

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

1 Histo & Deep Learning & SOA	7
1.1 Histological Images	7
1.1.1 Traditional Preparation of Histological Samples	7
1.1.2 Important Aspects	7
1.2 Introduction to Deep Learning	8
1.2.1 Perceptrons and Multilayer Feedforward Architecture	8
1.2.2 Training of a NN - Error Back-Propagation	10
1.3 Deep Learning-Based Segmentation Algorithms	15
1.3.1 State of the Art on Deep Learning Segmentation	16
1.3.2 Image Segmentation Datasets	20
2 Model Generations	22
2.1 Technical Tools for the Development	22
2.1.1 Quaternions	22
2.1.2 Parametric L-Systems	24
2.1.3 Voronoi Tassellation	26
2.1.4 Saltelli Algorithm - Randon Number Generation	28
2.1.5 Planar Section of a Polyhedron	30
2.1.6 Perlin Noise	31
2.1.7 Style-Transfer Neural Network	34
2.2 Model Development	37
2.2.1 Pancreatic Tissue Model	37
2.2.2 Dermal Tissue Model	44
2.2.3 Sectioning Process - Synthetic Images	45
3 Conclusions	46
3.1 conclusions	46
Bibliography	47

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 for the support of 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 anographical 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 the segmentation task, and it's quite widespread with countless different applications, allowing a sort of automatic image interpretation. In ML the segmentation problem is usually faced as a supervised task, hence the algorithm in order to be trained properly requires an appropriate 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 ap-

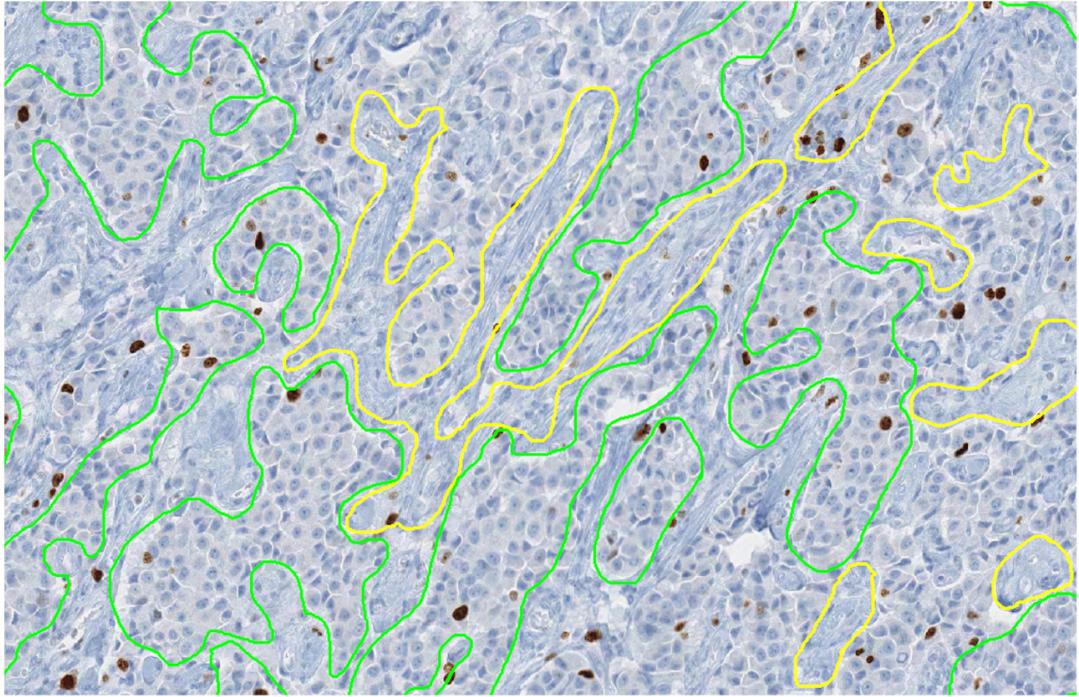


Figure 1: Interleaving of tumor (green annotation) and non-tumor (yellow annotation) regions [20].

plication, as would be the one on histological images, would require a lot of starting material, previously analyzed from the same qualified expert encharged of the visual inspection 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 and produce the corresponding 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 effective is the resulting algorithm.

