UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Generative Adversarial Networks for Electron Microscopy Image Segmentation

Autor:
Herreros Fraile, Eduardo

Responsable de Intercambio en la Uva:
Eusebio de la Fuente López

Universidad de destino:
Technische Universität Dresden

Valladolid, 09/2021.

TFG REALIZADO EN PROGRAMA DE INTERCAMBIO

TÍTULO: **Generative Adversarial Networks for Electron Microscopy Image Segmentation**

ALUMNO: **Eduardo Herreros Fraile**

FECHA: **20/09/2021**

CENTRO: **Machine Learning for Computer Vision / Computer Science**

UNIVERSIDAD: **Technische Universität Dresden**

TUTOR: **Mark Schöne**

Resumen:

El objetivo de este trabajo es estudiar la conveniencia del “entrenamiento adversario” para la segmentación de imágenes de microscopio electrónico, buscando reducir el esfuerzo de etiquetado y la necesidad de ser experto en el campo. El estudio se realiza sobre el dataset CREMI. Inicialmente se implementan diferentes modelos de aprendizaje supervisado, por medio de la arquitectura UNet de 2 y 3 dimensiones, fijando una referencia. A continuación, se explora la respuesta de nuestro dataset al entrenamiento no supervisado por medio de la arquitectura Style-Gan. Finalmente se opta por el framework Boundless, con el objetivo de que las representaciones, texturas y estructuras aprendidas durante la tarea auxiliar de extensión de imagen sean útiles para aplicar transferencia de aprendizaje en la tarea principal de segmentar o detectar las membranas de las células.

Palabras Clave:

Neural Network (Red Neuronal), GAN (Red Generativa Antagónica), UNet, Segmentación, Células.

Summary:

The aim of this work is to study the suitability of adversarial training for electron microscopy image segmentation, looking forward reducing the labeling efforts that this task requires, in addition to exceptional knowledge on the field. For such a task we work on the CREMI dataset. Initially, different supervised learning models are implemented, through the 2D and 3D UNet architecture, establishing a reference. Consecutively, different unsupervised trainings are carried out through the Style-Gan architecture in order to analyse the answer and behaviour of our dataset. Finally, the Boundless framework is chosen to learn representations through the auxiliary task of image-extension, seeking that these representations would be useful in the main objective of segmentation and membrane detection, after transfer learning is applied.

Key words:

Neural Network, GAN (Generative Adversarial Network), UNet, Segmentation (Instance segmentation), Cells.

TECHNISCHE
UNIVERSITÄT
DRESDEN

Faculty of Computer Science Institute of Artificial Intelligence, Machine Learning for Computer Vision

Bachelor Thesis

Generative Adversarial Networks for Electron Microscopy Image Segmentation

Eduardo Herreros

Born on: April 28, 1999 in Valladolid
Matriculation number: 4995481
Matriculation year: 2017

to achieve the academic degree

Industrial Electronics and Automation Engineering

First referee
Prof. Dr. Björn Andres

Second referee
Prof. Dr. Stefan Gumhold

Submitted on: August 18, 2021

I dedicate this thesis to .... Thanks to my professors and supervisors

Abstract

The seek of this work is to study the suitability of adversarial training for electron microscopy image segmentation, looking forward reducing the labeling efforts that this task requires, in addition to exceptional knowledge on the field. For such a task we have the CREMI dataset, which consists in a volume of neurons. The required segmentation context is provided by the state of the art architecture Unet, that we train to produce a first segmentation with supervised learning, and to be able to compare the results afterwards with unsupervised learning. Regarding adversarial training, we first train the StyleGan to check how our dataset reacted to Gans. Motivated by the results, we decided to dig deeper and trained a gan for Image Extension with the goal of later extraction of the learnt features, hoping that those features would improve the segmentation task. We found out the the learnt features did not improve the segmentation

Contents

1 Introduction **6**

2 Background **7**
2.1 Brief introduction to neural networks 8
2.1.1 Layers 9
2.2 Brief introduction to GANs: Generative Adversarial Networks 11

3 Methodology **13**
3.1 UNet 13
3.2 Segmentation 14
3.2.1 Affinity maps 14
3.2.2 Data augmentation 16
3.2.3 Batches 17
3.2.4 2D and 3D UNet models 17
3.2.5 Loss function: BCE 18
3.2.6 Optimizer: Adam 19
3.2.7 Accuracy 19
3.2.8 Post-processing 20
3.2.9 First contact with GANs: StyleGan 23
3.2.10 Transfer learning and GANs for Image Extension 24

4 Experiments **26**
4.1 Dataset 26
4.2 StyleGAN evaluation 26
4.3 2D UNet and 3D UNet for segmentation 26
4.3.1 Dataset 26
4.3.2 Training procedures 27
4.3.3 Post-processing 27
4.4 Image Extension: mask's shape/position influence 27
4.4.1 Image Extension 27
4.4.2 Segmentation 28
4.5 Evaluation of the segmentation quality: pretrained weights and frozen layers . . 28
4.5.1 Input data 28
4.5.2 Training procedures 29

5 Results **30**
5.1 StyleGan results (4.2) 30

5.2 2D and 3D UNet segmentation results (4.3) 31
5.2.1 2D-UNet results 31
5.2.2 3D-UNet results 34
5.2.3 Contrasting results 37
5.3 Image Extension results (4.4.1) 39
5.3.1 Mask on the right 39
5.3.2 Square mask in the middle 40
5.3.3 Square mask randomly located/different size 41
5.4 Segmentation (4.4.2) 42
5.5 Transfer learning results (4.5) 44
5.5.1 Pretrained weights vs random initialization 45
5.5.2 Hyperparameters 47
5.5.3 Early stopping and reduced dataset 48

6 Discussion **50**
6.1 Segmentation UNet 50
6.2 Adversarial training 53

7 Appendix **59**

1 Introduction

In the last years, image segmentation has been one of the most relevant tasks in computer vision. In the field of medicine is has become essential to have high quality segmentations for many different tesks, as for example the study of the brain. However, the image segmentation in this field have very little labeled data, as not only requires great effort but also exceptional knowledge on the field. Several studies of supervised learning have digged into the matter, reaching outstanding results in the segmentation task, however the is still plenty room for improvement. In recent years, the study of unsupervised learning for many different tasks has exponentially grown. Particularly, Generative Adversarial Networks have become very popular, offering a particular adversarial training that has proven to be very useful, and provides astonishing results with little effort. Therefore, many different GANs have been developed, for many different purposes. One specific type of gans are focused on predicting a feasible extension of an image, in such a way that the prediction correctly matches the context of the former image, and they merge into one. In order to achieve this, a deep understanding of the image is needed. This extraction of information would allow the Gan to produce image extensions in which a human eye would easily recognize familiar objects, shapes and textures. This two kinds of deep learning approaches, supervised and unsupervised, even though they seem to be different, they have the same foundation, and can meet in what is known as transfer learning. This would join the extraordinary behaviour of gans, with the better known supervised learning, and could lead to improvements in previous isolated training. For example, in the segmentation of brain cells, valuable information extracted by the gan could lead to reduce the efforts of labeling, and the need of larger datasets to achieve proper segmentation.

2 Background

In this context of biomedical image segmentation, it seems almost mandatory to begin mentioning the UNet. The UNet paper [RFB15] came out in 2015 proposing a new architecture, based on the 'fully convolutional network', looking forward overcoming the need of incredibly large training sets and networks, that these segmentations tasks usually demand. This architecture, formed by a contracting path with convolutions and downsampling, and a symetric expanding path with convolutions and upsampling (resembling a U-shape), offered notable improvements in biomedical segmentation, paving the way for deeper study on the field and playing a great role in several studies till this date. Besides the new architecture, a great amount of data augmentations were documented in [RFB15], in which the Unet heavily relied on to achieve great results. It wasn't long before a 3 dimensional version of the UNet was published [Çiç+16] for volumetric segmentation. A variant of this 3D-UNet was implement in [Lee+17], published in 2017, accomplishing outstanding results on the SNEMI3D Connectomics Challenge. The context, procedure and hyperparameter tunning presented on this paper has been followed closely in our work to succesfully perform segmentation.
In the other hand, our proposal in this work was to introduce the adversarial training in our segmentation task, and study its suitability in order to overcome the lack of a larger labeled dataset. Generative Adversarial Networks were first introduced in 2014 [Goo+14], and since then they been actively studied and one of the main interest within the machine learning community. These models were basically formed by two neural networks, a generator and a discriminator, that played an 'adversarial game', where the generator tries to produce fake images that look real, and the discriminator tries to distinguish whether the images are real or not. Related to our work, the first Gan we need to mention is the StyleGan [KLA18]. The proposed StyleGan architecture had serveral modifications in its generator with regard to previous Gans, generating high resolution images and allowing for the first time to take control over the style of the predictions, at different levels of detail. This has proven to have many applications, specially for generating faces as the style control gives you the ability of modifying the face features as desired. Besides fake image generation, other applications were given to Gans as, for example, Image Extension. The Gan must generate an extension of the image than not only looks real but also matches the context of the original portion. The common challenge to carry out this tasks is to correlate the low features of the image with the actual content that we, as humans, percieve (a person, a house...). An outstanding approach was submited in [Tet+19], where a semantic conditioning was introduce to the generator, pulling off great improvements in generating textures and the shapes of the objects and better results than the state of the art inpainting techniques.

2.1 Brief introduction to neural networks

Neural Networks can be described in few words as machine learning desings that allow a computer to learn how to perform an specific task by analizing some inputs or training examples. Bringing them to the computer vision field, some of this tasks may be image classification, object detection, image segmentation... If we look at the structure we can see them as a representation of the human brain, neurons interconnected to other neurons which yields a network. Therefore, the smallest and main unit we can find in a Neural Network is called 'node' or 'neuron', which takes some input, and after some calculations returns and output. The simplest Neural Network we can build is the perceptron:

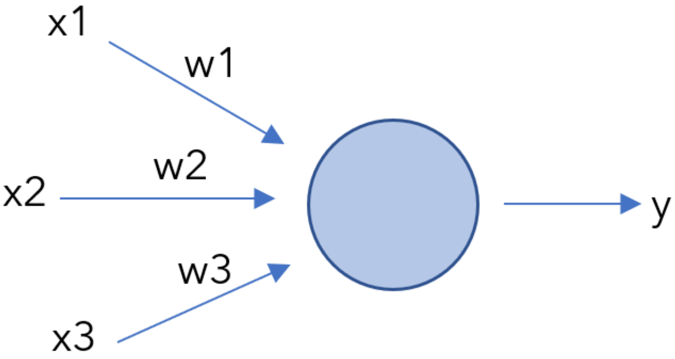

Figure 2.1: Perceptron

As we can see in the example 2.1, the node takes some inputs 'x1, x2, x3', and produces the output y. We introduce two concepts here, the weights and the bias. The weights are the only parameters of the network that are tuned or modified during the training process, and determine the importance that a neuron's output has in the next. In the other hand, the bias is a threshold that allows to control the neuron's activation. The output (y), is the result of summing up all inputs ('xi') multiplied by their respective weights ('wi'), adding a 'bias' or threshold ('b'), and then applying an activation function (ϕ), whose task is to standardize and keep the output values within a range.

$$y = \phi(\sum_{i=1}^{n} w_i x_i + b) = \phi(w^T x + b) \tag{2.1}$$

$$\phi(x) = \begin{cases} 1, if \sum wx + b \geq 0 \\ 0, if \sum wx + b < 0 \end{cases} \tag{2.2}$$

In this specific activation function we see that, when the output exceeds the given threshold, the neuron 'fires' and would pass information to the next node of the network connected. However, this linear True/False activation given on the perceptron is impractical as most real problems are non-linear. Thus, usually the output of an activation function can take any value within an specific range (non linear activation function such as sigmoid), smoothing the impact of any change in the variables to the network.

From the perceptron we can build very large and complicated neural networks in which the neurons are organised in 'columns' or layers. Every layer in beetwen the input and the ouput layers are considered 'hidden layers' 2.2.

The perceptron and the Neural Network we can see in 2.2 are both Fully Connected Networks, where every neuron of a specific layer is connected to every neuron of the next layer. Nevertheless, another networks can be constructed where the nodes of one layer are not fully

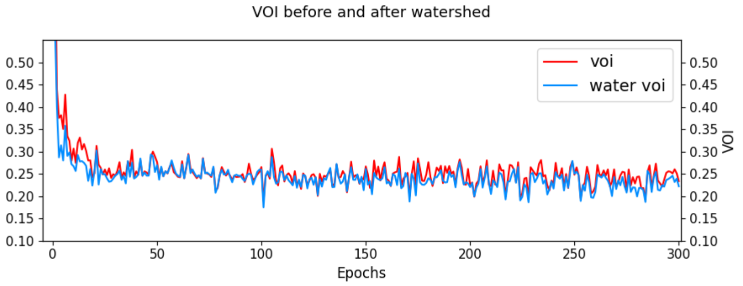

Figure 2.2: Fully connected neural network

Source: cs231n Stanford, Convolutional Neural Networks for Visual Recognition, `<https://cs231n.github.io/neural-networks-1/>`

connected to the next or even face a situtation where the nodes of a layer feed their outputs to a non-adjacent layer (skip-connections), instead of only to the following one.
To carry out the training process we will need a dataset in which the images are labeled, so the network knows 'what to learn'. Moreover, if we're start from scratch, we'll need to randomly initialize the weights and bias. The information of the images we feed to the input layer will travel throughout the network until it reaches the output. This first predictions will be poor since they depend on the weights that have been randomly initialize. It is in our best interest then to evaluate how accurate this results are by using what is called 'loss or cost function', which computes the distance between the predictions and the expected output. Ultimately, the aim is to minimize the loss or error so the neural network produces more accurate predictions. This is achieved through two different algorithms, Back-Propagation and Stochastic Gradient Descent. Back-Propagation is an algorithm that allows us to calculate or compute the gradients for the parameters in the newtork. Basically instead of going forward, it goes backwards from the output of the neural network, using the error of the prediction to compute the gradient for each weight. Then, the Stochastic Gradient Descent Algorithm ,or other optimizers, modifies the weights (optimizes the model) in order to achieve the minimum of the loss function. We can control the Step Size through a hyperparameter called Learning Rate, i.e. how much we want the weights to change with respect to the gradient or how quickly we want the net to adapt.

2.1.1 Layers

The architecture of a neural network is formed by a secuence of layers and operations, and each of them conducts an specific task. We are going to cover some of the most common layers and the mathematics behind them, as an attemp to put everything into context and facilitate the understanding of our work.

Convolutional layer

The main parameter in a convolutional layer is the filter. The filter or kernel is a matrix of weights that convolves or slides across the height and width of the input data performing dot products with the part that covers at any instance 2.3. Therefore, the height and width of the kernel must be smaller than the input's. The most common size of the kernel is 3x3.

There exist two techniques that can be applied to the convolution operation, padding and stride.

- Padding: this technique consists of padding or surrounding the edges of the input with random pixels (usually zeros, thus called zero-padding), which allows to manage the size

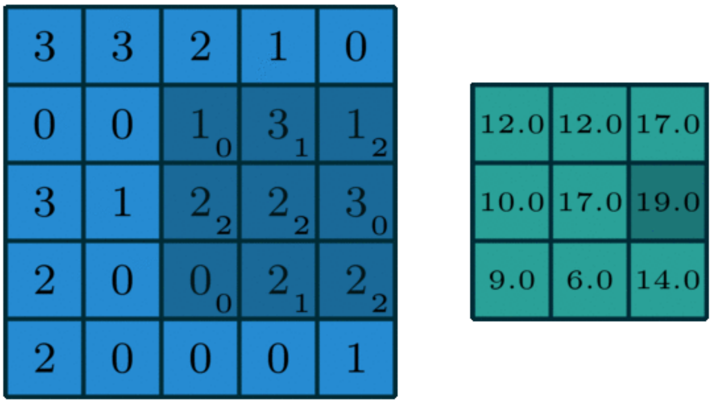

Figure 2.3: 2D convolution

Source: [DV18]

of the output image, as often we may be interest in having equal sizes for the inputs and outputs.

- Stride: the stride specifies how the kernel is going to move along the input. If the stride is 1, the kernel will slide one pixel at a time. If it is 2, the kernel will skip one pixel between convolutions. This technique is useful to reduce the size of the input.

Max pooling layer

The max pooling layer function is used to reduce the size of the input in order to get fewer parameters in the output and to decrease the chance of overfitting. Moreover, it introduces invariance against local perturbations such as shift. We have again a kernel that slides along the input, but instead of performing a dot product it simply selects the highest value, thus the one that better represents the context of the input with less information. The most common size for the kernel is 2x2, and the stride is the other editable parameter.

Transposed convolution layer

The transposed convolution, also known as up-convolution, can be seen as the opposite of a conv layer, as they upscale the input image. However, the methodoly and operations carried out are a little more complex. Let's say we have a 4x4 input image and a 3x3 kernel. Each of the input's pixels is going to be multiplied by each of the nine weights of the kernel, therefore every pixel is going to expand to a 3x3 dimension. Then, if the stride is 1, and zero padding, we're going to concatenate, skipping only one pixel, the 3x3 expansion of each pixel 4 times, which will result eventually in a 6x6 output. Stride and padding have the opposite effect here than in the convolution operations. The size of the output increases with the rise of the stride, whereas it drecreases with the growth of the padding. This layer are very convenient to produce high resolution images or to get back to the initial's resolution after downsampling with convolutional or max-pooling layers.

Batch normalization

Batch normalization [IS15] was presented in 2015 as a technique to reduce the concept of 'Internal Covariate Shift' defined as the change in the distribution of network activations due to the change in network parameters during training. However, it's been some disscusion about this statement since it was proposed and, until this day, the reason of the effectiveness remains unclear. Batch normalization provides stability and regularization, reduces the training time and makes the networks less sensitive to weight initialization. It's applied by normalizing the means and variances of each layer's input (to each mini-batch). Batch normalization can be added before or after an activation function. In the case of using Relu activation function, Batch normalization often comes before, as we have seen in more recent UNet approaches [Lee+17]. However, it remains open to discussion, and sometimes performs better after the activation function. Although it's become a common practice to incorporate batch normalization to the networks, there are others state of the art techniques, e.g. 'Instance Normalization' [UVL16], seen in recents GANs works [KLA18] [Tet+19].

ReLU activation function

The Rectified Linear Activation Function is a linear function that outputs a zero if the input is negative, otherwise the output will be the same as the input 5.8. The ReLU has replace traditional activation functions, as sigmoid or hyperbolic tangent, in a large amount of networks, as it helps to overcome the vanishing gradient problem*.

$$ReLU = \begin{cases} x, \; if \; x > 0 \\ 0, \; if \; x \leq 0 \end{cases} \tag{2.3}$$

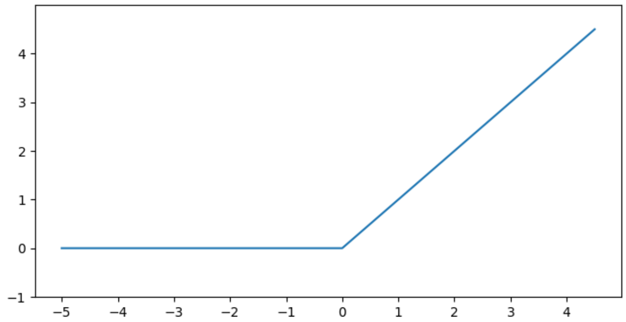

Figure 2.4: ReLU activation function

During the training of a neural network, a gradient is computed from the error of the predictions in order to update the weights of the layers and improve the model. This gradient propagates backwards from the last to the first layer of the network. Throughout this backpropagtion, the gradient diminishes and by the time it reaches the input, it might be so small that won't affect the weights, hence this layers won't 'learn'.

2.2 Brief introduction to GANs: Generative Adversarial Networks

Generative Adversarial Networks were proposed in [Goo+14] as a new framework to estimate generative models via an adversarial process. GANs are models formed by two submodels, a generator and a discriminator. The basic idea of this framework is that the generator produces some fake images from a domain and the discriminator has to decide whether the images are fake or not (binary classification, 1 for real 0 for fake). Both, the generator and discriminator, are trained simultaneously, thus the discriminator tries to maximize its ability to classify the images as real or fake, whereas the generator tries to reduce the probability that the discriminator can detect that an image is fake. This adversarial situation is known as two-player minmax game.

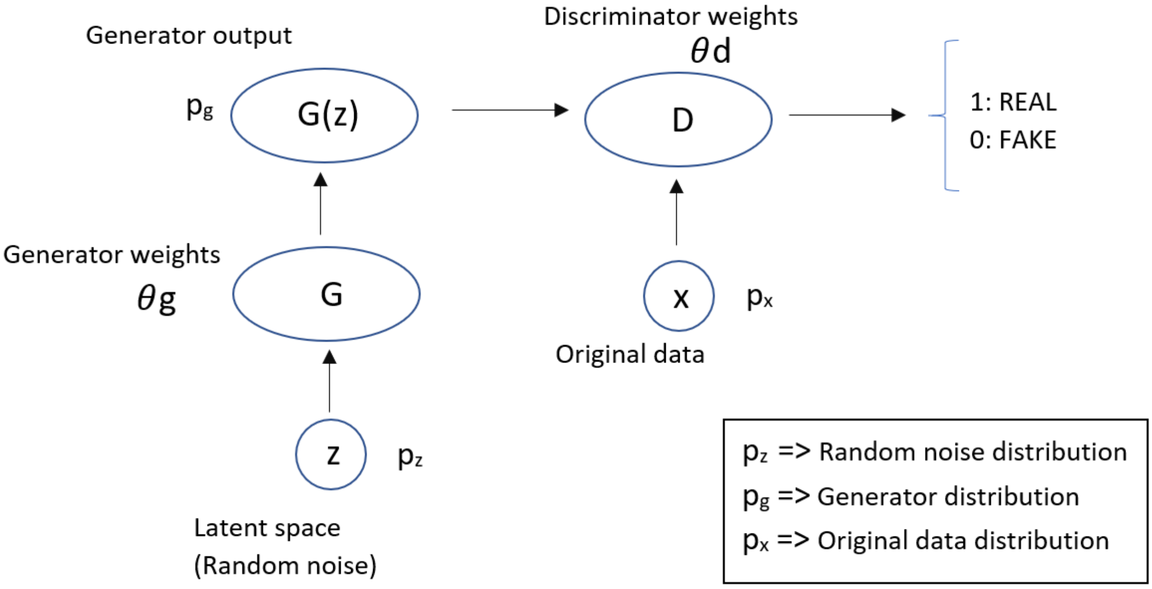

Figure 2.5: GAN architecture.

Following the figure 2.5, the generator takes some noise, a random vector from what is called latent space, and computes some outputs. Both, the completely random first output from the generator and the original data, arrive to the discriminator, whose task is to classify in real (1) or fake (0) its inputs. This process is carried out repeatedly, while updating first the discriminator and secondly the generator, until the generator manages to learn a distribution that is close enough to the one from the original data. This process can be written down with the Value Function 2.4, where D(x) represents the probability that real data is real, D(G(z)) represents the probability that fake data from the generator is real (thus 1-D(G(z)) is the probability that generator's data is fake) and E represents the expected values from the real data and the generator's input.

$$\min_{G} \max_{D} V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}[log D(x)] + \mathbb{E}_{z \sim p_z(z)}[log(1 - D(G(z)))] \tag{2.4}$$

We can see how the Value Function 2.4 and the loss functions of BCELoss 3.3 are quite similar. It's important to point out that the generator can not affect the first term of the equation, thus minimazing the loss for the generator means minimizing the second term. Moreover, as inidicated in [Goo+14], at early stage the task for the discriminator may be so easy that the second term of the equation saturates. In that case, the second term can be replaced for $log(D(G(z)))$, which now means that the generator will try to maximize this term (the probability that a fake image is considered real by the discriminator).
Is quite common to find that both submodels from a GAN are CNNs. The ideal situation is to reach the Nash Equilibria, where the generator computes perfect images and the discriminator is just randomly guessing ($\sim$ 50%).

3 Methodology

3.1 UNet

The neural network that we use to carry out the segmentation is the UNet [RFB15] with few modifications.

