to_TOF.txt | Transformation parameters used to align the structural image with the TOF image. |
| /orig/ScanParams_TOF.json | Scan parameters of original TOF-MRI image |
| /orig/ScanParams_struct.json | Scan parameters of original structural image |
| /orig/struct_aligned.nii.gz | The structural image aligned with the TOF-MRI image. |
| /pre/struct.nii.gz | Bias field corrected structural image. |
| /pre/TOF.nii.gz | Bias field corrected TOF-MRI image. |
| /pre/struct_aligned.nii.gz | Bias field corrected structural image, aligned with TOF-MRI |
| /aneurysms.nii.gz | Aneurysm Mask |
| /location.txt | Text file containing centre of massvoxel coordinates x,y,z and radius for all unruptured, untreated aneurysms in the TOF-MRI . |

Table 1: ADAM files brief description

### 3.1.2 Lausanne TOF-MRA Aneurysm Cohort

The Lausanne TOF-MRI Aneurysm Cohort [36] was created by Di Noto T et Al with the goal of democratising the acces to medical data to further the knowledge in the field of UIA automatic segmentation [37].

The dataset consisted of a cohort of patients that underwent TOF-MRI between 2010 and 2015. The dataset comprised a total of 284 subjects, with 157 presenting one or more UIAs, while 127 were healthy patients. Patients with ruptured/treated aneurysms, completely thrombosed aneurysms, infundibula (dilatations of the origin of an artery) or other vascular pathologies were excluded. The dataset was anonymized and organized following the Brain Imaging Data Structure (BIDS) standard ([38]) and it is publicly available at OpenNeuro: `https://openneuro.org/datasets/ds003949` under the CC0 license.

Each case consisted of a TOF-MRI image with its corresponding segmentation mask. The MRIs were performed at Lausanne University Hospital (CHUV), Switzerland, using various Philips and Siemens scanners with field strengths of 1.5, or 3T.

The acquired images had voxel spacing ranging from 0.27 to 0.46 mm in-plane and slice thickness ranging from 0.5 to 1 mm. Due to the clinical nature of the data and its collection from multiple studies conducted over a span of 2010 to 2015, there was no standardized acquisition protocol. The median patient age for subjects with UIAs was 56 years (sd = 14) with 66% of subjects being female, whilst, the median patient age for subjects without UIAs was 46 years (sd=17) with 52% of subjects being female.

Table 2 shows overall locations and size grouped according to the PHASES score ([39]).

| | | **Count** | % |
|---|---|---|---|
| **Location** | ICA | 59 | 29.8 (59/198) |
| | MCA | 57 | 28.8 (57/198) |
| | ACA/Pcom/Posterior | 82 | 41.4 (82/198) |
| **Size** | $d \leq 7mm$ | 180 | 91.0 (180/198) |
| | $7 - 9.9mm$ | 7 | 3.5 (7/198) |
| | $10 - 19.9mm$ | 10 | 5.0 (10/198) |
| | $d \geq 20mm$ | 1 | 0.5 (1/198) |

Table 2: Locations and sizes of aneurysms according to the PHASES score for the Lausanne TOF-MRA Aneurysm Cohort dataset. *ICA Internal Carotid Artery, MCA Middle Cerebral Artery, ACA Anterior Cerebral Arteries, Pcom Posterior communicating artery, Posterior posterior circulation, d maximum diameter.*

The protocol for aneurysm annotation was performed by a radialogist with 2 years of experience in neuroimaging, and supervised by a senior neuroradiologist with over 15 years of experience. Two annotation schemes were used:

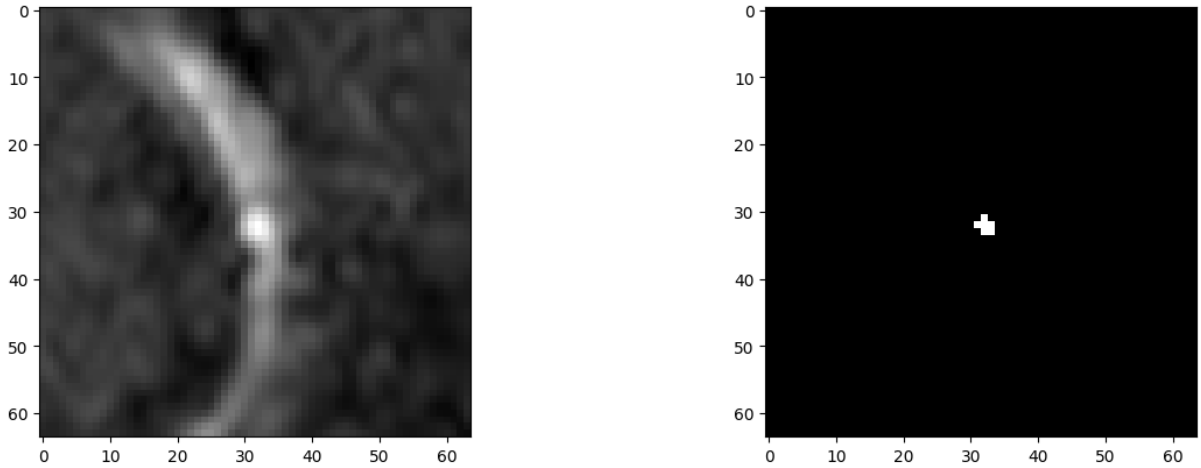
- **Weak labels**: For the majority of subjects (119 out of 157), the radiologist utilized the Multi-image Analysis GUI (Mango) software (version 4.0.1) to create weak labels. These labels represented spheres encompassing the entire aneurysm, irrespective of its shape. The radiologist manually selected the size of the spheres on a case-by-case basis, ensuring that the entire aneurysm was always eosed within the sphere.
- **Voxel-wise labels**: For the remaining subjects (38 out of 157), the radiologist employed

ITK-SNAP software (version 3.6.0) to create voxel-wise labels. These labels were drawn slice by slice while scrolling through the axial plane. There was no specific criterion used to select these 38 subjects, as they were consecutive to the 246 subjects in the first group.

### 3.1.3 Final dataset

Since the goal of the algorithm was to perform segmentation and not screening or detection, only subjects with aneurysm where used for this study. A manual inspection of the images was additionally performed, with bad quality images beeing dicarted. The final dataset consisted of 240 subjects with at least one untreated/unruptured aneursym. Cases were separated in 2 groups: Weak labels and Voxel-wise labels.

In the context of the location task, weak labels were employed to precisely identify the location of the aneurysm without necessitating the segmentation of its shape. By leveraging the spherical shape of the aneurysm masks for weak labels, the algorithm could focus exclusively on localising the aneurysm within the full-resolution image, obviating the need to learn intricate geometries. This approach facilitated efficient and accurate localisation, streamlining the algorithm's objective and reducing computational complexity. The total number of patients was 112 with 145 cases.




(a) TOF-MRA (b) Binary Mask

Figure 7: 64x64 crop centered around the aneurysm for an arbitrary slice

For the segmentation task, image crops of size 16x64x64 [DepthxWidthxHeight] were extracted from the TOF-MRI and segmentation masks (see figure 7), with each crop centered around the aneurysm. This approach allowed for a focused analysis of segmentation without

the requirement of locating the aneurysm within the full-resolution image. In order to maintain case separation, individual aneurysms from the same patient as well as aneurysms from follow-up scans were considered as distinct instances. The selection of crop size was determined based on the diameter sizes of the aneurysms present in the original datasets, ensuring adequate representation of the aneurysm region within the extracted crops. Location data was available for the ADAM dataset but had to be extracted for the Laussane dataset. The total number of patients was 138 with 169 cases.

The training set (n=222) and test set (n=28) were generated using a scheme that ensured both groups had a similar distribution of aneurysm sizes. Furthermore, all cases in the test dataset were extracted from the the vowel-wise label cases to assure they where fit to validate the whole model. The whole data cleanising pipeline its shown in figure 8.

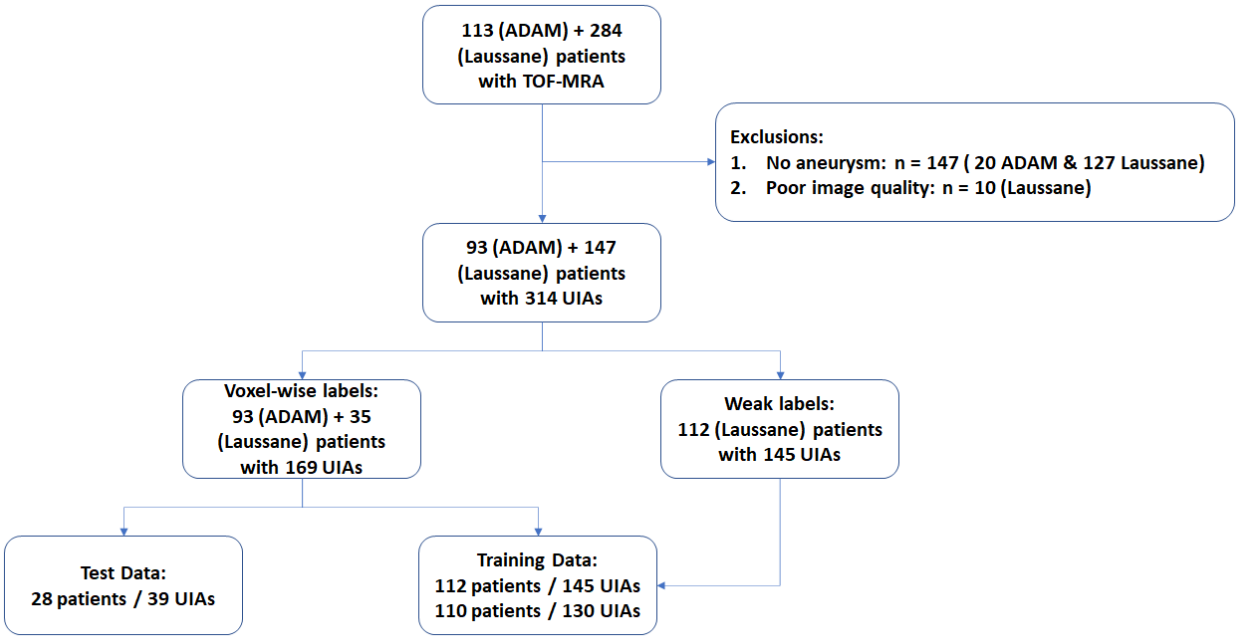



Figure 8: Data selection pipeline.

## 3.2 Aneurysm localisation

The chosen architecture for the localisation task was the nn-Unet architecture. nnU-Net is an automated deep learning-based segmentation method that dynamically configures itself for specific tasks in the biomedical domain. This comprehensive approach encompasses not only the network architecture but also preprocessing, training, and post-processing stages. With nnU-Net, the entire pipeline is tailored specifically to the specific task at hand, allowing for efficient and effective segmentation without the need for extensive manual configuration or parameter

tuning. By automating these processes, nnU-Net simplifies the deployment of deep learning models, saving time and effort for researchers and practitioners in the field [26].

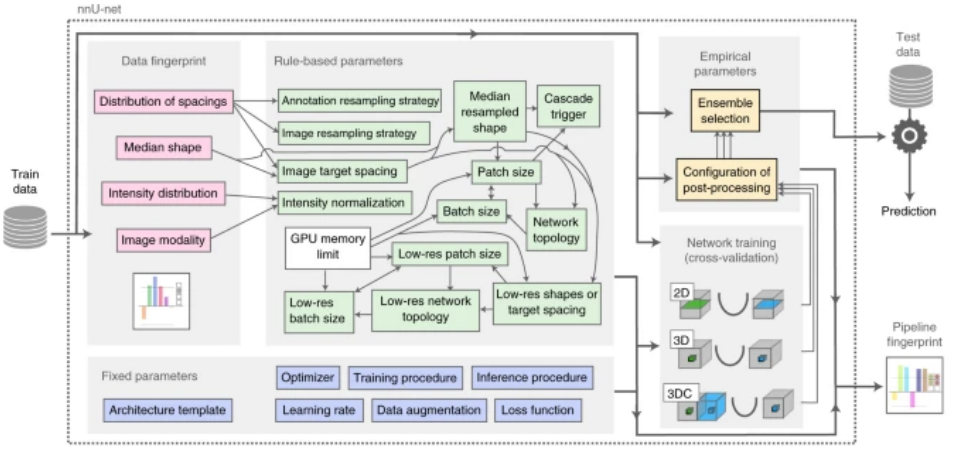
Although nnU-Net is primarily designed for image segmentation, obtaining 3D location information as the output can be easily achieved by calculating the center of mass for each predicted object. By analyzing the segmented regions, the center of mass can provide an estimate of the spatial location within the 3D space. This approach allows for straightforward localization without requiring significant modifications to the existing nnU-Net architecture.

### 3.2.1 Model Architecture

nnU-Net possesses the capability to automatically adapt to any new dataset. Figure 9 showcases the systematic approach employed by nnU-Net to configure complete segmentation pipelines. The figure provides a visual representation and detailed explanation of the key design choices adapted to the task at hand. This systematic approach ensures that nnU-Net efficiently addresses the entire configuration process, resulting in optimized segmentation pipelines tailored to the specific dataset being utilized.

When using nnU-Net on a new dataset, the configuration process is performed automatically without the need for manual intervention. This means that, apart from a few remaining empirical choices that need to be made, there is no additional computational cost beyond what is typically required for a standard network training procedure. The automated nature of nnU-Net's configuration allows for efficient and seamless deployment, eliminating the need for extensive manual adjustments and reducing computational overhead.

The initial step of this pipeline is the generation of the "Data Fingerprint". During the preprocessing step of nnU-Net, the provided training cases undergo cropping to their non-zero regions. This cropping process significantly reduces the image size of certain brain datasets, leading to improved computational efficiency. Subsequently, nnU-Net generates a dataset fingerprint based on the cropped training data, capturing essential parameters and properties. This includes information such as image size before and after cropping, image spacing (voxel size), modalities extracted from metadata, the number of classes for each image, and the total number of training cases. Additionally, the dataset fingerprint incorporates statistical measures, such as the mean, standard deviation, 0.5 percentile, and 99.5 percentile of intensity values within the foreground regions (voxels belonging to any class label) across all training cases. This comprehensive dataset fingerprint forms the foundation for further processing and customisation within




| Design choice | Required input | Automated (fixed, rule-based or empirical) configuration derived by distilling expert knowledge (more details in online methods) |
|---|---|---|
| Learning rate | – | Poly learning rate schedule (initial, 0.01) |
| Loss function | – | Dice and cross-entropy |
| Architecture template | – | Encoder–decoder with skip-connection ('U-Net-like') and instance normalization, leaky ReLU, deep supervision (topology-adapted in inferred parameters) |
| Optimizer | – | SGD with Nesterov momentum ($\mu = 0.99$) |
| Data augmentation | – | Rotations, scaling, Gaussian noise, Gaussian blur, brightness, contrast, simulation of low resolution, gamma correction and mirroring |
| Training procedure | – | 1,000 epochs × 250 minibatches, foreground oversampling |
| Inference procedure | – | Sliding window with half-patch size overlap, Gaussian patch center weighting |
| Intensity normalization | Modality, intensity distribution | If CT, global dataset percentile clipping & $z$ score with global foreground mean and s.d.<br>Otherwise, $z$ score with per image mean and s.d. |
| Image resampling strategy | Distribution of spacings | If anisotropic, in-plane with third-order spline, out-of-plane with nearest neighbor<br>Otherwise, third-order spline |
| Annotation resampling strategy | Distribution of spacings | Convert to one-hot encoding →<br>If anisotropic, in-plane with linear interpolation, out-of-plane with nearest neighbor<br>Otherwise, linear interpolation |

| | | |
|---|---|---|
| Image target spacing | Distribution of spacings | If anisotropic, lowest resolution axis tenth percentile, other axes median.<br>Otherwise, median spacing for each axis.<br>(computed based on spacings found in training cases) |
| Network topology, patch size, batch size | Median resampled shape, target spacing, GPU memory limit | Initialize the patch size to median image shape and iteratively reduce it while adapting the network topology accordingly until the network can be trained with a batch size of at least 2 given GPU memory constraints. for details see online methods. |
| Trigger of 3D U-Net cascade | Median resampled image size, patch size | Yes, if patch size of the 3D full resolution U-Net covers less than 12.5% of the median resampled image shape |
| Configuration of low-resolution 3D U-Net | Low-res target spacing or image shapes, GPU memory limit | Iteratively increase target spacing while reconfiguring patch size, network topology and batch size (as described above) until the configured patch size covers 25% of the median image shape. For details, see online methods. |
| Configuration of post-processing | Full set of training data and annotations | Treating all foreground classes as one; does all-but-largest-component-suppression increase cross-validation performance?<br>Yes, apply; reiterate for individual classes<br>No, do not apply; reiterate for individual foreground classes |
| Ensemble selection | Full set of training data and annotations | From 2D U-Net, 3D U-Net or 3D cascade, choose the best model (or combination of two) according to cross-validation performance |

Figure 9: Proposed automated method configuration for deep learning-based biomedical image segmentation [26].

nnU-Net.