Different approaches have already been explored to overcome this problem, and they are mainly based on the generation of synthetic labeled data to use 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 entire traditional sectioning process, which is made on real histological samples, is recreated virtually on this virtual model. This yields pairs of 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, overcoming 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 less sophisticated respect the target biological structures. I'll show two models: one of pancreatic tissue and another of dermal tissue, besides all the tools I used and the choices I made during the designing 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. Thesis

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

Like other artificial learning techniques, DL models aim to *learn* a relationship between some sort of input and a specific kind of output. In other words, approximating numerically 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 the detection of objects in a picture, or text manipulation/generation. The function is approximated employing a greatly complex network of simple linear and non-linear mathematical operations arranged in a so-called Neural Network (typically with millions of parameters). 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 [27].

The fundamental 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 of n numerical signals $\vec{x} = 1, x_1, \dots, x_n$ and computes a linear weighted combination with the weights vectors $\vec{w} = w_0, w_1, \dots, w_n$, where w_0 is a bias factor:

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

The results of this linear combination are given as input to a non-linear function $\chi(x)$ called the activation function. Typical choices as activation function are any sigmoidal function like $\text{sign}(x)$ and $\tanh(x)$, but in more advanced 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 and 0 if it is low enough, with a smooth modulation in-between the two values.

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 with a straight information flux. More complex architectures could implement also recursive connection, linking a layer to itself, but it should be regarded as sophistication to the standard case. There are endless possibilities

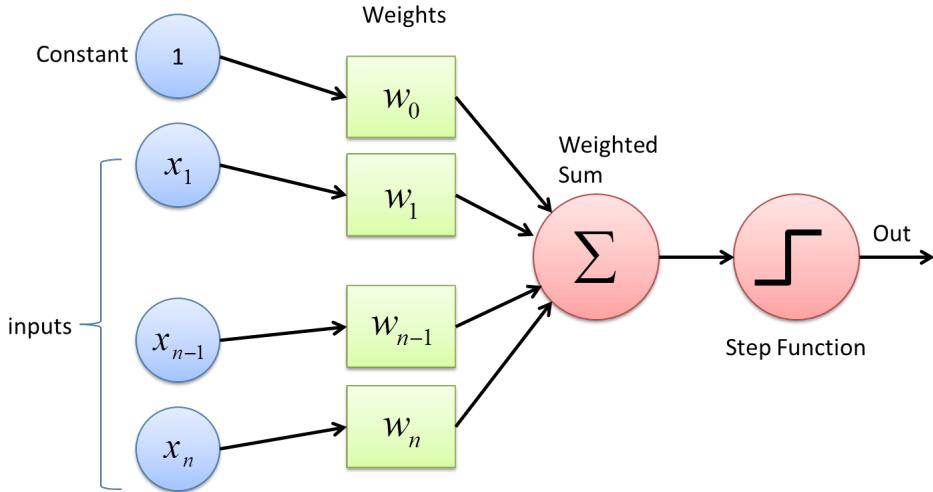


Figure 1.1: Schematic picture of a single layer perceptron. The input vector is linearly combined with the bias factor and sent to an activation function to produce the numerical “binary” output.

of combination and arrangement of neurons inside a NN’s layer, but the most simple ones are known as fully-connected layers, where every neuron is linked with each other 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 in the output generation is reached.

Although a fully connected network represents the simplest linking choice, the insertion of each weight increases the number of overall parameters, and so the complexity of the model. Thus we want to create links between neurons smartly, rejecting 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 of correlation between adjacent pixels of an image and their color channels, allowing a sort of spatial awareness. Furthermore, the majority of traditional computer vision techniques are based on the discrete convolution of images, and on the features extracted from them.

