Alma Mater Studiorum · University of Bologna

School of Science
Department of Physics and Astronomy
Master Degree in Physics

Dataset Generation for the Training of Neural Networks Oriented toward Histological Image Segmentation

Supervisor:
Dr. Enrico Giampieri

Submitted by:
Alessandro d'Agostino

Academic Year 2019/2020

Acknowledgements:

Abstract

Abstract.....

Contents

Introduction

In the last decades, the development of Machine Learning (ML) and Deep Learning (DL) techniques has contaminated every aspect of the scientific world, with interesting results in many different research fields. The biomedical field is no exception to this and a lot of promising applications are taking form, especially as Computer-Aided Detection (CAD) systems which are tools coming in support to physicians during the diagnostic process. Medical doctors and the Healthcare system in general collect a huge amount of data from patients during all the treatment, screening, and analysis activities in many different shapes, from anagraphical data, to blood analysis to clinical images.

In medicine the study of images is ubiquitous and countless diagnostic procedures rely on it, such as X-ray imaging (CAT), nuclear imaging (SPECT, PET), Magnetic resonance, and visual inspection of histological specimens after biopsies. The branch of Artificial Intelligence in the biomedical field that handles image analysis to assist physicians in their clinical decisions goes under the name of Digital Pathology Image Analysis (DPIA). In this thesis work, I want to focus on some of the beneficial aspects introduced by DPIA in the histological images analysis and some particular issues in the development of DL models able to handle this kind of procedure.

Nowadays the great majority of analysis of histological specimens occurs through visual inspection, carried out by highly qualified experts. Some analysis, as cancer detection, requires the ability to distinguish if a region of tissue is healthy or not with high precision in very wide specimens. This kind of procedure is typically very complex and requires prolonged times of analysis besides substantial economic efforts. Furthermore, the designated personnel for this type of analysis is often limited, leading to delicate issues of priority assignment while scheduling analysis, based on the estimated patient's clinical development. Some sort of support to this analysis procedure is therefore necessary.

The problem of recognizing regions with different features within an image and detect their borders is known in computer vision as segmentation task, and it's quite spread in many different applications, allowing a sort of automatic interpretation of the image. The segmentation problem is usually faced as a supervised task, hence the algorithm in order to be trained properly requires a reasonable quantity of pre-labeled images, from which learn the rules through which distinguish different regions. This means that the development of segmentation algorithms for a specific application, as would be the

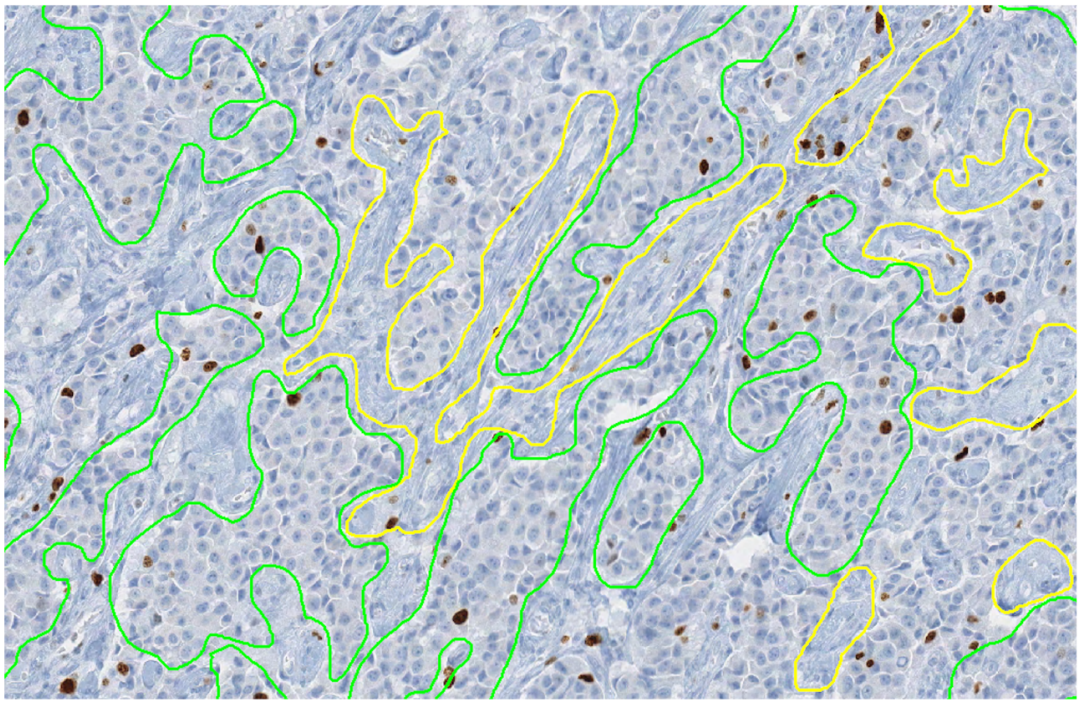

Figure 1: *Interleaving of tumor (green annotation) and non-tumor (yellow annotation) regions [15].*

one with histological images, requires a lot of starting material independently analyzed from the same qualified expert encharged of the visual inspection I mentioned before. A human operator thus is required to manually track the boundaries, for example, between healthy and tumoral regions within a sample of tissue and to label them with their identity, as in Figure 1. The more the algorithm to train is complex the more starting material is required to adjust the model's parameters and reach the desired efficacy.

The latest developed segmentation algorithms are based on DL techniques, hence based on the implementation of intricated Neural Networks (NN) which process the input images end produces the correspondent segmentation. Those models are typically very complex, with millions of parameters to adjust and tune, therefore they need a huge amount of pre-labeled images to learn their segmentation rules. This need for data is exactly the main focus of my thesis work. The shortage of ground truth images is indeed one of the toughest hurdles to overcome during the development of DL-based algorithms. Another important aspect to bear in mind is the quality of the ground truth material. It's impossible for humans to label boundaries of different regions with pixel-perfect precision, while for machines the more precise is the input the more tuned is the resulting algorithm.

There have already been explored different approaches to overcome this problem, and

they are mainly based on the generation of synthetic data to be used during the training phase. Some techniques achieve data augmentation manipulating already available images and then generating "new" images, but as we will see later this approach suffers from different issues. The technique that I propose in this work follows a generation from scratch of entire datasets suitable for the training of new algorithms, based on the 3D modelization of a region of human tissue at the cellular level. The sectioning of the virtual histological samples yields the synthetic images with their corresponding ground truth. Using this technique one would be able to collect sufficient material for the training (the entire phase or the preliminary part) of a model, avoiding then the shortage of hand-labeled data.

The 3D modeling of a region of particular human tissue is a very complex task, and it is almost impossible to capture all the physiological richness of a histological system. The models I implemented thus are inevitably schematic in their representation of the target biological structures. I'll show two models: one of pancreatic tissue and another of epidermic tissue, besides all the tools I used and the choices I made during the design phase.

In order to present organically all the steps of my work the thesis is organized in chapters as follows:

1. Structure
2. Of the
3. Thesys

Chapter 1

Histo & Deep Learning & SOA

1.1 Histological Images

Description of followed Approach

1.1.1 Traditional Preparation of Histological Samples

How to prepare a sample

1.1.2 Important Aspects

Problems with hand-labeling Important aspects in final images

1.2 Introduction to Deep Learning

Deep Learning is part of the broader framework of Machine Learning and Artificial Intelligence. Indeed all the problems typically faced using ML can also be addressed with DL techniques, for instance, regression, classification, clustering, and segmentation problems. We can think of DL as a universal methodology for iterative function approximation with a great level of complexity. In the last decades, this technology has seen a frenetic diffusion and an incredible development, thanks to the always increasing available computational power, and it has become a staple tool in all sorts of scientific applications.