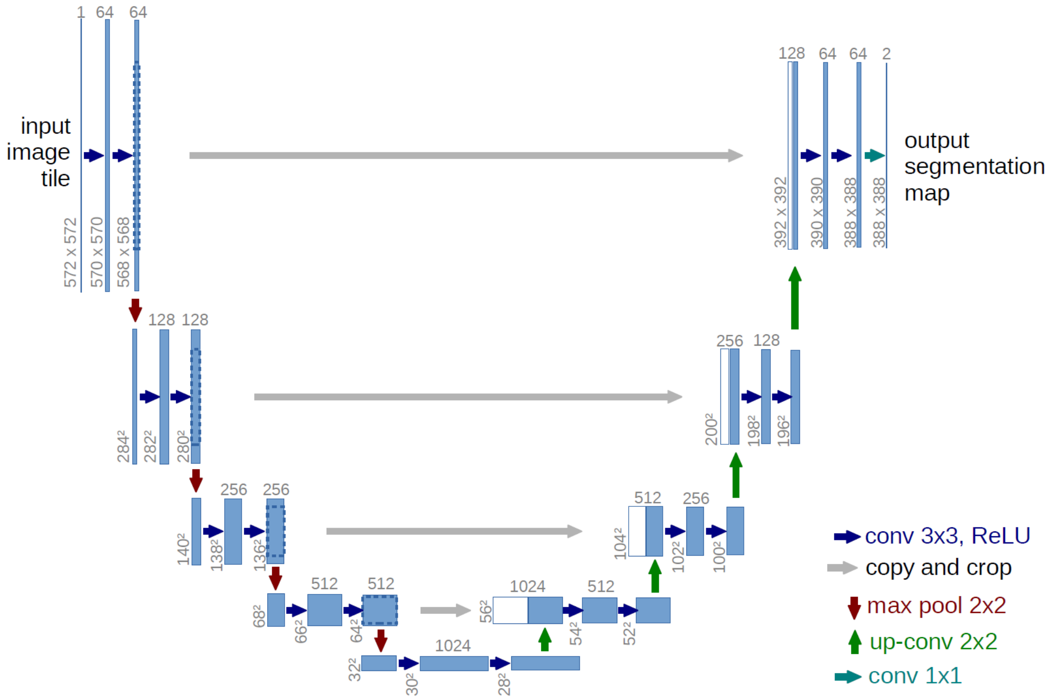

Figure 3.1: UNet architecture

Source: [RFB15]

If we look at 3.1 we can divide the architecture of the UNet in two sections, the encoder or contraction path (left side), and the decoder or expansion path (right side). The encoder is a sequence of convolutional and max pooling layers, where the size of the inputs is gradually reduced by half whereas the number of channels or feature maps is doubled. In the other hand, the decoder is the symmetric expanding path of the encoder, formed by a sequence of convolutional and transposed convolutional layers. In each step, the size of the images is doubled and the number of feature maps reduced by half.
In the enconder, the information or context of the image is extracted, while in the decoder,

the outputs of the up-convolutions are concatenated with the extracted features from the contracting path to help with the localization.This technique of concatenating the output of some layer to other a number of layers behind is known as 'skip connections', and have proven to increase performance, specially in the predictions of medical image segmentation. Even though there's still some discussion about why skip connections improve training, they are widely used, particularly when the network presents an encoder/decoder architecture, where the information lost troughout the encoder because of downsampling can be useful for the decoder in terms of spatial localization.

Each of the blue blocks that we can see in the figure 3.1 are a sequence of a 3x3 convolutional layer, batch normalization and Relu activation later (note that the batch normalization was not included in the original paper, but considering the benefits of this normalization we have added it to our implementation. The batch normalization was first published two months after the release of the UNet paper, and since then it has been included in a great variety of architectures). Besides the mentioned layers, we can find max pooling operations in the encoder and transpose convolutions in the decoder.

The convolution operations are performed in our architecture with a kernel size of 3 and both parameters, the Stride and Padding, are set to 1, which is very convinient, because it allows to keep the information with more context in the middle of the images. Moreover, with this combination we maintain the input's size, thus is easier to keep record of the size of the features throughout the network, and help when applying the skip connections, where the sizes of the concatenated features (height and width) must be equal. Regarding max pooling, both the kernel size and the stride are 2, which results in reducing the input by half. We find these same two values in the transposed convolutions, which with the addition of 0 padding results in an output size that is twice the input's.

Regarding the activation function, we have seen in previous work [Lee+17] that the ReLU activation function has been replaced in favour of the ELU activation function. The diference lies in the negative side, where instead of giving a 0 value for the negative inputs, it smoothly decreases towards a constant that is given (that must be positive).

$$ELU = \begin{cases} x & \text{if} \quad x > 0 \\ \alpha(e^x - 1), & \text{if} \quad x \leq 0 \end{cases} \tag{3.1}$$

However, for our architecture the results have shown that the there's not significant difference.

3.2 Segmentation

To perform our segmentation experiments we have a $(5\mu m)^3$ volume of Electron Microscopy (EM) Neuron Images (1250px * 1250px * 125px) of adult 'Drosophila melanogaster' brain, imaged with serial section Transmission Electron Microscopy (ssTEM). This dataset belongs to the CREMI challenge (Circuit Reconstruction from Electron Microscopy Images) [Fun+], where a competition was set up to evaluate different algorithms for automatic reconstruction of neurons and neural connectivity.The volumen is composed of nonisotropic voxels. A initial segmentation of the volume is also provided.

3.2.1 Affinity maps

The first task is to prepare the dataset for the training. From the segmentation images we can extract the targets for our network. Our goal is to get the affinity maps, that are binary masks where, a pixel whose neighbours (surrounding pixels) belong to the same instance takes the value 1, and if they don't it takes the value 0. So i.e. the expected result is to obtain

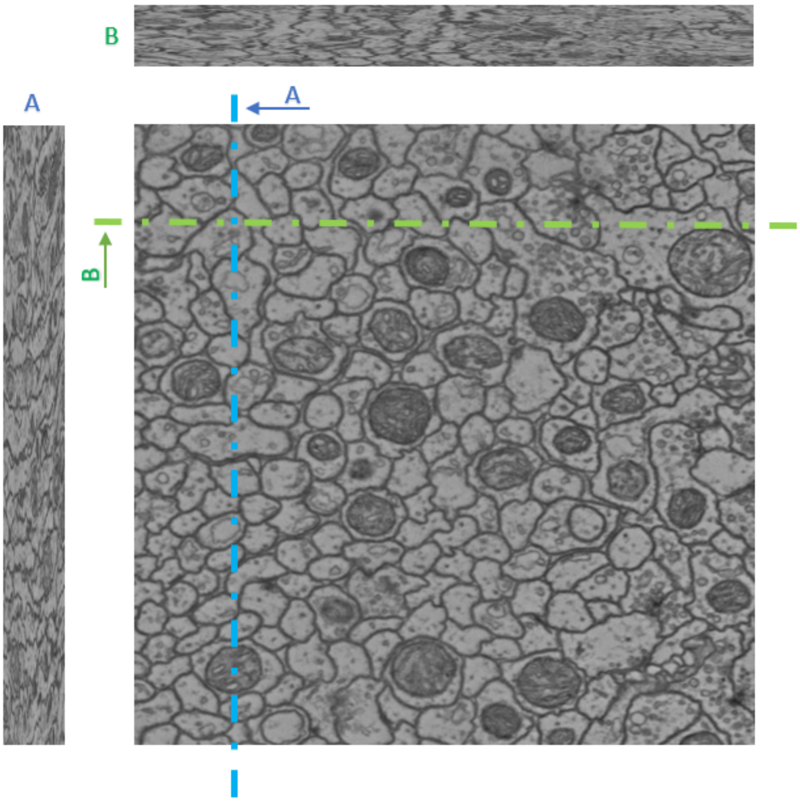

Figure 3.2: EM volume crops in three directions. A display of all the slices in the xy direction (1250x1250) can be found in `https://youtu.be/UvCIpkwMjIkl`

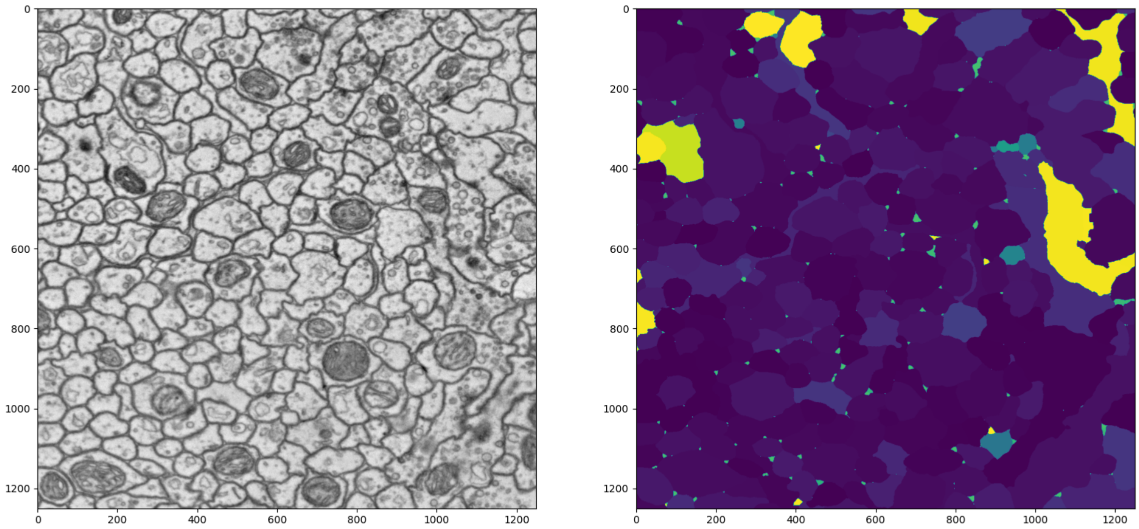

Figure 3.3: EM volume's slice and segmentation

binary targets where the neurons are white and the boundaries of the cells are black.
The pixels of the provided segmentation masks that belong to the same neuron share the same value. Therefore, applying the 3x3 kernel shown in 3.2 , followed by a Inverse-Binary Thresholding, we get the said affinity maps. Once we get them, it might be a good practice to dilate the image, as widen the boundaries ease the segmentation task.

$$kernel = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix} \tag{3.2}$$

Nevertheless, a more exhaustive processing has been studied and it's available in the Pytorch Connectomics Documentation [Gro19], that we adopt as we pursue best possible performance.

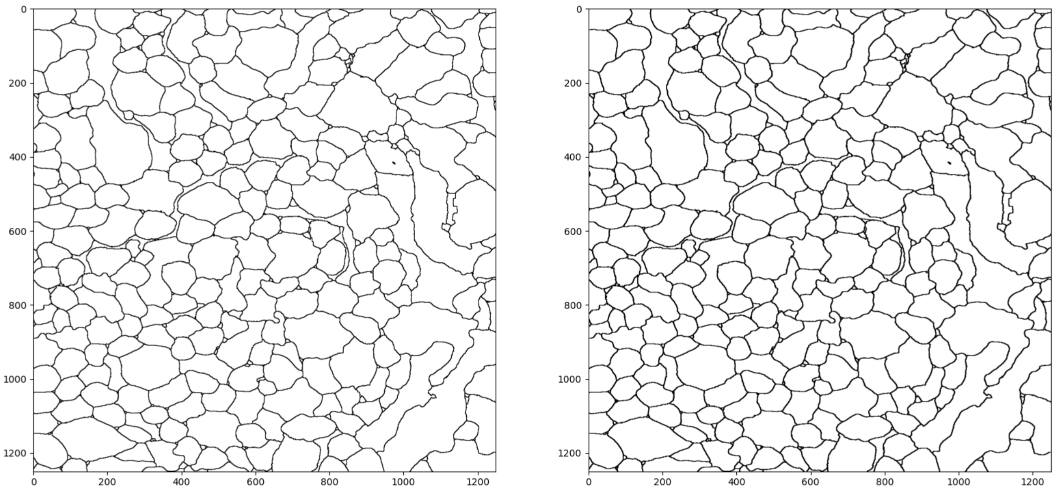

Figure 3.4: Affinity maps for segmentation. On the left the outcome of our proposed filtering, on the right the outcome of the Connectomics processing. We can notice that some dilation has been performed on the right image as the boundaries are wider.

3.2.2 Data augmentation

We use the first 100 slices as training set, the next 5 for validation, and the rest for testing. The amount of data we have is not large enough to achieve great results, thus we make use of data augmentations to increase our training set:

- Resize and randomly crop each of the slices. This way we can get from every 1250x1250 slice many different samples of smaller size.
- Randomly flip horizontally and vertically
- Randomly rotate the samples
- Blur the samples by a Gaussian function. Reproducing some imperfections present in the dataset, in this case blurred slices, has proved to be effective and helps with performance.
- Add Gaussian noise to the samples.

Besides varying the intensity or weight the blur and noise have in the samples, they're also applied randomly in a way that one input may have been blurred and added noise at the same time whereas other only blurred or maybe none of them. In addition, in some of the experiments we worked with a non-fixed dataset, so all the transformations were applied

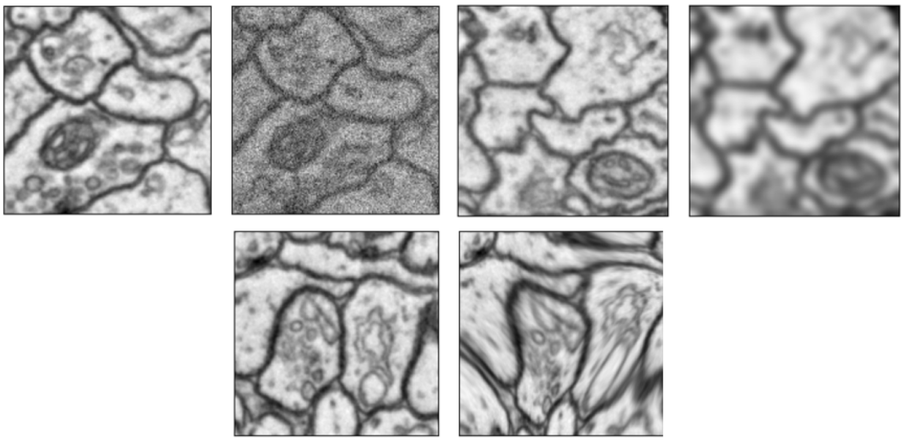

Figure 3.5: Examples of random crops with some of our data augmetations. On top Gaussian Noise and Blur, on bottom the elastic deformation that was dropped out because of the notable increase of running time.

during training, which means that the training samples were different every epoch, avoiding overfitting and saving memory.

Other data augmentation techniques as brightness modification or missing sections carried out in [Lee+17] might be worth considering. On the other hand, elastic deformation, that seems to play a big role in data augmentation for Biomedical Images, has been ruled out as training time became significantly larger.

3.2.3 Batches

Once the dataset is ready, the images are organised in batches before feeding the network. This means that a specific number of samples will be propagated through the network simultaneously instead of one at a time. This is beneficial as it requires less memory and reduces training time (less propagations means fewer weight updates). Moreover, this hyperparameter plays a mayor role in the generalization gap, which can be defined as the difference between performance on training data and test or unseen data. Large batch sizes may lead to poor generalization, whereas small sizes converge faster but might not reach the optima. In order to achieve best results, batch size and other hyperparameters are set following the previous works [RFB15],[Lee+17].

3.2.4 2D and 3D UNet models

Besides the batch size, we can modify the dimensionality of the inputs. Until now, we have considered a 2D implementation of the UNet. Hence the inputs to this model are random crops from the 125 slices that make up the volume, and have no correlation. However, shortly after UNet, a 3D implementation of this architecture was published [Çiç+16]. This model takes volumes (height, width and depth) as inputs, thus preserves the spatial information or context and enables new training approaches. However, the drawback is that more resources are needed and larger computation time.

We run a 3D-UNet 3.7 as proposed in [Lee+17] which, besides performing the pertinent 3d-convolutions instead of 2d, differs from our 2D model in others aspects, i.e., a smaller number of features channels throughout the network, the use of the Elu activation function instead of ReLU, and, in addition to the former encoder-decoder skip connections, the inclusion of

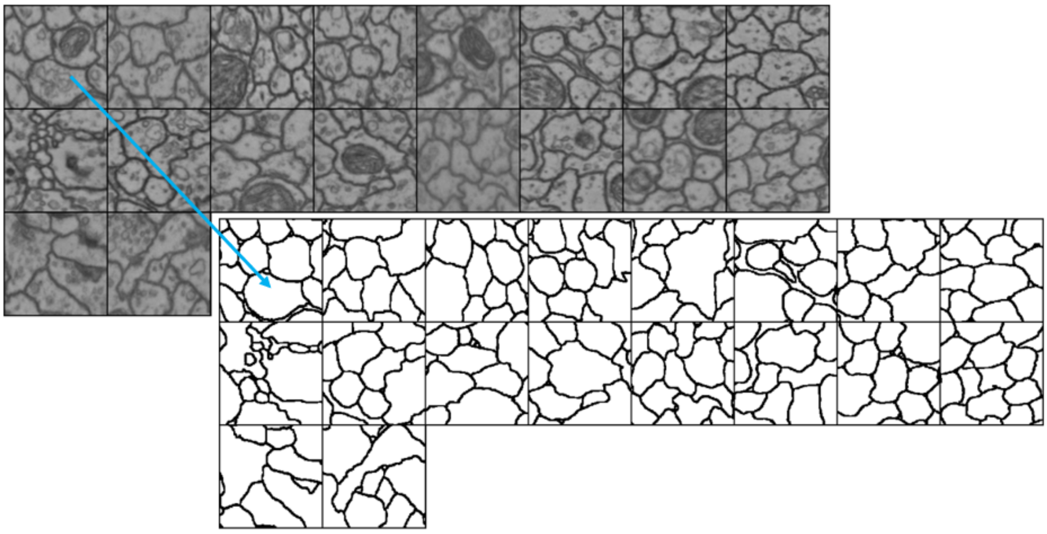

Figure 3.6: Training batch of 18 samples and their respective targets.

residual skip connections to each module, previously applied in [QHJ16]. Moreover, the skip connections between the encoder and the decoder are performed by summation, whereas in the 2d model, the encoder's features are 'concatenated' to the decoder's.

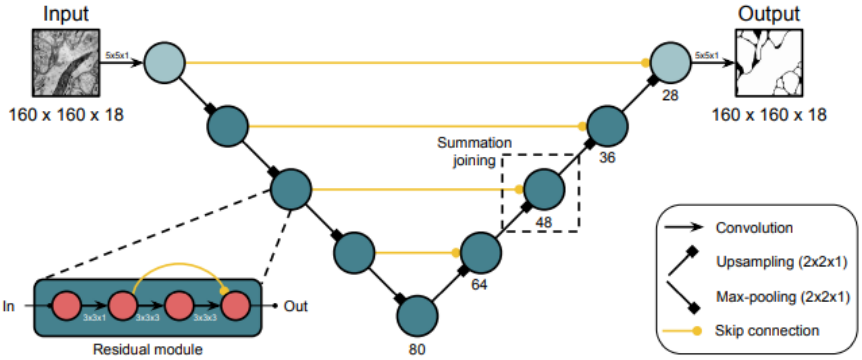

Figure 3.7: 3D-UNet architecture.

Source: [Lee+17]

3.2.5 Loss function: BCE

Throughout the training, the cost or loss function is going to calculate the error of the predictions, i.e., how different are the predictions from the actual targets of the network. Afterwards, these values are used to compute the gradients and update the weights, therefore the loss function plays a mayor role in the mode's learning. There are several types of functions, and depending on the context, some may be more appropiate than others. The choosen function is the Binary Cross Entropy with Logits Loss (BCEWithLogitsLoss). This loss function is the concatenation of a Sigmoid layer and the BCELoss, thus the inputs to the BCELoss are values within 0-1 range.

$$BCELoss = 1/N \sum_{i=1}^{N} -(y_i * log(p_i) + (1 - y_i) * log(1 - p_i)) \tag{3.3}$$

In the equation 3.3, y_i is the class and p_i the predicted propability. The output is the negative mean of the log of the corrected probabilities, which can be defined as the probability that something belongs to its original class. By default all predicted probabilities are referred to class 1. We have two classes, the boundaries and the background, hence two terms can be distinguished in the ecuation. If the actual class is 1, the second term of the equation is nule, and the predicted and corrected probability are the same (p_i). If the class is 0, the first term of the ecuation is nule, and the corrected probability is one minus the predicted probability ($1 - p_i$).

3.2.6 Optimizer: Adam

Following previous works, the algorithm or optimizer used to update the weights is the Adam optimizer [KB15], which has proven to be very effective for many neural networks with different architectures. This algorithm combines the effect of gradient descent with momentum, together with gradient descent with RMSprop. The Adam optimizer computes individual learning rates for each weight, using estimations of first (mean) and second (uncentered variance) moments of gradient (momentum accelerates the convergence towards the minima and reduces the fluctuation to the irrelevant direction). Therefore, besides the learning rate, two more hyperparameters are set, β_1 and β_2, for the estimations of mean and uncentered variance respectively .

3.2.7 Accuracy

In order to quantitatively measure how good the model performs and keep records of the training and results, three different methods are used:

- Pixel by pixel: every pixel of the predictions is compared with its target. The result is computed as the number of right predicted pixels by the total number of pixels in the image. However, this method has its limitations, as not all pixels have the same weight in the image composition. For example, one image might have 1000 wrong, leading to narrower boundaries and an accuracy of 94% , whereas another image might have only 100 wrong, 96% of accuracy, but the pixels leave an open boundary, thus two different cells are consider the same.
- Variation of information (Voi): this method measures the segmentation quality through the computation of the conditional entropies. If X is the ground-truth segmentation and Y the prediction from the model, the first term of the ecuation 3.4 measures the amount of under-segmentation ($H(X|Y)$), and the second term the amount of over-segmentation ($H(Y|X)$). This technique is more reliable than the pixel by pixel method to analyze the segmentation results.

$$VI(X,Y) = H(X|Y) + H(Y|X) \tag{3.4}$$

- Adapted Rand Error (ARand): this method measures the similarity between two data clusterings. It is computed as 3.5, where 'p' (precision) is the number of pairs of pixels that have the same label in the test label image and in the true image, divided by the number in the test image, and 'rec' (recall), is the number of pairs of pixels that have the same label in the test label image and in the true image, divided by the number in the true image.

$$ARand = 1 - \frac{2pr}{p + r} \tag{3.5}$$

3.2.8 Post-processing

Once the boundaries of the cells are predicted, the last step is to perform the segmentation. Our goal is to classify each of the cells as a different object in the image, which is known as instance segmentation. This task is done with the help of the module 'measure' from the Scikit-Image Algorithm Collection [Wal+14]. The function 'label' from this module allows us to label each of the cells as a different instance. The algorithm behind follows the same neighbouring logic than the one used to generate the affinity maps to behave as targets of the network. Two pixels belong to the same instance if they are neighbours and share the same value. The pixels with 0 value (boundaries) are consider as background, hence won't be processed and labeled. This function assigns a different value to each of the detected instances.

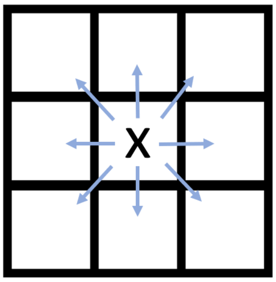

Figure 3.8: Neighbours of pixel X.