The next step in the configuration of nnU-Net is designing the model. Based on the data fingerprint described earlier, as well as any hardware constraints specific to the project, the algorithms designs the whole workflow automatically inferring these choices using a set of heuristic rules and some user inputs, generating a so-called pipeline fingerprint that contains all relevant design information about the model. All model paramethers are classified in one of 3 groups: fixed, rule-based and empirical parameters (see figure 9). The rule-based parameters work in conjunction with fixed parameters, which are independent of the data, and empirical parameters that undergo optimization during training. This comprehensive approach ensures that nnU-Net not only automates the design process but also maximizes the efficiency and effectiveness of the resulting segmentation models.

#### 3.2.1.1 Fixed parameters

- **Architecture template:** The nnU-Net architecture is based on the U-Net model with minor modifications. It does not use recently proposed architectural variations but employs instance normalization and leaky ReLU nonlinearity. Deep supervision is used with auxiliary losses in the decoder. Initial number of feature maps is set to 32 . Feature maps are capped at 320 for 3D U-Nets and 512 for 2D U-Nets.
- **Training schedule:** See section 3.2.2.
- **Inference:** During the inference stage, a sliding window approach is used with a window size matching the training patch size. Adjacent predictions overlap by half the patch size. To mitigate border-related issues, a Gaussian importance weighting technique is applied, giving higher weight to central voxels in softmax aggregation.

#### 3.2.1.2 Rule-based parameters

- **Intensity normalization**: Different normalization schemes are used for different modalities, with z-scoring as the default option. CT images employ a global normalization scheme.
- **Resampling**: Images are resampled to a target spacing using different interpolation methods based on the voxel spacing. Anisotropic cases are handled differently to minimize resampling artifacts.
- **Target spacing**: The target spacing is selected based on the median and percentile values of the voxel spacings in the training cases.
- **Patch size**: The patch size is initialized as the median image shape after resampling and padded if necessary.
- **Architecture topology:** nnU-Net's architecture is determined based on the patch size and voxel spacing. Downsampling operations are applied until the feature map size reaches a minimum of four voxels or the feature map spacings become anisotropic. The downsampling strategy prioritizes high-resolution axes, which are downsampled individually until their resolution is within twice that of the lowest resolution axis. Further downsampling is then performed simultaneously on all axes. The default kernel size for convolutions is 3x3x3 for 3D U-Net and 3x3 for 2D U-Net. In cases of initial resolution discrepancies between axes, the out-of-plane axis's kernel size is set to one until the resolutions align within a factor of two. Following that, the convolutional kernel size remains three for all axes.

- **Memory Consumption:** The maximum patch size is limited by GPU memory. Initially, the patch size is set to the median image shape after resampling, which often exceeds GPU capacity. nnU-Net estimates memory consumption by comparing feature map sizes to reference values. The patch size is iteratively reduced until the memory budget is met.

- **Batch size:** The final step involves configuring the batch size. If the patch size was reduced, the batch size is set to two. Otherwise, the remaining GPU memory is utilized to increase the batch size until it is fully utilized. To prevent overfitting, the batch size is capped to ensure that the total number of voxels in the mini-batch does not exceed 5% of the total number of voxels across all training cases.

#### 3.2.1.3 Empirical parameters

- **Ensembling and selection of U-Net configuration(s)**: The framework automatically determines the best U-Net configuration(s) for inference based on cross-validation results. Models can be ensembled by averaging softmax probabilities.

- **Post-processing**: Connected component-based post-processing is used to eliminate false positives. The framework determines the effect of suppressing smaller components and decides whether it should be performed for individual classes.

### 3.2.2 Model Training

By default, nnU-Net generates 4 different configurations:

- The **2D U-Net** configuration operates on full-resolution 2D images (2D slices in case of 3D images) data and is particularly effective for handling anisotropic data

- The **3D full-resolution U-Net** configuration also operates on full-resolution data but has a patch size limitation determined by GPU memory availability. This configuration generally achieves the best performance overall. However, for large datasets, the patch size may be insufficient to capture an adequate amount of contextual information.

- The **3D low-resolution U-Net** configuration operates on downsampled or low-resolution data. This configuration is particularly useful when dealing with extremely large datasets where memory constraints or computational limitations come into play.

- The **3D U-Net cascade** configuration is a combination of the 3D low-resolution and The 3D full-resolution specifically designed for handling large datasets. It involves two stages: first, a 3D U-Net operates on low-resolution data to generate coarse segmentation maps.

These maps are then refined by a second 3D U-Net operating on the full-resolution data, enhancing the segmentation quality.

For this particular problem and because of limitations on time and computational cost, only the 3D full-resolution was used, since this is said to yield the best results. A 5 folder cross validation scheme is also implemented in the U-net and was used to assess the performance and generalisation of the model.

The default optimiser is stochastic gradient descent (SGD) with Nesterov momentum ($\mu = 0.99$) [40], initializing the learning rate at 0.01. Throughout the training process, the learning rate is decayed using the 'poly' scheme, which follows the formula $(1 - \frac{epoch}{epochmax})^{0.9}$.

The loss function used the training pipeline is a combination of cross-entropy and Dice loss. For each deep supervision output, a downsampled ground truth segmentation mask was used to compute the loss. The training objective involved summing the losses (L) at all resolutions, denoted as L = w1 × L1 + w2 × L2 + ..., where the weights (w) are halved with each decrease in resolution. This resulted in w2 = $\frac{1}{2}$ × w1, w3 = $\frac{1}{4}$ × w1, and so on. Additionally, these weights are normalized to ensure they add to 1.

During the construction of mini-batches, samples from the training cases are randomly selected. To handle class imbalances effectively, oversampling techniques were also implemented. Specifically, 66.7% of the samples are selected randomly from various locations within the chosen training case, while the remaining 33.3% are guaranteed to contain one of the foreground classes present in the selected training sample, randomly selected. To ensure a minimum of one foreground patch per batch (resulting in a batch size of two),the number of foreground patches is rounded.

To enhance the robustness of the model, various data augmentation techniques were applied on-the-fly during training. These techniques included rotations, scaling, Gaussian noise, Gaussian blur, adjustments to brightness and contrast, simulation of low resolution, gamma correction, and mirroring. Results for the training are shown in annex B.

### 3.2.3 Evaluation Metrics

A commonly used metric to assess the accuracy of predicted locations in a 3D space is the Euclidean distance. The Euclidean distance measures the straight-line distance between two points in a three-dimensional Cartesian coordinate system. In the context of evaluating model predic-

tions, the Euclidean distance can be calculated between the predicted location and the ground truth location in 3D space. A smaller Euclidean distance indicates a more accurate prediction, as the predicted location is closer to the ground truth. Additionally, variations of the Euclidean distance, such as the root mean square error (RMSE) or the mean absolute error (MAE), can also be utilized to provide a single aggregated score that represents the overall accuracy of the model's predictions across multiple locations in 3D space.

However, the Dice coefficient was selected as the evaluation metric for the model due to its integration within the nn-Unet pipeline. The Sørensen–Dice coefficient [41, 42], also known as the Dice similarity coefficient or Dice index, is a widely used evaluation metric for image segmentation in medical image analysis. It quantifies the agreement between the predicted segmentation mask and the ground truth by measuring the overlap or similarity between the two. The coefficient is defined in equation 1, with X and Y beeing the two different sets beeing compared, in this case, the prediction and the ground truth.

$$DSC = \frac{2|X \cap Y|}{|X| + |Y|} \tag{1}$$

The Dice coefficient ranges from 0 to 1, where a value of 1 indicates a perfect match between the predicted and ground truth segmentation and 0 indicates no overlap at all. It takes into account both the true positives and false positives, making it suitable for assessing the overall performance of segmentation algorithms. Its intuitive interpretation and ease of computation make it a valuable tool for comparing different segmentation approaches and tracking the progress of algorithmic improvements in medical image segmentation tasks.

While the Dice coefficient is traditionally used to assess the quality of segmentation and not location prediction, it is important to note that achieving accurate segmentation inherently implies a reliable spatial localisation. Additionally, the emphasis of evaluation in this task shifts towards accurate localisation rather than precise segmentation, since contour delimitation is facilitated given the utilisation of weak labels. Therefore, obtaining an accurate segmentation also indicates a successful prediction of the object's location. Hence, despite the primary focus being on correct segmentation, the Dice coefficient serves as a useful metric for evaluating the model's spatial localisation performance in this context.

### 3.2.4 Hardware

The whole architecture was coded in python using the PyTorch framework [43] and trained in the cloud with Google Colaboratory[44] using a 40 GB NVIDIA A100-SXM4 GPU, 83 GB of RAM and an Intel(R) Xeon(R) CPU @ 2.20GHz processor. All the generated code is in the following GitHub repository: `https://github.com/ariajc/Intracranial-aneurysm-segmentation`.

## 3.3 Aneurysm segmentation

The chosen architecture for the segmentation task was a 3D ResU-Net architecture, a combination of the original U-Net architecture adapted to volumetric with the concept of residual connections introduced in [45].

### 3.3.1 Model Architecture

The 3D U-Net introduced in [31] presents a U-Net like architecture with the introduction of an additional dimension to be able to process volumetric data. The 3D U-Net architecture builds upon the principles of the 2D U-Net. As figure 15 shows, the architecture is comprised of an encoder a bridge and a decoder. By combining the contextual information captured by the encoding path with the high-resolution features obtained from the decoding path it is possible to achieve accurate and detailed segmentation results.

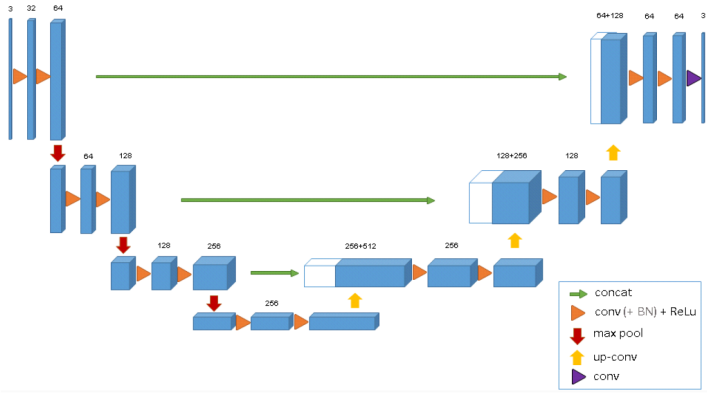
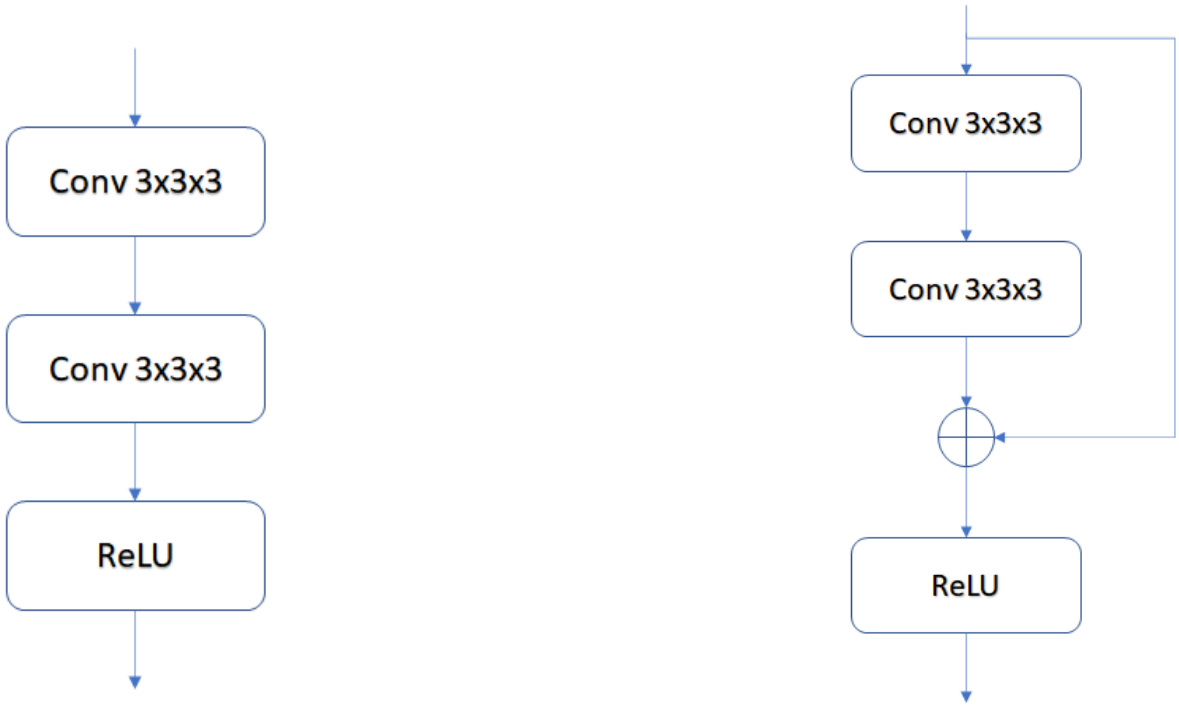



Figure 10: Schematic representation of the 3D U-Net architecture [31].

The difference with the 3D ResU-Net is the use of Residual Blocks instead of normal convolution blocks. Normal convolution blocks consist of a convolution followed by an activation function, while ResNet Blocks introduce residual connections, which address the problem of vanishing gradients when training very deep neural networks. Typically, as the number of layers in a neural network increases, the gradients tend to diminish during backpropagation, making it difficult to train deeper models effectively. ResNet addresses this issue by introducing skip connections, or shortcuts, that allow the network to directly learn the residual mapping between

input and output.

Residual block consists of a series of convolutional layers, typically with a small filter size (e.g., 3x3x3), followed by batch normalization and non-linear activation functions such as ReLU. Additionally a "skip connection" directly connects the input to the output of the residual block (see figure 11). This skip connection allows the network to learn the difference (or the "residual") between the input and the output, which is then added element-wise to the output of the convolutional layers. By introducing these residual connections, the network can learn to focus on modeling the residual information, making it easier to optimise the network and train much deeper architectures.




(a) Convolution Block (b) Residual Block

Figure 11: Schematic representation of a Convolution Block & Residual Block

The whole architecture consist of a batch normalisation layer, followed by an encoder network, a decoder and a final softmax layer to obtain the final prediction. Encoder and decoder are connected at each layer to pass on the information and provide additional context. Additional dropout layers were added to the model to prevent overfitting.

**3.3.1.1 Encoder** The encoder consist of 5 levels or layers each one containing a set of residual blocks followed by a max-pooling operation to reduce the spatial dimensions while increasing the receptive field. The number of filters usually increases with the depth of the

network, allowing for the extraction of more abstract features, with the initial layer having 64 feature maps that increase with a power of 2 scheme ([64,128,256,512,1024]).

**3.3.1.2 Decoder** The decoding path performs upsampling operations to gradually restore the spatial resolution and recover the detailed information. Each upsampling step involves a combination of 3D transposed convolutions (also known as deconvolutions) and concatenation with the corresponding feature maps from the encoding path.

### 3.3.2 Model Training

A 5 folder cross validation scheme was also implemented in the model and was used to assess its performance and generalisation.