1.2.1 Perceptrons and Multilayer Feedforward Architecture

As other artificial learning techniques, those models aim to "learn" a relationship between some sort of input and a specific kind of output. In other words, approximating numeri-

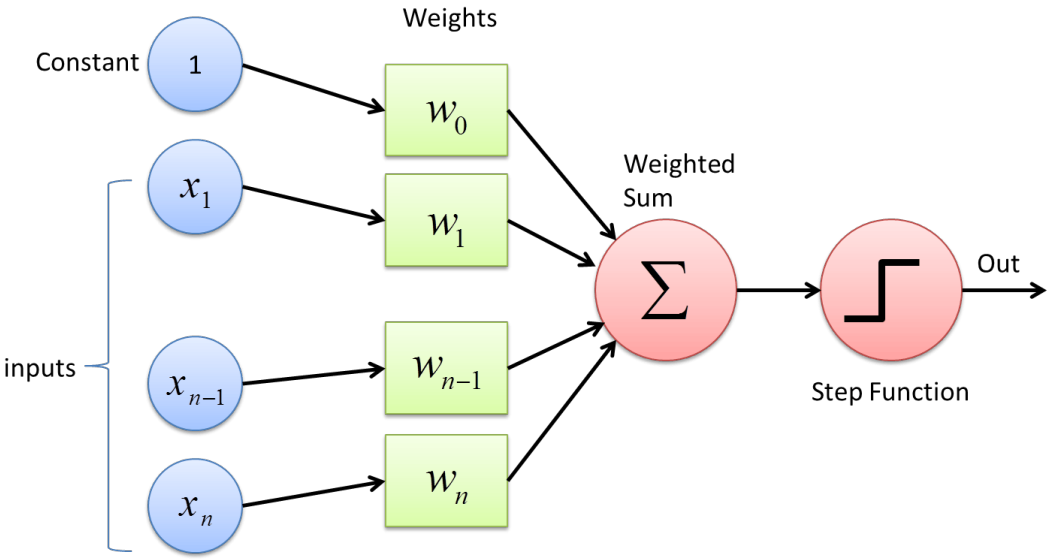

Figure 1.1: *Schematic picture of a single layer perceptron. The input vector is linearly combined with the bias factor and sent to an activation function to produce the numerical output.*

cally the function that processes the input data and produces the desired response. For example, one could be interested in clustering data in a multidimensional features space, or in the detection of objects in a picture, or in text manipulation and generation. The function is approximated by means of a greatly complex network of simple linear and non-linear mathematical operations arranged in a so-called Neural Network (typically with millions of parameters). In fact, the seed idea behind this discipline is to recreate the functioning of actual neurons in the human brain: their entangled connection system and their "ON/OFF" behavior [21].

The fondamental unit of a neural network is called perceptron, and it acts as a digital counterpart of a human neuron. As shown in Figure 1.1 a perceptron collects in input a series on n numerical signals $\vec{x} = 1, x_1, ..., x_n$ and computes a linear wieghted combination with the weights vectors $\vec{w} = w_0, w_1, ..., w_n$, where w_0 is the bias vector:

$$f(\vec{x}, \vec{w}) = \chi(\vec{x} \cdot \vec{w}) \tag{1.1}$$

The results of this linear combination is given as input to a non-linear function $\chi(x)$ called activation function. Typical choices as activation function are any sigmoidal function like $sign(x)$ and $tanh(x)$, but in more aadvanced applications other functions like ReLU [1] are used. The resulting function $f(\vec{x}, \vec{w})$ has then a simple non linear behaviour. It produces a binary output: 1 if the weighted combination is high enough, 0 otherwise.

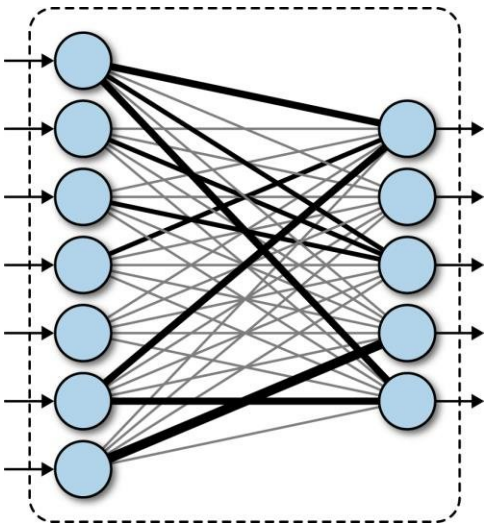

Figure 1.2: *Schematic representation of a fully connected (dense) layer. Every neuron from the first layer is connected with every output neuron. The link thickness represent the absolute value of the combination weight for that particular value.*

The most common architecture for a NN is the so-called feed-forward architecture, where many individual perceptrons are arranged in chained layers, which take as input the output of previous layers along the information flux. More complex architectures could implements also recursive connection, linking a layer to itself, but it should be regarded as an exception to the standard case. There are endless possibilities of combination and arrangement of neurons inside a NN but the most simple one is known as fully connected layer, where every neuron is linked with each toher neuron of the following layer, as shown in Figure 1.2. Each connection has its weight, which contributes to modulate the overall combination of signals. The training of a NN consists then in the adjustment and fine-tuning of all the Network's weights and parameters through iterative techniques until the desired precision is reached in the output generation.

Although a fully connected network represents the simplest linking choice, the insertion of each weight increases the number of parameters, and so the complexity of the model. Thus we want to create links between neurons smartly, avoiding the less useful ones. Depending on the type of data under analysis there are many different established typologies of layers. For example, in the image processing field, the most common choice is the convolutional layer, which implements a sort of discrete convolution on the input data, as shown in Figure 1.3. While processing images, the convolution operation confers to the perception some kind of correlation between adjacent pixels of an image and their color channels, allowing a sort of spatial awareness.

As a matter of principle a NN with just two successive layers, which is called a

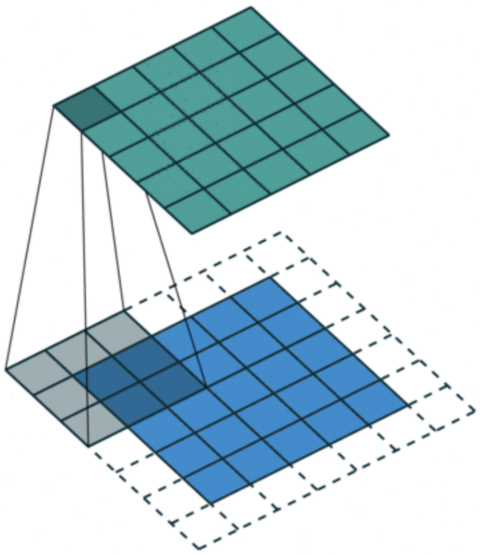

Figure 1.3: *Schematical representation of a convolutional layer. The input data are processed by a window kernel that slides all over the image.*

shallow network, and with an arbitrary number of neurons per layer, can approximate arbitrary well any kind of smooth enough function [16]. However, direct experience suggests that Networks with multiple layers, called *deep* networks, can reach equivalent results exploiting a lower number of parameters overall. This is the reason why this discipline goes under the name of *deep* learning: it focuses on deep networks with up to tents hidden layers. Such a deep structure allows the computation of which are called deep features, so features of the features of the input data, that allows the network to easily manage concepts that would be bearly understandable for humans.

1.2.2 Training of a NN - Error Back-Propagation

Depending on the task the NN is designed for, it will have a different architecture and number of parameters. Those parameters are initialized to completely random values, tough. The training process is exactly the process of seeking iteratively the right values to assign to each parameter in the network in order to accomplish the task. The best start to understanding the training procedure is to look at how a supervised problem is solved. In supervised problems, we start with a series of examples of true connections between inputs and correspondent outputs and we try to generalize the rule behind those examples. After the rule has been picked up the final aim is to exploit it and to apply it to unknown data, so new problem could be solved. In opposition to the concept of supervised problems there are the *unsupervised* problems, where the algorithm do not try

Figure 1.4: *Sample grid of images from the CIFAR10 dataset. Each one of the 32 × 32 image is labeled with one of the ten classes of objects: plane, car, bird, cat, deer, dog, frog, horse, ship, truck.*

to learn a rule from practical example but try to devise it from scratch. A task typically posed as unsupervised is clustering, when different data are separated in groups based on the values of their features in the feature space. Usually only the number of groups is taken in input from the algorithm, and the subdivision is completely performed by the machine. In the real world by the way, there are many different and sophisticated shapes between pure supervised and pure unsupervised learning, based on the actual availability of data and specific limitations to the individual task.

A good example of supervised problems is the classification of images. Let's assume we have a whole dataset of pictures of different objects as cats, dogs, cars, etc. like the CIFAR10 [11] dataset. This famous dataset is made of over $60K$ labeled images 32×32 divided into 10 categories of objects as shown in Figure 1.4. We could be interested in the creation of a NN able to assign at every image its belonging class. This NN could be arbitrarily complex but it certainly will take as input a $32\times32\times3$ RGB image and the output will be the predicted class. A typical output for this problem would be a probability distribution over all the 10 classes like:

$$\vec{p} = (p_1, p_2, \ldots, p_{10}), \tag{1.2}$$

$$\sum_{i=1}^{10} p_i = 1, \tag{1.3}$$

and it should be compared with the truth, that is represented just as a binary sequence $\vec{t}$ with the bit correspondent to the belonging class set as 1, and all the others value set to 0:

$$\vec{t} = (0, 0, \ldots, 1, \ldots, 0, 0). \tag{1.4}$$

Every time an image is given to the model an estimate of the output is produced. Thus, we need to measure the 'distance' between that prediction and the true value, to quantify the error made by the algorithm and try to improve the model's predictive power. The functions used for this purpose are called loss functions. The most common choice is the Mean Squared Error function that is simply the averaged L^2 norm of the difference vector between $\vec{p}$ and $\vec{t}$:

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(t_i - p_i)^2. \tag{1.5}$$

Let's say the NN we are training has L concecutive layers, each one with its activation function f^k and its wights vector $\vec{w}^k$, hence the prediction vector $\vec{p}$ could be seen as the result of the concesutive, nested, application through all the layers:

$$\vec{p} = f^L(\vec{w}^L \cdot (f^{L-1}(\vec{w}^{L-1} \cdot ... \cdot f^1(\vec{w}^1 \cdot \vec{x})))). \tag{1.6}$$

From both 1.5 and 1.6 it is clear that the loss function could be seen as a function of all the weights vectors of every layer of the network. So if we want to reduce the distance between the NN prediction and the true value we need to modify those weights to minimize the loss function. The most established algorithm to do so for a supervised task in a feed-forward network is the so-called Error Back-Propagation. The backpropagation method works essentially computing the gradient of the loss function with respect to the weights using the derivative chain rule and updating by a small amount the value of each parameter to lower the overall loss function. Each weight is *moved* counter-gradient, and summing all the contribution to every parameter the loss function approaches its minimum. In equation 1.7 is represented the variation applied to the j^{th} weight in the i^{th} layer:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}, \tag{1.7}$$

where E is the error function, and η is the *learning coefficient*, that modulate the effect of learning through all the training process. This iterative procedure is applied completely to each image in the training set several times, each time is called an *epoch*. The great majority of the dataset is exploited in the training phase to keep running this trial and error process and just a small portion is left out (typically 10% of the data) for a final performance test.