One recurrent error that we have confronted during the training of the network is that some of the boundaries of the predicted cells are left open. This would translate into inaccurate segmentations, as two cells with an open path in their borders will be labeled as one. Therefore, following the common practice, and the steps of [Lee+17] we introduced the 'Watershed' algorithm in our post-processing. The way we proceed is:

- First we take the outputs of the network through a sigmoid function to keep the values of the pixels within the range [0,1]. These are the probability maps, where we can understand each pixel's value as its probability to be white (value = 1). Next we apply a threshold (experimentally determined), to transform the probability maps into the boundary maps (binary images). 3.9

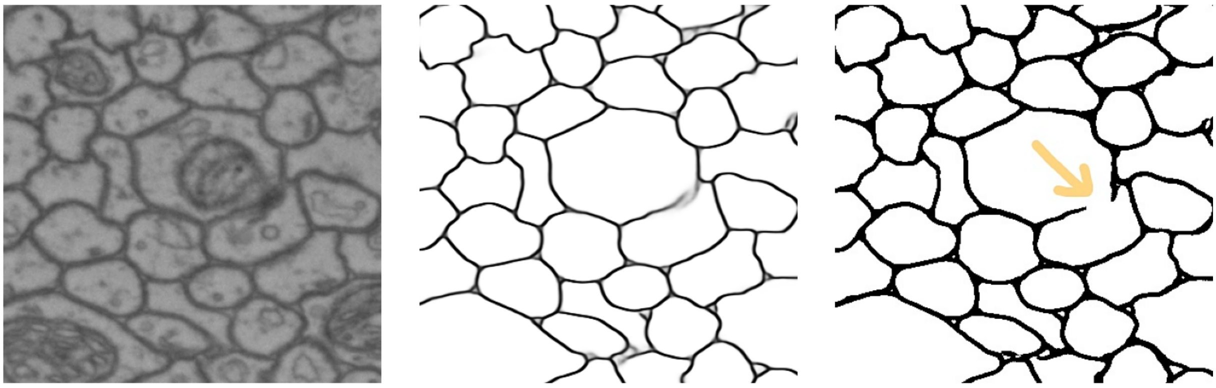

Figure 3.9: From left to right: crop of a nn input, probability map and boundary map after threshold with open border

- Now that we have the boundary maps we apply the watershed algorithm in order to avoid the undersegmentation that may occur if some boundaries were left open. The watershed algorithm reads the images as a topography map, the greater the pixel's value is, the higher. First it computes the euclidean distances, and creates a map where each of the former 1 value pixels now represents the distances to their closest boundary. If we invert the image, and keeping in mind the topographical interpretation, the cells would be basins, where their lowest point (further point from the boundaries) would be a minima, and the highest level would be the plane composed of the boundaries. Next, the algorithm 'floods' the basins, until the 'liquid' of the basins reach the surface and meet with the other's. That meeting point is what is known as 'wathersed line', and as we can see in 3.10, it roughly matches the boundary maps. Note that the number of minimas is a parameter that we can choose and tune to improve the results (experimentally determined).

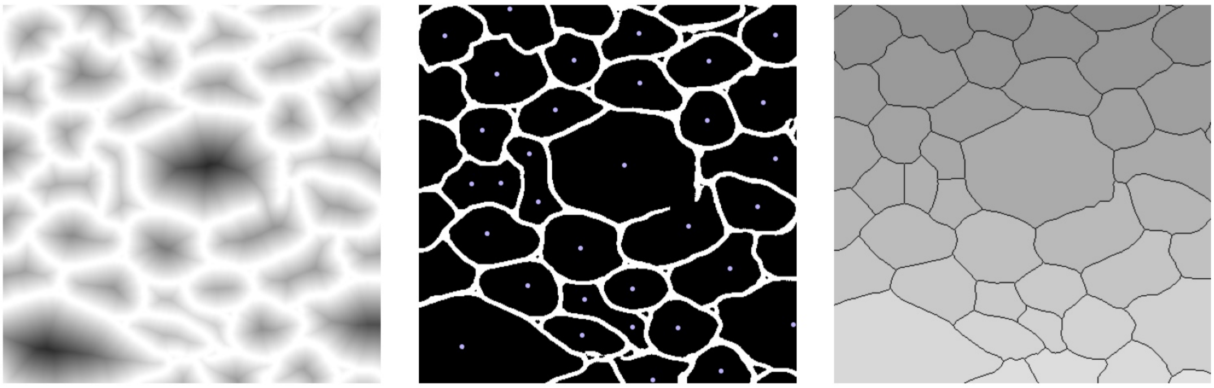

Figure 3.10: From left to right: euclidean distances (the darker the pixel the further to the boundary), computed minimas and watershed lines

- The watershed line is compared with the boundaries and a new map is created with the segments of the watershed line that don't match with the former boundaries. This segments are the potential new boundary segments. To decide wether they are added to the boundary map or not, we subtract their coordenates and for each segment we compute the mean value of the pixels that share the same coordinates on the probability map. Is fairly common to find that the probability where an oppening in a boundary happened is not close to 1 (meaning white pixel). Thus, we apply a threshold (experimentally determined) to reject the segments whose mean is close to the 1 value.

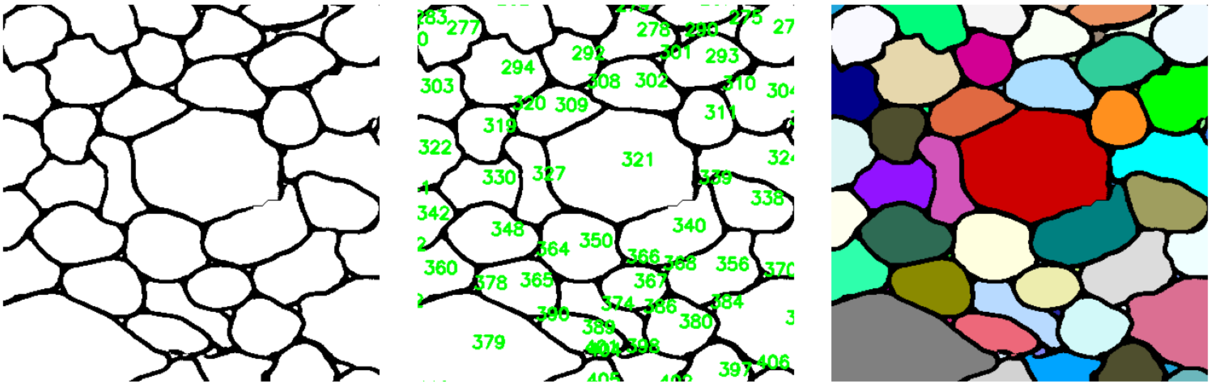

Figure 3.11: From left to right: boundary map with the segment computed through watershed, labels of the crop and colored image

- Once we have the chosen segments we add them to the boundary map and proceed to the labeling of each cell as explained. 3.11

In order to better analyse the results, the cells are displayed colored. The ideal situation would be to apply the 4-color theorem but this first approach was dropped out as the computing time was extraordinary. Because of that, we may encounter neighbour cells sharing the same color even though they are well-labeled.

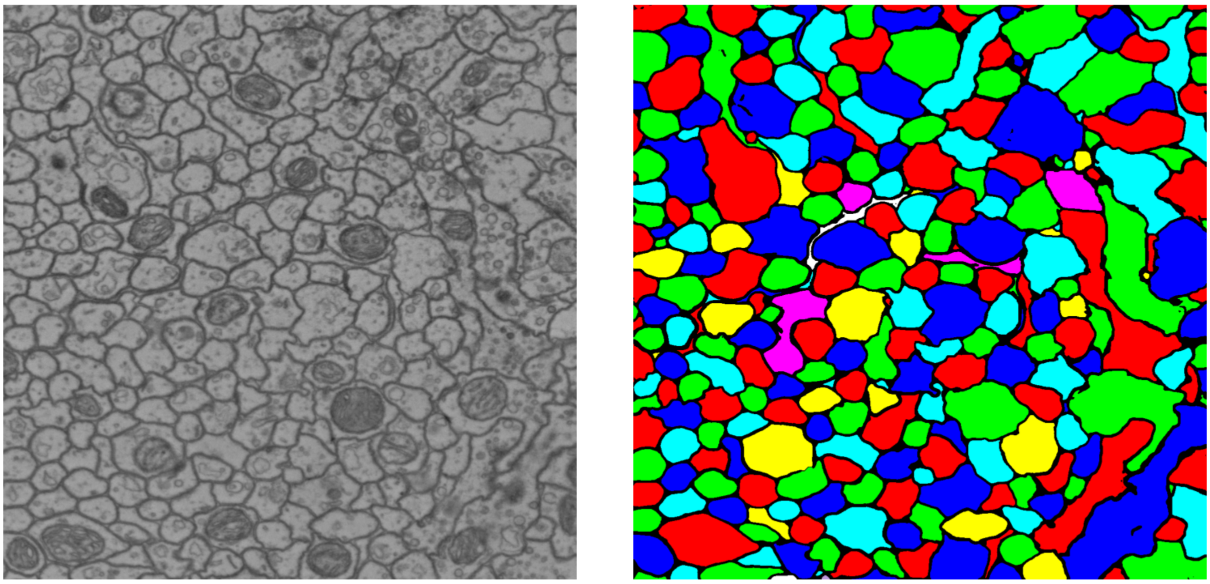

Figure 3.12: Original size input and post-processed output with 4-color theorem (NOTE: => 1). The image used for the post-processing explanation belongs to the training set, so it's seen data; 2). We have not use the 4,5..-color theorem in the next experiments to color the images because of extraordinary computational cost.)

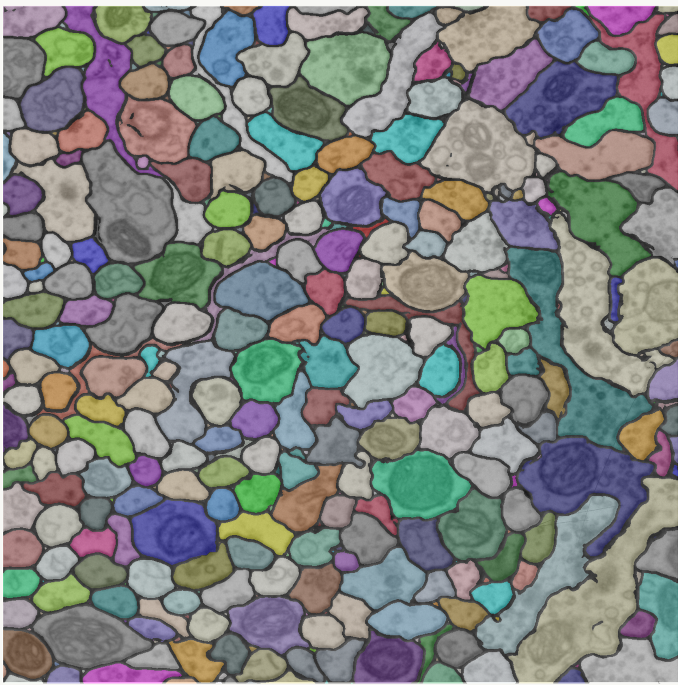

Figure 3.13: Segmentation overlay on the original image. The colors of this segmentation has ben applied cyclically from a limited palette, thus some nearby cells correctly labeled may share the same tone.

3.2.9 First contact with GANs: StyleGan

The StyleGan [KLA18] is a state of the art GAN model that has proven to produce outstanding high quality realistic photos. Besides that, is well-known because it proposes several modifications to the generator's architecture, which allows control over the style of the generated images. Consequently, is has been widely used to generate human faces, as the control over the style allows to modify the face features with ease. Taking a look at the architecture 3.14, some notorious modifications are: the addition of an eight fully connected layers Mapping network, one extra latent space (W), the addition of Adaptive Instance Normalization (AdaIN) which allows the style control, the addition of noise to each block of the network and the removal of the latent space as input of the generator in favour of a constant.

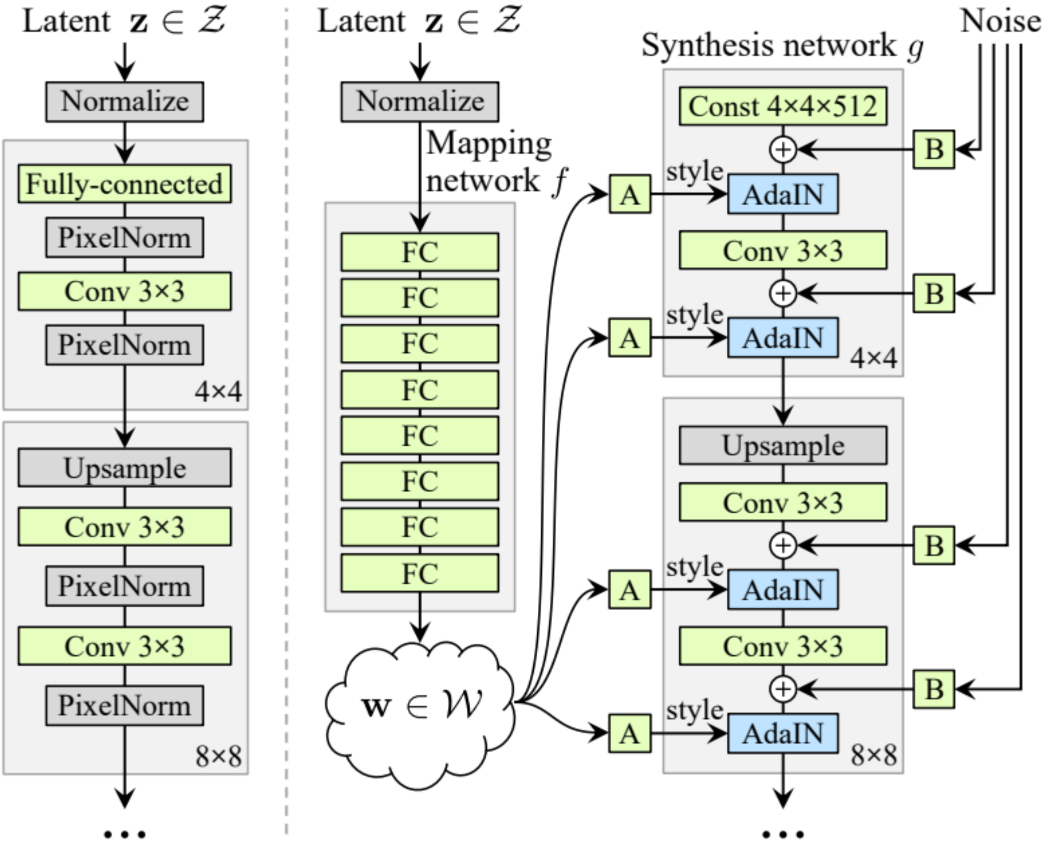

Figure 3.14: Traditional GAN's generator vs Style GAN's.

Source: [KLA18]

Even though being able to control the style may not seem to be an key thing for us at first sight, we carried out some training to see how it reacted and whether the results were promising or not. Moreover, keeping in mind that our dataset is very limited, if the outputs of the generator are quite similar to the real dataset, the StyleGan might be a valuable source for Data Augmentation. We can see the similarity in 3.15, where the right image is the ouput of the StyleGan with slightly contrast correction.

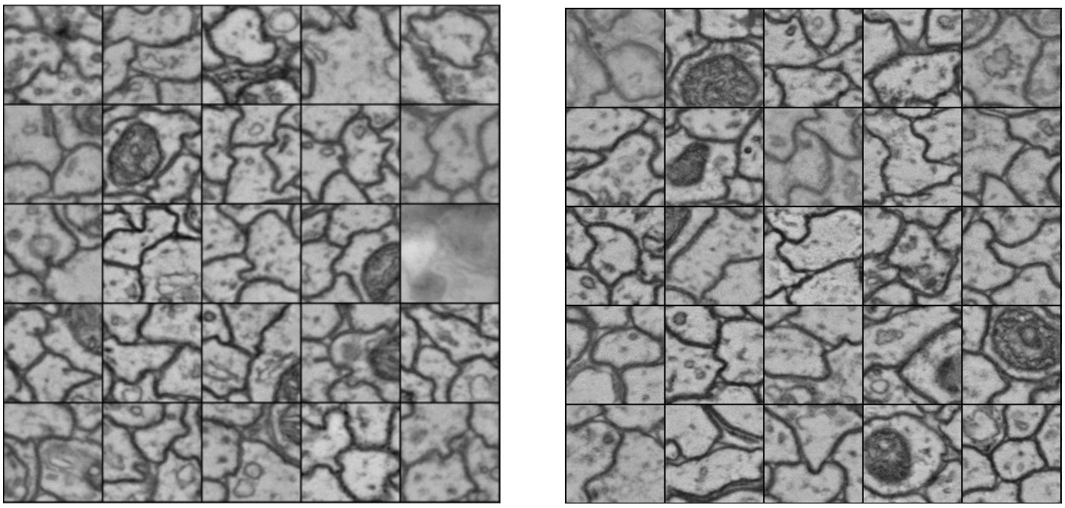

Figure 3.15: Original dataset (left) vs StyleGan results (right).

3.2.10 Transfer learning and GANs for Image Extension

Transfer learning is a machine learning technique that consists in saving the learned features from some task and using them for another purpose relatively related. Is commonly used both in situations where little data is available, or when the dataset is really large, but it would take a lot of time and resources to train. Therefore, transfer learning can help in reducing training time and improving the accuracy and performance of the model.

This technique might be really useful for our task given the fact that we have few labeled samples. The idea was to find a suitable gan to train with our dataset in an 'unserpervised' (self-supervision) way to perform some secundary job, hoping that the extracted features throughout the training were helpful for the segmentation. For this purpose GAN for Image Extension [Tet+19] was chosen.

Boundless: GAN for Image Extension

In [Tet+19] a new GAN architecture is proposed to compute image extension. The objective of this gan is to reach the point where it's not feasible to distinguish the extended portion of the image from the untreated one, acomplishing accurate image extension even thought the region to be filled is only surrounded by original data in one of the directions. It has proven to generate coherent semantics and better textures that the state of the art image extension gans and other proposed techniques.

The architecture of the model as well as the inputs of the generator and discriminator can be seen in 3.16, where:

- M: Mask of the image. The pixel values of the mask that correspond with the image portion to be filled are 1's, all the other pixels take the value 0.
- x: Real image without any processing
- z: Masked image. It is the original image but the value of the pixels that correspond with the portion to predict are 0's.
- $\hat{x}$: Masked image but with the part to predict filled with the generator's output.

One of the reasons of the model's success is the introduction of a semantic conditioned discriminator. Instead of feeding the discriminator directly with the generator's output, it takes $\hat{x}$ as input, in which the pixels of the generators output that weren't meant to be filled

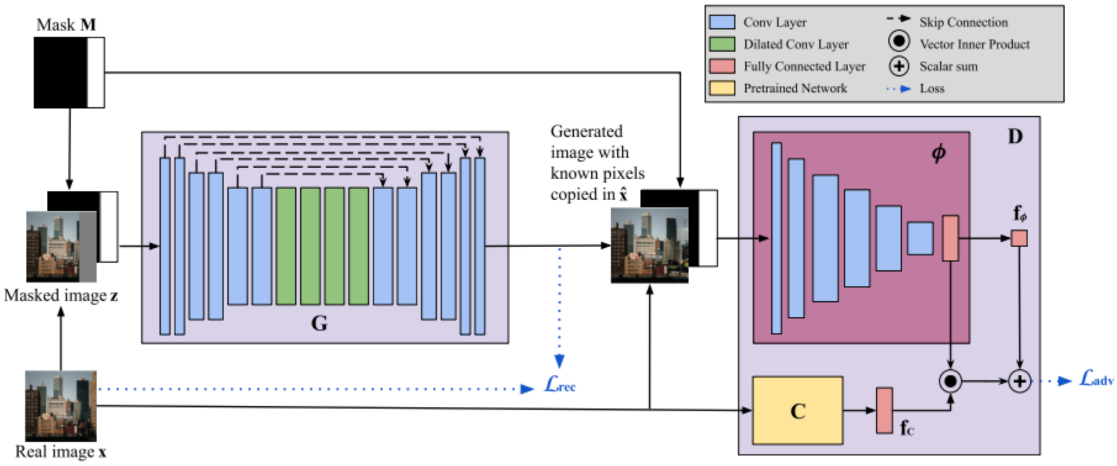

Figure 3.16: Model architecture of Boundless GAN for Image Extension.

Source: [Tet+19]

are replaced by the original ones. This, as well as the input of the mask M to the discriminator, gives it advantage in the adversarial game, as it 'knows' which section of the image to focus in to detect any change in the distribution. Moreover, looking forward solving the loss of quality that occurs when moving further from the real content of the image, a second form of conditioning is added, that consists in including a pretrained network (Inception V3) trained on ImageNet.
Concerning the generator, the architecture can be splitted in two, similarly to the UNet, in enconder and decoder. There are 16 Gated Convolutional layers [Yu+18] which are a strong way to extract features, where the input is used to compute gating values (sigmoid after convolution operation), and the output is the instance normalization of the multiplication of the learned features (elu after convolution operation) by the gating values. After the training on image extension, the learned features by the generator are used looking forward improving the segmentation task. However, not all of the learned features may be useful. Therefore, several experiments will take place with different number of 'frozen layers'. This means that, at the time of initializing weights, some of the layers won't require gradients (frozen layers), hence those layer's weights won't be modified during training. It's common practice in transfer learning to initialize the layers of the enconder with the pre-trained weights, frezing those layers, and randomly initialize the rest.

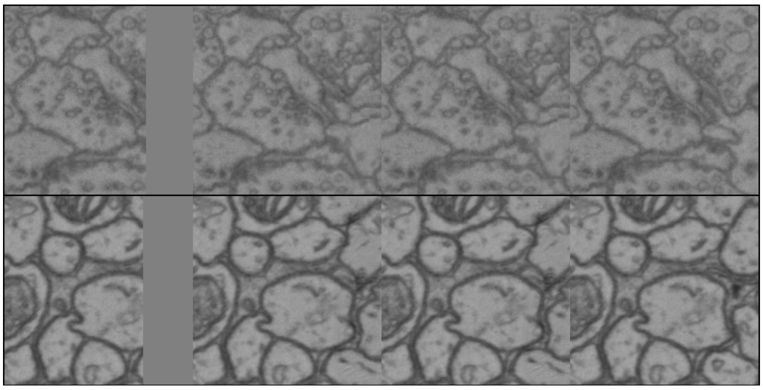

Figure 3.17: Image Extension with mask on the right edge. From left to right, masked image (z), generator's output, z + (M*generator's output), groundtruth.

4 Experiments

4.1 Dataset

To carry out the experiments we divide our (1250px * 1250px * 125px) dataset in 3 different groups. The first 100 slices are for training, the next 5 for validation and tuning, and the last 20 for testing.

4.2 StyleGAN evaluation

The StyleGAN [KLA18] training was chosen with the objective of testing and evaluating the suitability of our dataset for adversarial training. Furthermore, due to the limited amount of training data, great results in generating 'fake' images of cells could turn the StyleGan into a valuable source for data augmentation. For the training we adopted the [Seo18] Pytorch implementation of the Style Gan, and reshaped the architecture for grayscale images. All the hyperparameters were left as in the original implementation. The input was the training dataset (first 100 slices), randomly cropped to size 128, flipped and rotated, and normalize to mean and std 0.5, obtaining the amount of 20000 crops. We run the experiment for 25000 iterations, with a batch size of 16. Regarding post-processing, the outputs' contrast was slightly increased to better resemble the real images.

4.3 2D UNet and 3D UNet for segmentation

The UNet architecture was selected due to great amount of previous documented work and its suitability and exceptional results in segmentation tasks. Keeping in mind that the main goal is to introduce the adversarial training in the segmentation duty, looking forward improving the results, the UNet training sets the perfect background to compare with. In this experiment we train a two dimensional and a three dimensional UNet, and compare the results between both of them. For the 2D UNet architecture we have tried to implement as close as possible as described in [RFB15], with the addition of batch normalization after each convolution. In the other hand, we adopted the 3D UNet implementation from [Gro19].

4.3.1 Dataset

The traininig was carried out with the training dataset. For the 2D UNet we used batches of 18 random crops of size 160x160, whereas for the 3D UNet, minibatches of size 1, with a volume of size 18x160x160. The dataset was resized to 900x900 before cropping. Regarding data augmentation, we applied random crops, horizontal and vertical flips, random

rotations ($angles = \{0, 90, 180, 270\}$), blur ($kernel\_size = 11$), following a gaussian distribution ($mean = 0.01$, $std = 3$), and noise, with a gaussian distribution of ($mean = 0.0$, $std = 0.2$). Considering that the 3D-Unet takes volumes as inputs, the augmentations that supposed change of position, i.e. rotation and flipping, where applied for the whole volume, whereas the blurring and the noise where randomly applied within the volume.
The targets of the networks were the affinity maps mentioned in 3.2.1, binary images computed using the Pytorch Connectomics toolbox [Gro19].