As a matter of principle a NN with just two successive layers, which is called a *shallow* network, and with an arbitrary number of neurons per layer, can approximate arbitrary well any kind of smooth enough function [21]. However, direct experience suggests that networks with multiple layers, called *deep* networks, can reach equivalent

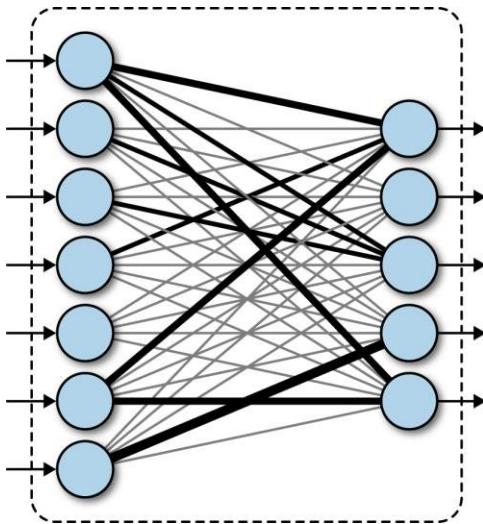


Figure 1.2: Schematic representation of a fully connected (or dense) layer. Every neuron from the first layer is connected with every output neuron. The link thickness represent the absolute value of the combination weight for that particular value.

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 tens of hidden layers. Such deep structures allow the computation of what is called deep features, so features of the features of the input data, that allows the network to easily manage concepts that would be barely 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 the new problem could be solved. In opposition to the concept of supervised problems, there are the *unsupervised* problems, where the algorithm does not try to learn a rule from a 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

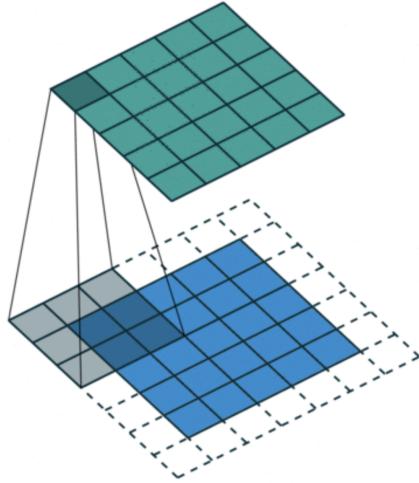


Figure 1.3: Schematic representation of a convolutional layer. The input data are processed by a window kernel that slides all over the image. This operation can recreate almost all the traditional computer vision techniques, and can overcome them, creating new operations, which would be unthinkable to hand-engineered.

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 creative 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 [15] dataset. This famous dataset is made of over $60K$ labeled colored 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 $32 \times 32 \times 3$ 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:

$$\vec{p} = (p_1, p_2, \dots, p_{10}), \quad (1.2)$$

$$\sum_{i=1}^{10} p_i = 1, \quad (1.3)$$

and it should be compared with the true label, 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:

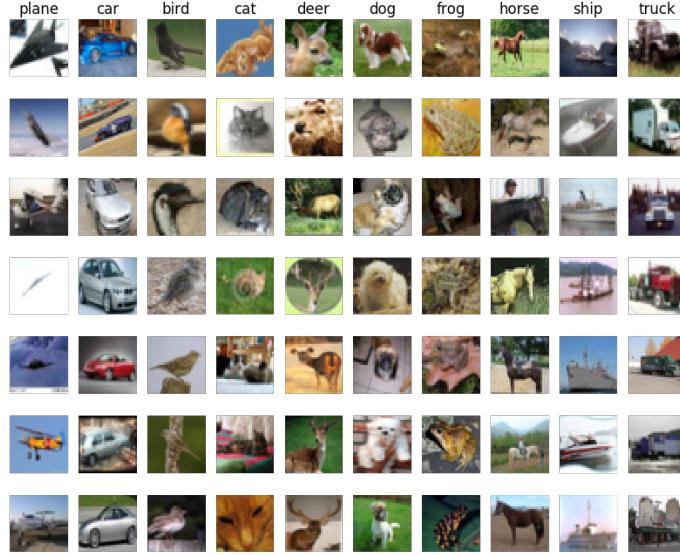


Figure 1.4: Sample grid of images from the CIFAR10 dataset. Each one of the 32×32 image is labeled with one of the ten classes of objects: plane, car, bird, cat, deer, dog, frog, horse, ship, truck.

$$\vec{t} = (0, 0, \dots, 1, \dots, 0, 0). \quad (1.4)$$

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 (MSE) function that is simply the averaged L^2 norm of the difference vector between \vec{p} and \vec{t} :

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

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

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

From both equations 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 back-propagation method is an iterative technique that 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 at each step. 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 in a single step of the method:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}, \quad (1.7)$$

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 the whole dataset is reprocessed 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 the gradient descent process would probably 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 [14].