The loss function shall inevitably be differentiable, and its behavior heavily influences the success of the training. If the loss function presents a gradient landscape rich of local minima it's very probable that the gradient descent process would get stuck in one of them. More sophisticated algorithms capable of avoiding this issue have been devised, with the insertion of some degree of randomness in them, as the Stochastic Gradient Descent algorithm, or the wide used *Adam* optimizer [10].

The training phase is the pulsing heart of a DL model development and it could take even weeks on top-level computers for the most complicated networks. In fact, one of the great limits to the complexity of a network during the designing phase is exactly the available computational power. There are many more further technical details necessary for proper training, the adjustment of which can heavily impact on the quality of the algorithm.

However, after the training phase, we need to test the performance of the NN. This is usually done running the trained algorithm on never seen before inputs, the test dataset, and comparing the prediction with the ground-truth value. A good way to evaluate the quality of the results is to use the same function used as loss function during the training, but there is no technical restriction to the choice of this quality metric. The average score on the whole test set is then used as a numerical score for the network, and it allows straightforward comparison with other models, trained for the same task.

All this training procedure is coherently customized to every different application, depending on which the problem is posed as supervised or not and depending on the more or less complex network's architecture. The leitmotif is always finding a suitable loss function that quantifies how well the network does what it has been designed to do and trying to minimize it, operating on the parameters that define the network structure.

1.3 Deep Learning-Based Segmentation Algorithms

In digital image processing, image segmentation is the process of recognizing and subdividing an image into different regions of pixels that show similar features, like color, texture, or intensity. Typically, the task of segmentation is to recognize the edges and boundaries of the different objects in the image and assigning a different label to every detected region. The result of the segmentation process is an image with the same dimensions of the starting one made of solid color regions, representing the detected objects. This image is called *segmentation mask*. In Figure 1.5 is shown an example of segmentation of a picture of an urban landscape: different colors are linked to different classes of objects like persons in magenta and scooters in purple. This technology has a significant role in a wide variety of application fields such as scene understanding, medical image analysis, augmented reality, etc.

A relatively easy problem and one of the first to be tackled could be distinguishing an object from the background in a grey-scale image. The easiest technique to per-

Figure 1.5: *Example of the resulting segmentation mask of an image of an urban landscape. Every interesting object of the image is detected and a solid color region replaces it in the segmentation mask. Every color corresponds to a different class of objects, for example, persons are highlighted in magenta and scooters in purple. The shape and the boundaries of every region should match as precisely as possible the edges of the objects.*

form segmentation in this kind of problem is based on thresholding. Thresholding is a binarization technique based on the image's grey-level histogram: to every pixel with luminosity above that threshold is assigned the color *white*, and vice versa the color *black*. However, this is a very primitive and fallacious yet very fast method, and it manages poorly complex images or images with un-uniformity in the background.

Many other traditional techniques improve this first segmentation method. Some are based on the object's edges recognition, exploiting the sharp change in luminosity typically in correspondence of the boundary of a shape. Other techniques exploit instead a region-growing technology, according to which some *seed* region markers are scattered on the image, and the regions corresponding to the objects in the image are grown to incorporate adjacent pixels with similar properties.

1.3.1 State of the Art on Deep Learning Segmentation

Similarly to many other traditional tasks, also for segmentation, there has been a thriving development lead by the diffusion of deep learning, that boosted the performances resulting in what many regards as a paradigm shift in the field [14].

In further detail, image segmentation can be formulated as a classification problem of pixels with semantic labels (semantic segmentation) or partitioning of individual objects (instance segmentation). Semantic segmentation performs pixel-level labeling with a set of object categories (e.g. boat, car, person, tree) for all the pixels in the image, hence it is typically a harder task than image classification, which requires just a single label for the whole image. Instance segmentation extends semantic segmentation scope further by detecting and delineating each object of interest in the image (e.g. partitioning of individual nuclei in a histological image).

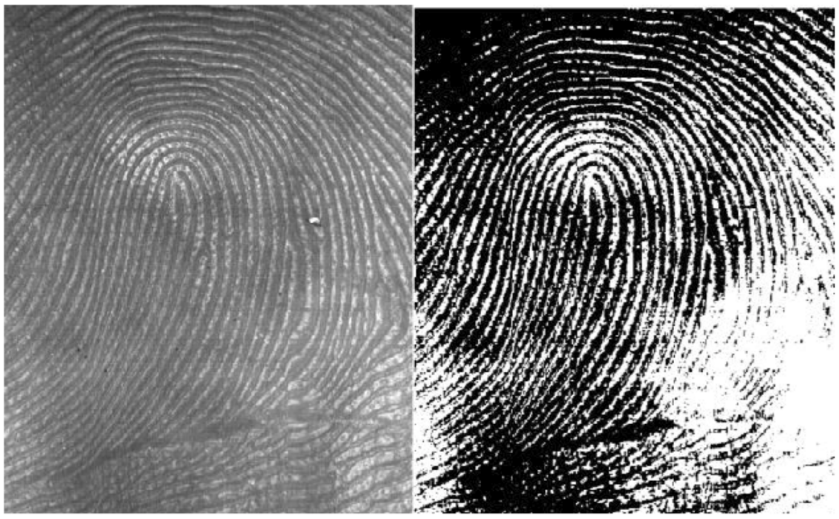

Figure 1.6: *Example of the resulting segmentation mask of an image of a fingeprint obtained trhough a thresholding algorithm. The result is not extremely good, but this techinque is very easy to implement and runs very quickly.*

There are many prominent Neural Network architectures used in the computer vision community nowadays, based on very different concepts such as convolution, recursion, dimensionality reduction, and image generation. This section will provide an overview of the state of the art of this technology and will dwell briefly on the details behind some of those innovative architectures.

Recurrent Neural Networks (RNNs) and the LSTM

The typical application for RNN is processing sequential data, as written text, speech or video clips, or any other kind of time-series signal. In this kind of data, there is a strong dependency between values at a given time/position and values previously processed. Those models try to implement the concept of *memory* weaving connections, outside the main information flow of the network, with the previous NN input. At each time-stamp, the model collects the input from the current time X_i and the hidden state from the previous step h_{i-1} and outputs a target value and a new hidden state Figure 1.7. Typically RNN cannot manage easily long-term dependencies in long sequences of signals. There is no theoretical limitation in this direction, but often it arises vanishing (or exploding) gradient problematics. A specific type of RNN has been designed to avoid this situation, the so-called Long Short Term Memory (LSTM) [9]. The LSTM architecture includes three gates (input gate, output gate, forget gate), which regulate the flow of information into and out from a memory cell, which stores values over arbitrary time intervals.

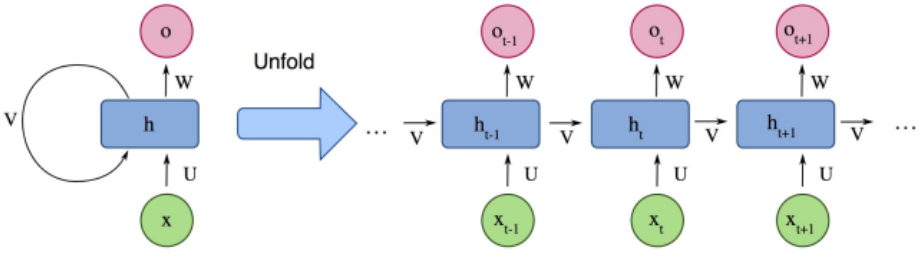

Figure 1.7: *Example of the structure of a simple RNN from [14].*

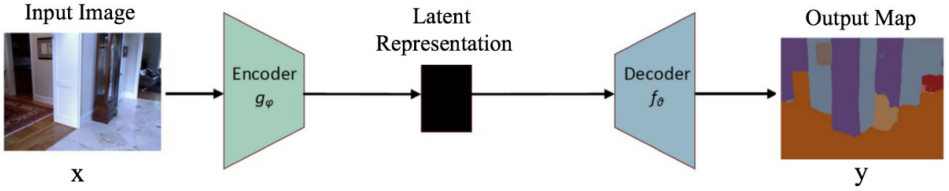

Figure 1.8: *Example of the structure of a simple ED NN from [14].*

Encoder-Decoder and Auto-Encoder Models

Encoder-Decoder models try to learn the relation between an input and the corresponding output with a two steps process. The first step is the so-called *encoding* process, in which the input x is compressed in what is called the *latent-space* representation $z = f(x)$. The second step is the *decoding* process, where the NN predicts the output starting from the latent-space representation $y = g(z)$. The idea underneath this approach is to capture in the latent-space representation the underlying semantic information of the input that is useful for predicting the output. ED models are widely used in image-to-image problems (where both input and output are images) and for sequential-data processing (like Natural Language Processing NLP). In Figure 1.8 is shown a schematic representation of this architecture. Usually, these model follow a supervised training, trying to reduce the reconstruction loss between the predicted output and the ground-truth output provided while training. Typical applications for this technology are image-enhancing techniques like de-noising or super-resolution, where the output image is an improved version of the input image.