4.3.2 Training procedures

The networks weights were initialized by 'Kaiming Initialization'. Both were trained using the Binary Cross-Entropy Loss. We used the Adam optimizer, with $\alpha = 0.01$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 0.01$. Setting up the 'ReduceLROnPlateau' scheduler, the step size was reduced by half, up to four times, when the validation loss stopped decreasing for 5 consecutive epochs.

4.3.3 Post-processing

First, a sigmoid function was applied to the ouputs of the network. Next, the probability maps ($range = [0, 1]$) were thresholded at $p > 0.81$ (experimentally computed), an the watershed algorithm was applied in order to close some open boundaries and improve the overall segmentation quality. Finally, the labels are assigned as described in 3.2.8.

4.4 Image Extension: mask's shape/position influence

The purpose of this experiments was to evaluate the impact of the mask's shape and position in the image extension process, in terms of quality of the prediction of the masked area, and how this conditioned the segmentation performance when the network was initialized with the pretrained weights. For this task we embraced the pytorch implementation [Ish19] of the Boundless paper [Tet+19] and adapted the code for 1 channel images (grayscale). (Note that in [Tet+19] the architecture InceptionV3, pretrained on ImageNet (3 channel images), is used to better extract the features with a low computational cost. Therefore the pretrained weights were computed from a 3 input channel first layer. Thus, as we modified the first layer of InceptionV3 to take as input 1 channel images, we loaded the ImageNet pretrained weights of the first layer from only one of the channels.)

4.4.1 Image Extension

Input data

We randomly cropped to a size of 257, following the suggestions of [Tet+19], the slices of our training dataset. We run experiments with three different masks. The mask may be a square right in the middle of the image, whose size is three time less the input's size, a rectangle on the right side of the image, with the same height as the image and a quarter of its size, and a square randomly located in the image, with a size that goes from 1/3 to 2/3 the input's size (see 4.1) The batch size was 20. The input data was normalize to a $range = [-1, 1]$. The generator takes three inputs, two explained in 3.2.10 ,the Mask (M) and the masked image (z), and an image with 'True' to all the pixels added by the implementation.

Training procedures

The training was carried out with two different losses. The 'L1Loss', that measures the pixel-wise loss of the prediction against the ground truth, and the Adversarial loss, with a weight of

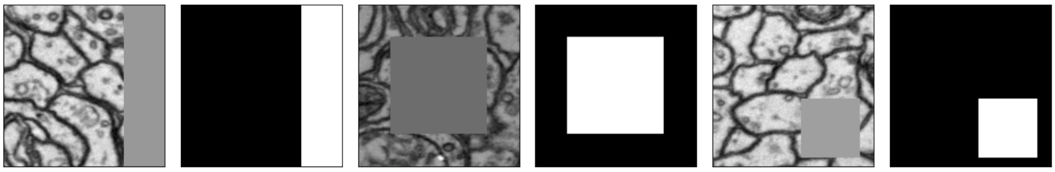

Figure 4.1: Masked image (z) and Mask in the three different scenarios

$\lambda = 10^{-2}$. Both the generator and the discriminator used the Adam optimizer, with $\alpha_d = 10^{-3}$, $\alpha_g = 10^{-4}$, $\beta_1 = 0.5$, $\beta_2 = 0.9$. We run the experiments for 200 epochs.

4.4.2 Segmentation

Input data

We resized to 900x900 and randomly cropped to a size of 257 the images from the training set. In order to keep the input as close as possible to the Image Extension Input, we mantained the 3 channels input. Since we didn't want to apply any mask to the input data, the masked image (z) was replaced by the original image, and the Mask (M) was replaced by an image with all its pixels set to zero, which in the Image Extension training would mean that no mask has been applied to the input image (however, this was tested and results were vaguely better this way, rather than replacing the Mask with and image with all its pixels set to one). The input data was normalized to a $range = [-1, 1]$ as in 4.4.1 and performed data augmentation as introduced in 3.2.2.

Training procedures

The architecture was the Image Extension's generator. Only the last operation, where the model's output is fixed within the values [-1, 1], was removed from the original architecture. The pretrained weights were loaded for the first 8 layers (encoder) and the following layers were randomly initialize. This first eight layers were frozen, i.e. these layers did not require gradients during training, thus their weights were not updated. The loss was computed with the Binary Cross-Entropy function, and optimization was performed by the Adam optimizer $\alpha = 0.01$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 0.01$. (we did not use the same Hyperparameters than the Image Extension task as we experimentally concluded that were inappropiate). The targets of the networks were the affinity maps mentioned in 3.2.1, binary images computed using the Pytorch Connectomics toolbox [Gro19].

4.5 Evaluation of the segmentation quality: pretrained weights and frozen layers

The aim of these concatenation of experiments was to study the segmentation results obtained with and without the pretrained weights, and to decide whether the chosen adversarial training was a good call looking forward improving the segmentation task. At the same time, we evaluated the variability in our results depending on the number of layers that were 'pretrained' and the number of layers that were randomly initialize.

4.5.1 Input data

Keeping the same tone as previous experiments, our training set was randomly cropped to 257x257 images, previously resized to 900x900, and normalized to a $range = [-1, 1]$. Data augmentation was performed as introduced in 3.2.2. The targets of the networks were the affinity maps. The batch size was 18. The input data were as described in previous experiments, the transformed original image, a black image (pixels' value = 0) replacing the Mask, and a blank image (pixels' value = 1) following the adopted implementation. All the mentioned data augmentations techniques in 3.2.2 were included, and were performed durring running time, thus the input were different every epoch of the training.

4.5.2 Training procedures

As in 4.4.2, the architecture was the generator from the Image Extension training, removing the final squishing of the outputs between [-1, 1]. The loss was computed by the Binary Cross Entropy function, and the optimizer used was Adam. We run tests with both the parameters from [Tet+19], ($\alpha = 10^{-4}$, $\beta_1 = 0.5$, $\beta_2 = 0.9$, $\epsilon = 0.01$), and from [Lee+17], ($\alpha = 0.01$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 0.01$). The learning was not scheduled for proper evaluation. Many experiments were performed in which a different number and combination of layers were frozen and initialized by the pretrained weights, as well as randomly initialized.

5 Results

5.1 StyleGan results (4.2)

In 5.1 we can see the evolution of the loss throughout the training process, 5.2 shows samples of the output of the GAN at different stages of the training, and 5.3 compares crops from the real images with the generated samples with contrast.

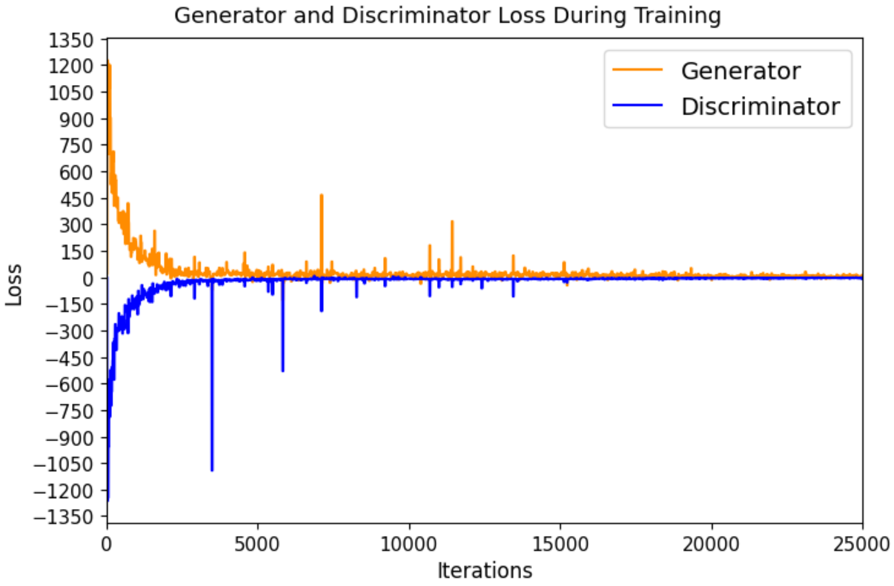

Figure 5.1: Generator and Discriminator loss over the iterations

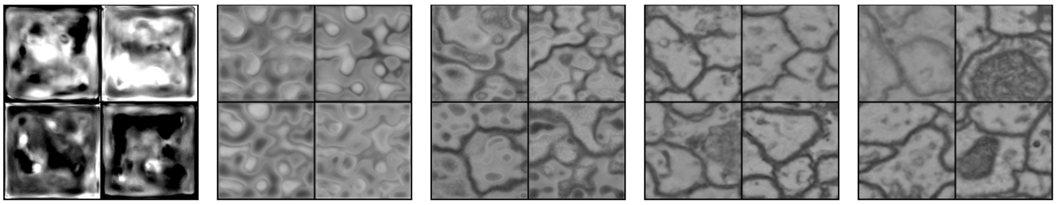

Figure 5.2: Outputs of the Gan during the training process. From left to right: 900, 3600, 7500, 15200 and 25000 iterations

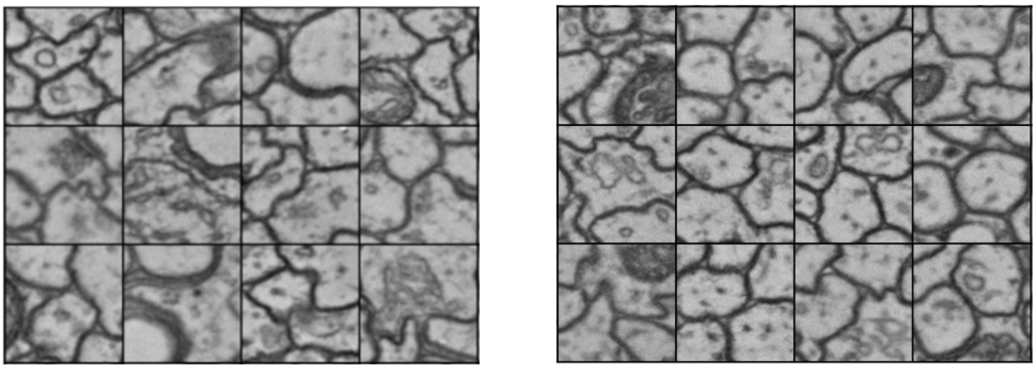

Figure 5.3: Training batch on the left and predictions of the GAN with added contrast on the right

5.2 2D and 3D UNet segmentation results (4.3)

5.2.1 2D-UNet results

The following graphs show the results computed troughout the training process. Before parameter tunning, the predictions were post-processed with a $sigmoid\ threshold = 0.73$ and a $watershed\ threshold = 0.81$ (see 3.2.8).

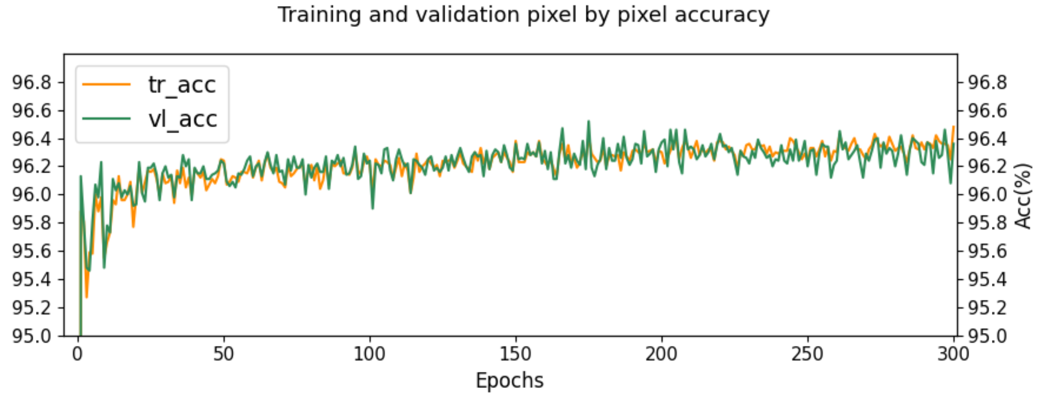

Figure 5.4: UNet2D Pixel accuracy on the training and validation set for 300 epochs

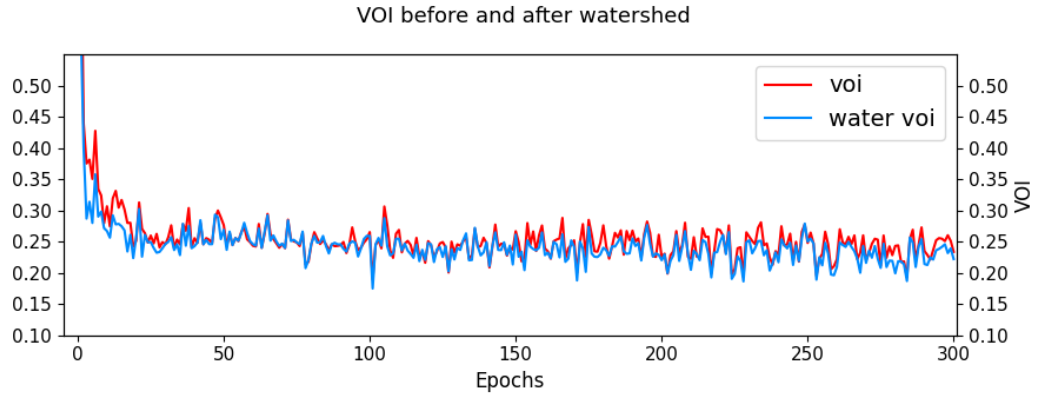

Figure 5.5: UNet2D VOI on the validation set before and after watershed.

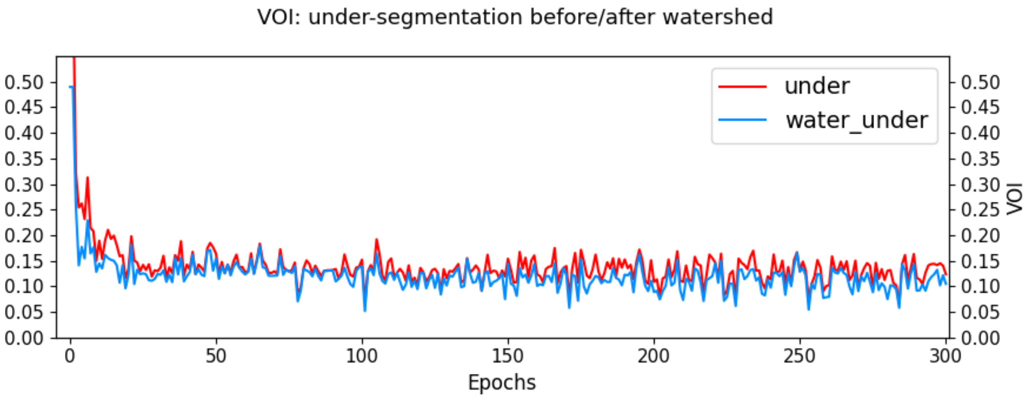

Figure 5.6: UNet2D VOI undersegmentation on the validation set before and after watershed.

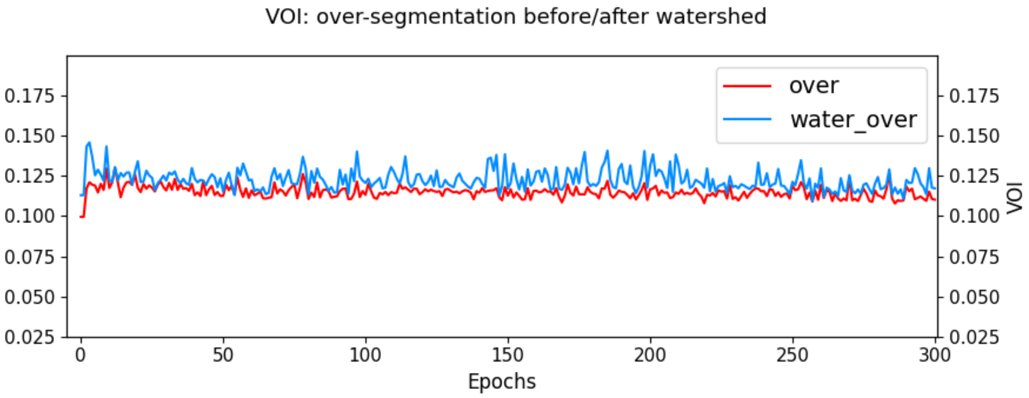

Figure 5.7: UNet2D VOI oversegmentation on the validation set before and after watershed.

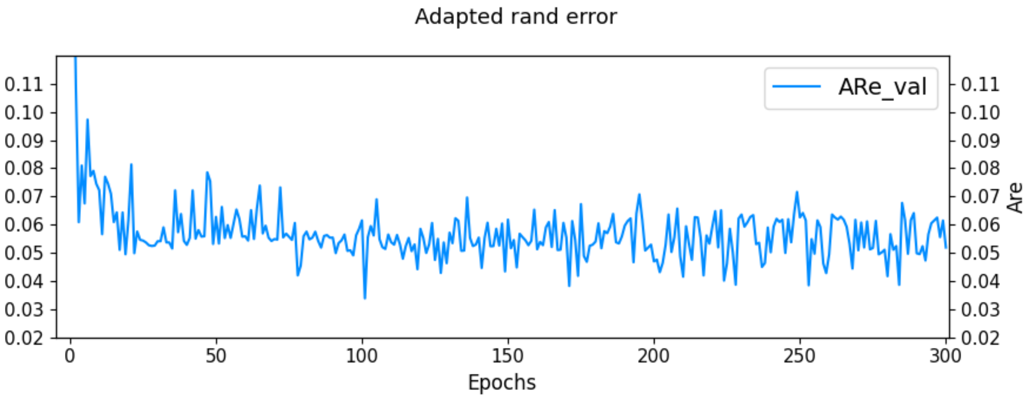

Figure 5.8: Adapted rand error on the validation set after watershed.

	Acc	Voi watershed	Adapted rand error
Mean value	96.294	0.23604	0.06151

Table 5.1: Mean for the last 5 epochs of the acc, voi and Are computed on the batches of the validation set (**crop size** = **160**).

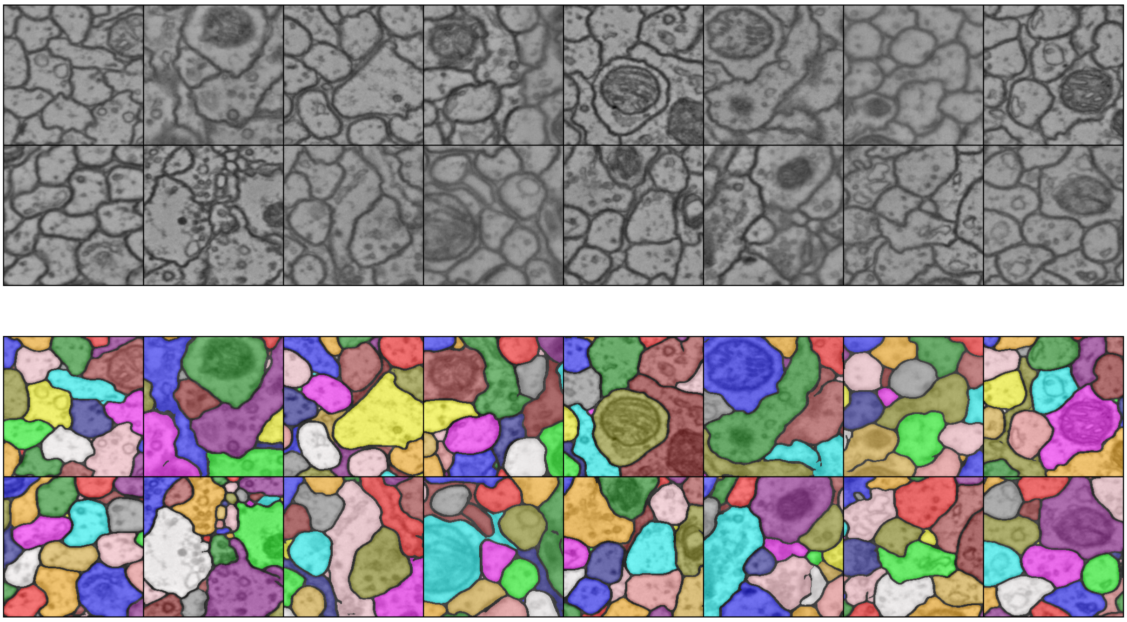

Figure 5.9: Example of a batch segmentation from the validation set.

Parameter tunning

The next tables show the Voi of the validation set for different post-processing thresholds (4.3).

sig_th\edge_th	sig_th	sig_th + 0.02	sig_th + 0.04	sig_th + 0.06	sig_th + 0.08	sig_th + 0.1
0.73	0.34518211	0.34024846	0.3311448	0.32876741	0.31015961	0.29805295
0.8	0.30170726	0.29820824	0.29095355	0.28878042	0.29087292	0.29087188
0.83	0.28979098	0.2870637	0.28173661	0.28354939	0.28320302	0.28188659
0.86	0.279248	0.27825378	0.27473227	0.27745733	0.27666762	0.27717771
0.88	0.27247987	0.27279436	0.27267743	0.27135408	0.27235543	0.26663057
0.9	0.27012443	0.2675271	0.26864067	0.26913416	0.26637362	0.46496631

Table 5.2: VOI's mean of the validation set for different thresholds.

sig_th\edge_thh	sig_th	sig_th + 0.02	sig_th + 0.04	sig_th + 0.06	sig_th + 0.08	sig_th + 0.1
0.73	0.21314544	0.21326956	0.21385464	0.21637996	0.21978404	0.22213061
0.8	0.21306922	0.21470234	0.21988121	0.22522749	0.22734242	0.22734214
0.83	0.21715414	0.2215466	0.22365832	0.22752809	0.23098226	0.23128799
0.86	0.22064403	0.22605554	0.22753674	0.23104534	0.23255361	0.23513137
0.88	0.22601899	0.22667582	0.2299904	0.23027891	0.23403996	0.24314562
0.9	0.22567901	0.22813892	0.22926271	0.23042022	0.23992665	0.44705647

Table 5.3: Oversegmentation's mean of the validation set for different thresholds.

sig_th\edge_th	sig_th	sig_th + 0.02	sig_th + 0.04	sig_th + 0.06	sig_th + 0.08	sig_th + 0.1
0.73	0.13203666	0.1269789	0.11729016	0.11238745	0.09037557	0.07592234
0.8	0.08863804	0.0835059	0.07107233	0.06355293	0.06353049	0.06352973
0.83	0.07263684	0.06551711	0.05807829	0.05602129	0.05222076	0.0505986
0.86	0.05860397	0.05219824	0.04719554	0.04641199	0.04411401	0.04204634
0.88	0.04646088	0.04611855	0.04268703	0.04107517	0.03831547	0.02348494
0.9	0.04444541	0.03938818	0.03937796	0.03871394	0.02644696	0.01790984

Table 5.4: Undersegmentation's mean of the validation set for different thresholds.

Final 2D results

The lowest values of our evaluation are highlighted in the tables 5.2, 5.3, 5.4. However, we have chosen a *sigmoid threshold* = 0.83 and a *edge threshold* = 0.91 as explained in the discussion of results. Thus, the following results are computed with these specific thresholds, as well as the segmentations that we can find in the appendix .

	VOI	VOI watershed	Adapted rand error
Mean value	0.24291	0.23812	0.08513

Table 5.5: Final results fo the 2D-UNet segmentation computed on the test set ($20x1250x1250$) for a sigmid threshold = 0.81 and a edge threshold = 0.91.