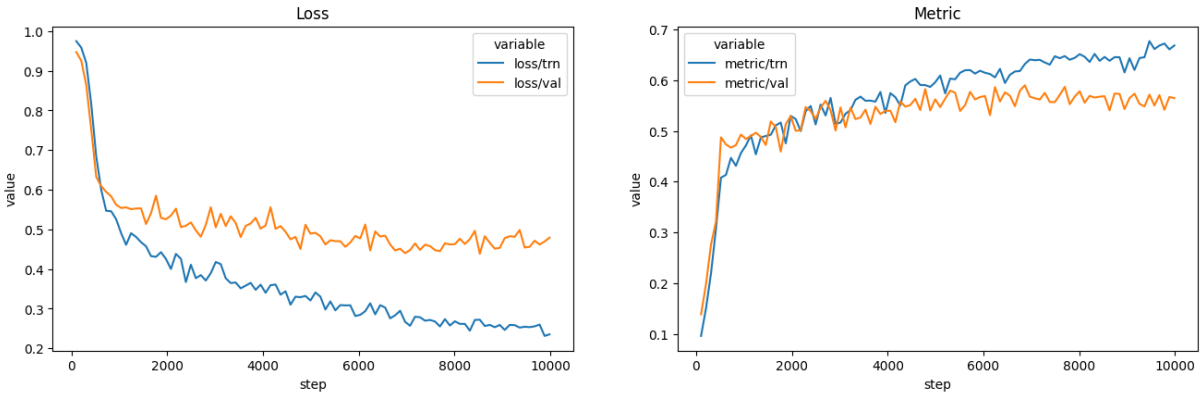
The chosen optimiser was the ADAM optimiser [46], with a constant learning rate of 1e-5. Different to SGD, Adam maintains a learning rate for each parameter and calculates first and second moments of the gradients, combining the benefits of AdaGrad [47] and RMSProp. The algorithm updates the parameters using the moments and the calculated gradients. This adapts learning rates for different parameters, handles sparse gradients, and converges quicker than other optimiser in general, making it a suitable optimiser in most cases. Additionally L2 regularization [48], also known as weight decay, was also applied to prevent overfitting with $\lambda = 1e{-}5$.

The loss function used the training pipeline is the Generalized Dice Loss (GDL)[49] defined in equation 2. Introduced originally as an evaluation metric for multiple class segmentation tasks, it was later reconverted to a loss function able to capture the similarity between predicted and ground truth segmentation by considering each class contributions and providing a single score. Although design for multi-class problems, it can be adapted to binary problems by separating foreground and background in different channels. This loss addresses the issue of class imbalance by giving more weight to underrepresented classes. This is done by assigning a weight to each class based on its inverse frequency, giving more importance to the underrepresented classes during training.

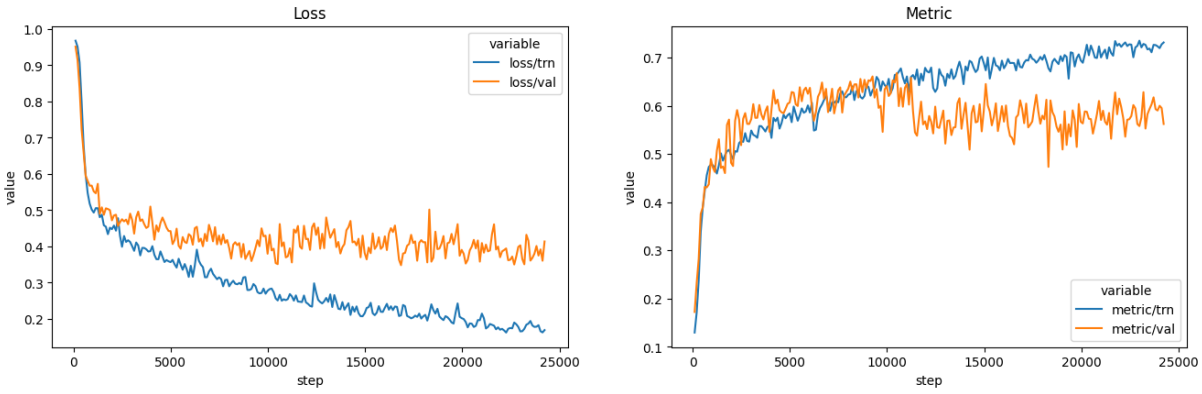
$$GDL = 1 - 2\frac{\sum_{l=1}^{2} w_l \sum_n r_{ln} p_{ln}}{\sum_{l=1}^{2} w_l \sum_n r_{ln} + p_{ln}} \tag{2}$$

with $r_{ln}$ the ground truth label, $p_{ln}$ the predicted label and $w_l$ the weight assigned to each label defined as $w_l = 1/(\sum_{n=1}^{N} r_{ln})^2$

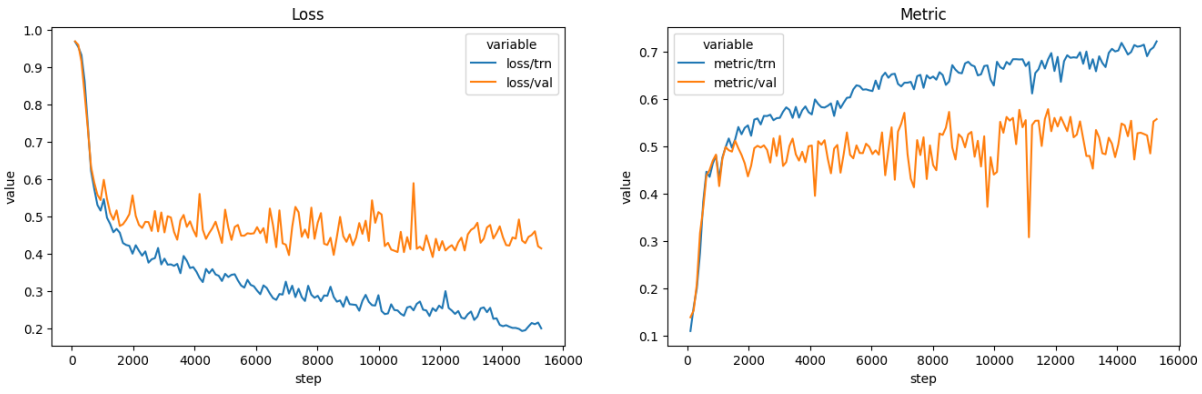
The model showed signs of overfitting despite the use of L2 regularization and dropout layers in all of its training/validation configurations (see figure 11), however configuration number 2 showed the best results and hence was selected as the final model.



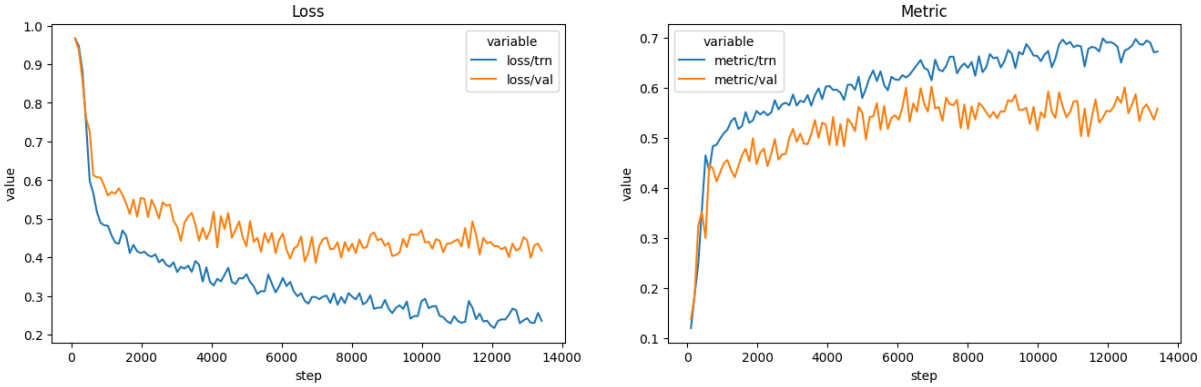

(a) Configuration 1



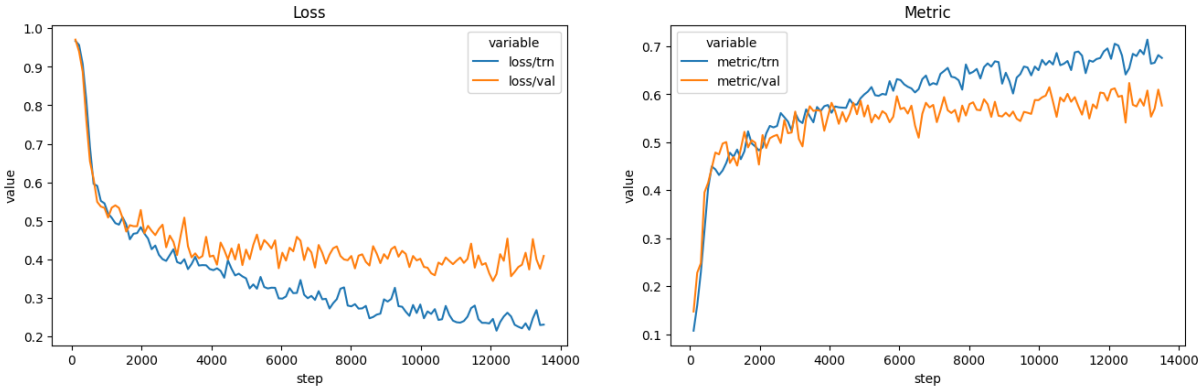

(b) Configuration 2




(c) Configuration 3




(d) Configuration 4




(e) Configuration 5

Figure 11: Training and Validation results for the 5 cross validation configurations

### 3.3.3 Evaluation Metrics

There exist several metrics to evaluate the accuracy of a segmentation, however for this task we will be using the following:

- **Dice Similarity Coefficient (DSC)**: See 3.2.3.
- **Modified Hausdorff Distance (MHD)**: The Hausdorff Distance is a metric that quantifies the dissimilarity between two sets of points, by measuring the maximum distance between a point in one set and its closest point in the other set, considering both the forward and reverse directions. The MHD, defined by equation 3, modifies the standard method to address the sensitivity of the original Hausdorff Distance to outliers [50]. In the context of segmentation evaluation, MHD provides a measure of the shape dissimilarity between the predicted and ground truth segmentations. A lower MHD value indicates a better alignment between the segmentations and represents a higher level of similarity.

$$d_H(X,Y) = \max\left\{\frac{1}{N_x}\sum_{x\in X} d(x,Y), \frac{1}{N_y}\sum_{y\in Y} d(y,X)\right\} \tag{3}$$

with $d$ the Euclidean distance, $X$ and $Y$ defined as the two sets being compared, in this case the prediction and the ground truth and $N_i$ as the number of points in the set.

- **Volumetric Similarity (VS)**: The volumetric similarity is a metric that considers the volumes of the segments to determine similarity. VS is defined by equation 4, with a higher value indicating better segmentation accuracy. It is important to note that Volumetric Similarity (VS), despite being defined using four cardinalities, is not classified as an overlap-based metric. Unlike overlap-based metrics, it compares the absolute volumes of the segmented regions between two segmentations, without considering the actual overlap between the segments. Therefore, its assessment focuses solely on the comparison of the volumes in each segmentation, disregarding the degree of overlap between them.

$$VS = 1 - \frac{|FN - FP|}{2TP + FP + FN} \tag{4}$$

with $FN$ being false negatives, $FP$ being false positives and $TP$ being true positives.

These three segmentation metrics provide different perspectives on the quality of segmentations. DSC assesses the voxel-level overlap, MHD quantifies shape dissimilarity, and VS evaluates volumetric agreement. When used together, these metrics offer a comprehensive evaluation of the segmentation performance, considering both the spatial alignment and volumetric correspondence between the predicted and ground truth segmentations.

### 3.3.4 Hardware

The whole algorithm was coded in python using the PyTorch framework [43] and trained on a local device with a 4GB NVIDIA GeForce GTX 1650 with Max-Q Design GPU, 32 GB of RAM and an 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00 GHz processor. All the generated code is in annex A and in the following GitHub repository: `https://github.com/ariajc/Intracranial-aneurysm-segmentation`.

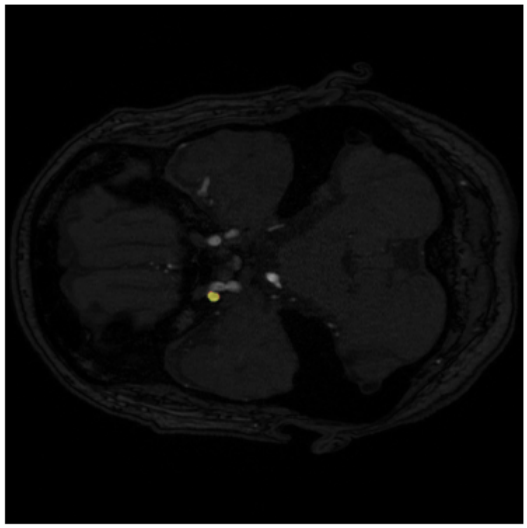
# 4 Results

The model underwent testing on a cohort comprising 28 patients, encompassing a total of 39 aneurysms. A summary of the statistical description for this cohort is presented in Table 3.

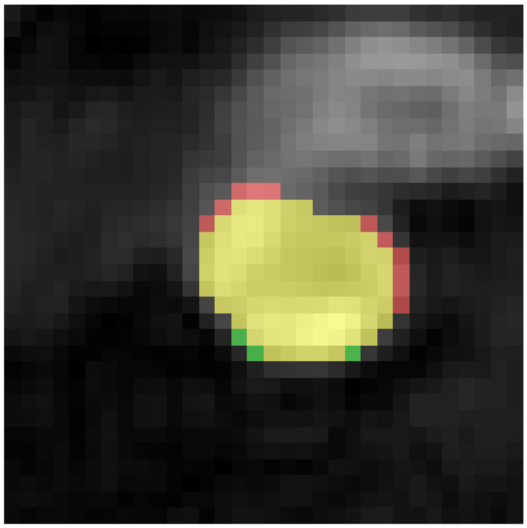
| | | **Count** | % |
|---|---|---|---|
| **Location** | ICA | 13 | 33.3 (13/39) |
| | MCA | 18 | 46.1 (18/39) |
| | ACA/Pcom/Posterior | 7 | 17.9 (7/39) |
| | B | 1 | 2.7 (1/39) |
| **Size** | $d \leq 7mm$ | 30 | 76.9 (30/39) |
| | $7 - 9.9mm$ | 6 | 15.4 (6/39) |
| | $10 - 19.9mm$ | 2 | 5.1 (2/39) |
| | $d \geq 20mm$ | 1 | 2.6 (1/39) |

Table 3: Statistical description of the test datset. ICA: Internal Carotid Artery; MCD: Middle Cerebral Artery; ACA: Anterior Cerebral Artery; B: Basilar Artery

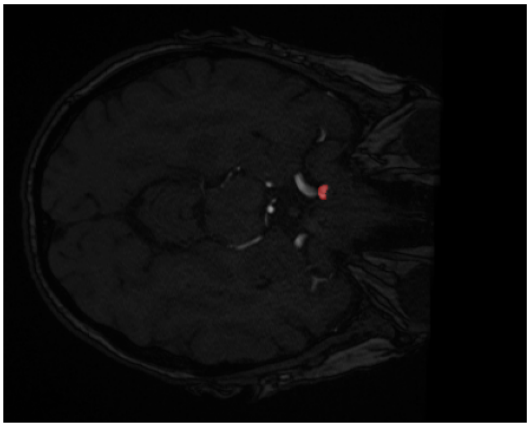
Figure 12 displays the segmentation results for three arbitrarily selected 2D slices from three different cases. In the figure, yellow represents true positives, red corresponds to false negatives, and green denotes false positives. The evaluated model demonstrates a sensitivity of 50% and an average of 0.43 false positives per case. These findings indicate a relatively lower ability to accurately detect positive instances. However, the model showcases a noteworthy characteristic of maintaining a low false positive rate, highlighting a high level of specificity. This outcome suggests that the model places emphasis on minimizing false positive errors, reducing the risk of incorrectly labeling negative instances as positive.