Generative Adversarial Networks (GANs)

The peculiarity of Generative Adversarial Network lies in its structure. It is actually made of two distinct and independent modules: a generator and a discriminator, as shown in Figure 1.9. The first module G, responsible for generation, typically learns to map a prior random distribution of input z to a target dis-

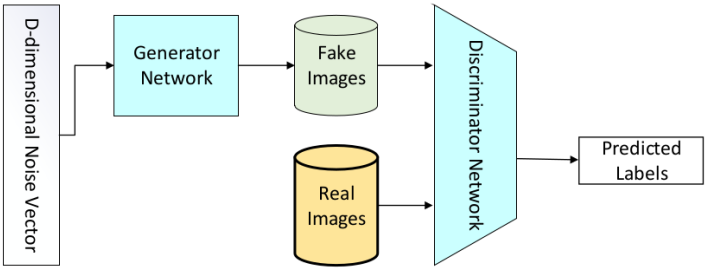

Figure 1.9: *Schematical representation of a Generative Adversarial Networks, form [14].*

tribution y, as similar as possible to the objective (e.g. the generation of a new likely human face or a hybrid image from two starting sample, as we will see in section 2.1.7) $G = z \rightarrow y$. The second module, the discriminator D, instead is trained to distinguish between *real* and *fake* images of the target category. These two networks are trained alternately in the same training process. The generator tries to fool the discriminator and vice versa. To this *competition* within different parts of the network is due the name adversarial. The formal manner to set up this adversarial training lies in the accurate choice of a suitable loss function, that will look like:

$$L_{GAN} = \mathbb{E}_{x \sim p_{data}(x)}[logD(x)] + \mathbb{E}_{z \sim p_z(z)}[log(1 - D(G(z)))]$$

The GAN is thus based on a minimax game between G and D. D aims to reduce the classification error in distinguishing fake samples from real ones, and as a consequence maximizing the L_{GAN}. On the other hand, G wants to maximize the D's error, hence minimizing L_{GAN}. The result of the training process is the trained generator G^*, capable of produce an arbitrary number of new data (images, text or whatever else)

$$G^* = \arg\min_{\mathrm{G}} \max_{\mathrm{D}} L_{GAN}$$

This peculiar architecture has yielded several interesting results and it has been developed in many different directions, with influences and contaminations with other architectures. In section 2.1.7 will be shown a particular application of this architecture, known as *style-transfer* network, which is a particular algorithm capable of implanting the visual texture of a *style* image onto the content of a different image, producing interesting hybrid images.

Convolutional Neural Networks (CNNs)

As stated before CNNs are a staple choice in image processing DL applications. They mainly consist of three types of layers:

i convolutional layers, where a kernel window of parameters is convolved with the image pixels and produce numerical features maps.

ii nonlinear layers, which apply an activation function on feature maps (usually element-wise). This step allows the network to introduce non-linear behavior and then increasing its modeling capabilities.

iii pooling layers, which replace a small neighborhood of a feature map with some statistical information (mean, max, etc.) about the neighborhood and reduce the spatial resolution.

Given the arrangement of successive layers, each unit receives weighted inputs from a small neighborhood, known as the receptive field, of units in the previous layer. The stack of layers allows the NN to perceive different resolutions: the higher-level layers learn features from increasingly wider receptive fields. The leading computational advantage given by CNN architecture lies in the sharing of kernels' weights within a convolutional layer. The result is a significantly smaller number of parameters than fully-connected neural networks. Some of the most notorious CNN architectures include: AlexNet [12], VGGNet [20], and U-Net [17].

For this work, U-net architecture is particularly interesting. The U-net model was initially developed for biomedical image segmentation, and in its structure reflects characteristics of both CNN and Encoder-Decoder models. Ronneberger et al.[17] proposed this model for segmenting biological microscopy images in 2015. The U-Net architecture is made of two branches, a contracting path to capture context, and a symmetric expanding path (see Figure 1.10). The down-sampling flow is made of a Fully Convolutional Network (FCN)-like architecture that computes features with 3×3 kernel convolutions. On the other hand, the up-sampling branch exploits up-convolution operations (or deconvolution), reducing the number of feature maps while increasing their dimensions. Another characteristic of this architecture is the presence of direct connections between layers of a similar level of compression in compressing and decompressing branches. Those links allow the NN to preserve spatial and pattern information. The Network flow eventually ends with a 1×1 convolution layer responsible of the generation of the segmentation mask of the input image.

[HOW MUCH SHOULD I DEEPEN THE TECHNICAL DESCRIPTION?]

1.3.2 Image Segmentation Datasets

Besides the choice of suitable architecture the most important aspect while developing a NN is the dataset on which perform the training process. Let confine the discussion only to image-to-image problems, like segmentation problems. There are a lot of widely used datasets, but I want to mention just a few of them to give the idea of their typical dimensions.

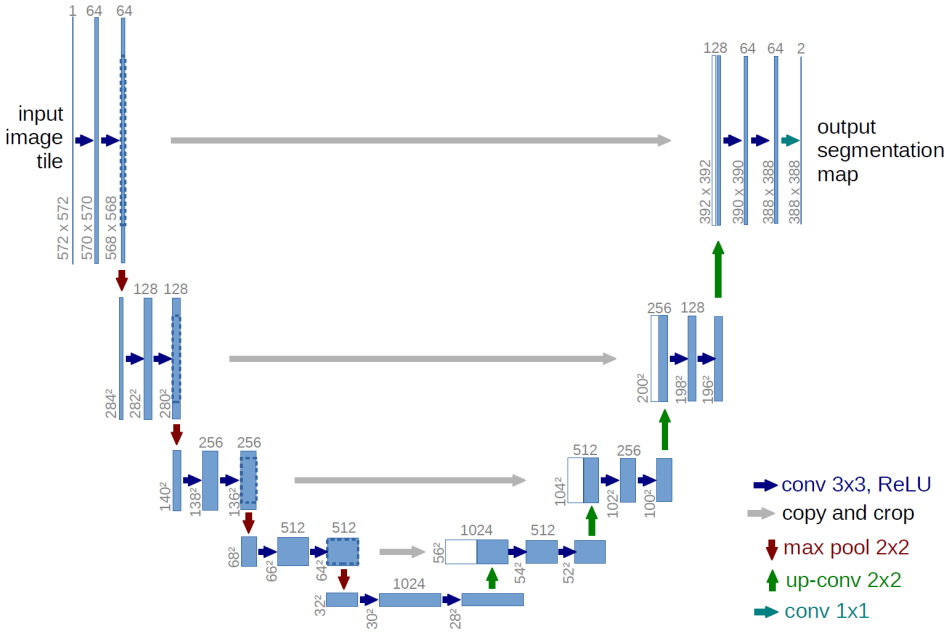

Figure 1.10: *Scheme of the typical architecture of a U-net NN. This particular model was firstly proposed by Ronneberger et al. in [17].*

A good example of segmentation is the Cityscapes dataset [4], which is a large-scale database with a focus on semantic understanding of urban street scenes. The dataset is made of video sequences from the point of view of a car in the road traffic, from 50 different cities in the world. The clips are made of 5K frames, labeled with extremely high quality at pixel-level and an additional set of 20K weakly-annotated frames. Each pixel in the segmentation mask contains the semantic classification, among over 30 classes of objects. An example of an image from this dataset is shown in Figure 1.5.

The PASCAL Visual Object Classes (VOC) [7] is another of the most popular datasets in computer vision. This dataset is designed to support the training of algorithms for 5 different tasks: segmentation, classification, detection, person layout, and action recognition. In particular, for segmentation, there are over 20 classes of labeled objects (e.g. planes, bus, car, sofa, TV, dogs, person, etc.). The dataset comes divided into two portions: training and validation, with 1,464 and 1,449 images, respectively. In Figure 1.11 is shown an example of an image and its corresponding segmentation mask.