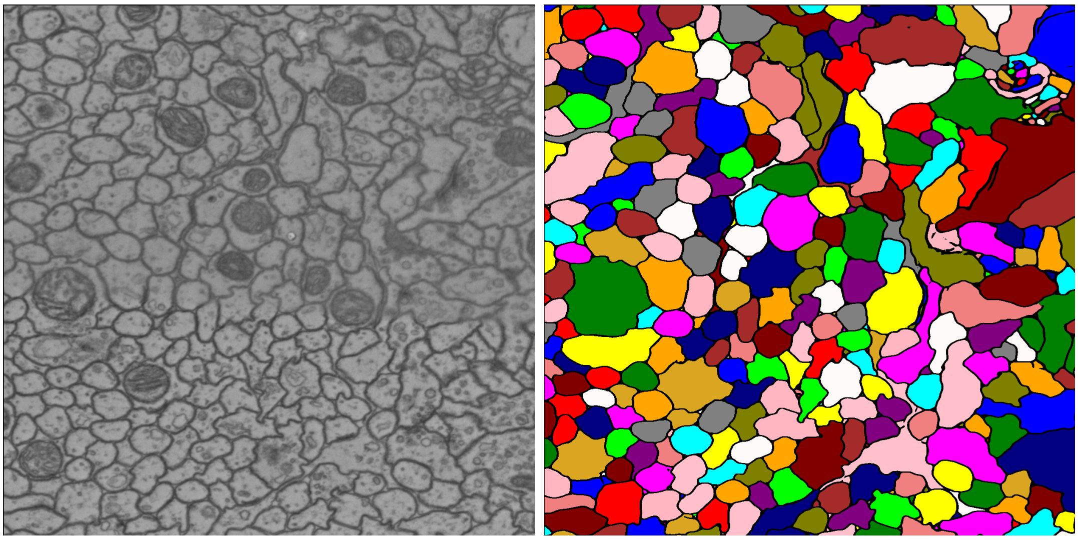

Figure 5.10: Example of the segmentation carried out with the tunned parameters on one sample of the test set.

5.2.2 3D-UNet results

The following graphs show the results computed troughout the traning process. Before parameter tunning, the predictions were post-processed with a *sigmoid threshold* = 0.73 and a *watershed threshold* = 0.81 (see 3.2.8).

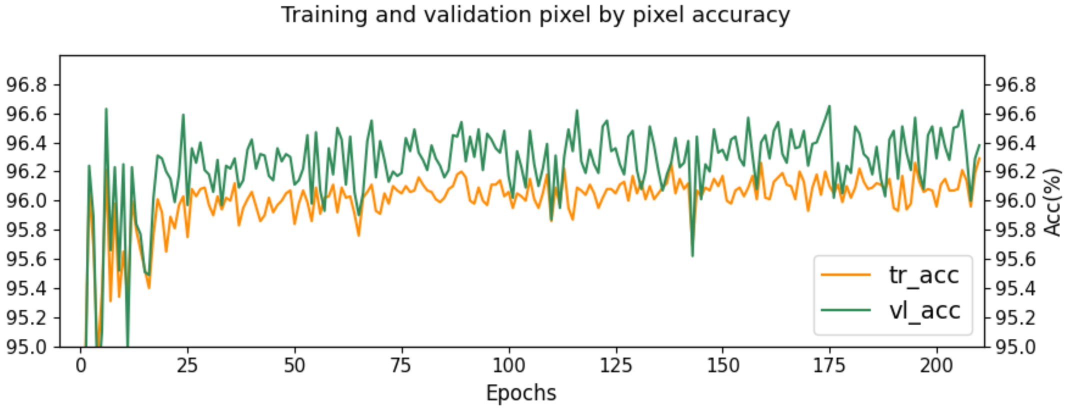

Figure 5.11: UNet3D Pixel accuracy on the training and validation set for 210 epochs

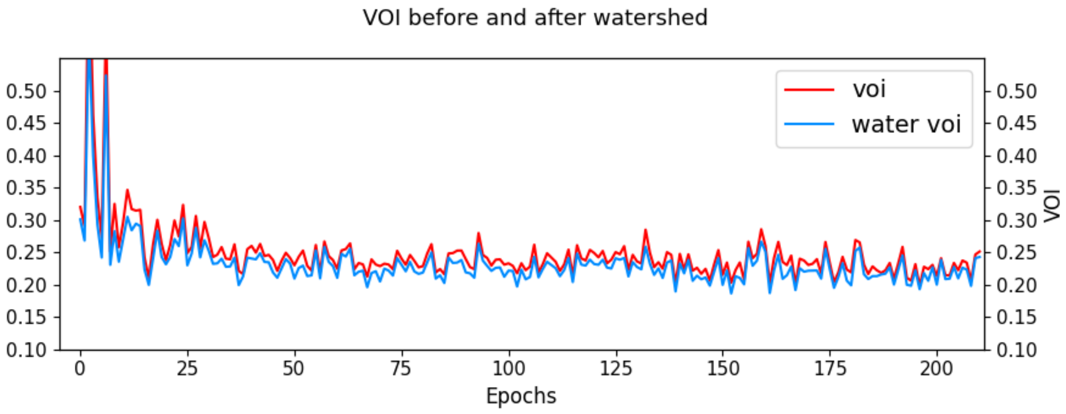

Figure 5.12: UNet3D VOI on the validation set before and after watershed.

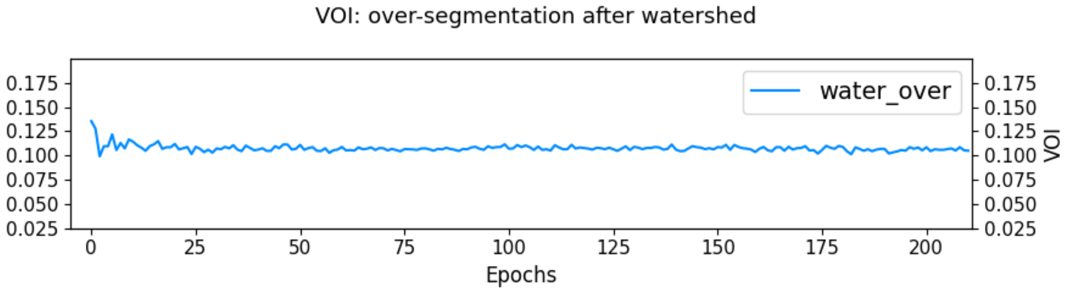

Figure 5.13: UNet3D VOI undersegmentation on the validation set after watershed.

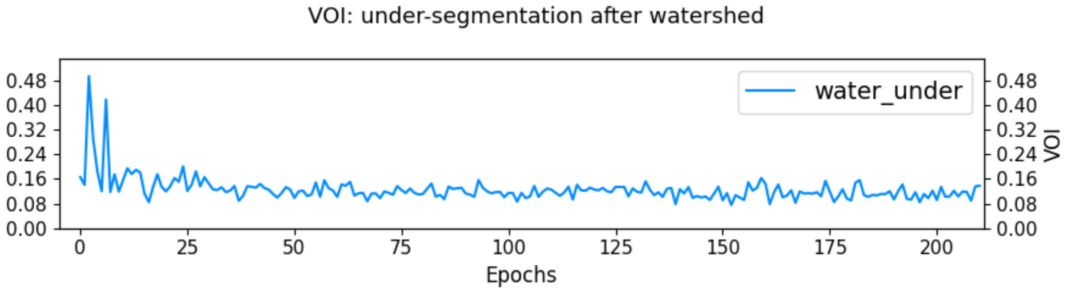

Figure 5.14: UNet3D VOI oversegmentation on the validation set after watershed.

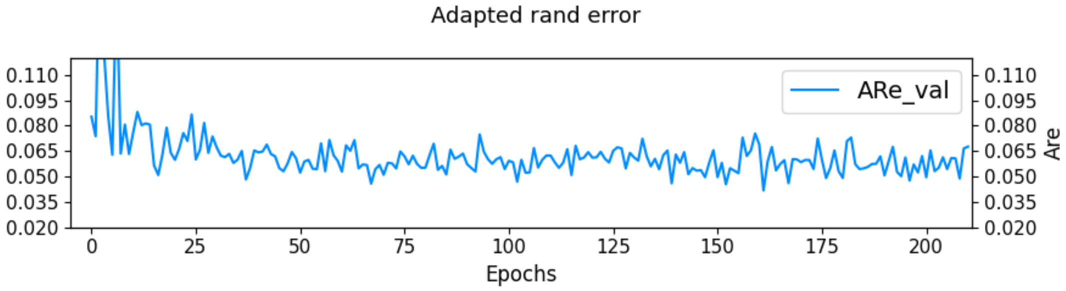

Figure 5.15: Adapted rand error on the validation set after watershed.

	Acc	Voi watershed	Adapted rand error
Mean value	96.324	0.22834	0.06094

Table 5.6: Mean for the last 5 epochs of the acc, voi and Are computed on the batches of the validation set (**crop size = 160**).

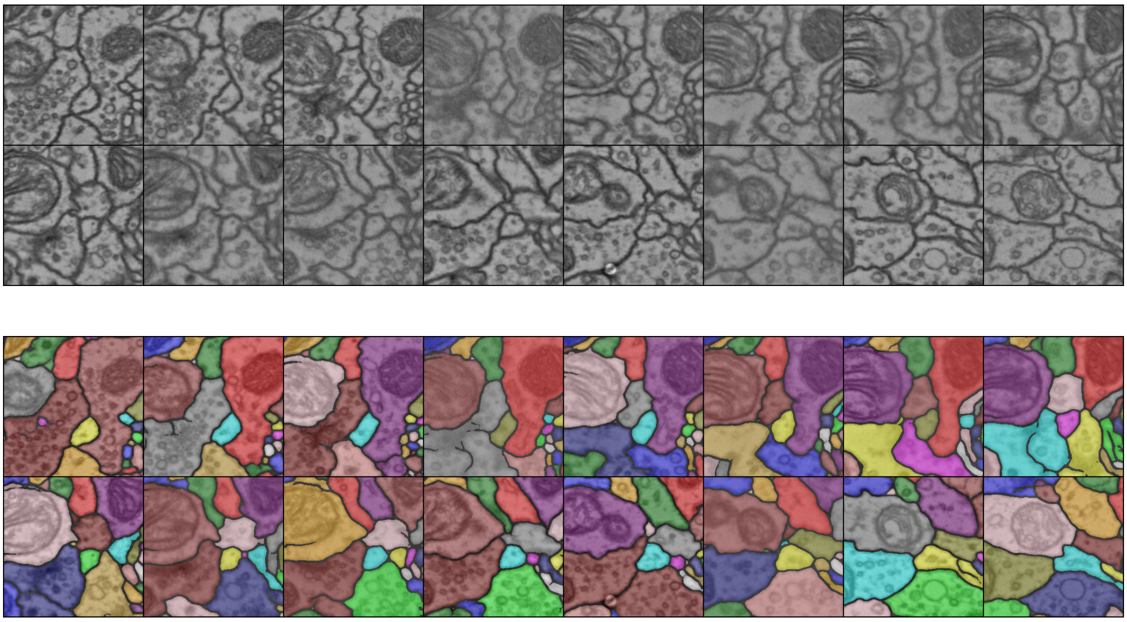

Figure 5.16: From top-to-bottom: example of a batch (volume) from the validation set, and predicted segmentation displayed above the actual input.

Parameter tunning

The following tables present the mean VOI of the validation set for different post-processing thresholds (4.3).

sig_th\edge_th	sig_th	sig_th + 0.02	sig_th + 0.04	sig_th + 0.06	sig_th + 0.08	sig_th + 0.1
0.73	0.4290547	0.41365697	0.41094326	0.40268915	0.3938805	0.37892296
0.8	0.3801884	0.36878481	0.35618074	0.33913657	0.33495362	0.30628901
0.83	0.35483158	0.34744605	0.33498273	0.32767994	0.30596649	0.30568959
0.86	0.33528227	0.32437826	0.30500718	0.28715934	0.28556377	0.28819076
0.88	0.3262903	0.30292415	0.28686556	0.28628755	0.28328314	0.28981275
0.9	0.29941567	0.28601196	0.2816527	0.27959717	0.28890729	0.512121

Table 5.7: VOI's mean of the validation set for different thresholds.

sig_th\edge_th	sig_th	sig_th + 0.02	sig_th + 0.04	sig_th + 0.06	sig_th + 0.08	sig_th +0.1
0.73	0.20304354	0.20392226	0.20602648	0.20614893	0.21032239	0.21420085
0.8	0.21150062	0.21797665	0.21818641	0.22195473	0.22468605	0.22977515
0.83	0.21682496	0.22046487	0.22060938	0.22545161	0.23259457	0.23358679
0.86	0.2224703	0.22305666	0.22769009	0.23633155	0.23842581	0.24823244
0.88	0.22227452	0.22723789	0.23079307	0.23311774	0.248604	0.26410456
0.9	0.22764389	0.23115767	0.23227592	0.24899573	0.26368471	0.49273844

Table 5.8: Oversegmentation's mean of the validation set for different thresholds.

sig_th\edge_th	sig_th	sig_th + 0.02	sig_th + 0.04	sig_th + 0.06	sig_th + 0.08	sig_th +0.1
0.73	0.22601116	0.20973471	0.20491678	0.19654021	0.1835581	0.16472211
0.8	0.16868778	0.15080816	0.13799433	0.11718184	0.11026757	0.07651385
0.83	0.13800662	0.12698118	0.11437334	0.10222834	0.07337192	0.0721028
0.86	0.11281197	0.1013216	0.0773171	0.05082779	0.04713796	0.03995831
0.88	0.10401578	0.07568626	0.05607249	0.05316981	0.03467915	0.02570819
0.9	0.07177178	0.05485429	0.04937678	0.03060144	0.02522257	0.01938256

Table 5.9: Undersegmentation's mean of the validation set for different thresholds.

Final 3D results

The lowest values of our evaluation are highlighted in the tables 5.7, 5.8, 5.9. However, we have chosen a *sigmoid threshold* = 0.83 and a *edge threshold* = 0.91 as explained in the discussion of results, and in order to better compare with the 2D-Unet results. Thus, the following results are computed with these specific thresholds, as well as the segmentations that we can find in the appendix VALUE.

	VOI	VOI watershed	Adapted rand error
Mean value	0.27045	0.24607	0.09029

Table 5.10: Final results fo the 3D-UNet segmentation computed on the test set ($20x1250x1250$) for a sigmid threshold = 0.81 and a edge threshold = 0.91.

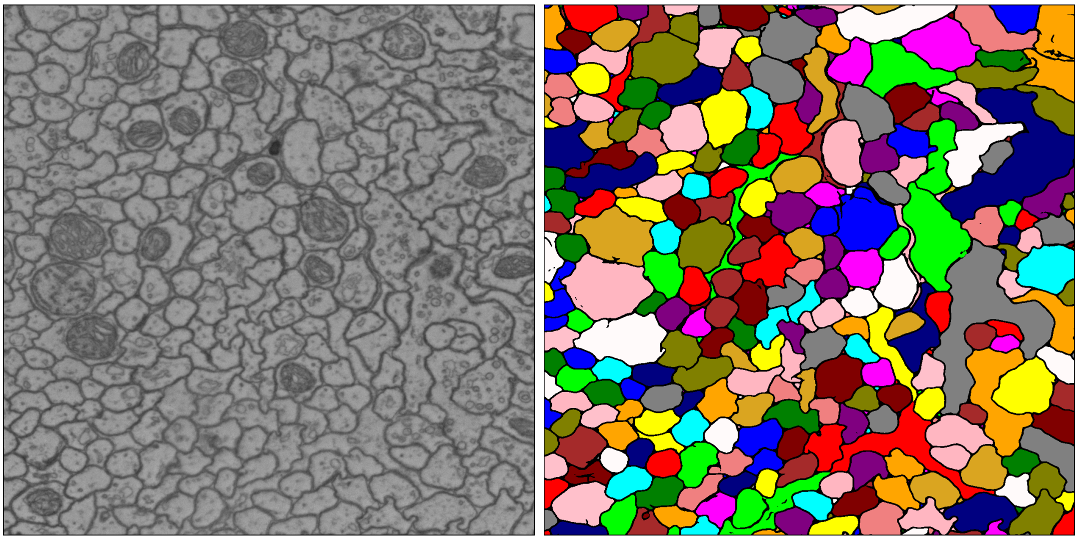

Figure 5.17: Example of the segmentation carried out with the tunned parameters on one sample of the test set.

5.2.3 Contrasting results

	VOI split	VOI merge	ARAND
2D-UNet	0.18757	0.05062	0.08513
3D-UNet	0.18770	0.05837	0.09029
CREMI best submit	0.339	0.115	0.108

Table 5.11: Over-segmentation, under-segmentation and adapted rand error for the 2D and 3D UNet models in contrast with the best submit in thesegmentation leaderboard of the CREMI challenge. Our model and the CREMI submition do not follow the same pre-evaluation procedure, what makes the contrasting of very little meaning and should be better taken as a informative data. See discussion of results.

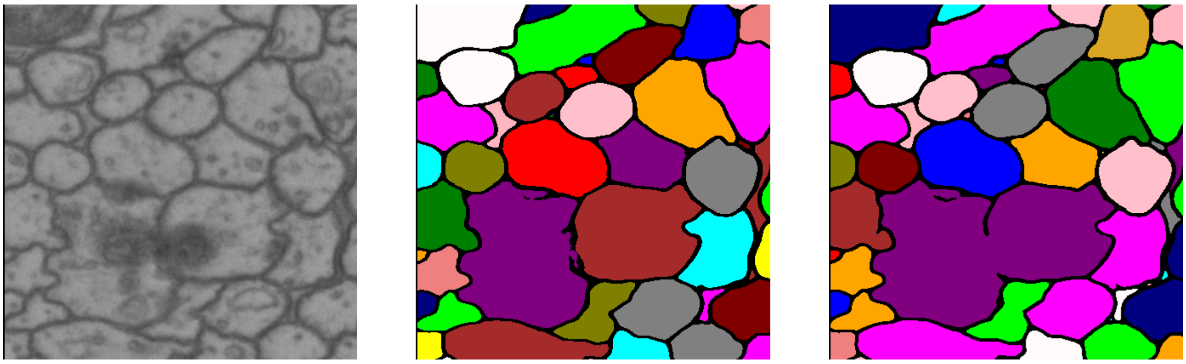

Figure 5.18: Crop of input

Figure 5.19: 2D-UNet

Figure 5.20: 3D-UNet

Figure 5.21: Differences on the predicted segmentations of the same crop by each of the model

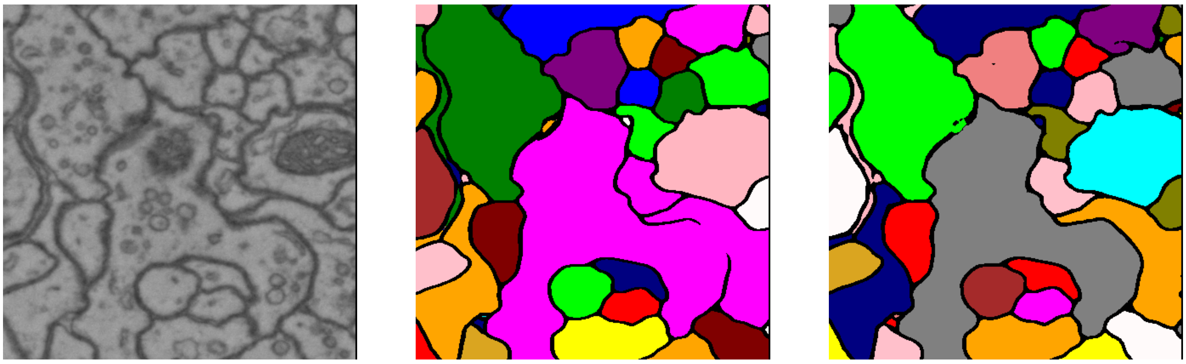

Figure 5.22: Crop of input

Figure 5.23: 2D-UNet

Figure 5.24: 3D-UNet

Figure 5.25: Differences on the predicted segmentations of the same crop by each of the model

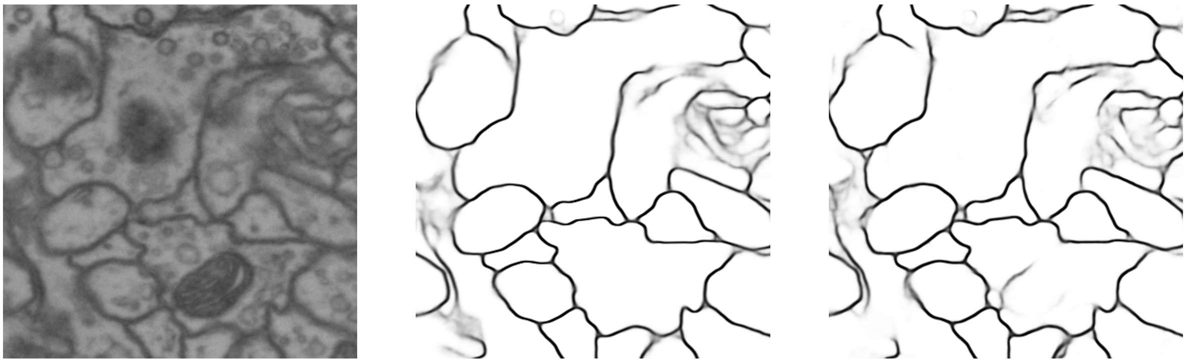

Figure 5.26: Crop of input

Figure 5.27: 2D-UNet

Figure 5.28: 3D-UNet

Figure 5.29: Differences on the models' output (probability maps) of the same crop

5.3 Image Extension results (4.4.1)

5.3.1 Mask on the right

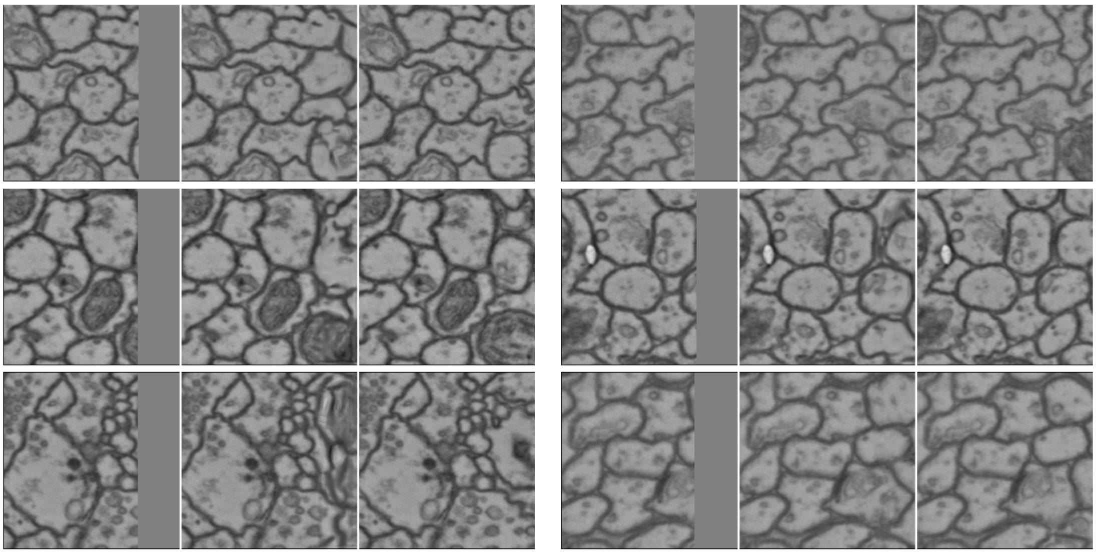

Figure 5.30: Different examples from the validation set. From left to right: masked image, prediction and groundtruth.

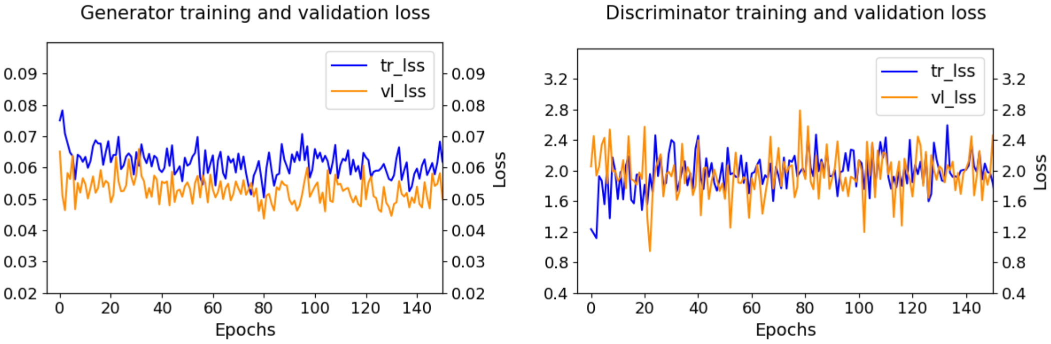

Figure 5.31: Loss throughout the training of the Generator and the Discriminator for the training and validation set.