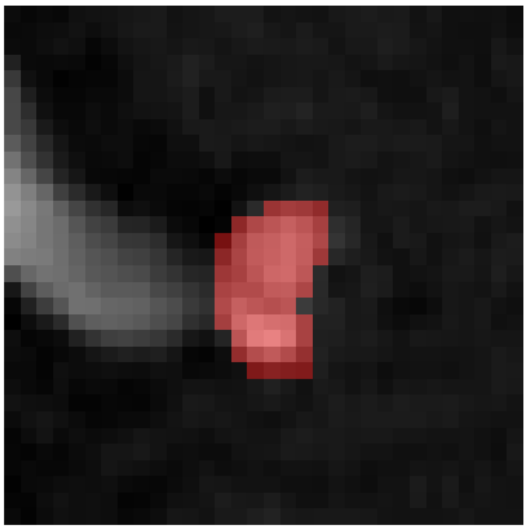
(a) Results for case 14



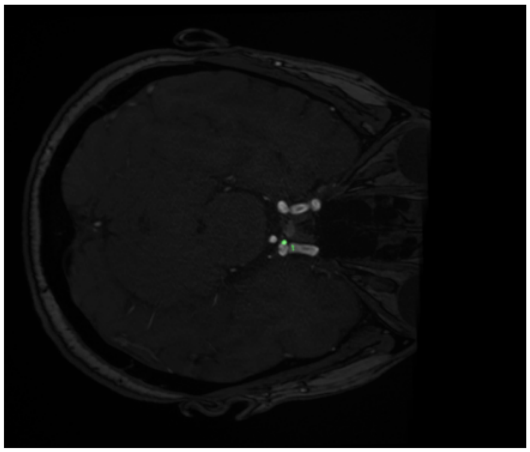
(b) Results for case 14 cropped



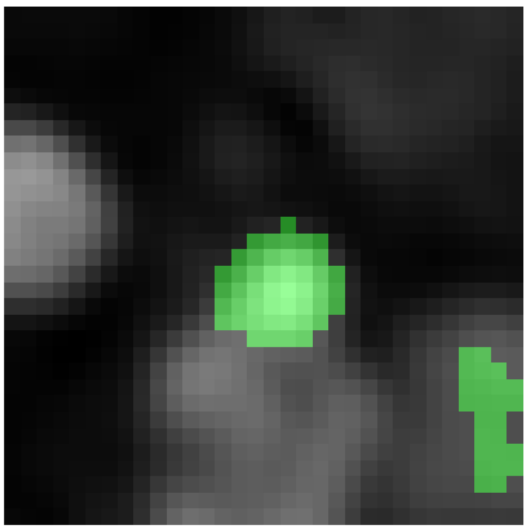
(c) Results for case 5



(d) Results for case 5 cropped



(e) Results for case 10



(f) Results for case 10 cropped

Figure 12: Results for arbitrary slices for cases 5,10,14

To establish a benchmark for the model's performance, the results obtained from the ADAM segmentation challenge were used (available at `https://adam.isi.uu.nl/results/results-miccai-2020/`). Table 4 presents the average metrics for each team participating in the challenge, including the algorithm developed in this study. The 95% confidence intervals, determined using a bootstrapping algorithm, are enclosed in brackets. The results indicate that the model achieved the third-best Dice Similarity Coefficient (DSC), the fourth-best Modified Hausdorff Distance (MDH), and the third-best Volumetric Similarity (VS) scores, positioning it as the third-best overall model. However, it should be noted that the model developed in this work was trained using a larger cohort of cases. Furthermore, the interobserver metrics, which involve comparing manual segmentations from two different observers on a subset of the scans, underscore the challenging nature of intracranial aneurysm segmentation even for human experts. These findings emphasize the considerable variability in segmentation outcomes, influenced by individual practitioners' criteria and expertise.

| **Team** | **DSC** | **MHD (mm)** | **VS** |
|---|---|---|---|
| **junma** | **0.41 (0.35 - 0.47)** | 8.96 (5.59 - 12.71) | **0.5 (0.43 - 0.56)** |
| **joker** | 0.40 (0.34 - 0.46) | **8.67 (5.35 - 12.32)** | 0.48 (0.42 - 0.54) |
| ***This model*** | 0.28 (0.15 - 0.40) | 16.37 (9.47 - 24.10) | 0.43 (0.29 - 0.58) |
| **kubiac** | 0.28 (0.23 - 0.33) | 18.13 (12.73 - 24.07) | 0.39 (0.33 - 0.45) |
| **inteneural** | 0.17 (0.13 - 0.21) | 23.98 (19.65 - 28.04) | 0.36 (0.30 - 0.41) |
| **xlim** | 0.21 (0.18 - 0.25) | 36.82 (32.72 - 41.3) | 0.39 (0.34 - 0.44) |
| **zelosmediacorp** | 0.09 (0.06 - 0.13) | 9.79 (4.66 - 15.5) | 0.13 (0.09 - 0.18) |
| **stronge** | 0.07 (0.04 - 0.11) | 24.42 (18.72 - 30.36) | 0.21 (0.15 - 0.28) |
| **IBBM** | 0.01 (0 - 0.02) | 12.77 (0.97 - 25.81) | 0.01 (0 - 0.03) |
| **TUM IBBM** | 0.07 (0.05 - 0.1) | 65.02 (60.93 - 69.24) | 0.31 (0.26 - 0.36) |
| **Interobserver** | **0.63 (0.60 - 0.67)** | **2.42 (1.56 - 3.48)** | **0.76 (0.73 - 0.79)** |

Table 4: Average metrics and ranking for each team in the ADAM challenge as well as the model developed in this work. DSC: Dice Similarity Coefficient; Modified Hausdorff Distance: MHD; VS: Volumetric Similarity.

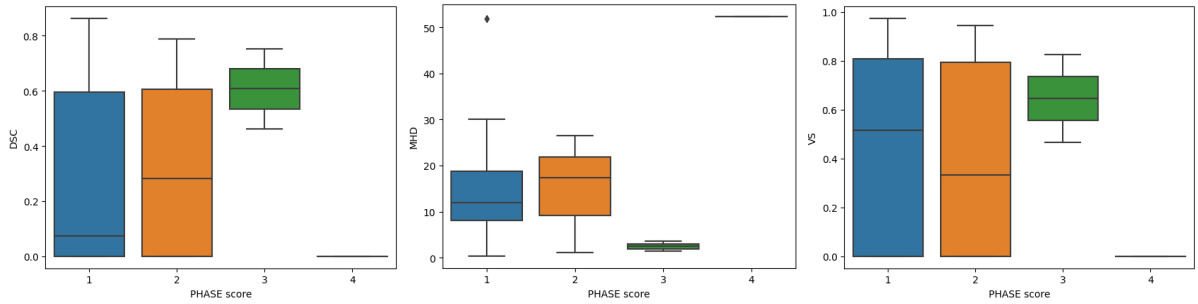
Table 5 presents the average metrics specifically for cases in which the aneurysm was detected. This analysis focuses on instances where the presence of an aneurysm was correctly identified by the model.

In the final analysis, the prediction results were examined with respect to two variables:

| **Team** | **DSC** | **MHD (mm)** | **VS** |
|---|---|---|---|
| ***This model*** | 0.55 (0.41 - 0.67) | 14.03 (6.87 - 22.51) | 0.70 (0.57 - 0.81) |

Table 5: Average metrics and ranking for each team in the ADAM challenge as well as the model developed in this work when only considering identified aneurysms. DSC: Dice Similarity Coefficient; Modified Hausdorff Distance: MHD; VS: Volumetric Similarity.

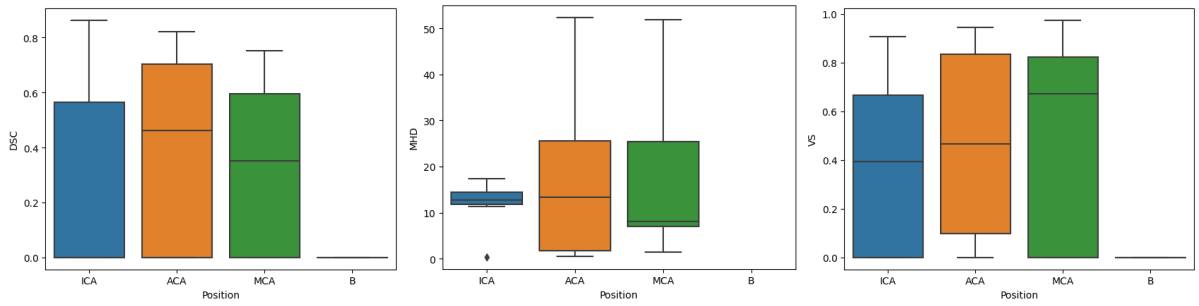
aneurysm position and size, which were determined using the PHASE score [5]. The obtained results are presented in figures 13 and 14. Box-plot representations presented in figure 13 suggest that the model achieves optimal performance when segmenting aneurysms with sizes ranging from 10 to 19.9 mm in diameter. For aneurysms measuring between 0-7 mm and 7-10 mm, the model demonstrates comparable performance, while it struggles to accurately segment aneurysms larger than 20 mm in diameter. However, statistical analysis using an ANOVA test indicates no significant difference in accuracy across different aneurysm sizes for any of the three metrics examined (DSC: p-value = 0.63, MDH: p-value = 0.75674, VS: p-value = 0.55492). These results suggest that the accuracy of the model remains consistent regardless of the size of the aneurysm, emphasizing its ability to effectively segment aneurysms across a broad range of sizes.




(a) DSC distribution according to PHASE score (b) MHD distribution according to PHASE score (c) VS distribution according to PHASE score

Figure 13: Different metric distibution according to PHASE score [5]

Similarly results showed in figure 14 demonstrate that the model consistently performs well across different locations of the aneurysm, as indicated by the three metrics used for assessment. Statistical analysis using an ANOVA test reveals no significant difference in accuracy for different aneurysm locations across all three metrics (DSC: p-value = 0.714457, MDH: p-value = 0.281292, VS: p-value = 0.626736). These findings suggest that the model's accuracy is not influenced by the specific location of the aneurysm, indicating its robustness and ability to accurately segment aneurysms regardless of their position within the intracranial region.




(a) DSC distribution according to position

(b) MHD distribution according to position

(c) VS distribution according to position

Figure 14: Different metric distribution according to position. ICA: Internal Carotid Artery; MCD: Middle Cerebral Artery; ACA: Anterior Cerebral Artery; B: Basilar Artery

All the code generated during evaluation is in the following GITHUB repository: `https://github.com/ariajc/Intracranial-aneurysm-segmentation`.

# 5 Conclusion

In conclusion, this study focused on developing a fully automatic algorithm for unruptured intracranial aneurysm segmentation. A two step strategy was used, where the problem was separated in 2 tasks, localisation and segmentation, with separate deep learning models being trained for each task and later ensembled. While the model may not have achieved the highest performance compared to the state-of-the-art models, it still provided results in line with the current advancements in the field.

However, despite the progress made in the field and the promising performance of deep learning models, there is still a noticeable gap between the performance of algorithms and manual segmentation. Manual segmentation, although time-consuming and labor-intensive, is often considered the gold standard for accurate aneurysm segmentation. It requires the expertise and knowledge of experienced radiologists or clinicians, and it allows for fine-grained anatomical details to be captured.

Further research and development are necessary to bridge this gap and improve the performance of automated segmentation algorithms. This includes exploring novel deep learning architectures, incorporating more diverse and extensive training data, and refining the model's ability to handle challenging cases and anatomical variations.

While deep learning models for intracranial aneurysm segmentation may not yet rival the accuracy of experienced radiologists, they offer distinct advantages in terms of speed and scalability that can compensate for their potential shortcomings.

Deep learning models have demonstrated the capability to process large volumes of medical imaging data rapidly, providing efficient and automated segmentation results. This accelerated processing time can significantly reduce the turnaround time for diagnosis and treatment planning, leading to improved patient outcomes.

Furthermore, deep learning models have the potential to handle higher workloads compared to individual radiologists. With the increasing demand for medical imaging analysis, the ability of these models to analyze a large number of scans consistently and objectively can enhance the efficiency of healthcare systems. By leveraging their efficiency and scalability, this models can alleviate the burden on radiologists, contributing to improved healthcare delivery, enabling

timely diagnoses and facilitating the processing of large-scale medical imaging data.

In conclusion, while the work presented in this document may not have surpassed the state-of-the-art performance, it represents a valuable step towards the automation of this critical task. The ongoing advancements in the field and the collaboration between medical professionals and researchers will contribute to the continued improvement of automated segmentation algorithms, ultimately enhancing clinical decision-making and patient care in the context of intracranial aneurysms and other clinical areas.

## References


[1] Mayo Clinic. *The Hidden Dangers of Brain Aneurysm*. 2022. URL: https://www.mayoclinic.org/es-es/hidden-dangers-brain-aneurysm-infographic/ifg-20404403.

[2] Mayo Clinic. *Hemorragia subaracnoidea*. 2022. URL: https://www.mayoclinic.org/es-es/diseases-conditions/subarachnoid-hemorrhage/symptoms-causes/syc-20361009.

[3] Keedy A. "An overview of intracranial aneurysms". In: *Mcgill J Med* 9.2 (2006), pp. 141–146.

[4] Backes D. "ELAPSS score for prediction of risk of growth of unruptured intracranial aneurysms". In: *Neurology* 88.17 (2017), pp. 1600–1606.

[5] Jacoba P Greving et al. "Development of the PHASES score for prediction of risk of rupture of intracranial aneurysms: a pooled analysis of six prospective cohort studies". In: *The Lancet Neurology* 13.1 (2014), pp. 59–66. ISSN: 1474-4422. DOI: https://doi.org/10.1016/S1474-4422(13)70263-1. URL: https://www.sciencedirect.com/science/article/pii/S1474442213702631.

[6] Cipolla MJ. *The Cerebral Circulation*. San Rafael (CA): Morgan Claypool Life Sciences, 2009.

[7] STEHBENS WE. "ANEURYSMS AND ANATOMICAL VARIATION OF CEREBRAL ARTERIES". In: *Arch Pathol* 75 (1963), pp. 45–64.

[8] Hazama F. Kayembe KN Sasahara M. "Cerebral aneurysms and variations in the circle of Willis". In: *Stroke* 15.5 (1984), pp. 846–50. DOI: 10.1161/01.str.15.5.846.

[9] Mediline. *Circle of Willis*. Accessed: 2023-06-04. 2023. URL: %5Curl%7Bhttps://medlineplus.gov/ency/imagepages/18009.htm%7D.

[10] NIH. *Cerebral Aneurysms*. Accessed: 2023-06-04. 2023. URL: %5Curl%7Bhttps://www.ninds.nih.gov/health-information/disorders/cerebral-aneurysms%7D.

[11] *Baylor Medicine Brain Aneurysms*. https://www.bcm.edu/healthcare/specialties/neurosurgery/cerebrovascular-and-stroke-surgery/brain-aneurysms. Accessed: 2023-06-03.

[12] Newell DW Brisman JL Song JK. "Cerebral aneurysms". In: *N Engl J Med* 355.9 (2006), pp. 928–39. DOI: 10.1056/NEJMra052760.

[13] Adel Malek. "Hemodynamic Shear Stress and Its Role in Atherosclerosis". In: *JAMA* 282 (Dec. 1999), p. 2035. DOI: 10.1001/jama.282.21.2035.

[14] Edelman ER Turjman AS Turjman F. "Role of fluid dynamics and inflammation in intracranial aneurysm formation". In: *Circulation* 129(3) (2014), pp. 373–82. DOI: 10.1161/CIRCULATIONAHA.113.001444.

[15] Daan Backes et al. "PHASES Score for Prediction of Intracranial Aneurysm Growth". In: *Stroke* 46.5 (2015), pp. 1221–1226. DOI: 10.1161/STROKEAHA.114.008198. eprint: https://www.ahajournals.org/doi/pdf/10.1161/STROKEAHA.114.008198. URL: https://www.ahajournals.org/doi/abs/10.1161/STROKEAHA.114.008198.

[16] White PM Wardlaw JM. "The detection and management of unruptured intracranial aneurysms". In: *Brain* 123.2 (2000), pp. 205–221. DOI: 10.1093/brain/123.2.205..

[17] Annah Lane, Philip Vivian, and Alan Coulthard. "Magnetic resonance angiography or digital subtraction catheter angiography for follow-up of coiled aneurysms: Do we need both?" In: *Journal of Medical Imaging and Radiation Oncology* 59.2 (2015), pp. 163–169. DOI: https://doi.org/10.1111/1754-9485.12288. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1754-9485.12288. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1754-9485.12288.