It is worth mentioning that in the medical image processing domain typically the available dataset is definitely not that rich and vast (that is actually the seed of this work) and thus many techniques of data augmentation have been devised, to get the best out of the restricted amount of material. Generally, data augmentation manipulates the starting material applying a set of transformation to create new material, like rotation, reflection, scaling, cropping and shifting, etc. Data augmentation has been

Figure 1.11: *An example image from the PASCAL dataset and its corresponding segmentation mask [7].*

proven to improve the efficacy of the training, making the model less prone to overfitting, increasing the generalization power of the model, and helping the convergence to a stable solution during the training process.

Chapter 2

Model Generations

2.1 Technical Tools for the Development

Before delving into the details of the development of the two histological models, which are the heart of this work, it should be convenient to dwell on every tool employed during the design phase.

All the work has been done in a pure `Python` environment, using several already established libraries and writing on my own missing code, for some specific applications. All the code produced during the development, the images, and the data produced have been collected in a devoted repository on GitHub [5]. I decided to code in `Python` given the thriving variety of available libraries geared toward scientific computation, image processing, data analysis, and last but not least for its ease of use (compared to other programming languages).

In this section, it will follow a description, in no particular order, of the less common tools I used during my work.

2.1.1 Quaternions

Quaternions are, in mathematics, a number system that expands in four dimensions the complex numbers. They have been described for the first time by the famous mathematician William Rowan Hamilton in 1843. This number system define three independent *imaginary* units $\boldsymbol{i}$, $\boldsymbol{j}$, $\boldsymbol{k}$ as in (2.1), which allows the general representation of a quaternion $\boldsymbol{q}$ is (2.2) and its inverse $\boldsymbol{q}^{-1}$ (2.3) where a, b, c, d are real numbers:

$$\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = \boldsymbol{ijk} = -1, \tag{2.1}$$

$$\boldsymbol{q} = a + b\boldsymbol{i} + c\boldsymbol{j} + d\boldsymbol{k}, \tag{2.2}$$

$$\boldsymbol{q}^{-1} = (a + b\boldsymbol{i} + c\boldsymbol{j} + d\boldsymbol{k})^{-1} = \frac{1}{a^2 + b^2 + c^2 + d^2}\,(a - b\boldsymbol{i} - c\boldsymbol{j} - d\boldsymbol{k}). \tag{2.3}$$

Furthermore, the multiplication operation between quaternionn does not benefit from commutativity, hence the product between basis elements will behave as follows:

$$\begin{aligned} \boldsymbol{i}\cdot 1 = 1\cdot\boldsymbol{i} = \boldsymbol{i}, \qquad \boldsymbol{j}\cdot 1 = 1\cdot\boldsymbol{j} = \boldsymbol{j}, \qquad \boldsymbol{k}\cdot 1 = 1\cdot\boldsymbol{k} = \boldsymbol{k} \\ \boldsymbol{i}\cdot\boldsymbol{j} = \boldsymbol{k}, \qquad \boldsymbol{j}\cdot\boldsymbol{i} = -\boldsymbol{k} \\ \boldsymbol{k}\cdot\boldsymbol{i} = \boldsymbol{j}, \qquad \boldsymbol{i}\cdot\boldsymbol{k} = -\boldsymbol{j} \\ \boldsymbol{j}\cdot\boldsymbol{k} = \boldsymbol{i}, \qquad \boldsymbol{k}\cdot\boldsymbol{j} = -\boldsymbol{i}. \end{aligned} \tag{2.4}$$

This number system has plenty of peculiar properties and applications, but for the purpose of this project, quaternions are important for their ability to represent, in a very convenient way, rotations in three dimensions. In fact, the particular subset of quaternions with vanishing real part ($a = 0$) has a useful, yet redundant, correspondence with the group of rotations in tridimensional space. Every 3D rotation of an object can be represented by a 3D vector $\vec{u}$: the vector's direction indicates the axis of rotation and the vector magnitude $|\vec{u}|$ express the angular extent of rotation. However, the matrix operation which expresses the rotation around an arbitrary vector $\vec{u}$ it is quite complex and does not scale easily for multiple rotations [3], which brings to very heavy and entangled computations.

Using quaternions for expressing rotations in space, instead, it is very convinient. Given the unit rotation vector $\vec{u}$ and the rotation angle θ, the corresponding rotation quaternion $\boldsymbol{q}$ becomes (2.6):

$$\vec{u} = (u_x, u_y, u_z) = u_x\boldsymbol{i} + u_y\boldsymbol{j} + u_z\boldsymbol{k}, \tag{2.5}$$

$$\boldsymbol{q} = e^{\frac{\theta}{2}(u_x\boldsymbol{i}+u_y\boldsymbol{j}+u_z\boldsymbol{k})} = \cos\frac{\theta}{2} + (u_x\boldsymbol{i} + u_y\boldsymbol{j} + u_z\boldsymbol{k})\sin\frac{\theta}{2}, \tag{2.6}$$

$$\boldsymbol{q}^{-1} = \cos\frac{\theta}{2} - (u_x\boldsymbol{i} + u_y\boldsymbol{j} + u_z\boldsymbol{k})\sin\frac{\theta}{2}, \tag{2.7}$$

where in (2.6) we can clearly see a generalization of the Euler's formula for the exponential notation of complex numbers, which hold for quaternions. It can be shown that the application of the rotation represented by $\boldsymbol{q}$ on an arbitrary 3D vector $\vec{v}$ should be easily expressed as:

$$\vec{v}\,' = \boldsymbol{q}\vec{v}\boldsymbol{q}^{-1}, \tag{2.8}$$

using the Hamilton product defined on quaternions (2.4). This rule raises a very convinient and an extremily scalable way to compute consecutive rotations in space. Given two independent and consecutive rotations represented by the two quaternions $\boldsymbol{q}$ and $\boldsymbol{p}$ applyed on the vector $\vec{v}$ the resulting rotated vector $\vec{v}\,'$ is simply yielded as:

$$\vec{v}\,' = \boldsymbol{p}(\boldsymbol{q}\vec{v}\boldsymbol{q}^{-1})\boldsymbol{p}^{-1} = (\boldsymbol{p}\boldsymbol{q})\vec{v}(\boldsymbol{q}\boldsymbol{p})^{-1}, \tag{2.9}$$

which essentially is the application of the rotation $\boldsymbol{r} = \boldsymbol{qp}$ on the vector $\vec{v}$. This representation is completely coherent with the algebra of 3D rotations, which does not benefit from commutativity in turn.

Given this convenient property, quaternions are indeed widely used in all sorts of applications of digital 3D space design, as for simulations and for videogame design. The position of an object in the space in simulations is generally given by the application of several independent rotations, typically in the order of a tenth of rotations, which with quaternions is given easily by the product of simple objects. Every other alternative method would imply the use of matrix representation of rotations or other rotation systems as Euler's angles and would eventually make the computation prohibitive.

The use of quaternions in this work will be justified in section [??], while speaking of parametric L-systems in 3D space, used to build the backbone of the ramified structure of blood vessels in the reconstruction of a sample of pancreatic tissue.

I was able to find many useful `Python` libraries for computation with quaternions, but the one I appreciated the most for its interface and ease of use was the `pyquaternion`. With this library, it's almost immediate the definition of a quaternion by its correspondent rotation vector, and the multiplication between quaternions is very reliable.

2.1.2 Parametric L-Systems

Lindenmayer systems, or simply L-systems, were conceived as a mathematical theory of plant development [13] in 1968 by Aristid Lindenmayer. Successively, a lot of geometrical interpretations of L-systems were proposed with the intention to make them a versatile instrument for modeling the morphology typical of plants and other organic structures. As a biologist, Lindenmayer studied different species of yeast and fungi and worked the growth patterns of various types of bacteria (e.g. as the *cyanobacteria Anabaena catenula*). The main purpose for which L-systems were devised was to allow a formal description of the development of simple multicellular living organisms. Subsequently, the potentiality of these systems was expanded to describe higher-order plants and complex branching structures.

An L-system is in general defined by an *axiom* sequence and some development *rules*, which are recursively applied to the sequence and lead its development. The original proposed L-system was fairly simple and shows really well the idea underneath:

$$\begin{aligned} &axiom \ : \ A \\ &rules \ : \ (A \to AB), \quad (B \to A) \end{aligned}$$

where A and B could be any two different patterns in the morphology of an algae, or could be different bifurcations in a ramificated structure. The iterative application of the

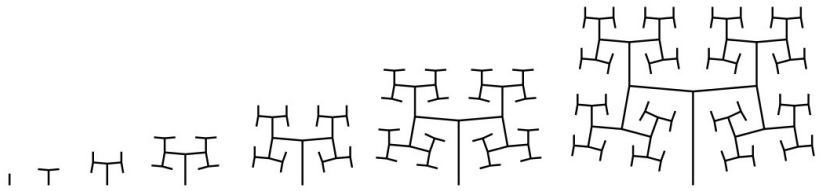

Figure 2.1: *Growth pattern for the space-filling fract-like system, used to mimic the blood vassel bifurcations in sec [??]*

rules to the axiom sequence, let's say for 7 times, will produce the following sequence:

$$
\begin{aligned}
n = 0 \ &: A \\
n = 1 \ &: AB \\
n = 2 \ &: ABA \\
n = 3 \ &: ABAAB \\
n = 4 \ &: ABAABABA \\
n = 5 \ &: ABAABABAABAAB \\
n = 6 \ &: ABAABABAABAABABAABABA \\
n = 7 \ &: ABAABABAABAABABAABABAABAABABAABAAB\,.
\end{aligned}
$$

This kind of tool, as will be shown also in [??], is particularly suited for the creation of structures with fractal behavior, and it has been used in this work to create the backbone of the entangled bifurcation in blood vessels in the modelization of pancreatic tissue. In particular, there was the need for a fractal-like space-filling ramification as the one shown in Figures 2.1.