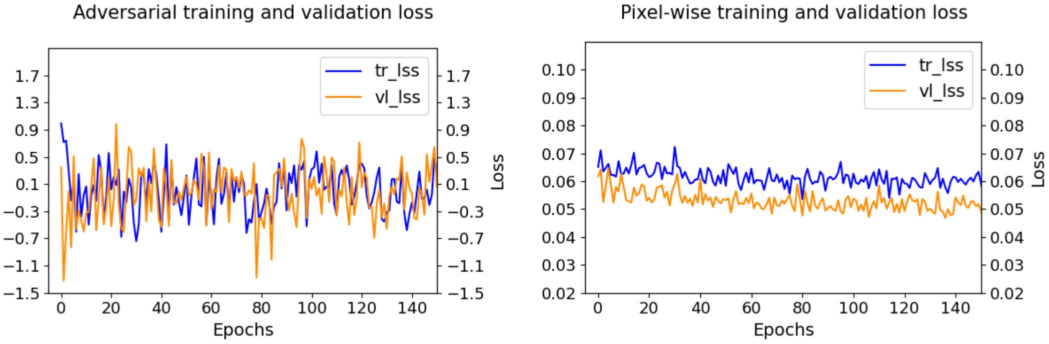

Figure 5.32: Adversarial and pixel wise loss throughout the training for the training and validiation set.

	D. loss	G. loss	Adv. loss	Pixel-wise loss
Training	1.97093	0.06830	0.48965	0.06341
Validation	1.96588	0.05822	0.64427	0.05177

Table 5.12: Loss values for the training and the validation set after 150 epochs (18:51.51 running time)

5.3.2 Square mask in the middle

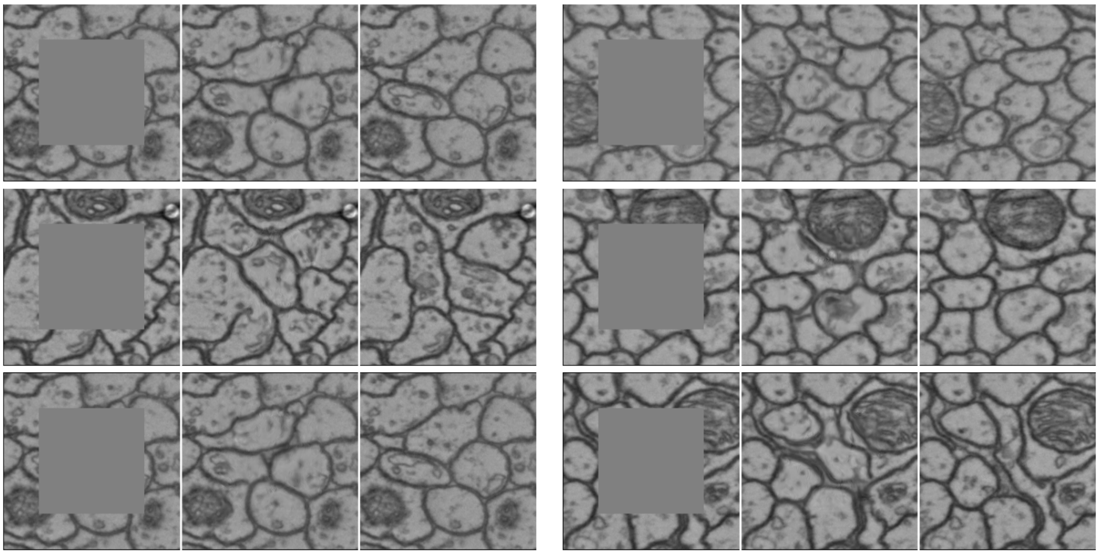

Figure 5.33: Different examples from the validation set. From left to right: masked image, prediction and groundtruth.

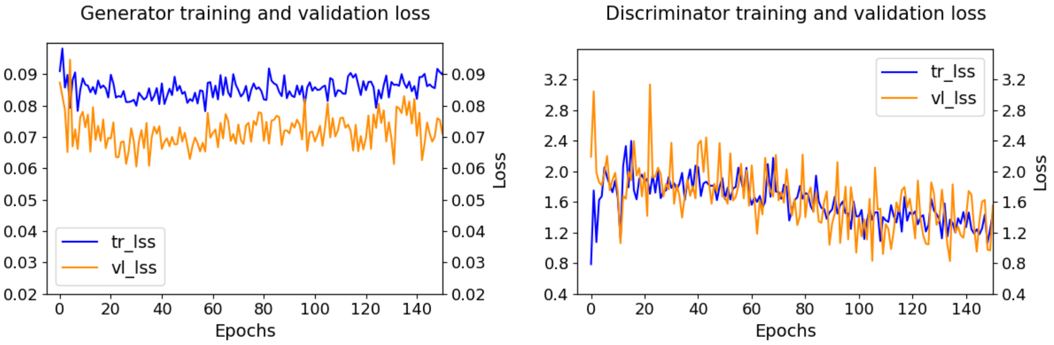

Figure 5.34: Loss throughout the training of the Generator and the Discriminator for the training and validation set.

	D. loss	G. loss	Adv. loss	Pixel-wise loss
Training	1.24309	0.09067	0.73436	0.08332
Validation	0.96793	0.07540	0.73706	0.06803

Table 5.13: Loss values for the training and the validation set after 150 epochs (18:51.51 running time)

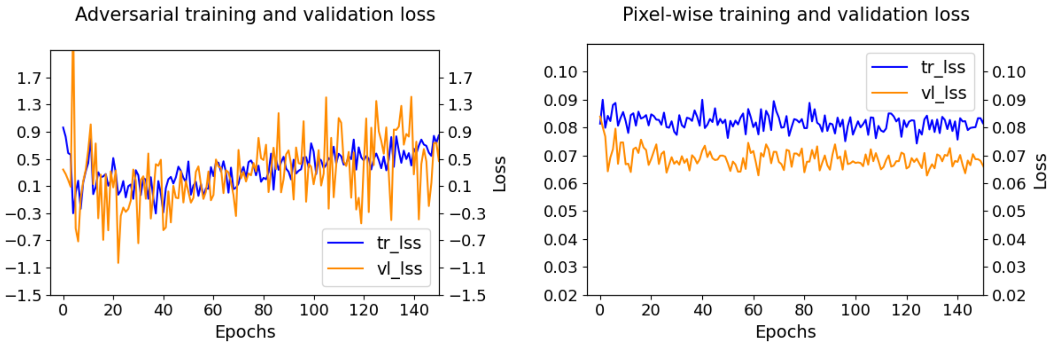

Figure 5.35: Adversarial and pixel wise loss throughout the training for the training and valdiation set.

5.3.3 Square mask randomly located/different size

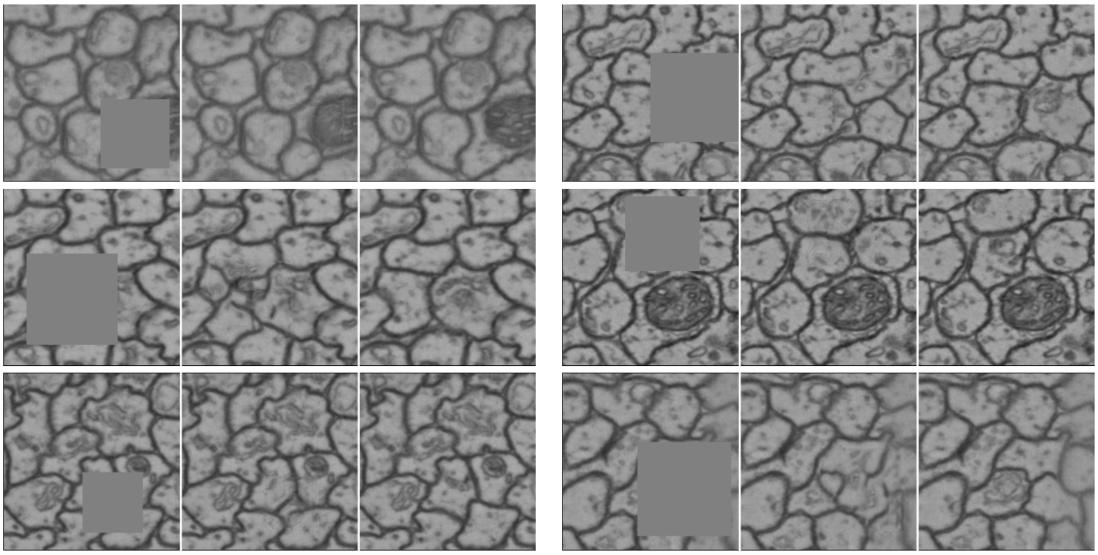

Figure 5.36: Different examples from the validation set. From left to right: masked image, prediction and groundtruth.

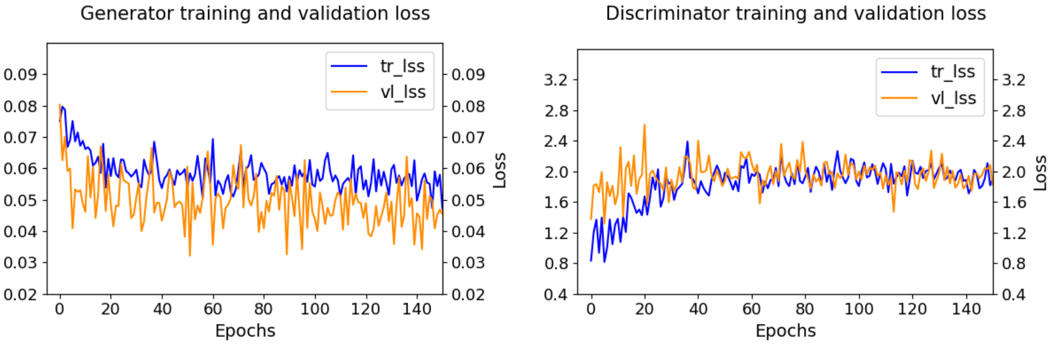

Figure 5.37: Loss throughout the training of the Generator and the Discriminator for the training and validation set.

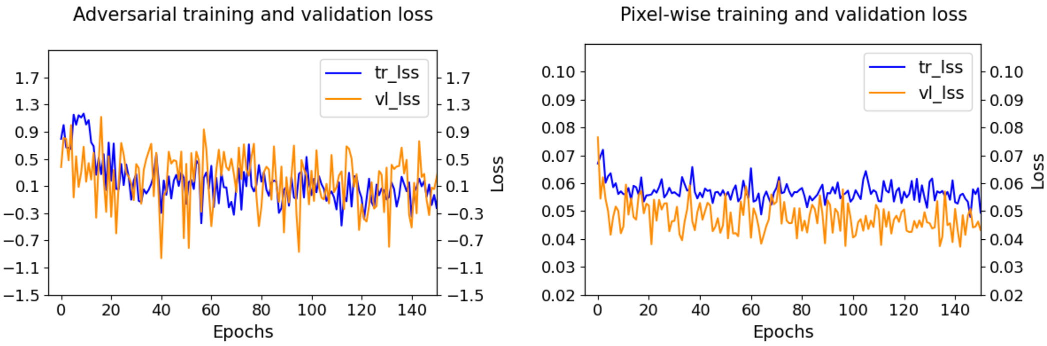

Figure 5.38: Adv and pixel-wise loss for the training and validation set.

	D. loss	G. loss	Adv. loss	Pixel-wise loss
Training	1.81831	0.05792	-0.03082	0.05823
Validation	2.08587	0.04681	0.05492	0.04627

Table 5.14: Loss values for the training and the validation set after 150 epochs (18:51.51 running time)

5.4 Segmentation (4.4.2)

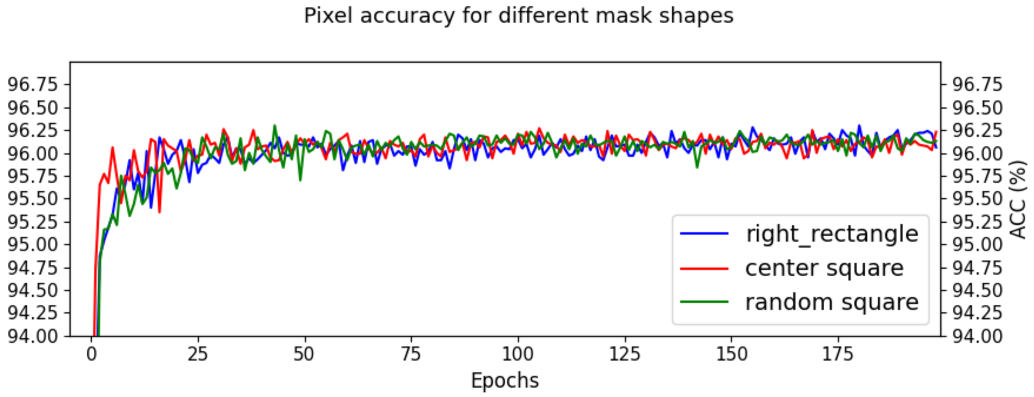

Figure 5.39: Pixel acc. on the validation set for transfer learning with different mask's shapes

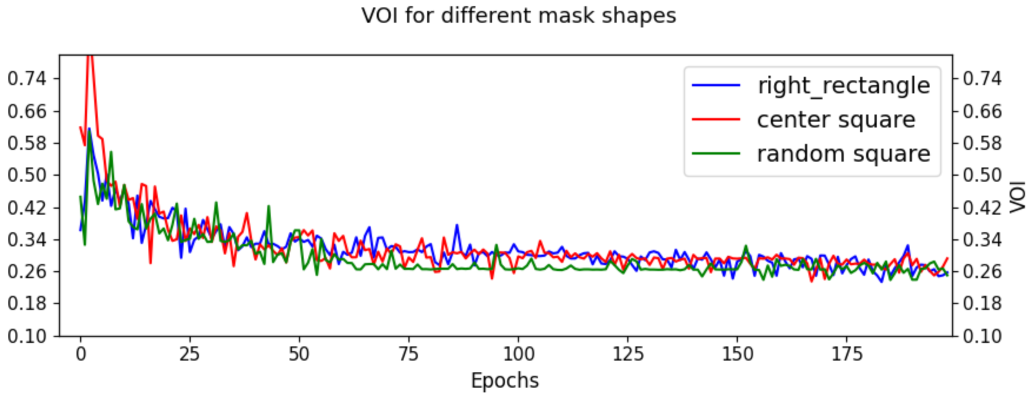

Figure 5.40: Voi on the validation set for different shapes and locations of the masks .

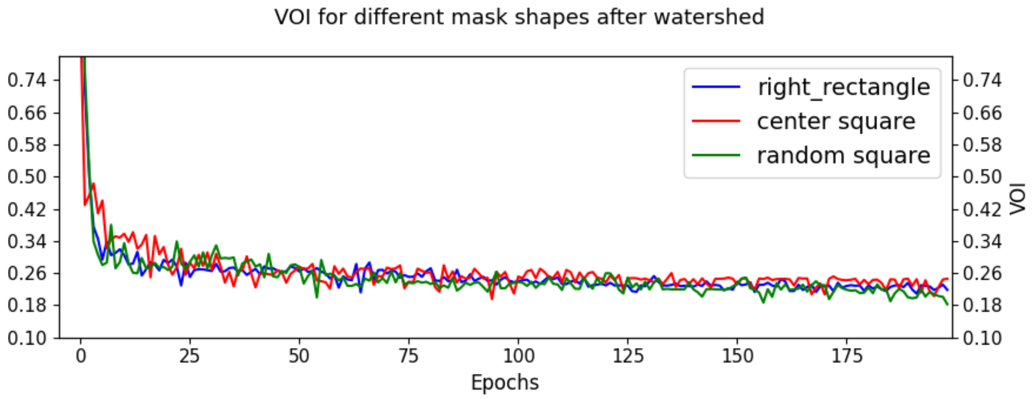

Figure 5.41: Voi after watershed on the validation set for different shapes and locations of the masks.

	Pixel Acc	VOI	VOI after watershed
Right mask	96.19	0.25597	0.22354
Square mask	96.1	0.26661	0.22644
Random mask	96.13	0.26897	0.20149

Table 5.15: Results of the pixel accuracy and voi for transfer learning with the weights obtained from the image extension with each of the masks (last 5 epochs mean)

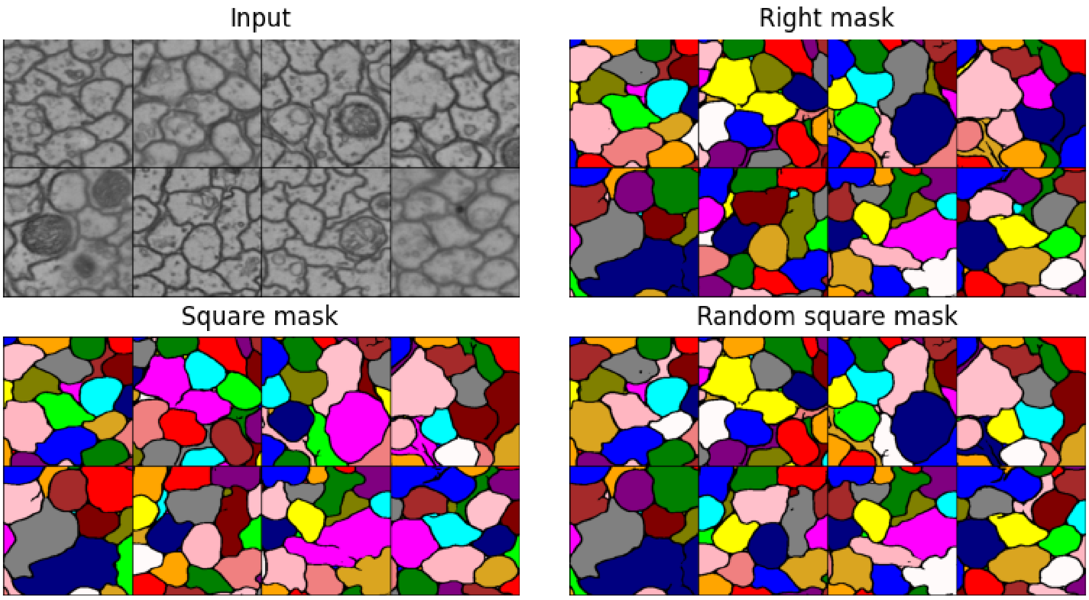

Figure 5.42: Segmentation predictions for transfer learning with different shapes/locations of mask. From left-to-right, top-to-bottom: crop from an input batch from the validation set, segmentation for pretraining with mask on the right, segmentation with pretraining with square mask on the center and segmentation with square mask randomly located and sized

5.5 Transfer learning results (4.5)

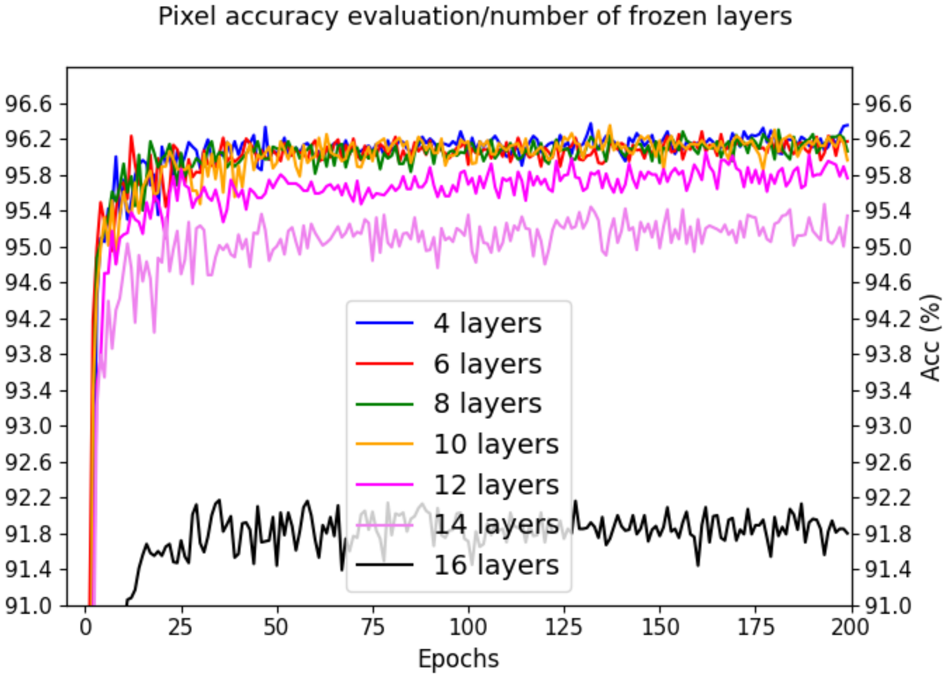

Figure 5.43: Accuracy on the validation set for different number of frozen layers with pretrained weights loaded.

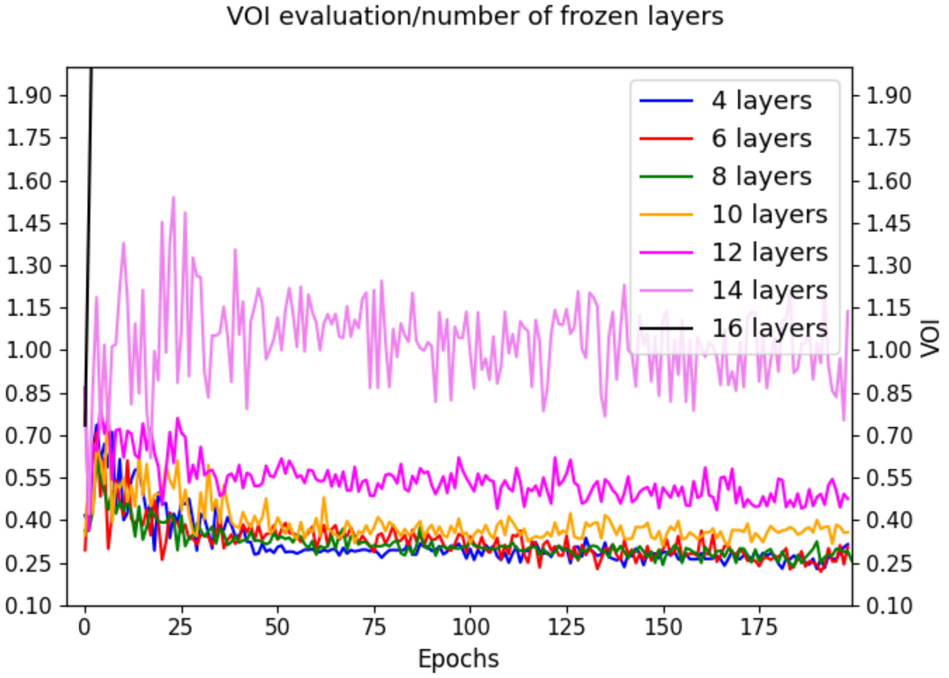

Figure 5.44: VOI on the validation set for different number of frozen layers with pretrained weights loaded.