[18] S.D. Hobbs et al. "LDL Cholesterol is Associated with Small Abdominal Aortic Aneurysms". In: *European Journal of Vascular and Endovascular Surgery* 26.6 (2003), pp. 618–622. ISSN: 1078-5884. DOI: https://doi.org/10.1016/S1078-5884(03)00412-X. URL: https://www.sciencedirect.com/science/article/pii/S107858840300412X.

[19] et Al Belavadi R. "Surgical Clipping Versus Endovascular Coiling in the Management of Intracranial Aneurysms". In: *Cureus* 13.12 (2021). DOI: 10.7759/cureus.20478.

[20] et Al Briganti F. "Endovascular treatment of cerebral aneurysms using flow-diverter devices: A systematic review". In: *Neuroradiol J* 28.4 (2015). DOI: 10.1177/1971400915602803.

[21] Pierot L et Al. "WEB Treatment of Intracranial Aneurysms: Feasibility, Complications, and 1-Month Safety Results with the WEB DL and WEB SL/SLS in the French Observatory." In: *AJNR Am J Neuroradiol* 35.5 (2015), pp. 922–7. DOI: 10.3174/ajnr.A4230.

[22] Rubén Cardenes et al. "Automatic Aneurysm Neck Detection Using Surface Voronoi Diagrams". In: *IEEE Transactions on Medical Imaging* 30.10 (2011), pp. 1863–1876. DOI: 10.1109/TMI.2011.2157698.

[23] Clemens M. Hentschke et al. "Automatic cerebral aneurysm detection in multimodal angiographic images". In: *2011 IEEE Nuclear Science Symposium Conference Record*. 2011, pp. 3116–3120. DOI: 10.1109/NSSMIC.2011.6152566.

[24] Žiga Bizjak et al. "Modality agnostic intracranial aneurysm detection through supervised vascular surface classification". In: *Medical Imaging 2021: Computer-Aided Diagnosis*. Ed. by Maciej A. Mazurowski and Karen Drukker. Vol. 11597. International Society for Optics and Photonics. SPIE, 2021, 115970O. DOI: 10.1117/12.2580868. URL: https://doi.org/10.1117/12.2580868.

[25] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *CoRR* abs/1505.04597 (2015). arXiv: 1505.04597. URL: http://arxiv.org/abs/1505.04597.

[26] Fabian Isensee et al. "nnU-Net: a self-configuring method for deep learning-based biomedical image segmentation". In: *Nature methods* 18.2 (Feb. 2021), pp. 203–211. ISSN: 1548-7091. DOI: 10.1038/s41592-020-01008-z. URL: https://doi.org/10.1038/s41592-020-01008-z.

[27] Kimberley M. Timmins et al. "Comparing methods of detecting and segmenting unruptured intracranial aneurysms on TOF-MRAS: The ADAM challenge". In: *NeuroImage* 238 (2021), p. 118216. ISSN: 1053-8119. DOI: https://doi.org/10.1016/j.neuroimage.2021.118216. URL: https://www.sciencedirect.com/science/article/pii/S1053811921004936.

[28] Leonard Berrada, Andrew Zisserman, and M Pawan Kumar. "Smooth loss functions for deep top-k classification". In: *arXiv preprint arXiv:1802.07595* (2018).

[29] Jun Ma et al. "Loss odyssey in medical image segmentation". In: *Medical Image Analysis* 71 (2021), p. 102035.

[30] Guangyu Zhu et al. "Deep learning-based recognition and segmentation of intracranial aneurysms under small sample size". In: *Frontiers in Physiology* 13 (2022). ISSN: 1664-042X. DOI: 10.3389/fphys.2022.1084202. URL: https://www.frontiersin.org/articles/10.3389/fphys.2022.1084202.

[31] Özgün Çiçek et al. "3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation". In: *CoRR* abs/1606.06650 (2016). arXiv: 1606.06650. URL: http://arxiv.org/abs/1606.06650.

[32] Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. "V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation". In: *CoRR* abs/1606.04797 (2016). arXiv: 1606.04797. URL: http://arxiv.org/abs/1606.04797.

[33] Rahil Shahzad et al. "Fully automated detection and segmentation of intracranial aneurysms in subarachnoid hemorrhage on CTA using deep learning". In: *Scientific Reports* 10 (Dec. 2020). DOI: 10.1038/s41598-020-78384-1.

[34] Konstantinos Kamnitsas et al. "Efficient Multi-Scale 3D CNN with Fully Connected CRF for Accurate Brain Lesion Segmentation". In: *CoRR* abs/1603.05959 (2016). arXiv: 1603.05959. URL: http://arxiv.org/abs/1603.05959.

[35] Stefan Klein et al. "elastix: A Toolbox for Intensity-Based Medical Image Registration." In: *IEEE Trans. Med. Imaging* 29.1 (2010), pp. 196–205. URL: http://dblp.uni-trier.de/db/journals/tmi/tmi29.html#KleinSMVP10.

[36] Tommaso Di Noto et al. *"Lausanne$_T$OF − MRA$_A$neurysm$_C$ohort"*. OpenNeuro, 2022. DOI: doi:10.18112/openneuro.ds003949.v1.0.1.

[37] Tourbier S Di Noto T Marie G. "Towards Automated Brain Aneurysm Detection in TOF-MRA: Open Data, Weak Labels, and Anatomical Knowledge." In: *Neuroinformatics* 21.1 (2023), pp. 21–34. DOI: 10.1007/s12021-022-09597-0.

[38] K. Gorgolewski. "The brain imaging data structure, a format for organizing and describing outputs of neuroimagin experiments". In: *Scientific Data* 3 (2016). DOI: https://doi.org/10.1038/sdata.2016.44.

[39] Jacoba P Greving et al. "Development of the PHASES score for prediction of risk of rupture of intracranial aneurysms: a pooled analysis of six prospective cohort studies". In: *The Lancet Neurology* 13.1 (2014), pp. 59–66.

[40] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, 17–19 Jun 2013, pp. 1139–1147. URL: https://proceedings.mlr.press/v28/sutskever13.html.

[41] T. Sørensen. "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". In: *Kongelige Danske Videnskabernes Selskab* 5.4 (1948), pp. 1–34.

[42] Lee R. Dice. "Measures of the Amount of Ecologic Association Between Species". In: *Ecology* 26.3 (1945), pp. 297–302. DOI: 10.2307/1932409.

[43] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[44] Ekaba Bisong. "Google Colaboratory". In: *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*. Berkeley, CA: Apress, 2019, pp. 59–64. ISBN: 978-1-4842-4470-8. DOI: 10.1007/978-1-4842-4470-8_7. URL: https://doi.org/10.1007/978-1-4842-4470-8_7.

[45] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[46] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations* (Dec. 2014).

[47] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12 (July 2011), pp. 2121–2159.

[48] Corinna Cortes, Mehryar Mohri, and Afshin Rostamizadeh. "L2 Regularization for Learning Kernels". In: *CoRR* abs/1205.2653 (2012). arXiv: 1205.2653. URL: http://arxiv.org/abs/1205.2653.

[49] Carole H. Sudre et al. "Generalised Dice overlap as a deep learning loss function for highly unbalanced segmentations". In: *CoRR* abs/1707.03237 (2017). arXiv: 1707.03237. URL: http://arxiv.org/abs/1707.03237.

[50] M.-P. Dubuisson and A.K. Jain. "A modified Hausdorff distance for object matching". In: *Proceedings of 12th International Conference on Pattern Recognition*. Vol. 1. 1994, 566–568 vol.1. DOI: 10.1109/ICPR.1994.576361.

[51] Adrian Wolny et al. "Accurate and versatile 3D segmentation of plant tissues at cellular resolution". In: *eLife* 9 (July 2020). Ed. by Christian S Hardtke et al., e57613. ISSN: 2050-084X. DOI: 10.7554/eLife.57613. URL: https://doi.org/10.7554/eLife.57613.

# Appendices

## A Segmentation model code

A part of the following code was based on the work presented in [51].

### A.1 Dataset Creation

```
import csv
import functools
import glob
import os
import random

from collections import namedtuple


import numpy as np

import nibabel as nib
import torch
import torch.cuda
from torch.utils.data import Dataset


from util.disk import getCache
from util.util import IrcTuple


from util.logconf import logging

log = logging.getLogger(__name__)
log.setLevel(logging.DEBUG)

raw_cache = getCache('code_raw')


CandidateInfoTuple = namedtuple(
    'CandidateInfoTuple',
    'diameter_mm, series_name, center_xyz'
)
def xyz2zxy(X):
    return np.transpose(X, (2, 0, 1))
```

```
@functools.lru_cache(1)
def getCandidateInfoList():
    path='C:\\Users\\Usuari\\Desktop\\TFM\\DATA'

    candidateInfo_list = []
    for file in os.listdir(path):
        for files in os.listdir(f"{path}\{file}"):
            if files.endswith("location.txt"):
                if os.path.getsize(f"{path}\{file}\{files}") != 0:
                    series_name = file
                    with open(f"{path}\{file}\{files}", "r") as f:
                        for row in list(csv.reader(f)):
                            candidateCenter_xyz = tuple([float(x) for x in row
    [0:3]])
                            candidateDiameter_mm = float(row[3])

                            candidateInfo_list.append(CandidateInfoTuple(
                                candidateDiameter_mm,
                                series_name,
                                candidateCenter_xyz,
                            ))

    candidateInfo_list.sort(reverse=True)
    return candidateInfo_list

@functools.lru_cache(1)
def getCandidateInfoDict():
    candidateInfo_list = getCandidateInfoList()
    candidateInfo_dict = {}

    for candidateInfo_tup in candidateInfo_list:
        candidateInfo_dict.setdefault(candidateInfo_tup.series_name,
                                      []).append(candidateInfo_tup)

    return candidateInfo_dict

class MRA:
    def __init__(self, series_name):
        nii_path = glob.glob(
            'C:\\Users\\Usuari\\Desktop\\TFM\\DATA\\{}\\TOF.nii.gz'.format(
    series_name)
        )
        mra_nii = nib.load(nii_path[0])
```

```
77          mra_a = np.array(mra_nii.get_fdata(), dtype=np.float32)
78          mra_a=xyz2zxy(mra_a)
79
80
81          self.series_name = series_name
82          self.a = mra_a
83
84          #get scpacing in each direction
85          self.vxSize_xyz = IrcTuple(*mra_nii.header.get_zooms())
86
87          self.positive_mask = self.buildAnnotationMask() ## returns anotation
    mask
88
89      def buildAnnotationMask(self):
90          nii_mask_path = glob.glob(
91              'C:\\Users\\Usuari\\Desktop\\TFM\\DATA\\{}\\aneurysms.nii.gz'.format
    (self.series_name)
92          )
93          mask_nii = nib.load(nii_mask_path[0])
94          mask_a = np.array(mask_nii.get_fdata(), dtype=np.int_)
95          mask_a=xyz2zxy(mask_a)
96
97          return mask_a
98
99      def getRawCandidate(self, center_xyz, width_xyz):
100
101         slice_list = []
102         for axis, center_val in enumerate(center_xyz):
103             start_ndx = int(round(center_val - width_xyz[axis]/2))
104             end_ndx = int(start_ndx + width_xyz[axis])
105
106             if start_ndx < 0:
107                 start_ndx = 0
108                 end_ndx = int(width_xyz[axis])
109
110             if end_ndx > self.a.shape[axis]:
111                 end_ndx = self.a.shape[axis]
112                 start_ndx = int(self.a.shape[axis] - width_xyz[axis])
113
114             slice_list.append(slice(start_ndx, end_ndx))
115
116         pos_chunk = self.positive_mask[tuple(slice_list)]
117         mra_chunk = self.a[tuple(slice_list)]
```

```
118
119         return mra_chunk,pos_chunk
120
121
122
123 @functools.lru_cache(1, typed=True)
124 def getMRA(series_name):
125     return MRA(series_name)
126
127 @raw_cache.memoize(typed=True)
128 def getMRARawCandidate(series_name, center_xyz, width_xyz):
129     mra = getMRA(series_name)
130     mra_chunk, pos_chunk= mra.getRawCandidate(center_xyz,width_xyz)
131     return mra_chunk, pos_chunk
132
133 class ADAM2dSegmentationDataset(Dataset):
134     def __init__(self,
135                  isValSet_bool=None,
136                  config=0
137             ):
138         self.isValSet_bool = isValSet_bool
139
140         self.series_list = sorted(getCandidateInfoList(),reverse=True)
141
142         self.config = config
143         self.lists = self.dividir_lista()
144
145         if isValSet_bool:
146             self.series_list = self.lists[self.config]
147             assert self.series_list
148         else:
149             self.series_list = [e for ls in self.lists for e in ls if ls != self
    .lists[self.config]]
150             assert self.series_list
151
152
153         log.info("{!r}: {} {} series".format(
154             self,
155             len(self.series_list),
156             {None: 'general', True: 'validation', False: 'training'}[
    isValSet_bool]
157         ))
158
```

```
159     def dividir_lista(self):
160         k_1 = []
161         k_2 = []
162         k_3 = []
163         k_4 = []
164         k_5 = []
165 
166         for i, elemento in enumerate(self.series_list, start=1):
167             if i % 5 == 1:
168                 k_1.append(elemento)
169             elif i % 5 == 2:
170                 k_2.append(elemento)
171             elif i % 5 == 3:
172                 k_3.append(elemento)
173             elif i % 5 == 4:
174                 k_4.append(elemento)
175             elif i % 5 == 0:
176                 k_5.append(elemento)
177 
178         return k_1,k_2,k_3,k_4,k_5
179 
180     def shuffleSamples(self):
181         random.shuffle(self.series_list)
182 
183     def __len__(self):
184         return len(self.series_list)
185 
186     def __getitem__(self, ndx):
187         candidateInfo_tup = self.series_list[ndx % len(self.series_list)]
188         return self.getitem_trainingCrop(candidateInfo_tup)
189 
190     def getitem_trainingCrop(self, candidateInfo_tup):
191 
192         center_zxy = candidateInfo_tup.center_xyz[-1:] + candidateInfo_tup.
    center_xyz[:-1]
193         mra_a, pos_a= getMRARawCandidate(
194             candidateInfo_tup.series_name,
195             center_zxy,
196             (16, 64, 64)
197         )
198 
199         mra_t = torch.from_numpy(mra_a).to(torch.float32)
200         pos_t = torch.from_numpy(pos_a).to(torch.long)
```

```
201
202         return mra_t.unsqueeze(0), pos_t.unsqueeze(0)
203
204
205
206 class PrepcacheADAMDataset(Dataset):
207     def __init__(self, *args, **kwargs):
208         super().__init__(*args, **kwargs)
209
210         self.candidateInfo_list = getCandidateInfoList()
211
212         self.seen_set = set()
213         self.candidateInfo_list.sort(key=lambda x: x.series_name)
214
215     def __len__(self):
216         return len(self.candidateInfo_list)
217
218     def __getitem__(self, ndx):
219
220         candidateInfo_tup = self.candidateInfo_list[ndx]
221         center_zxy = candidateInfo_tup.center_xyz[-1:] + candidateInfo_tup.
    center_xyz[:-1]
222         getMRARawCandidate(candidateInfo_tup.series_name, center_zxy, (16, 64,
    64))
223
224         series_name = candidateInfo_tup.series_name
225         if series_name not in self.seen_set:
226             self.seen_set.add(series_name)
227
228         return 0, 1
```