The system in Figure 2.1 represent the successive ramification of a structure which grows adding segments gradually shorter, by a ratio parameter R and inclined of $\delta = \pm 85°$ respect the previous branch. The axiom and the rules that produce this structure are the following:

$$
\begin{aligned}
&axiom \ : \ A \\
&rule_1 \ : \ A \rightarrow F(1)[+A][-A] \\
&rule_2 \ : \ F(s) \rightarrow F(s \cdot R)
\end{aligned}
\tag{2.10}
$$

where A represent the start of a new branch and $F(s)$ represent a branch of lenght s. The presence of a rule which acts dfferently depending on the target object, is an further sophistication respect to the standard L-system. For this reason these systems are called parametric L-systems.

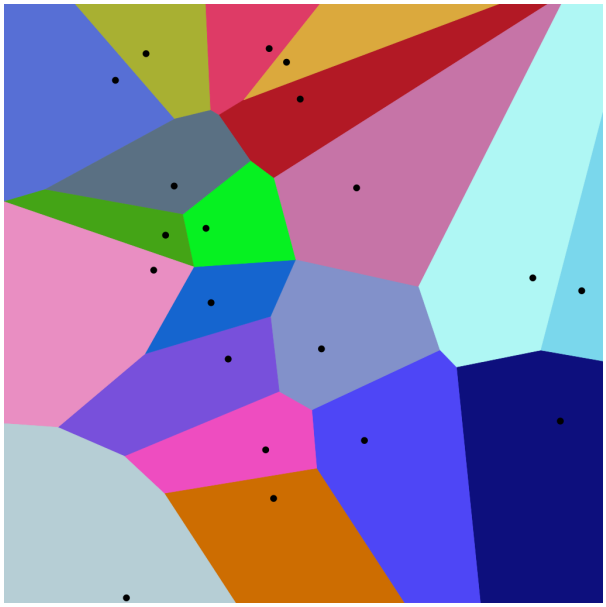

Figure 2.2: *Example of a Voronoi decomposition of a plane into 20 regions corresponding to 20 starting points*

The use of standard L-systems turned out to be widespread, and there were a lot of different `Python` libraries at my disposal for coding. By the way, parametric L-systems were not just as popular, and I was not able to find a reliable library on which to build my work. I decided then to code a parametric branching system able to recreate the structure with rules (2.10) at any desired level of iteration. Creating the tool I needed on my own I was able to add all the optional features I would have needed during the development, like an adjustable degree of angular noise in the branch generation.

2.1.3 Voronoi Tassellation

Voronoi diagrams, or Voronoi decompositions, are space-partitioning systems, which divides an n-dimensional Euclidian space into sub-regions depending on the proximity to a given set of objects. More precisely, given an n-dimensional space and m starting point $p_1, \ldots, p_m$ inside it, the whole space will be subdivided in m adjacent regions. Every point of the space is assigned to the region correspondent to the nearest starting point. In Figure 2.2 is shown a practical example of a Voronoi decomposition of a plane into 20 regions corresponding to the 20 starting points. Informal use of Voronoi diagrams can be traced back to Descartes in 1644, and many other mathematicians after him. But, Voronoi diagrams are named after Georgy Feodosievych Voronoy who defined and studied the general n-dimensional case in 1908 [24].

More precisely, let X be a metric space and d the distance defined on it. Let K be

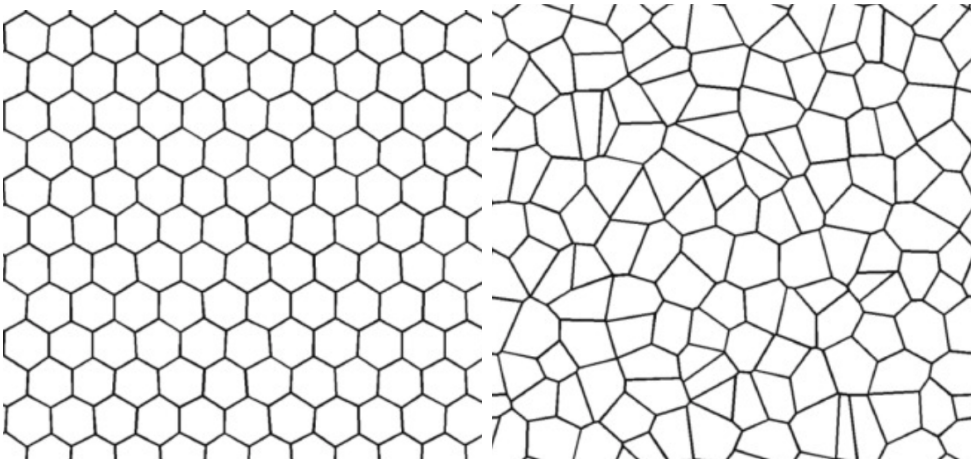

Figure 2.3: *On the right an example of 2D Voronoi decomposition resulting from a omogenously distribuited points in the plane. On the left the resulting decomposition obtained from randomly distribuited points in the plane, from [2].*

the set of indices and let $(P_k)_{k\in K}$ be the tuple of sites in the space X. The k^{th} Voronoi cell R_k, associated with the site P_k is the set of all the points in X whose distance to P_k is smaller than the distance to any other site P_j, with $j \neq k$, or in other words:

$$R_k = \{x \in X \mid d(x, P_k) \leq d(x, P_j)\ \forall j \in K,\ j \neq k\}, \tag{2.11}$$

depending on the notion of distance defined on the space X the final redistribution in subregions will look very differently.

In addittion to the choice of the distance function, another foundamental factor is the dstribution of sites in the space to be divided. If the points are chosen equally and omogenously distributed the final distribution will appear as a simple regular lattice, while a complete random distribution of points in the space will provide a decomposition in cells with very different shapes and volumes, as shown in Figure 2.3. Interesting results concerning points from a semi-random distribution will be shown in section [??], which lead to a decomposition with a good richness in shapes but with the desired homogeneity in volumes.

The Voronoi Decomposition has been of great interest in this project for the division of a 3D space in subregions, to recreate the spatial distribution of cells in a sample of human tissue, as will be shown in section [??]. The formal definition of Voronoi regions 2.11 ensure the convexity of each decomposition's tassel, which in three-dimensional space would be adjacent convex polyhedrons. Every tassel of the decomposition will be represented by a bounded 3-dimensional convex hull [1], with except for those most external cells which are unbounded and requires special attention while using.

[1]See section 2.1.5 for further details.

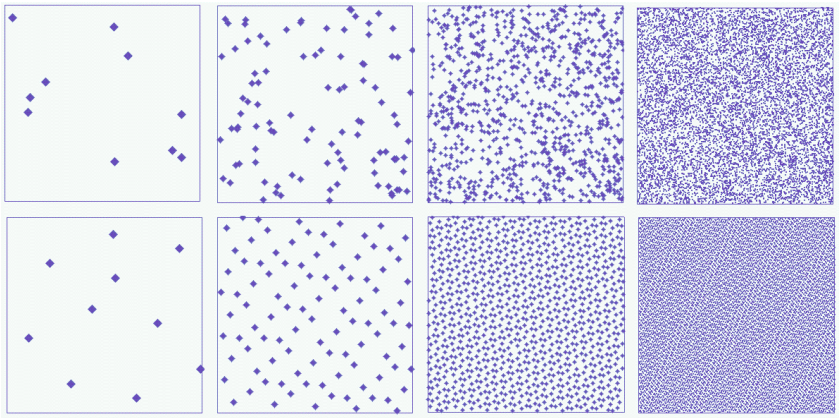

Figure 2.4: *Coverage of the unit square with an additive quasirandom numbers sequence as in 2.12 (UP) and for uniformly sampled random numbers (BOTTOM). From top to bottom: 10, 100, 1000, 10000 points.*

The most widespread tool for the computation of Voronoi decompositions in `Python` is contained in the `spatial` submodule of the famous library `SciPy`, which is a staple tool for an incredible variety of scientific algorithms. The `Voronoi` object from `Scipy` library offers a very efficient algorithm for space-partitioning, and it has been one of the pillars for the modelization of tissues.

2.1.4 Saltelli Algorithm - Randon Number Generation

As mentioned in section 2.1.3, in this project there was the need of quasi-random number generation for the production of Voronoi tassellations. Quasi-random sequences (or low-discrepancy sequences) are patterns of numbers which emulate the behaviour of uniform random distributions, but have a more homogeneous and quick coverage of the sampling domain, which provide an important advantege in applications as in quasi-Monte Carlo integration techniques, as shown in Figure 2.4. In computer science there is not any possibility of recreating *true* random sequences, hence any stocasticity is completely deterministic in its essence even if produced by very chaotic processes [2]. Indeed, every algorithm for random number generation is completely repeatable given its starting status. Quasi-random sequences, are completely deterministic too, but implements more *predictable* algorithm.