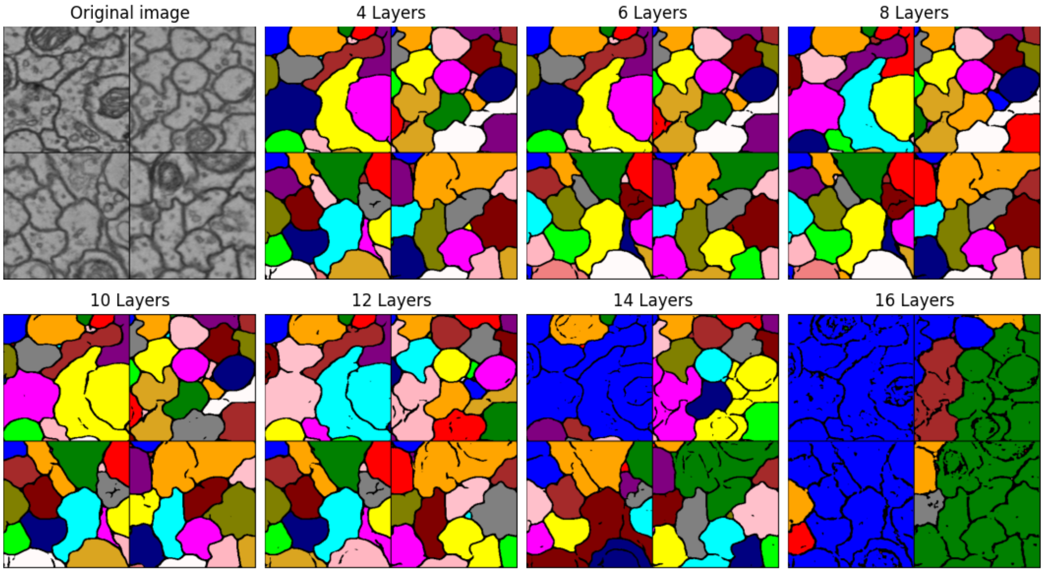

Figure 5.45: Segmentation results in a crop of the validation set for different number of frozen layers with pretrained weights.

	4 layers	6 layers	8 layers	10 layers	12 layers	14 layers	16 layers
Acc (%)	96.26	96.15	96.19	96.19	95.768	95.13	90.96
VOI	0.28572	0.27405	0.28573	0.35764	0.49225	0.91028	2.96603

Table 5.16: Mean of the acc and voi for the last 5 epochs for different number of frozen layers with the pretrained weights.

5.5.1 Pretrained weights vs random initialization

	All random	All pretr.	8 random frozen	8 pre. frozen + 9 random
Acc (%)	96.2	96.22	95.86	96.14
Voi	0.248	0.26866	0.49214	0.36446

Table 5.17: Mean of the acc and voi for the last 5 epochs for different combinations of initializations and frozen layers

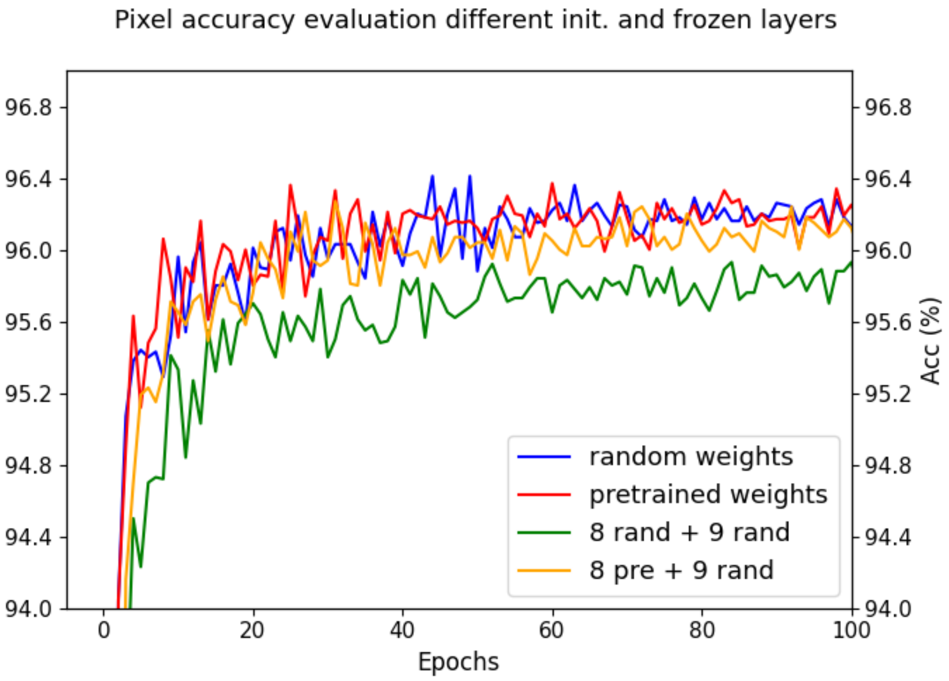

Figure 5.46: Evaluation of the influence of using pretrained weights on the segmentation task. Pixel acc. of 4 different scenarios:(blue) all layers randomly initialize,(red) all layers with pretrained weights, (green) all layers randomly initialized with the first 8 frozen, (orange) first 8 layers with pretrained weights and frozen, randomly initialized the reminder.

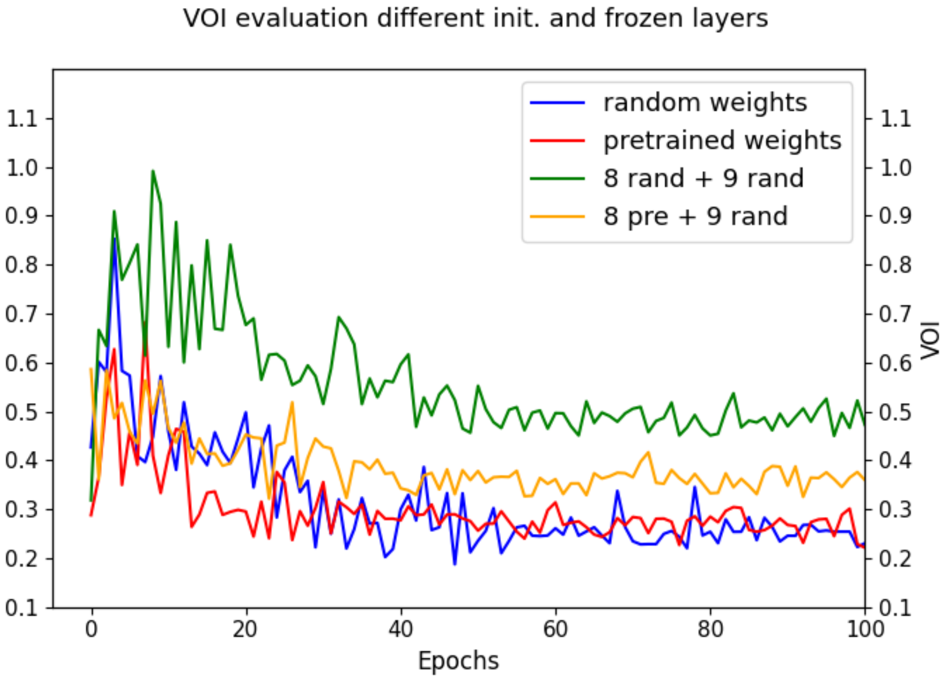

Figure 5.47: Evaluation of the influence of using pretrained weights on the segmentation task.VOI of 4 different scenarios:(blue) all layers randomly initialize,(red) all layers with pretrained weights, (green) all layers randomly initialized with the first 8 frozen, (orange) first 8 layers with pretrained weights and frozen, randomly initialized the reminder.

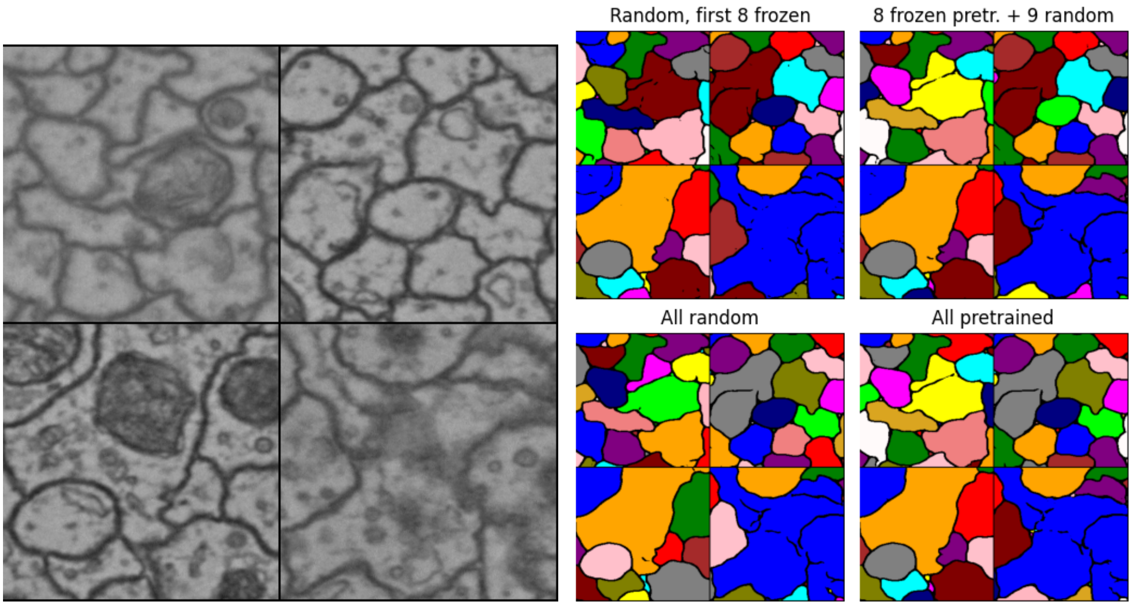

Figure 5.48: Segmentation results after 200 epochs for different scenarios regardin the initialization of the layers and the number of layers that require gradients.

5.5.2 Hyperparameters

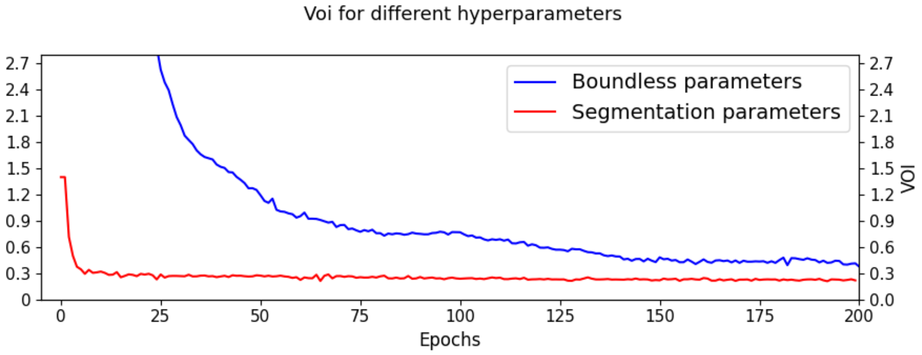

Figure 5.49: Constrat on the Pixel accuracy on transfer learning using the hyperparameters from [Tet+19] and [Lee+17] .

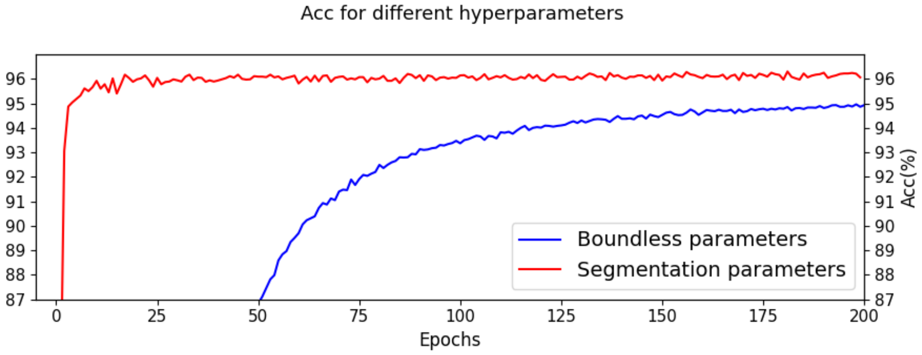

Figure 5.50: Constrat on VOI on transfer learning using the hyperparameters from [Tet+19] and [Lee+17] .

	Segmentation hyp.	Boundless hyp.	Boundless hyp. 400 epochs
Acc (%)	96.06	94.93	95.42
Voi	0.25715	0.70453	0.59491

Table 5.18: Constrat on VOI on transfer learning using the hyperparameters from [Tet+19] and [Lee+17]

5.5.3 Early stopping and reduced dataset

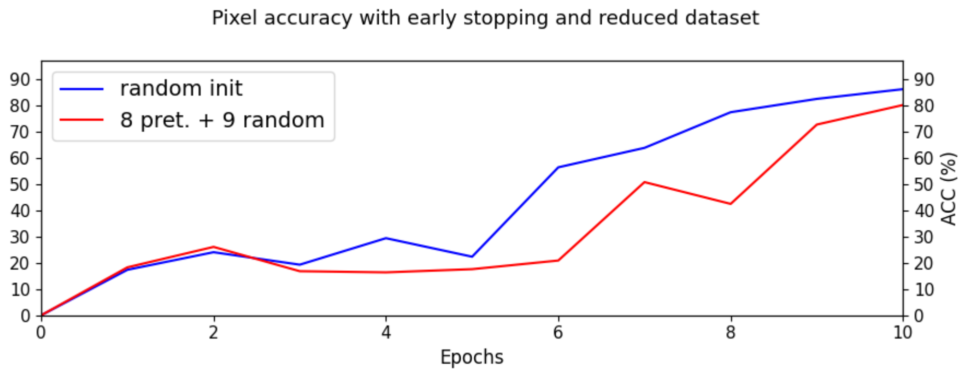

Figure 5.51: Pixel accuracy when early stopping and using a small training set. In the graph we can see the comparisson between our model, with pretrained weights for the first 8 layers and randomly initialized the reminder, and our model randomly initialized.

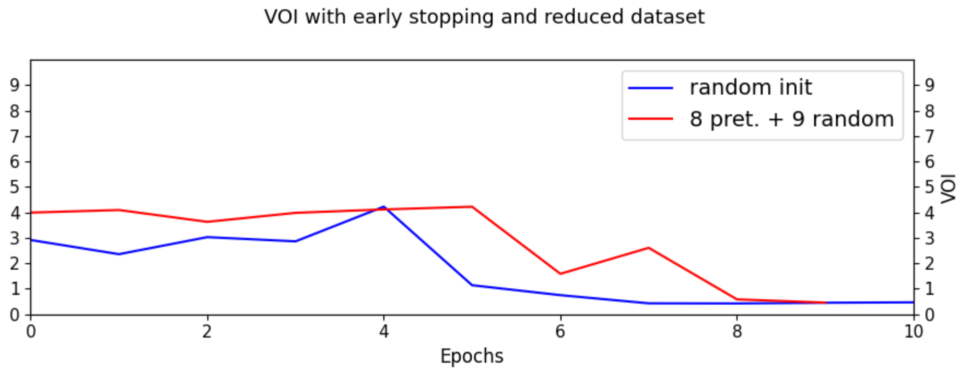

Figure 5.52: VOI when early stopping and using a small training set. In the graph we can see the comparisson between our model, with pretrained weights for the first 8 layers and randomly initialized the reminder, and our model randomly initialized.

	8 pretr. + 9 random	Random init.
Acc (%)	80.09	86.12
Voi	0.681	0.63

Table 5.19: Acc and voi for early stopping with reduced dataset

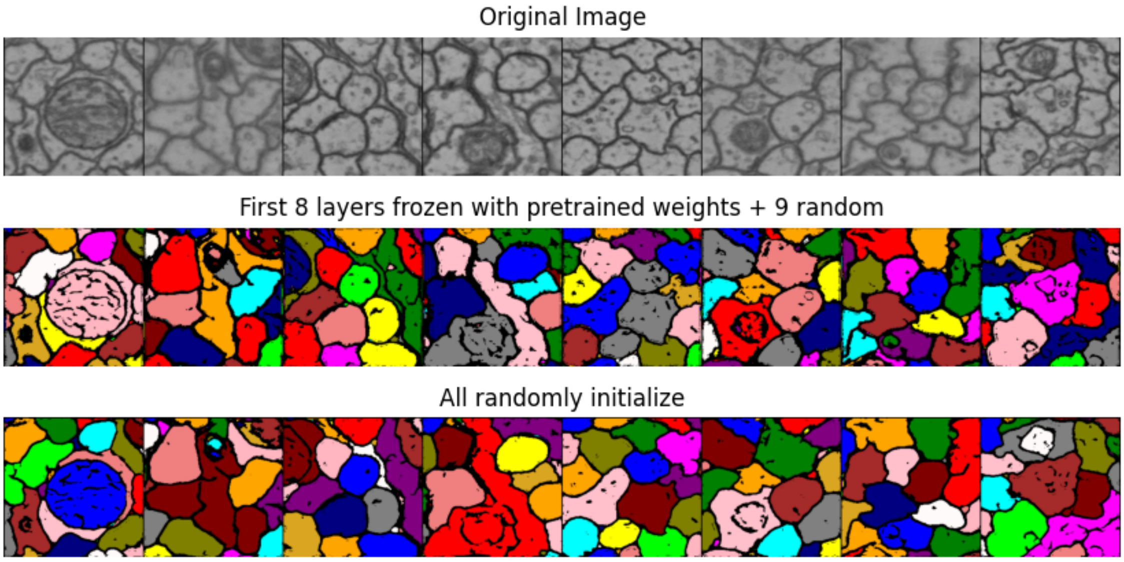

Figure 5.53: Segmentation results obtained after 10 epochs with a reduced training dataset.

6 Discussion

6.1 Segmentation UNet

As we have stated at the beginning of this work, the role of the supervised segmentation was to bring context to our task, to get familiar with the biomedical image segmentation and our specific dataset, and eventually obtain some results that we would study and try to improve through adversarial training. Eventually, the supervised segmentation has ended playing a big part in our study. We chose the UNet architecture for this end, because of its great background in matters of segmentation, and specifically with datasets that were quite related to ours. We have carried out training with both, a 2 dimensional and a 3 dimensional architecture of the UNet, in order to get more results and compare different outcomes.
We can see the results of training these two models in 5.2. If we follow along the training we can see that both are quite alike. We start evaluating the results of the training without tunning the thresholds of the sigmoid and the watershed, as mention in 3.2.8, so we took the same values ($sig\_th = 0.73$ and $watershed\_th = 0.81$) for better comparison. Starting taking a look at the pixel-accuracy graphs (5.4, 5.11), we can quickly get a first idea that both networks are learning and outputing decent predictions of the affinity maps. However, this plots are not a good source to check whether the training is heading towards nice results or not for two different reasons. First, we can not get fool by the high accuracy value. The percentage of the pixels that form part of the cell boundaries in our image might be approximately 10 times less than the actual pixels of cells. Therfore, we could be plotting a high acc when predicting completely white images. Secondly, as our goal is to perform good segmentation, the weight or importance tha every pixel plays in the image is not the same. If the network wrongly predicts a pixel in a place where the boundary width was 1, it would translate in an open boundary, which in terms of segmentation is way worse than failing in the prediction of several pixels that don't play a mayor role in the image and won't lead to open contours. Having said this, we can also notice that the validation accuracy is oddly close to the training accuracy, even higher, specially for the 3D-UNet 5.11. This is most likely do to the fact that we have introduced many data augmentations, to make up for the poor dataset, and so the the network more easily predicts the validation inputs (without any noise or blur) than the training ones.

Keeping on with the graphs of the training, the variance of information (5.5, 5.12) seems quite decent, perhaps slightly better in the 2 dimensional architecture. If we take close look to the VOI of both models, we can see how the voi of the 2D-UNet before and after applying watershed is quite similar whereas the voi of the 3D-UNet model after watershed seems to have improved more noticeably. This may be interpreted as the 3D-UNet not predicting good enough (could use a greater sigmoid threshold) and requiring the watershed algorithm to close some open boundaries. These graphs, in constrat with the pixel accuracy, are quite reliable, as the VOI informs us about the splits and the merges of the cells. Same goes for the ARand (5.8, 5.15), which measures the similarity between the groundtruth and the prediction as explained in 3.5.

In 5.1 , 5.10 we can see the results from both models on the validation set, for inputs of the same size used for training (160), farly smaller than the actual size we want to predict (1250x1250), thus the results are likely to be better than when feeding the network with a larger size. The results for both models are rather similar.

Once both models are trained, the next step is to fine-tuning, to find the most suitable values for the sigmoid and watershed threshold, so the VOI is as low as possible. We can see the contingency tables in 5.2 and 5.7. To understand the values and why we can not completly rely on them is necessary to clearly know the importance of both, the sigmoid and the watershed threshold, and the issue we faced when measuring the VOI. The sigmoid threshold is going to dermine whether the pixels from the probability maps take the value 0 or 1. In the other hand, the watershed threshold is going to determine whether a new potential cell border is added or not to the boundary map, after performing watershed. Summing up, the bigger the sigmoid th., the more number of pixels are consider boundaries, and the bigger the watershed th., the more new potential borders are added to the boundary map. If we take a look at the tables 5.3, 5.4, 5.8, 5.9, it makes sense that the largest oversegmentation happens for the highest sigmoid threshold and consequently, the largest undersegmentation happens for the lowest sigmoid threshold. However, the tricky situation arrives when measuring the Voi. The provided masks for the CREMI challenge did not have any boundaries, which means that all the pixels that are boundaries in our predictions are going to be classified as wrong, and the Voi is going to be useless. In contrast, we can set the Voi to ignore all the pixels that are considered boundaries in our predictions. That way, it will not classify the cell's borders as wrong. Nevertheless, it would also mean that if we use a larger sigmoid threshold, the boundaries would probably be wider, maybe some cells that happened to be opened before now are closed, but many more pixels would be ignored, hence a 'fake' Voi would be computed. So it seems that one procedure will lead to innacurate bad results and the other to innacurate good results respectively. There exist two ways to proper solve this: one would be to slim down the boundaries in our predictions as much as possible (without creating open boundaries in the task), and then computing the voi; the second one would be to add boundaries on the segmentation masks and to ignore the pixels that correspond to boundaries (would be advisable to slim down the boundaries of the predictions too). However, we had limited time to write this thesis, and we were not able to implement the mentioned.

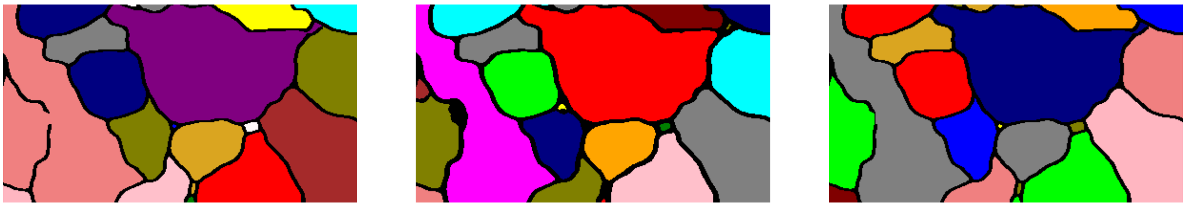

Figure 6.1 Figure 6.2 Figure 6.3

Figure 6.4: From left to right: $sigmoid\_th = 0.7$; $sigmoid\_th = 0.92$; $sigmoid\_th = 0.7$ and $watershed\_th = 0.8$

Therefore, as the contingency tables show the results when ignoring the boundary pixels of the predictions, the threshold selection criteria has not being choosing the ones that produced the lower Voi (the ones highlighted), but choosing an intermidiate value to get the less possible open borders and at the same time avoid too wide boundaries that fake the results and may even turn some small cells into a wide boundary region. The combination of a sigmoid threshold of 0.81 and a watershed threshold of 0.91 give us the wanted results. An example of our described issue is ilustrated in 6.4. For a low sigmoid threshold, the predicted segmentation produces an open contour, whereas if we use an extremely high sigmoid threshold, we manage to close it by computing too wide contours that decrease the segmentation quality, even though the Voi as implemented is not going to reflect it. The image 6.3 shows the best solution, where we don't use an extraordinary high sigmoid threshold, thus the boundaries are kept thin, but we rely on a high watershed threshold value, so the feasible open contours are clossed. Moreover, we want to point out that is better to choose high values that would produce oversegmentation that having to deal with undersegmentation.