Listing 1: Code to generate the datasets to tarin and validate the model

## A.2 Architecture

```
1 import torch
2 from torch import nn
3
4 import torch.nn.functional as F
5
6
7 import torch.nn as nn
8
9 from functools import partial
```

```
10
11 import importlib
12
13
14
15 def create_conv(in_channels, out_channels, kernel_size, order, num_groups,
       padding, is3d):
16     """
17     Create a list of modules with together constitute a single conv layer with
       non-linearity
18     and optional batchnorm/groupnorm.
19
20     Args:
21         in_channels (int): number of input channels
22         out_channels (int): number of output channels
23         kernel_size(int or tuple): size of the convolving kernel
24         order (string): order of things, e.g.
25             'cr' -> conv + ReLU
26             'gcr' -> groupnorm + conv + ReLU
27             'cl' -> conv + LeakyReLU
28             'ce' -> conv + ELU
29             'bcr' -> batchnorm + conv + ReLU
30         num_groups (int): number of groups for the GroupNorm
31         padding (int or tuple): add zero-padding added to all three sides of the
        input
32         is3d (bool): is3d (bool): if True use Conv3d, otherwise use Conv2d
33     Return:
34         list of tuple (name, module)
35     """
36     assert 'c' in order, "Conv layer MUST be present"
37     assert order[0] not in 'rle', 'Non-linearity cannot be the first operation
       in the layer'
38
39     modules = []
40     for i, char in enumerate(order):
41         if char == 'r':
42             modules.append(('ReLU', nn.ReLU(inplace=True)))
43         elif char == 'l':
44             modules.append(('LeakyReLU', nn.LeakyReLU(inplace=True)))
45         elif char == 'e':
46             modules.append(('ELU', nn.ELU(inplace=True)))
47         elif char == 'c':
48             # add learnable bias only in the absence of batchnorm/groupnorm
```

```
            bias = not ('g' in order or 'b' in order)
            if is3d:
                conv = nn.Conv3d(in_channels, out_channels, kernel_size, padding=padding, bias=bias)
            else:
                conv = nn.Conv2d(in_channels, out_channels, kernel_size, padding=padding, bias=bias)

            modules.append(('conv', conv))
        elif char == 'g':
            is_before_conv = i < order.index('c')
            if is_before_conv:
                num_channels = in_channels
            else:
                num_channels = out_channels

            # use only one group if the given number of groups is greater than the number of channels
            if num_channels < num_groups:
                num_groups = 1

            assert num_channels % num_groups == 0, f'Expected number of channels in input to be divisible by num_groups. num_channels={num_channels}, num_groups={num_groups}'
            modules.append(('groupnorm', nn.GroupNorm(num_groups=num_groups, num_channels=num_channels)))
        elif char == 'b':
            is_before_conv = i < order.index('c')
            if is3d:
                bn = nn.BatchNorm3d
            else:
                bn = nn.BatchNorm2d

            if is_before_conv:
                modules.append(('batchnorm', bn(in_channels)))
            else:
                modules.append(('batchnorm', bn(out_channels)))
        else:
            raise ValueError(f"Unsupported layer type '{char}'. MUST be one of ['b', 'g', 'r', 'l', 'e', 'c']")

    return modules

```

```
85
86  class SingleConv(nn.Sequential):
87      """
88      Basic convolutional module consisting of a Conv3d, non-linearity and
        optional batchnorm/groupnorm. The order
89      of operations can be specified via the `order` parameter
90
91      Args:
92          in_channels (int): number of input channels
93          out_channels (int): number of output channels
94          kernel_size (int or tuple): size of the convolving kernel
95          order (string): determines the order of layers, e.g.
96              'cr' -> conv + ReLU
97              'crg' -> conv + ReLU + groupnorm
98              'cl' -> conv + LeakyReLU
99              'ce' -> conv + ELU
100         num_groups (int): number of groups for the GroupNorm
101         padding (int or tuple): add zero-padding
102         is3d (bool): if True use Conv3d, otherwise use Conv2d
103     """
104
105     def __init__(self, in_channels, out_channels, kernel_size=3, order='gcr',
        num_groups=8, padding=1, is3d=True):
106         super(SingleConv, self).__init__()
107
108         for name, module in create_conv(in_channels, out_channels, kernel_size,
        order, num_groups, padding, is3d):
109             self.add_module(name, module)
110
111
112 class ResNetBlock(nn.Module):
113     """
114     Residual block that can be used instead of standard DoubleConv in the
        Encoder module.
115     Motivated by: https://arxiv.org/pdf/1706.00120.pdf
116
117     Notice we use ELU instead of ReLU (order='cge') and put non-linearity after
        the groupnorm.
118     """
119
120     def __init__(self, in_channels, out_channels, kernel_size=3, order='cge',
        num_groups=8, is3d=True, **kwargs):
121         super(ResNetBlock, self).__init__()
```

```
122
123         if in_channels != out_channels:
124             # conv1x1 for increasing the number of channels
125             if is3d:
126                 self.conv1 = nn.Conv3d(in_channels, out_channels, 1)
127             else:
128                 self.conv1 = nn.Conv2d(in_channels, out_channels, 1)
129         else:
130             self.conv1 = nn.Identity()
131
132         # residual block
133         self.conv2 = SingleConv(out_channels, out_channels, kernel_size=
    kernel_size, order=order, num_groups=num_groups,
134                                 is3d=is3d)
135         # remove non-linearity from the 3rd convolution since it's going to be
    applied after adding the residual
136         n_order = order
137         for c in 'rel':
138             n_order = n_order.replace(c, '')
139         self.conv3 = SingleConv(out_channels, out_channels, kernel_size=
    kernel_size, order=n_order,
140                                 num_groups=num_groups, is3d=is3d)
141
142         # create non-linearity separately
143         if 'l' in order:
144             self.non_linearity = nn.LeakyReLU(negative_slope=0.1, inplace=True)
145         elif 'e' in order:
146             self.non_linearity = nn.ELU(inplace=True)
147         else:
148             self.non_linearity = nn.ReLU(inplace=True)
149             # self.non_linearity = nn.Tanh()
150
151     def forward(self, x):
152         # apply first convolution to bring the number of channels to
    out_channels
153         residual = self.conv1(x)
154
155         # residual block
156         out = self.conv2(residual)
157         out = self.conv3(out)
158
159         out += residual
160         out = self.non_linearity(out)
```

```
161
162         return out
163
164
165
166 class Encoder(nn.Module):
167     """
168     A single module from the encoder path consisting of the optional max
169     pooling layer (one may specify the MaxPool kernel_size to be different
170     from the standard (2,2,2), e.g. if the volumetric data is anisotropic
171     (make sure to use complementary scale_factor in the decoder path) followed
        by
172     a basic module (DoubleConv or ResNetBlock).
173
174     Args:
175         in_channels (int): number of input channels
176         out_channels (int): number of output channels
177         conv_kernel_size (int or tuple): size of the convolving kernel
178         apply_pooling (bool): if True use MaxPool3d before DoubleConv
179         pool_kernel_size (int or tuple): the size of the window
180         pool_type (str): pooling layer: 'max' or 'avg'
181         basic_module(nn.Module): either ResNetBlock or DoubleConv
182         conv_layer_order (string): determines the order of layers
183             in `DoubleConv` module. See `DoubleConv` for more info.
184         num_groups (int): number of groups for the GroupNorm
185         padding (int or tuple): add zero-padding added to all three sides of the
         input
186         is3d (bool): use 3d or 2d convolutions/pooling operation
187     """
188
189     def __init__(self, in_channels, out_channels, conv_kernel_size=3,
        apply_pooling=True,
190                  pool_kernel_size=2, pool_type='max', basic_module=ResNetBlock,
        conv_layer_order='gcr',
191                  num_groups=8, padding=1, is3d=True):
192         super(Encoder, self).__init__()
193         assert pool_type in ['max', 'avg']
194         if apply_pooling:
195             if pool_type == 'max':
196                 if is3d:
197                     self.pooling = nn.MaxPool3d(kernel_size=pool_kernel_size)
198                 else:
199                     self.pooling = nn.MaxPool2d(kernel_size=pool_kernel_size)
```

```
            else:
                if is3d:
                    self.pooling = nn.AvgPool3d(kernel_size=pool_kernel_size)
                else:
                    self.pooling = nn.AvgPool2d(kernel_size=pool_kernel_size)
        else:
            self.pooling = None

        self.basic_module = basic_module(in_channels, out_channels,
                                         encoder=True,
                                         kernel_size=conv_kernel_size,
                                         order=conv_layer_order,
                                         num_groups=num_groups,
                                         padding=padding,
                                         is3d=is3d)

    def forward(self, x):
        if self.pooling is not None:
            x = self.pooling(x)
        x = self.basic_module(x)
        return x


class Decoder(nn.Module):
    """
    A single module for decoder path consisting of the upsampling layer
    (either learned ConvTranspose3d or nearest neighbor interpolation)
    followed by a basic module (DoubleConv or ResNetBlock).

    Args:
        in_channels (int): number of input channels
        out_channels (int): number of output channels
        conv_kernel_size (int or tuple): size of the convolving kernel
        scale_factor (tuple): used as the multiplier for the image H/W/D in
            case of nn.Upsample or as stride in case of ConvTranspose3d, must
    reverse the MaxPool3d operation
            from the corresponding encoder
        basic_module(nn.Module): either ResNetBlock or DoubleConv
        conv_layer_order (string): determines the order of layers
            in `DoubleConv` module. See `DoubleConv` for more info.
        num_groups (int): number of groups for the GroupNorm
        padding (int or tuple): add zero-padding added to all three sides of the
     input
```

```
241         upsample (bool): should the input be upsampled
242     """
243
244     def __init__(self, in_channels, out_channels, conv_kernel_size=3,
    scale_factor=(2, 2, 2), basic_module=ResNetBlock,
245                  conv_layer_order='gcr', num_groups=8, mode='nearest', padding
    =1, upsample=True, is3d=True):
246         super(Decoder, self).__init__()
247
248         if upsample:
249             # if basic_module == DoubleConv:
250             #     # if DoubleConv is the basic_module use interpolation for
    upsampling and concatenation joining
251             #     self.upsampling = InterpolateUpsampling(mode=mode)
252             #     # concat joining
253             #     self.joining = partial(self._joining, concat=True)
254             # else:
255                 # if basic_module=ResNetBlock use transposed convolution
    upsampling and summation joining
256                 self.upsampling = TransposeConvUpsampling(in_channels=
    in_channels, out_channels=out_channels,
257                                                           kernel_size=
    conv_kernel_size, scale_factor=scale_factor)
258                 # sum joining
259                 self.joining = partial(self._joining, concat=False)
260                 # adapt the number of in_channels for the ResNetBlock
261                 in_channels = out_channels
262         else:
263             # no upsampling
264             self.upsampling = NoUpsampling()
265             # concat joining
266             self.joining = partial(self._joining, concat=True)
267
268         self.basic_module = basic_module(in_channels, out_channels,
269                                          encoder=False,
270                                          kernel_size=conv_kernel_size,
271                                          order=conv_layer_order,
272                                          num_groups=num_groups,
273                                          padding=padding,
274                                          is3d=is3d)
275
276     def forward(self, encoder_features, x):
277         x = self.upsampling(encoder_features=encoder_features, x=x)
```

```
278         x = self.joining(encoder_features, x)
279         x = self.basic_module(x)
280         return x
281 
282     @staticmethod
283     def _joining(encoder_features, x, concat):
284         if concat:
285             return torch.cat((encoder_features, x), dim=1)
286         else:
287             return encoder_features + x
288 
289 
290 def create_encoders(in_channels, f_maps, basic_module, conv_kernel_size,
    conv_padding, layer_order, num_groups,
291                     pool_kernel_size, is3d):
292     # create encoder path consisting of Encoder modules. Depth of the encoder is
     equal to `len(f_maps)`
293     encoders = []
294     for i, out_feature_num in enumerate(f_maps):
295         if i == 0:
296             # apply conv_coord only in the first encoder if any
297             encoder = Encoder(in_channels, out_feature_num,
298                               apply_pooling=False,  # skip pooling in the firs
    encoder
299                               basic_module=basic_module,
300                               conv_layer_order=layer_order,
301                               conv_kernel_size=conv_kernel_size,
302                               num_groups=num_groups,
303                               padding=conv_padding,
304                               is3d=is3d)
305         else:
306             encoder = Encoder(f_maps[i - 1], out_feature_num,
307                               basic_module=basic_module,
308                               conv_layer_order=layer_order,
309                               conv_kernel_size=conv_kernel_size,
310                               num_groups=num_groups,
311                               pool_kernel_size=pool_kernel_size,
312                               padding=conv_padding,
313                               is3d=is3d)
314 
315         encoders.append(encoder)
316 
317     return nn.ModuleList(encoders)
```

```
318
319
320 def create_decoders(f_maps, basic_module, conv_kernel_size, conv_padding,
    layer_order, num_groups, is3d):
321     # create decoder path consisting of the Decoder modules. The length of the
    decoder list is equal to `len(f_maps) - 1`
322     decoders = []
323     reversed_f_maps = list(reversed(f_maps))
324     for i in range(len(reversed_f_maps) - 1):
325         # if basic_module == DoubleConv:
326         #     in_feature_num = reversed_f_maps[i] + reversed_f_maps[i + 1]
327         # else:
328         in_feature_num = reversed_f_maps[i]
329
330         out_feature_num = reversed_f_maps[i + 1]
331
332         decoder = Decoder(in_feature_num, out_feature_num,
333                           basic_module=basic_module,
334                           conv_layer_order=layer_order,
335                           conv_kernel_size=conv_kernel_size,
336                           num_groups=num_groups,
337                           padding=conv_padding,
338                           is3d=is3d)
339         decoders.append(decoder)
340     return nn.ModuleList(decoders)
341
342
343 class AbstractUpsampling(nn.Module):
344     """
345     Abstract class for upsampling. A given implementation should upsample a
    given 5D input tensor using either
346     interpolation or learned transposed convolution.
347     """
348
349     def __init__(self, upsample):
350         super(AbstractUpsampling, self).__init__()
351         self.upsample = upsample
352
353     def forward(self, encoder_features, x):
354         # get the spatial dimensions of the output given the encoder_features
355         output_size = encoder_features.size()[2:]
356         # upsample the input and return
357         return self.upsample(x, output_size)
```

```
358
359
360 class InterpolateUpsampling(AbstractUpsampling):
361     """
362     Args:
363         mode (str): algorithm used for upsampling:
364             'nearest' | 'linear' | 'bilinear' | 'trilinear' | 'area'. Default: '
    nearest'
365             used only if transposed_conv is False
366     """
367
368     def __init__(self, mode='nearest'):
369         upsample = partial(self._interpolate, mode=mode)
370         super().__init__(upsample)
371
372     @staticmethod
373     def _interpolate(x, size, mode):
374         return F.interpolate(x, size=size, mode=mode)
375
376
377 class TransposeConvUpsampling(AbstractUpsampling):
378     """
379     Args:
380         in_channels (int): number of input channels for transposed conv
381             used only if transposed_conv is True
382         out_channels (int): number of output channels for transpose conv
383             used only if transposed_conv is True
384         kernel_size (int or tuple): size of the convolving kernel
385             used only if transposed_conv is True
386         scale_factor (int or tuple): stride of the convolution
387             used only if transposed_conv is True
388
389     """
390
391     def __init__(self, in_channels=None, out_channels=None, kernel_size=3,
    scale_factor=(2, 2, 2)):
392         # make sure that the output size reverses the MaxPool3d from the
    corresponding encoder
393         upsample = nn.ConvTranspose3d(in_channels, out_channels, kernel_size=
    kernel_size, stride=scale_factor,
394                                       padding=1)
395         super().__init__(upsample)
396
```

```
397
398 class NoUpsampling(AbstractUpsampling):
399     def __init__(self):
400         super().__init__(self._no_upsampling)
401
402     @staticmethod
403     def _no_upsampling(x, size):
404         return x
405
406
407
408
409
410 def number_of_features_per_level(init_channel_number, num_levels):
411     return [init_channel_number * 2 ** k for k in range(num_levels)]
412
413
414
415 def get_class(class_name, modules):
416     for module in modules:
417         m = importlib.import_module(module)
418         clazz = getattr(m, class_name, None)
419         if clazz is not None:
420             return clazz
421     raise RuntimeError(f'Unsupported dataset class: {class_name}')
422
423
424 class AbstractUNet(nn.Module):
425     """
426     Base class for standard and residual UNet.
427
428     Args:
429         in_channels (int): number of input channels
430         out_channels (int): number of output segmentation masks;
431             Note that the of out_channels might correspond to either
432             different semantic classes or to different binary segmentation mask.
433             It's up to the user of the class to interpret the out_channels and
434             use the proper loss criterion during training (i.e. CrossEntropyLoss
     (multi-class)
435             or BCEWithLogitsLoss (two-class) respectively)
436         f_maps (int, tuple): number of feature maps at each level of the encoder
    ; if it's an integer the number
437             of feature maps is given by the geometric progression: f_maps ^ k, k
```

```
    =1,2,3,4
438         final_sigmoid (bool): if True apply element-wise nn.Sigmoid after the
    final 1x1 convolution,
439             otherwise apply nn.Softmax. In effect only if `self.training ==
    False`, i.e. during validation/testing
440         basic_module: basic model for the encoder/decoder (DoubleConv,
    ResNetBlock, ....)
441         layer_order (string): determines the order of layers in `SingleConv`
    module.
442             E.g. 'crg' stands for GroupNorm3d+Conv3d+ReLU. See `SingleConv` for
    more info
443         num_groups (int): number of groups for the GroupNorm
444         num_levels (int): number of levels in the encoder/decoder path (applied
    only if f_maps is an int)
445             default: 4
446         is_segmentation (bool): if True and the model is in eval mode, Sigmoid/
    Softmax normalization is applied
447             after the final convolution; if False (regression problem) the
    normalization layer is skipped
448         conv_kernel_size (int or tuple): size of the convolving kernel in the
    basic_module
449         pool_kernel_size (int or tuple): the size of the window
450         conv_padding (int or tuple): add zero-padding added to all three sides
    of the input
451         is3d (bool): if True the model is 3D, otherwise 2D, default: True
452     """
453
454     def __init__(self, in_channels, out_channels, final_sigmoid, basic_module,
    f_maps=64, layer_order='gcr',
455                  num_groups=8, num_levels=4, is_segmentation=True,
    conv_kernel_size=3, pool_kernel_size=2,
456                  conv_padding=1, is3d=True):
457         super(AbstractUNet, self).__init__()
458
459         if isinstance(f_maps, int):
460             f_maps = number_of_features_per_level(f_maps, num_levels=num_levels)
461
462         assert isinstance(f_maps, list) or isinstance(f_maps, tuple)
463         assert len(f_maps) > 1, "Required at least 2 levels in the U-Net"
464         if 'g' in layer_order:
465             assert num_groups is not None, "num_groups must be specified if
    GroupNorm is used"
466
```

```
467         # create encoder path
468         self.encoders = create_encoders(in_channels, f_maps, basic_module,
    conv_kernel_size, conv_padding, layer_order,
469                                         num_groups, pool_kernel_size, is3d)
470 
471         # create decoder path
472         self.decoders = create_decoders(f_maps, basic_module, conv_kernel_size,
    conv_padding, layer_order, num_groups,
473                                         is3d)
474 
475         # in the last layer a 1 1  convolution reduces the number of output
    channels to the number of labels
476         if is3d:
477             self.final_conv = nn.Conv3d(f_maps[0], out_channels, 1)
478         else:
479             self.final_conv = nn.Conv2d(f_maps[0], out_channels, 1)
480 
481         if is_segmentation:
482             # semantic segmentation problem
483             if final_sigmoid:
484                 self.final_activation = nn.Sigmoid()
485             else:
486                 self.final_activation = nn.Softmax(dim=1)
487         else:
488             # regression problem
489             self.final_activation = None
490 
491     def forward(self, x):
492         # encoder part
493         encoders_features = []
494         for encoder in self.encoders:
495             x = encoder(x)
496             # reverse the encoder outputs to be aligned with the decoder
497             encoders_features.insert(0, x)
498 
499         # remove the last encoder's output from the list
500         # !!remember: it's the 1st in the list
501         encoders_features = encoders_features[1:]
502 
503         # decoder part
504         for decoder, encoder_features in zip(self.decoders, encoders_features):
505             # pass the output from the corresponding encoder and the output
506             # of the previous decoder
```

```
507             x = decoder(encoder_features, x)
508
509         x = self.final_conv(x)
510
511         # apply final_activation (i.e. Sigmoid or Softmax) only during
    prediction.
512         # During training the network outputs logits
513         if not self.training and self.final_activation is not None:
514             x = self.final_activation(x)
515
516         return x
517
518
519 class ResidualUNet3D(AbstractUNet):
520     """
521     Residual 3DUnet model implementation based on https://arxiv.org/pdf
    /1706.00120.pdf.
522     Uses ResNetBlock as a basic building block, summation joining instead
523     of concatenation joining and transposed convolutions for upsampling (watch
    out for block artifacts).
524     Since the model effectively becomes a residual net, in theory it allows for
    deeper UNet.
525     """
526
527     def __init__(self, in_channels, out_channels, final_sigmoid=True, f_maps=64,
     layer_order='gcr',
528                  num_groups=8, num_levels=5, is_segmentation=True, conv_padding
    =1, **kwargs):
529         super(ResidualUNet3D, self).__init__(in_channels=in_channels,
530                                              out_channels=out_channels,
531                                              final_sigmoid=final_sigmoid,
532                                              basic_module=ResNetBlock,
533                                              f_maps=f_maps,
534                                              layer_order=layer_order,
535                                              num_groups=num_groups,
536                                              num_levels=num_levels,
537                                              is_segmentation=is_segmentation,
538                                              conv_padding=conv_padding,
539                                              is3d=True)
```