A first good example to understand the concept of quasi-random generation could be

[2] A chaotic process is a deterministic process which has an extremily sensible dependence on its starting conditions. This property mimics very effectively the behaviour of true random processes, which are intrinsically forbidden in computer science.

an additive reccurrence, as the following:

$$s_{n+1} = (s_n + \alpha) \bmod 1, \tag{2.12}$$

which for every seed element s_0 and real parameter α produced completely different sequences.

[THE FOLLOWING PARAGRAPH SOUNDS TOO HEAVY AND TECHNICAL]

In the bottom line of Figure 2.4 is clearly visible the good and homogeneous coverage of the sampling domain, altough it strongly visible a regular pattern between points, which does not convey an *organic* sensation at all. However, increasing the complexity of our very simple starting model 2.12 it is possible to overcome this *artificial* appereance of sampled points and to produce very good samples.

A notorious algorithm for quasi-random number generation is the Sobol sequence, introduced by the russian mathematician Ilya M. Sobol in 1967 [23]. In its work, Sobol wanted to construct a sequence x_n of points in the s-dimensional unitary hybercube $I^s = [0,1]^s$ such as for any integrable function f:

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} f(x_i) = \int_{I_s} f. \tag{2.13}$$

Sobol wanted to minimize the *holes* in the sampled domain (which it could be shown to be a property that helps the convergence of the sequence) and minimize as well the *holes* in every lower-dimension projection of the sampled points. The particularly good distributions that fullfill those requirments are known as (t, m, s)-nets and (t, s)-sequences in base b. To better understand them we need first to define the concept of s-interval in base b, which is a subset of I_s such as:

$$E_s^b = \prod_{j=1}^{s} \left[\frac{a_j}{b^{d_j}}, \frac{a_j + 1}{b^{d_j}} \right), \tag{2.14}$$

where a_j and d_j are non-negative integers, and $a_j < b^{d_j}$ for all j in {1, ...,s}.

Let be t and m two integers such as $0 \leq t \leq m$. A (t, m, s)-net in base b is defined as a sequence x_n of b^m points of I_s such that:

$$\textbf{Card } \mathbf{P} \cap \{x_1, \ldots, x_n\} = b^t \tag{2.15}$$

for all the elementary interval $\mathbf{P}$ in base b of hypervolume $\lambda(\mathbf{P}) = b^{t-m}$.

Given a non-negative integer t, a (t, s)-sequence in base b is an infinite sequence of points x_n such that for all integers $k \geq 0$, $m \geq t$ the sequence $\{x_{kb^m}, \ldots, x_{(k+1)b^m-1}\}$ is a (t, m, s)-net in base b.

Sobol in his article described in particular (t, m, s)-net and (t, s)-sequence in base 2. A more torough description of all the formal properties of those particular sequences could be find in [22].

In order to perform the actual sampling during the modelization, it has been used the `saltelli` module from the `SALib` library, which performs a sampling in an s-dimensional space following the Saltelli algorithm, which is a specific improved version of the Sobol algorithms oriented toward the parameter sensitivity analysis [18], [19].

2.1.5 Planar Section of a Polyhedron

As will be shown in section [??] a foundamental step for the functioning of the modelization is the planar section of a three dimensional polyhedron. It tourned out that there is no general rule to perform a planar section of a convex polyhedron with an arbitrary number of faces, respect to an arbitrary sectioning plane. Hence, I devised an algorithm to handle this task. In the general case the result of the sectioning process of a polyhedron is a polygonal surface in the case of full intersection. Otherwise it could be an empty set of points or a segment in case of particular tangency, but those two cases are not of interest for the model.

Given a convex polyhedron with n vertices and the a sectioning plane p, let V be the set of all the vertices and $f_p(\vec{x})$ the equation defining the plane. The algorithm is defined by the following steps:

1. Divide the V in two subsets: A made of those vertices which lie above and B, made of those which lie below the sectioning plane. Like in 2.16:

$$\begin{aligned} A &= \{v \in V \mid f_p(v) \geq 0\} \\ B &= \{v \in V \mid f_p(v) \leq 0\} \end{aligned} \tag{2.16}$$

   If any of the two subsets tourns out to be empty the plane p does not intersect the polyhedron, and the section is empty. A and B are represented in different colors in Figure 2.5.

2. Detect, and *draw*, any possible line that crosses two points respectively from A and B. If n_A and n_B are the number of points above and below the plane then there will be $n_A \times n_B$ possible lines. In Figure 2.5 all the lines between the two classes of point are drawn in white.

3. Detect all the points P from the intersection between the $n_A \times n_B$ lines from the previous step and the sectioning plane p. All these points will lie on the same plane, within the boundaries of the polygonal section.

4. The final polygon, is then yielded by computing the convex hull of the points in P. The convexity of the starting polyhedron in fact ensure the convexity of any section of the solid.

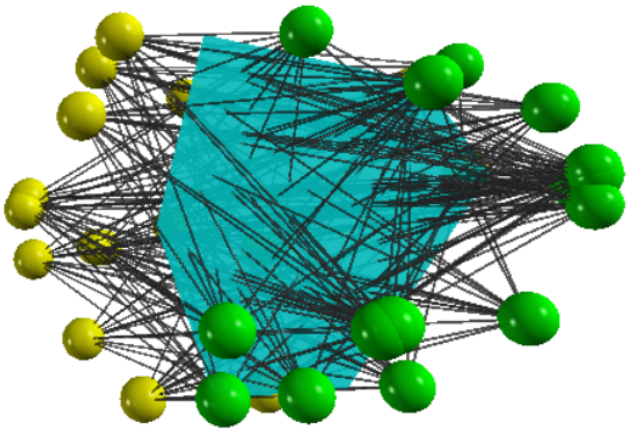

Figure 2.5: *In this picture is shown an example of application of the algoritm for the planar section on a polyhedrion. The vertices are divided into two groups, with different colors yellow and green. All the possible lines between any couple of vertices picked from the two classes is drawn in black. In Turquoise the resulting planar section, obtained as the convex hull containing all the intersections between the lines and the plane.*

The result of the algorithm is then a convex hull, which in geometry is defined as the smallest convex envelope or convex closure of a set of points. In 2 dimensions is the smallest convex polygon containing a certain set of points in a plane (Figure 2.6), and in 3 dimensions it is the smallest convex polyhedron containing a set of points in the space.

In `Python`, the most convinient way to work with convex hulls was to use the submodule `spatial.ConvexHull` from the `SciPy` library. This module allows also a convinient way for plotting images with `MatPlotLib`, which is the point of refernce for plotting and images formation in `Python`.

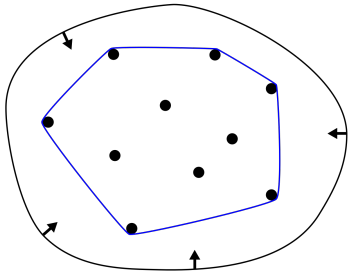

Figure 2.6: *Representation of the convex hull of a bounded planar set.*

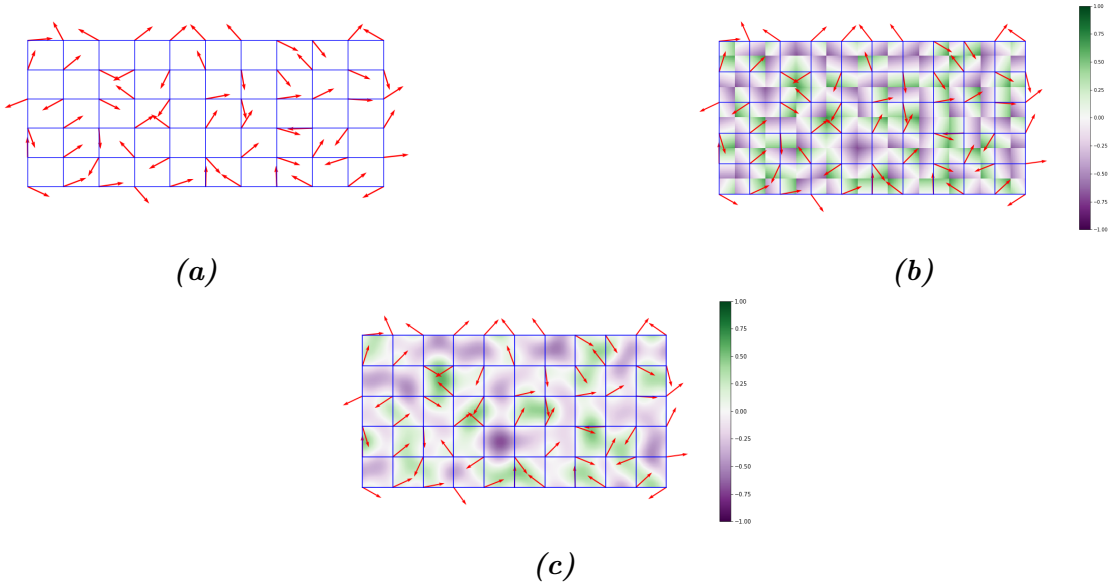

Figure 2.7: *The three main steps of the algorithm to produce Perlin noise. In 2.7a the plane discretization and the assignment of a gradient vector to every node of the grid. In 2.7b the computation of the dot product with all the points inside the discretization and in 2.7c the interpolation of the values to create the final function.*

2.1.6 Perlin Noise

Perlin noise is a widely used form of noise in computer graphics, which mimics very well natural and smooth fluctuations around a constant value. It has been developed by Ken Perlin in 1983, and it is now the staple tool for giving texture to object in virtual modelization, often considered the *salt* of computer graphics texturization. The Perlin noise is a gradient-based algorithm defined on grid discretization of a n-dimensional space. The algorithm involves three subsequent steps:

1. The first step is to discretize the n-dimensional space in a regular lattice: the dimension of the grid will impact heavily on the scale of the noise. As in Figure 2.7a at every node of the grid is assigned a randomly oriented n-dimensional unitary gradient vector. This is the preliminary setup which will allow the computation of the actual noise function in every point of the space.