Finally, if we evaluate the results we see that our 2D-Unet slightly outperforms the 3D-Unet 5.11 (note that the table shows the best submission of the CREMI challenge [Fun+], but they add background around object boundaries to the groundtruths before evaluation, so the comparison with our results is not quite reliable). Is not an easy task to compare both networks as they differ in several aspects. In order to carry out the trainings of both models as close as possible we used the same hyperparameters, based on previous documentation [Lee+17]. First, there's the plain difference in the dimensionality of the data they process. The 2D-Unet takes flat images as inputs whereas the 3D-Unet takes volumes. This makes it harder to compare, as we can not pass the exact same data to both of them. Let's say we want to input a dataset composed of 6000 different random crops, with flips, rotations, blur and noise randomly applied, to the 2D-Unet. For a depth of 18, the equivalent number of volumes for the 3D-Unet would be almost 334 (6000/18). However, the 18 slices of each of the volumes, as they belong to the nearby slices, have similar information, thus the 3D-Unet would see less different information than the 2D-Unet. In the other hand, if we decide to input 6000 volumes, we could say that both models see the same amount of different data (clearly saving the distances), but this way the 3D-Unet would take as inputs the aproximately equivalent of 18 times the dataset of the 2D-Unet. As we did not find a proper way to equally input data, we used a fix dataset for the 2D-UNet and for the 3D-UNet we applied the transformations in running time.

The architectures of both models do have some differences too. The 3D-Unet implementation adopted from [Lee+17] uses residual blocks and skip connections (by addition), whereas the 2D-UNet model do not have residual blocks and performs the skip connections by concatenation. This could be a determinant factor, as when running trainings we noticed that the residual blocks were counterproductive for the 2D-UNet.

Another factor could be the width or number of feature channels. Both models share the same depth, but the 2D-UNet is notoriously wider, from 64 to 1024 filters or channels, whereas the 3D-UNet goes from 28 to 80 channels. This translates in a difference from 31,036,481 trainable parameters in the 2 dimensional model to 923,201 trainable parameters in the 3 dimensional model. Over the years, the networks have become deeper over wider, as it's been proven that more layers usually means that a larger number of different features are learned. However, a very wide network may easily lead to overfitting (if not deep enough), thus not successfully generalizing. Nevertheless, this does not seem to be the case, and the amount of learnable parameters could play a major role in the results.

Regarding some improvements that we could have applied to our implementation having had more time, we could mention first mention the implementation of an algorithm to slim down the boundaries as mentioned before, thus obtaining better overall sementation results and being able to better measure the predictions with the standard procedures (Voi, ARand...). Secondly we could have improved the way we measure the affinity aglomerations, i.e. the way we decide wether a potential new contour computed by the watershed algorithm is added or not to our boundary map. It is likely that computing the mean for a wider range of neighbour pixels would have produced better results. Finally, finding an alternative way to solve the problem of the open boundaries would have been valuable.

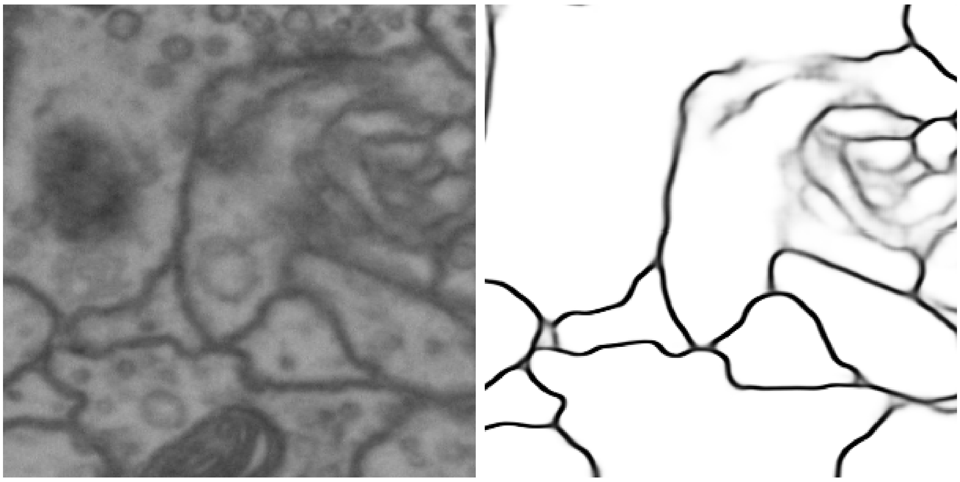

Figure 6.5: Crop from an input of the validation set and the probability map outputed.

We have noticed that the open boundary issue happens in locations of the images where the borders are not proper defined or they are some kind blurred. Having the problem detected, we could have thought of some kind of addition to our network in order to solve it, for example, located blur inside an image as part of data augmentations to try to reproduce as close as possible the original one. Another way to approach this matter would be to introduce adversarial training, to use the UNet as the generator and pass to the discriminator the affinity maps (targets of the UNet) and the probability maps. As an open boundary is easily detectable, it may be quite straightforward for the whole model to output generated predictions without the open contours.

6.2 Adversarial training

The first approach we had to Generative Adversarial Networks in this work was the Style Gan. The idea of running experiments with this gan was mainly to see how our dataset reacted to adversarial training, and decide whether to take a step further and keep with the goal of introducing the adversarial training in the segmentation task. Therefore, we did not stay longer running experiments with the Style Gan. If we take a quick look at the graph 5.1 we can see how the training seemed to be carried out succesfully, and then the outputed data confirmed it 5.3. The predictions of the gan were remarkably good, and after minor contrast adjusment, were almost impossible to distinguish from the real ones. Therefore, it became also a feasible valuable resource to extend our dataset, even though we did not use it at the end. In the light of the results, we moved forward and searched for a potential suitable gan for our objective. We decided to try with the gan proposed in [Tet+19] for image extension, as the results were quite inspiring and the extracted information from the images fairly precise. We can see the computation of different losses throughout the training of the image extension in the figures 5.31, 5.32, for the mask on the right side of the image, 5.34, 5.35, for the square mask in the middle of the image, and 5.37, 5.38, for the square mask with random size and location. We can see how after 150 epochs, almost 19 hours of training, the results were quite decent, and how the validation and loss curves were quite alike, achieving good generalization on the unseen data. If we take a deeper look into the graphs, we can notice that the model does not only generalize well, but produce better results on the validation set. This is because, as well as we mentioned before in the segmentation with the UNet discussion, we performed several data augmentations to the training set, thus the samples from the validation set (without the gaussian blur and noise) are in some way easier to predict. Before this experiment, we run the training for image extension without introducing the blur and gaussian noise and we could clearly see by the graphs that the model was not generalizing 6.6.

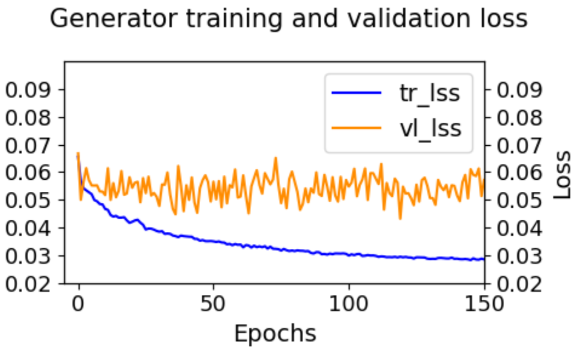

Figure 6.6: Example of the generator's loss throughout the training for image extension without Gaussian noise and blur as data augmentations.

The predicted extensions of the images for the three different masks were rather satisfactory 5.30, 5.33, 5.36, generating with more or less accuracy the shapes of the cells and producing good textures. There is not significant difference in the image extension in terms of the chosen mask. Both the mask in the right side of the image, and the square mask in the middle, output great reconstructions, even though the mask in the middle might produce better results as it has more context than the right mask, as being surrounded by the original image, whereas the right mask has only one side in contact with the information. In the other hand, the randomly located/sized square mask seems to produce nice results as well, but we have noticed that when it reaches the size of the fixed-sized square mask, the prediction gets a little blurry and more innacurate.

Once we had the convincing results in the image extension gan, the last step was to separete the generator of the image extension gan from the rest of the model and use it as a neural network to compute the segmentation. In order to load the pretrained weights to the former generator we had to decide which mask's pretrained weights we were going to use, looking forward the best possible results. This evaluation is carried out in 5.4. We can observe in the graphs 5.39 and 5.40 that the accuracy and the variance of information for the segmentation task, when loading the pretrained weights from the three different mask shapes, are fairly similar. This training was carried out freezing the first 8 layers, which means that these layers did not requiere gradients throughout the training, thus their weights were not updated. This is a common practice when applying transfer learning as the idea is to preserve the learnt features from the previous model, to see if they can help improve other often related task. Before running any further experiment, we decided to freeze the first 8 layers, as those layers made up the enconder of the model (section of the model where the image is downsampled and the number of feature channels is increased), and was the section of the former generator where the features could had been learnt. Going back to the evaluation of the graphs, the fact that the accuracy and the voi of the model were almost identical for the three different sets of pretrained weights, could have already indicate at an early stage that either there was not significant variance in the learnt features from one kind of mask to the other, or the transfer learning was probably not going to improve the training. We conclude to run the following transfer learning experiments with the pretrained wieghts from the right mask, as it got the lowest score on the voi before performing watershed.

From now on, all the remaining experiments that we carried out show that the transfer learning did not work out as expected. We started by evaluating the influence of the number of frozen layers on the segmentation, measuring the accuracy and the variance of information. We can see in the graphs 5.43 and 5.44 how the more layers frozen, the worse the accuracy and the results, and how from 4 to 8 frozen layers the values were quite similar. This was the first sign that made us realised that the transfer learning might actually not work. We were expecting to see some kind of inverted 'u shape', i. e. to record the best values with a number of frozen layers close to the encoder's end (6..8th layer), and the worst values in the ends of the whole architecture, with either 1 or 16 layers. This could be the expected output if the pretrained weights had extracted useful features, as freezing only the early layers would mean that the weights from the middle-end section of the encoder would be updated, and some convenient features might be lost, and freezing layers close to the architecture's end would prevent the layers from the decoder to update their weights, when those layers from the middle-end of the decoder are not likely to have extracted functional features. This is not completly true as freezing only a few layers would most probably output better results than freezing until almost the end of the model in most cases, especially if we do not apply early stopping and let it train for many epochs, because the first case would have many learnable parameters whereas the second only a few from the last layers. Summing up we can say that it depends on many different factors. However, the fact that the graphs show that the results are better when more layers are able to udpate their weights is a first evidence that the extracted features may not be suitable.

The goal of the next experiment 5.5.1 is to run test with and without the pretrained weights using the same architecture to clearly see if they have any influence in the segmentation task. In the graphs 5.46 and 5.47 we evaluate four different training scenarios: all the layers randomly initialized and none frozen, all layers with the pretrained weights and none frozen, all layers randomly initialized and the first 8 (encoder) frozen, and the first 8 layers with the pretrained weights loaded and frozen and the remaining randomly initialized. We see that there's not significant difference between randomly initialization and loading the pretrained weights when all layers do require gradients. This two tests clearly outperform the test with the 8 first layers frozen with the pretrained weights, which is also a strong evidence that the extracted features are not useful. In the other hand, the test with the first 8 layers frozen and randomly initialized outputs notoriously worse results than the test with the 8 first layers pretrained and frozen. This does not provide reliable information because as the layers are randomly initialized, by a matter of chance the first layers might be initialized more or less adequately for our task, and the outputs would differ from test to test.

As we metioned before, if we evaluate these experiments for many epochs and with a large enough dataset, the feasible advantage that may have the model with pretrained weights might pass unnoticed, as we give the other model enough time to update its weights and adjust to the targets. For that reason we run an experiment for only a few epochs and a reduced dataset. 5.5.3 We have to be careful, because using a dataset too small or containing samples with similar information (basically crops from nearby slices in the same coordinates) would fake the results, as the models would easily overfit. We see in the graphs 5.51 and 5.52 how the at an early stage of the training, and with a small dataset, the results from the randomly initialized model are once again better than the results from the model with pretrained weights.

Finally, in 5.49 and 5.50 we provide some justification of why we did not use for the experiments the same hyperparameters that were used to perform the image extension, even though we adopt its architecture. We can see how choosing the hyperparameters from [Lee+17] that we used for the segmentation clearly outperforms the hyperparameters from the image extension [Tet+19]. As the learning rate for the image extension is smaller than the one used for segmentation, we run the training for 200 more epochs. We can see in 5.18 that still, after more epochs, the hyperparameters from [Lee+17] are the most suitable for this task.

After all the studied experiments we conclude that the chosen adversarial training is not suitable for our segmentation task, and does not improve performance, neither shows any clear evidence of conditioning our segmentation task. Further study would be needed to shed light on the reasons behind, but due to the limited time we had to write this work, we were not in the position to go any deeper. The most straightforward reason might be that the learnt features from the image extension task are not suitable for transfer learning to our segmentation task and do not contain valuable information related with the boundaries detection. Another reason might be the diferences regarding the training of the image extension and the segmentation. The one that might be most obvious is the need of feeding the model with masks along with the image to extend in the image extension training. When we adopt the generator architecture to perform segmentation with the pretrained weights, the empty masks that we have to input along with the image that we want to segment loose their value, and therfore, might interfere in the transfer learning.

Bibliography

[Goo+14] Ian J. Goodfellow et al. *Generative Adversarial Networks.* 2014. arXiv: `1406.2661 [stat.ML]`.

[Wal+14] Stéfan van der Walt et al. "scikit-image: image processing in Python". In: *PeerJ* 2 (June 2014), e453. ISSN: 2167-8359. DOI: `10.7717/peerj.453`. URL: `https://doi.org/10.7717/peerj.453`.

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: `1502.03167`. URL: `http://arxiv.org/abs/1502.03167`.

[KB15] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.* Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: `http://arxiv.org/abs/1412.6980`.

[RFB15] 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`.

[Çiç+16] 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`.

[QHJ16] Tran Minh Quan, David G. C. Hildebrand, and Won-Ki Jeong. "FusionNet: A deep fully residual convolutional neural network for image segmentation in connectomics". In: *CoRR* abs/1612.05360 (2016). arXiv: `1612.05360`. URL: `http://arxiv.org/abs/1612.05360`.

[UVL16] Dmitry Ulyanov, Andrea Vedaldi, and Victor S. Lempitsky. "Instance Normalization: The Missing Ingredient for Fast Stylization". In: *CoRR* abs/1607.08022 (2016). arXiv: `1607.08022`. URL: `http://arxiv.org/abs/1607.08022`.

[Lee+17] Kisuk Lee et al. "Superhuman Accuracy on the SNEMI3D Connectomics Challenge". In: *CoRR* abs/1706.00120 (2017). arXiv: `1706.00120`. URL: `http://arxiv.org/abs/1706.00120`.

[DV18] Vincent Dumoulin and Francesco Visin. *A guide to convolution arithmetic for deep learning.* 2018. arXiv: `1603.07285 [stat.ML]`.

[KLA18] Tero Karras, Samuli Laine, and Timo Aila. "A Style-Based Generator Architecture for Generative Adversarial Networks". In: *CoRR* abs/1812.04948 (2018). arXiv: `1812.04948`. URL: `http://arxiv.org/abs/1812.04948`.

[Seo18] Kim Seonghyeon. *Style Based Gan Pytoch.* `https://github.com/rosinality/style-based-gan-pytorch.git`. 2018.

[Yu+18] Jiahui Yu et al. "Free-Form Image Inpainting with Gated Convolution". In: *CoRR* abs/1806.03589 (2018). arXiv: `1806.03589`. URL: `http://arxiv.org/abs/1806.03589`.

[Gro19] Zudi Lin - Visual Computing Group. *Pytorch Connectomics: segmentation toolbox for EM connectomics.* `https://github.com/zudi-lin/pytorch_connectomics.git`. `https://connectomics.readthedocs.io/en/latest/index.html`. 2019.

[Ish19] Ryogo Ishikawa. *Boundless: Generative Adversarial Networks for Image Extension in Pytorch.* `https://github.com/recong/Boundless-in-Pytorch.git`. 2019.

[Tet+19] Piotr Teterwak et al. "Boundless: Generative Adversarial Networks for Image Extension". In: *CoRR* abs/1908.07007 (2019). arXiv: `1908.07007`. URL: `http://arxiv.org/abs/1908.07007`.

[Fun+] Jan Funke et al. *CREMI: MICCAI Challenge on Circuit Reconstruction from Electron Microscopy Images.* `https://cremi.org/`.

7 Appendix

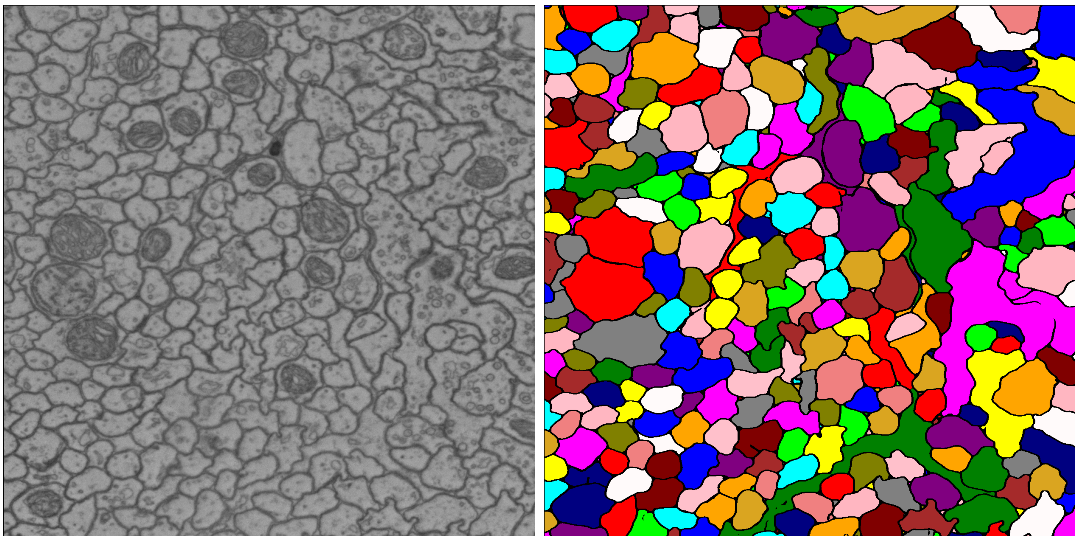

Figure 7.1: 1250x1250 slice from the test set and the predicted segmentation.

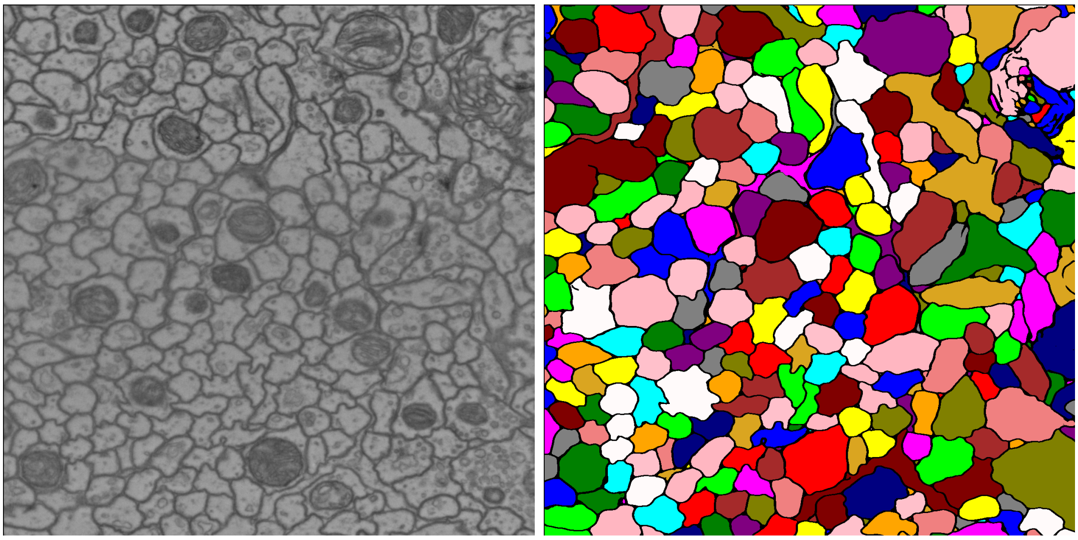

Figure 7.2: 1250x1250 slice from the test set and the predicted segmentation.

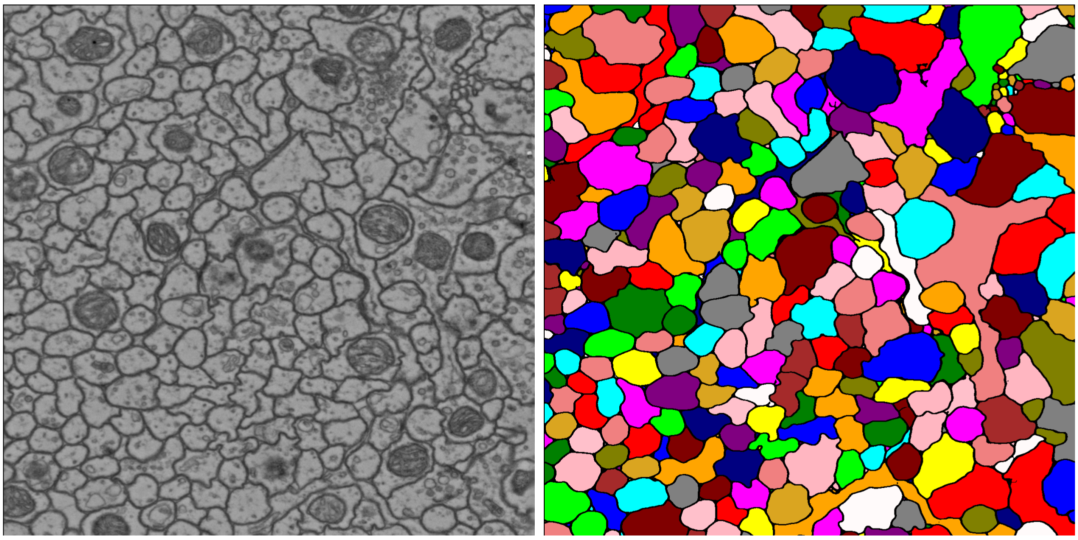

Figure 7.3: 1250x1250 slice from the test set and the predicted segmentation.

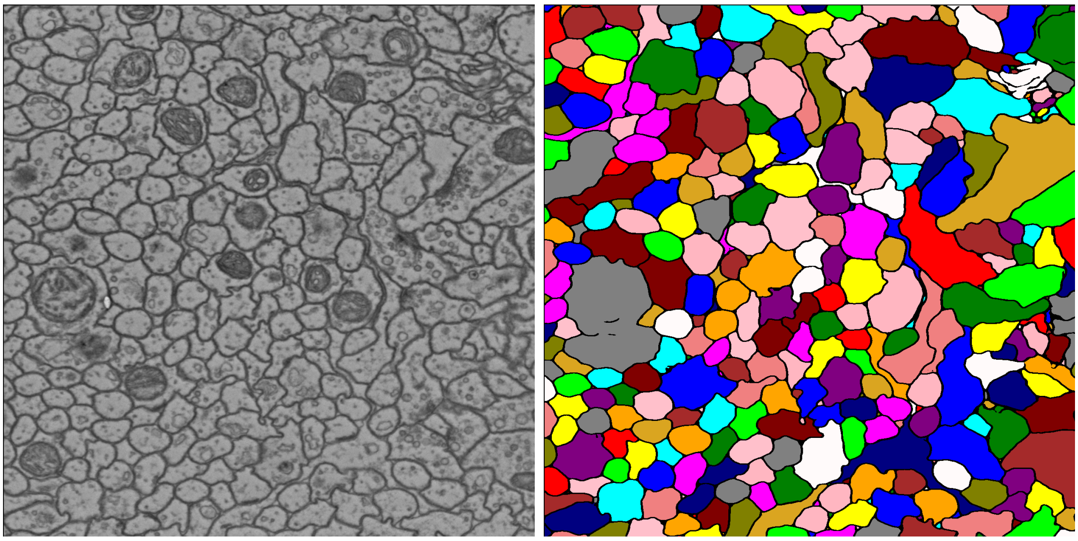

Figure 7.4: 1250x1250 slice from the test set and the predicted segmentation.

Statement of authorship

I hereby certify that I have authored this document entitled *Generative Adversarial Networks for Electron Microscopy Image Segmentation* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, August 18, 2021

Eduardo Herreros