Listing 2: Python code to generate the model architecture

## A.3 Model

```
import math
import random

import torch
from torch import nn as nn
import torch.nn.functional as F

from util.logconf import logging
from util.unetPlusPlus import ResidualUNet3D


log = logging.getLogger(__name__)
log.setLevel(logging.DEBUG)

class UNetWrapper(nn.Module):
    #kwarg is a dictionary containing all keyword arguments passed to the
    constructor
    def __init__(self, **kwargs):
        super().__init__()

        self.input_batchnorm = nn.BatchNorm3d(kwargs['in_channels'])
        self.unet = ResidualUNet3D(**kwargs)

    def _init_weights(self):
        init_set = {
            nn.Conv2d,
            nn.Conv3d,
            nn.ConvTranspose2d,
            nn.ConvTranspose3d,
            nn.Linear,
        }
        for m in self.modules():
            if type(m) in init_set:
                nn.init.kaiming_normal_(
                    m.weight.data, mode='fan_out', nonlinearity='relu', a=0
                )
                if m.bias is not None:
                    fan_in, fan_out = \
                        nn.init._calculate_fan_in_and_fan_out(m.weight.data)
                    bound = 1 / math.sqrt(fan_out)
                    nn.init.normal_(m.bias, -bound, bound)


```

```
43      def forward(self, input_batch):
44          bn_output = self.input_batchnorm(input_batch)
45          un_output = self.unet(bn_output)
46
47          return un_output
48
49
50
51  class SegmentationAugmentation(nn.Module):
52      def __init__(
53              self, flip=None, offset=None, scale=None, rotate=None, noise=None
54      ):
55          super().__init__()
56
57          self.flip = flip
58          self.scale = scale
59          self.rotate = rotate
60          self.noise = noise
61
62      def forward(self, input_g, label_g):
63          transform_t = self._build3dTransformMatrix()
64          transform_t = transform_t.expand(input_g.shape[0],-1, -1)
65          transform_t = transform_t.to(input_g.device, torch.float32)
66          affine_t = F.affine_grid(transform_t[:,:3],
67                  input_g.size(), align_corners=False)
68
69          augmented_input_g = F.grid_sample(input_g,
70                  affine_t, padding_mode='border',
71                  align_corners=False)
72          augmented_label_g = F.grid_sample(label_g.to(torch.float32),
73                  affine_t, padding_mode='border',
74                  align_corners=False)
75
76          if self.noise:
77              noise_t = torch.randn_like(augmented_input_g)
78              noise_t *= self.noise
79
80              augmented_input_g += noise_t
81
82          return augmented_input_g, augmented_label_g
83
84
85      def _build3dTransformMatrix(self):
```

```
            transform_t = torch.eye(4)

            for i in range(2):
                if self.flip:
                    if random.random() > 0.5:
                        transform_t[i,i] *= -1

                if self.scale:
                    scale_float = self.scale
                    random_float = (random.random() * 2 - 1)
                    transform_t[i,i] *= 1.0 + scale_float * random_float

            if self.rotate:
                alpha = random.random() * math.pi * 2 #Takes a random angle in radians, so in the range 0 .. 2{pi}

                rotation_z = torch.tensor([ #Rotation matrix for the 2D rotation by the random angle in the first two dimensions
                    [math.cos(alpha),-math.sin(alpha), 0, 0],
                    [math.sin(alpha), math.cos(alpha), 0, 0],
                    [0, 0, 1, 0],
                    [0, 0, 0, 1]
                ])

                transform_t @= rotation_z #Applies the rotation to the transformation matrix using the Python matrix multiplication operator

            return transform_t
```

Listing 3: Python code to asemble the compelte model

## A.4 Training

```
import argparse
import datetime
import hashlib
import os
import shutil
import socket
import sys

import numpy as np
from torch.utils.tensorboard import SummaryWriter