2. Given the candidate point $\vec{x}$ in the grid onto which evaluate the noise there are 2^n nearest grid nodes. For each one of these 2^n nodes it is evaluated the distance vector from $\vec{x}$ as the offset between the two points. Then it is computed the dot product between every pair made of a gradient vector and the offset vector. This operation should be tought as made on every point in the lattice, as in Figure 2.7b,

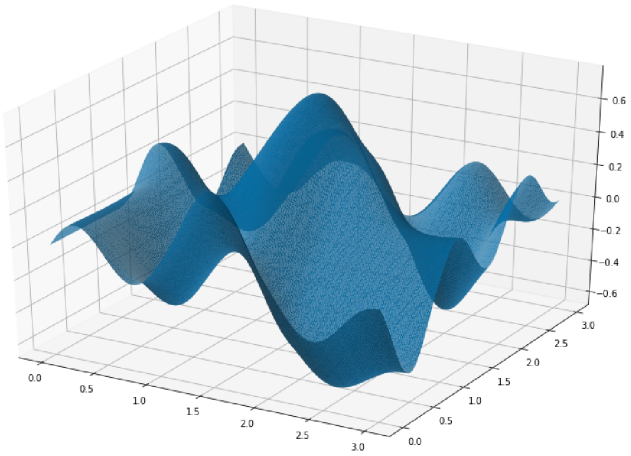

Figure 2.8: *Example of Perlin noise 2D funciton produced while working on this project. This surface offers a smooth variation around the value 0 with amplituded* $\in [0; 1]$.

where at every point of the grid is represented just one of the $2^2 = 4$ series of dot products.

3. The final step is the interpolation between the 2^n series of dot. To perform the interpolation usually is used a function with vanishing first degree (and preferably also second degree) derivative in correspondence of the 2^n grid nodes [3]. This means that the noise function will pass through zero at every node and have a gradient equal to the precomputed grid node gradient. These properties give Perlin noise its characteristic spatial scale and smoothness.

In general the final result of the algorithm is a smooth function with a random-like behaviour that mimics really well an organic appearance, like in Figure 2.8, with fluctuation around the value 0, with amplituted $\in [0; 1]$. The surface in Figure 2.8 has been produced plotting in 3D the results of the function `pnoise2` from the library `noise`, which offers a good and intuitive tool for the production of different type of noise.

2.1.7 Style-Transfer Neural Network

Style-Transfer Neural Networks are common models, able of creating new hybrid images implanting the visual style from an image preserving the visual content of another image. The two images necessary for the algorithm are called *style* image S and *content* image C, and the resulting *styled* picture X, as in Figure 2.9.

[3] Usually are used functions with a sigmoidal behaviour, like any smoothstep function, which are a family of very common items in computer graphics.

Figure 2.9: *Different examples of application of a style-transfer NN on the same content image, with different style images, from [8]*

There are many different tested and comparable architectures to compute this kind of algorithm. In my work I decided to use in particular the procedure described in [8], using the *PyTorch* ecosystem to implement the necessary code.

The backbone of the architecture is the VGG-19 network, which s a convolutional neural network 19 layers deep. This huge model has been pre-trained on over a million-images from the ImageNet database [6], for the classification into over than 1000 classes of objects. As a result, the network has learned rich feature representations for a wide range of images. The best (and conceptually the only) way to load a pre-trained model is to load the ordered set of weights which define the network and to initilize an empty module with those values. This is the perfect start for creating a style transfer network, which require an further and briefer training phase, to completely customize the network.

The key ingredient for finalize the model is to insert some little but foundamental modifications and to extend the training on the couple of input images. This final training should be aware of the *concepts* of visual style and visual content of the image, and the operation should try to preserve them both. This is usually done minimizing two new loss function, computed between the staring image and the produced image:

Content Loss

The content loss is a function that represents a weighted version of the content distance for an individual layer. The most comonly used function to evaluate the preservation of content between two images is the simple Mean Squared Error as in eqution (1.5). It can be computed between any couple of same-sized object, hence

also between the results of the same feature maps on the images X and C at the same layer L.

$$L_{Cont} = \|F_{XL}F_{CL}\|^2 \tag{2.17}$$

In order to evaluate this content loss, it is necessary to insert a custom transparent[4] layer directly after the convolution layer(s) that are being used to compute the content distance.

Style Loss

The concept of *style* loss function is the true novelty introduced by [8]. This loss function is implemented similarly to the content loss module, as it will act as a transparent layer in the network. The computation of the style loss requires in advanced the evaluation of the Gram matrix G_{XL} at a certain layer L. A Gram matrix is the result of multiplying a given matrix by its transposed matrix. In this case the matrix to multiplicate is a reshaped version of the feature maps F_{XL}: $\hat{F}_{XL}$, a $K \times N$ matrix, where K is the number of feature maps at layer L and N is the length of any vectorized feature map F^k_{XL}. Furthermore, the Gram matrix must be normalized by dividing each element by the total number of elements in the matrix. The style distance is now computed using the mean square error between G_{XL} and G_{SL}:

$$L_{Style} = \|G_{XL}G_{SL}\|^2 \tag{2.18}$$

[4]A transparent layer is a layer that performs some operations, like evaluating a function on its input, but returns as output an uncahnged copy of its input.

Chapter 3

Conclusions

3.1 conclusions

Bibliography

[1] Abien Fred Agarap. Deep learning using rectified linear units (relu), 2018.

[2] J. Alsayednoor and P. Harrison. Evaluating the performance of microstructure generation algorithms for 2-d foam-like representative volume elements. *Mechanics of Materials*, 98:44 – 58, 2016.

[3] R. W. Brockett. Robotic manipulators and the product of exponentials formula. In P. A. Fuhrmann, editor, *Mathematical Theory of Networks and Systems*, pages 120–129, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg.

[4] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. *CoRR*, abs/1604.01685, 2016.

[5] Alessandro d'Agostino. Dataset generation for the training of neural networks oriented toward histological image segmentation, april 2020.

[6] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[7] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[8] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style, 2015.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[11] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[13] Aristid Lindenmayer. Mathematical models for cellular interactions in development ii. simple and branching filaments with two-sided inputs. *Journal of theoretical biology*, 18(3):300–315, 1968.

[14] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey, 2020.

[15] Muhammad Niazi, Thomas Tavolara, Vidya Arole, Douglas Hartman, Liron Pantanowitz, and Metin Gurcan. Identifying tumor in pancreatic neuroendocrine neoplasms from ki67 images using transfer learning. *PLOS ONE*, 13:e0195621, 04 2018.

[16] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta Numerica*, 8:143195, 1999.

[17] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[18] Andrea Saltelli. Making best use of model evaluations to compute sensitivity indices. *Computer Physics Communications*, 145(2):280 – 297, 2002.

[19] Andrea Saltelli, Paola Annoni, Ivano Azzini, Francesca Campolongo, Marco Ratto, and Stefano Tarantola. Variance based sensitivity analysis of model output. design and estimator for the total sensitivity index. *Computer Physics Communications*, 181(2):259 – 270, 2010.

[20] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[21] Sandro Skansi. *Introduction to Deep Learning: From Logical Calculus to Artificial Intelligence*. Springer Publishing Company, Incorporated, 1st edition, 2018.

[22] I.M. Sobol. Uniformly distributed sequences with an additional uniform property. *USSR Computational Mathematics and Mathematical Physics*, 16(5):236 – 242, 1976.

[23] I.M Sobol. Global sensitivity indices for nonlinear mathematical models and their monte carlo estimates. *Mathematics and Computers in Simulation*, 55(1):271 – 280, 2001. The Second IMACS Seminar on Monte Carlo Methods.

[24] Georges Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. premier mémoire. sur quelques propriétés des formes quadratiques positives parfaites. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 1908:102 – 97.