```

```
12 import torch
13 import torch.nn as nn
14 import torch.optim
15
16 from torch.optim import SGD, Adam, AdamW
17 from torch.utils.data import DataLoader
18
19 from util.util import enumerateWithEstimate
20 from .dsets import ADAM2dSegmentationDataset, getMRA
21 from util.logconf import logging
22 from .model import UNetWrapper, SegmentationAugmentation
23 from .loss import GeneralizedDiceLoss
24 from torch.optim.lr_scheduler import ExponentialLR
25
26 log = logging.getLogger(__name__)
27 log.setLevel(logging.DEBUG)
28
29
30 METRICS_LOSS_NDX = 1
31 METRICS_DICE_NDX = 2
32 METRICS_TP_NDX = 7
33 METRICS_FN_NDX = 8
34 METRICS_FP_NDX = 9
35
36 METRICS_SIZE = 10
37
38 class SegmentationTrainingApp:
39     def __init__(self, sys_argv=None):
40         if sys_argv is None:
41             sys_argv = sys.argv[1:]
42
43         parser = argparse.ArgumentParser()
44         parser.add_argument('--batch-size',
45             help='Batch size to use for training',
46             default=5,
47             type=int,
48         )
49         parser.add_argument('--num-workers',
50             help='Number of worker processes for background data loading',
51             default=8,
52             type=int,
53         )
54         parser.add_argument('--epochs',
```

```
            help='Number of epochs to train for',
            default=1,
            type=int,
        )

        parser.add_argument('--augmented',
            help="Augment the training data.",
            action='store_true',
            default=False,
        )
        parser.add_argument('--augment-flip',
            help="Augment the training data by randomly flipping the data left-right, up-down, and front-back.",
            action='store_true',
            default=False,
        )
        parser.add_argument('--augment-offset',
            help="Augment the training data by randomly offsetting the data slightly along the X and Y axes.",
            action='store_true',
            default=False,
        )
        parser.add_argument('--augment-scale',
            help="Augment the training data by randomly increasing or decreasing the size of the candidate.",
            action='store_true',
            default=False,
        )
        parser.add_argument('--augment-rotate',
            help="Augment the training data by randomly rotating the data around the head-foot axis.",
            action='store_true',
            default=False,
        )
        parser.add_argument('--augment-noise',
            help="Augment the training data by randomly adding noise to the data.",
            action='store_true',
            default=False,
        )

        parser.add_argument('--tb-prefix',
            default='p2ch13',
```

```
            help="Data prefix to use for Tensorboard run. Defaults to chapter.",
        )

        parser.add_argument('comment',
            help="Comment suffix for Tensorboard run.",
            nargs='?',
            default='none',
        )


        self.cli_args = parser.parse_args(sys_argv)
        self.time_str = datetime.datetime.now().strftime('%Y-%m-%d_%H.%M.%S')
        self.totalTrainingSamples_count = 0
        self.trn_writer = None
        self.val_writer = None
        self.augmentation = True

        self.augmentation_dict = {}
        if self.cli_args.augmented or self.cli_args.augment_flip:
            self.augmentation_dict['flip'] = True
        if self.cli_args.augmented or self.cli_args.augment_offset:
            self.augmentation_dict['offset'] = 0
        if self.cli_args.augmented or self.cli_args.augment_scale:
            self.augmentation_dict['scale'] = 0.2
        if self.cli_args.augmented or self.cli_args.augment_rotate:
            self.augmentation_dict['rotate'] = True
        if self.cli_args.augmented or self.cli_args.augment_noise:
            self.augmentation_dict['noise'] = 25.0

        self.use_cuda = torch.cuda.is_available()
        self.device = torch.device("cuda" if self.use_cuda else "cpu")

        self.segmentation_model, self.augmentation_model = self.initModel()
        self.optimizer = self.initOptimizer()
        self.criterion = GeneralizedDiceLoss().to(self.device)

    def initModel(self):
        segmentation_model = UNetWrapper(
            in_channels=1,
            out_channels=1,
        )

        augmentation_model = SegmentationAugmentation(**self.augmentation_dict)
```

```
136
137         if self.use_cuda:
138             log.info("Using CUDA; {} devices.".format(torch.cuda.device_count())
    )
139             if torch.cuda.device_count() > 1:
140                 segmentation_model = nn.DataParallel(segmentation_model)
141                 augmentation_model = nn.DataParallel(augmentation_model)
142             segmentation_model = segmentation_model.to(self.device)
143             augmentation_model = augmentation_model.to(self.device)
144
145         return segmentation_model, augmentation_model
146
147     def initOptimizer(self):
148
149         return Adam(self.segmentation_model.parameters(),weight_decay=1e-5,lr
    =0.00001)
150
151     def initTrainDl(self):
152         train_ds = ADAM2dSegmentationDataset(
153             config=4,
154             isValSet_bool=False,
155         )
156
157         batch_size = self.cli_args.batch_size
158         if self.use_cuda:
159             batch_size *= torch.cuda.device_count()
160
161         train_dl = DataLoader(
162             train_ds,
163             batch_size=batch_size,
164             num_workers=self.cli_args.num_workers,
165             pin_memory=self.use_cuda,
166         )
167
168         return train_dl
169
170     def initValDl(self):
171         val_ds = ADAM2dSegmentationDataset(
172             config=4,
173             isValSet_bool=True,
174         )
175
176         batch_size = self.cli_args.batch_size
```

```
177         if self.use_cuda:
178             batch_size *= torch.cuda.device_count()
179 
180         val_dl = DataLoader(
181             val_ds,
182             batch_size=batch_size,
183             num_workers=self.cli_args.num_workers,
184             pin_memory=self.use_cuda,
185         )
186 
187         return val_dl
188 
189     def initTensorboardWriters(self):
190         if self.trn_writer is None:
191             log_dir = os.path.join('runs', self.cli_args.tb_prefix, self.
    time_str)
192 
193             self.trn_writer = SummaryWriter(
194                 log_dir=log_dir + '_trn_seg_' + self.cli_args.comment)
195             self.val_writer = SummaryWriter(
196                 log_dir=log_dir + '_val_seg_' + self.cli_args.comment)
197 
198     def main(self):
199         log.info("Starting {}, {}".format(type(self).__name__, self.cli_args))
200 
201         train_dl = self.initTrainDl()
202         val_dl = self.initValDl()
203 
204         best_score = 0.0
205         self.validation_cadence = 1
206         for epoch_ndx in range(1, self.cli_args.epochs + 1):
207             log.info("Epoch {} of {}, {}/{} batches of size {}*{}".format(
208                 epoch_ndx,
209                 self.cli_args.epochs,
210                 len(train_dl),
211                 len(val_dl),
212                 self.cli_args.batch_size,
213                 (torch.cuda.device_count() if self.use_cuda else 1),
214             ))
215             trnMetrics_t = self.doTraining(epoch_ndx, train_dl)
216             self.logMetrics(epoch_ndx, 'trn', trnMetrics_t)
217 
218             if epoch_ndx == 1 or epoch_ndx % self.validation_cadence == 0:
```

```
                # if validation is wanted
                valMetrics_t = self.doValidation(epoch_ndx, val_dl)
                score = self.logMetrics(epoch_ndx, 'val', valMetrics_t)
                best_score = max(score, best_score)

                self.saveModel('seg', epoch_ndx, score == best_score)

                self.logImages(epoch_ndx, 'trn', train_dl)
                self.logImages(epoch_ndx, 'val', val_dl)

        self.trn_writer.close()
        self.val_writer.close()

    def doTraining(self, epoch_ndx, train_dl):
        trnMetrics_g = torch.zeros(METRICS_SIZE, len(train_dl.dataset), device=
    self.device)
        self.segmentation_model.train()
        train_dl.dataset.shuffleSamples()

        batch_iter = enumerateWithEstimate(
            train_dl,
            "E{} Training".format(epoch_ndx),
            start_ndx=train_dl.num_workers,
        )
        for batch_ndx, batch_tup in batch_iter:
            self.optimizer.zero_grad()
            loss_var = self.computeBatchLoss(batch_ndx, batch_tup, train_dl.
    batch_size, trnMetrics_g)
            loss_var.backward()
            self.optimizer.step()

        self.totalTrainingSamples_count += trnMetrics_g.size(1)

        return trnMetrics_g.to('cpu')

    def doValidation(self, epoch_ndx, val_dl):
        with torch.no_grad():
            valMetrics_g = torch.zeros(METRICS_SIZE, len(val_dl.dataset), device
    =self.device)

            batch_iter = enumerateWithEstimate(
                val_dl,
                "E{} Validation ".format(epoch_ndx),
```

```
259                 start_ndx=val_dl.num_workers,
260             )
261             for batch_ndx, batch_tup in batch_iter:
262                 self.computeBatchLoss(batch_ndx, batch_tup, val_dl.batch_size,
    valMetrics_g)
263
264         return valMetrics_g.to('cpu')
265
266     def computeBatchLoss(self, batch_ndx, batch_tup, batch_size, metrics_g,
267                          classificationThreshold=0.5):
268         input_t, label_t = batch_tup
269
270         input_g = input_t.to(self.device, non_blocking=True)
271         label_g = label_t.to(self.device, non_blocking=True)
272
273         if self.segmentation_model.training and self.augmentation_dict:
274             input_g, label_g = self.augmentation_model(input_g, label_g)
275
276
277         prediction_g = self.segmentation_model(input_g)
278         loss = self.criterion(prediction_g, label_g).mean()
279
280
281         start_ndx = batch_ndx * batch_size
282         end_ndx = start_ndx + input_t.size(0)
283
284         with torch.no_grad():
285
286             predictionBool_g = (prediction_g[:, 0:1]
287                                 > classificationThreshold).to(torch.float32)
288
289             label_g = label_g.to(torch.float32)
290
291             tp = (predictionBool_g *  label_g).sum(dim=[1,2,3,4])
292             fn = ((1-predictionBool_g) *  label_g).sum(dim=[1,2,3,4])
293             fp = (predictionBool_g *  (1-label_g)).sum(dim=[1,2,3,4])
294
295             intersection = (predictionBool_g * label_g).sum(dim=[1,2,3,4])
296             dice = (2. * intersection )/(predictionBool_g.sum(dim=[1,2,3,4]) +
    label_g.sum(dim=[1,2,3,4]))
297             dice = torch.where(torch.isnan(dice), torch.tensor(1.), dice)
298
299             metrics_g[METRICS_LOSS_NDX, start_ndx:end_ndx] = loss
```

```
300             metrics_g[METRICS_DICE_NDX, start_ndx:end_ndx] = dice.mean()
301             metrics_g[METRICS_TP_NDX, start_ndx:end_ndx] = tp
302             metrics_g[METRICS_FN_NDX, start_ndx:end_ndx] = fn
303             metrics_g[METRICS_FP_NDX, start_ndx:end_ndx] = fp
304
305         return loss
306
307
308     def logImages(self, epoch_ndx, mode_str, dl):
309         self.segmentation_model.eval()
310
311         images = sorted(dl.dataset.series_list)[:12]
312         for series_ndx, tup in enumerate(images):
313             mra_t, label_t=dl.dataset.getitem_trainingCrop(tup)
314
315             input_g = mra_t.to(self.device).unsqueeze(0)
316
317             prediction_g = self.segmentation_model(input_g)[0]
318
319             prediction_a = (prediction_g.to('cpu').detach().numpy() > 0.5)
320             prediction_a = prediction_a[0,8,:,:]
321             label_a = label_t.numpy() > 0.5
322             label_a = label_a[0,8,:,:]
323
324
325             mra_t[0,8,:,:] /= torch.max(mra_t[0,8,:,:])
326             mra_t[0,8,:,:] += 0.5
327
328             mraSlice_a = mra_t[0,8,:,:].numpy()
329
330             image_a = np.zeros((64, 64, 3), dtype=np.float32)
331             image_a[:,:,:] = mraSlice_a.reshape((64,64,1))
332             image_a[:,:,0] += prediction_a & (1 - label_a)
333
334             image_a[:,:,1] += prediction_a & label_a
335             image_a *= 0.5
336             image_a.clip(0, 1, image_a)
337
338             writer = getattr(self, mode_str + '_writer')
339             writer.add_image(
340                 f'{mode_str}/{series_ndx}_prediction',
341                 image_a,
342                 self.totalTrainingSamples_count,
```

```
343                 dataformats='HWC',
344             )
345 
346             if epoch_ndx == 1:
347                 image_a = np.zeros((64, 64, 3), dtype=np.float32)
348                 image_a[:,:,:] = mraSlice_a.reshape((64, 64,1))
349                 image_a[:,:,1] += label_a  # Green
350 
351                 image_a *= 0.5
352                 image_a[image_a < 0] = 0
353                 image_a[image_a > 1] = 1
354                 writer.add_image(
355                     '{}/{}_label'.format(
356                         mode_str,
357                         series_ndx
358                     ),
359                     image_a,
360                     self.totalTrainingSamples_count,
361                     dataformats='HWC',
362                 )
363                 # This flush prevents TB from getting confused about which
364                 # data item belongs where.
365             writer.flush()
366 
367     def logMetrics(self, epoch_ndx, mode_str, metrics_t):
368         log.info("E{} {}".format(
369             epoch_ndx,
370             type(self).__name__,
371         ))
372 
373         metrics_a = metrics_t.detach().numpy()
374         sum_a = metrics_a.sum(axis=1)
375         assert np.isfinite(metrics_a).all()
376 
377         allLabel_count = sum_a[METRICS_TP_NDX] + sum_a[METRICS_FN_NDX]
378 
379         metrics_dict = {}
380         metrics_dict['loss/all'] = metrics_a[METRICS_LOSS_NDX].mean()
381         metrics_dict['dice/all'] = metrics_a[METRICS_DICE_NDX].mean()
382 
383         metrics_dict['percent_all/tp'] = \
384             sum_a[METRICS_TP_NDX] / (allLabel_count or 1) * 100
385         metrics_dict['percent_all/fn'] = \
```

```
386             sum_a[METRICS_FN_NDX] / (allLabel_count or 1) * 100
387         metrics_dict['percent_all/fp'] = \
388             sum_a[METRICS_FP_NDX] / (allLabel_count or 1) * 100
389 
390 
391         precision = metrics_dict['pr/precision'] = sum_a[METRICS_TP_NDX] \
392             / ((sum_a[METRICS_TP_NDX] + sum_a[METRICS_FP_NDX]) or 1)
393         recall    = metrics_dict['pr/recall']    = sum_a[METRICS_TP_NDX] \
394             / ((sum_a[METRICS_TP_NDX] + sum_a[METRICS_FN_NDX]) or 1)
395 
396         metrics_dict['pr/f1_score'] = 2 * (precision * recall) \
397             / ((precision + recall) or 1)
398 
399         log.info(("E{} {:8} "
400                   + "{loss/all:.4f} loss, "
401                   + "{dice/all:.4f} dice, "
402                   + "{pr/precision:.4f} precision, "
403                   + "{pr/recall:.4f} recall, "
404                   + "{pr/f1_score:.4f} f1 score"
405                    ).format(
406             epoch_ndx,
407             mode_str,
408             **metrics_dict,
409         ))
410         log.info(("E{} {:8} "
411                    + "{loss/all:.4f} loss, "
412                    + "{dice/all:.4f} dice, "
413                    + "{percent_all/tp:-5.1f}% tp, {percent_all/fn:-5.1f}% fn, {
    percent_all/fp:-9.1f}% fp"
414         ).format(
415             epoch_ndx,
416             mode_str + '_all',
417             **metrics_dict,
418         ))
419 
420         self.initTensorboardWriters()
421         writer = getattr(self, mode_str + '_writer')
422 
423         prefix_str = 'seg_'
424 
425         for key, value in metrics_dict.items():
426             writer.add_scalar(prefix_str + key, value, self.
    totalTrainingSamples_count)
```

```
427
428         writer.flush()
429
430         score = metrics_dict['pr/recall']
431
432         return score
433
434
435     def saveModel(self, type_str, epoch_ndx, isBest=False):
436         file_path = os.path.join(
437             'data-unversioned',
438             'models',
439             self.cli_args.tb_prefix,
440             '{}_{}_{}.{}.state'.format(
441                 type_str,
442                 self.time_str,
443                 self.cli_args.comment,
444                 self.totalTrainingSamples_count,
445             )
446         )
447
448         os.makedirs(os.path.dirname(file_path), mode=0o755, exist_ok=True)
449
450         model = self.segmentation_model
451         if isinstance(model, torch.nn.DataParallel):
452             model = model.module
453
454         state = {
455             'sys_argv': sys.argv,
456             'time': str(datetime.datetime.now()),
457             'model_state': model.state_dict(),
458             'model_name': type(model).__name__,
459             'optimizer_state' : self.optimizer.state_dict(),
460             'optimizer_name': type(self.optimizer).__name__,
461             'epoch': epoch_ndx,
462             'totalTrainingSamples_count': self.totalTrainingSamples_count,
463         }
464         torch.save(state, file_path)
465
466         log.info("Saved model params to {}".format(file_path))
467
468         if isBest:
469             best_path = os.path.join(
```

```
                'data-unversioned', 'models',
                self.cli_args.tb_prefix,
                '{}_{}_{}.best.state'.format(
                type_str,
                self.time_str,
                self.cli_args.comment)
            )
            torch.save(state, file_path)

            log.info("Saved model params to {}".format(best_path))

        with open(file_path, 'rb') as f:
            log.info("SHA1: " + hashlib.sha1(f.read()).hexdigest())


if __name__ == '__main__':

    SegmentationTrainingApp().main()
```

Listing 4: Python code to train the model

## A.5 Loss

```
import torch
from torch import nn as nn


class _AbstractDiceLoss(nn.Module):
    """
    Base class for different implementations of Dice loss.
    """

    def __init__(self, weight=None, normalization='sigmoid'):
        super(_AbstractDiceLoss, self).__init__()
        self.register_buffer('weight', weight)
        # The output from the network during training is assumed to be un-
    normalized probabilities and we would
        # like to normalize the logits. Since Dice (or soft Dice in this case)
    is usually used for binary data,
        # normalizing the channels with Sigmoid is the default choice even for
    multi-class segmentation problems.
        # However if one would like to apply Softmax in order to get the proper
    probability distribution from the
        # output, just specify `normalization=Softmax`
```

```
        assert normalization in ['sigmoid', 'softmax', 'none']
        if normalization == 'sigmoid':
            self.normalization = nn.Sigmoid()
        elif normalization == 'softmax':
            self.normalization = nn.Softmax(dim=1)
        else:
            self.normalization = lambda x: x

    def dice(self, input, target, weight):
        # actual Dice score computation; to be implemented by the subclass
        raise NotImplementedError

    def forward(self, input, target):
        # get probabilities from logits
        input = self.normalization(input)

        # compute per channel Dice coefficient
        per_channel_dice = self.dice(input, target, weight=self.weight)

        # average Dice score across all channels/classes
        return 1. - torch.mean(per_channel_dice)


class GeneralizedDiceLoss(_AbstractDiceLoss):
    """Computes Generalized Dice Loss (GDL) as described in https://arxiv.org/pdf/1707.03237.pdf.
    """

    def __init__(self, normalization='sigmoid', epsilon=1e-6):
        super().__init__(weight=None, normalization=normalization)
        self.epsilon = epsilon

    def dice(self, input, target, weight):
        assert input.size() == target.size(), "'input' and 'target' must have the same shape"

        input = flatten(input)
        target = flatten(target)
        target = target.float()

        if input.size(0) == 1:
            # for GDL to make sense we need at least 2 channels (see https://arxiv.org/pdf/1707.03237.pdf)
```

```
            # put foreground and background voxels in separate channels
            input = torch.cat((input, 1 - input), dim=0)
            target = torch.cat((target, 1 - target), dim=0)

        # GDL weighting: the contribution of each label is corrected by the inverse of its volume
        w_l = target.sum(-1)
        w_l = 1 / (w_l * w_l).clamp(min=self.epsilon)
        w_l.requires_grad = False

        intersect = (input * target).sum(-1)
        intersect = intersect * w_l

        denominator = (input + target).sum(-1)
        denominator = (denominator * w_l).clamp(min=self.epsilon)

        return 2 * (intersect.sum() / denominator.sum())


def flatten(tensor):
    """Flattens a given tensor such that the channel axis is first.
    The shapes are transformed as follows:
       (N, C, D, H, W) -> (C, N * D * H * W)
    """
    # number of channels
    C = tensor.size(1)
    # new axis order
    axis_order = (1, 0) + tuple(range(2, tensor.dim()))
    # Transpose: (N, C, D, H, W) -> (C, N, D, H, W)
    transposed = tensor.permute(axis_order)
    # Flatten: (C, N, D, H, W) -> (C, N * D * H * W)
    return transposed.contiguous().view(C, -1)
```

Listing 5: Python code for the loss function

# B nnU-Net training results

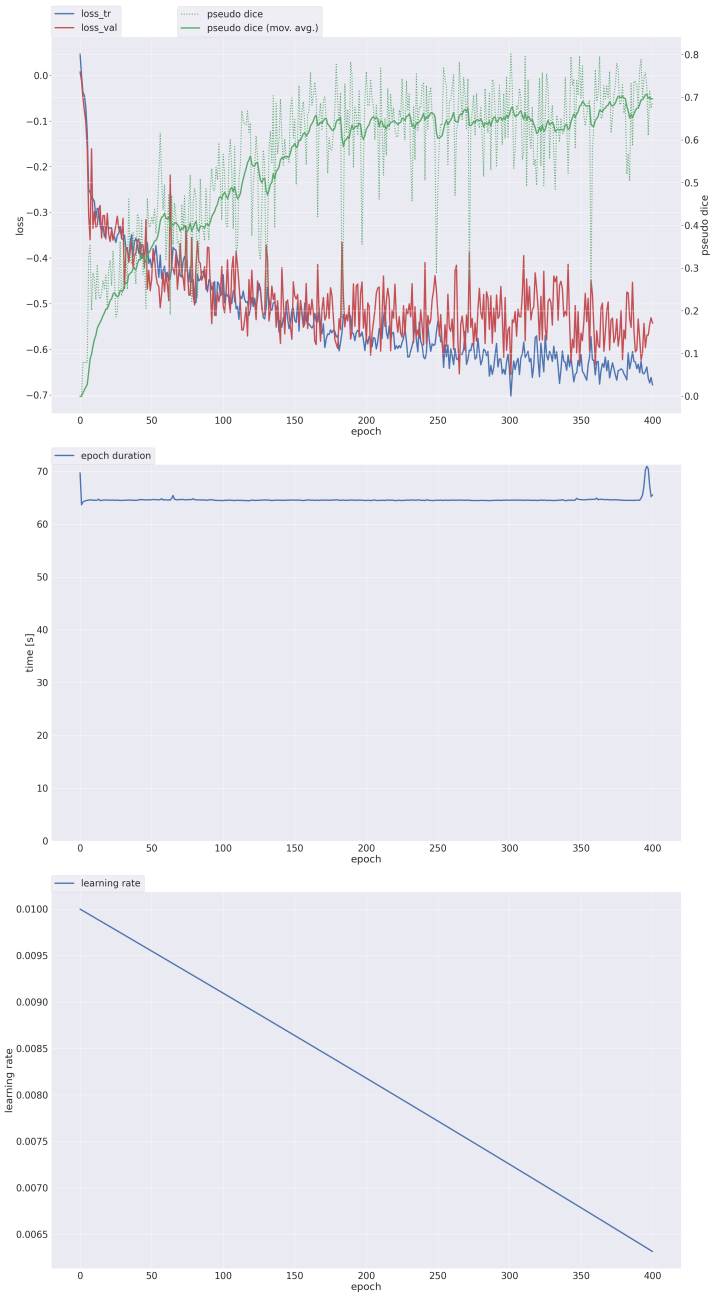



Figure 15: Results displaying nnU-Net training process for the aforementioned dataset