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 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 the 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' performances, 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.



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.

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 perform 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.

A lot of other traditional techniques improve this first segmentation method [6]. 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



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

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 [19].

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).

There are many prominent Neural Network architectures used in the computer vision community nowadays, based on very different ideas 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,

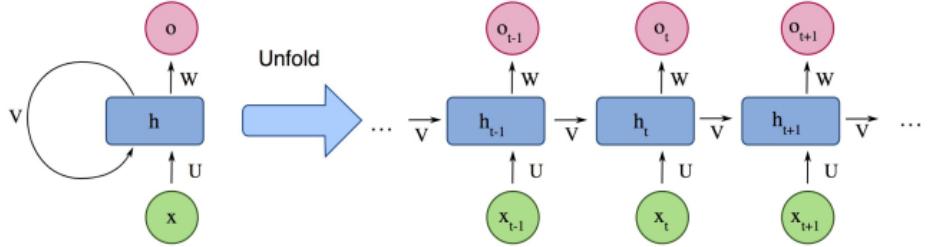


Figure 1.7: Example of the structure of a simple Recurrent Neural Network from [19].

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’s 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 during the training phase. A specific type of RNN has been designed to avoid this situation, the so-called Long Short Term Memory (LSTM) [12]. 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.

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

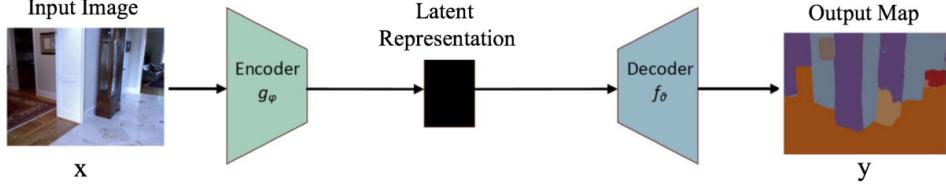


Figure 1.8: Example of the structure of a simple Encoder-Decoder Neural Network from [19].

techniques like de-noising or super-resolution, where the output image is an improved version of the input image. Or image generation problems (e.g. plausible new human faces generation) in which all the properties which define the type of image under analysis should be learned in the representation latent space.

Generative Adversarial Networks (GANs)

The peculiarity of Generative Adversarial Network (GAN) 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 the generation, typically learns to map a prior random distribution of input z to a target distribution y , as similar as possible to the target $G = z \rightarrow y$ (i.e. almost any kind of image-to-image problem could be addressed with GANs, as in [13]). 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. The name adversarial is actually due to this *competition* within different parts of the network. The formal manner to set up this adversarial training lies in the accurate choice of a suitable loss function, that will look like:

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

. The GAN is thus based on a min-max 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):

$$G^* = \arg \min_G \max_D L_{GAN}$$

. This peculiar architecture has yielded several interesting results and it has been developed in many different directions, with influences and contaminations with other architectures [13].

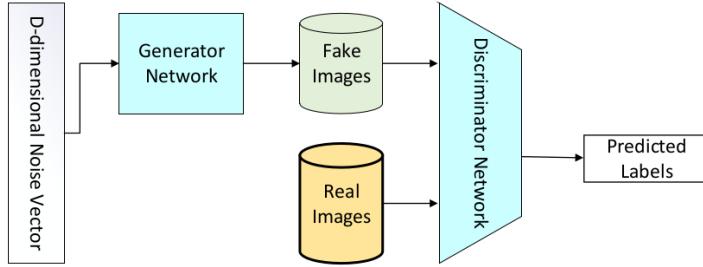


Figure 1.9: Schematic representation of a Generative Adversarial Networks, form [19].

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. 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. Some of the most notorious CNN architectures include: AlexNet [16], VGGNet [26], and U-Net [22].

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.[22] proposed

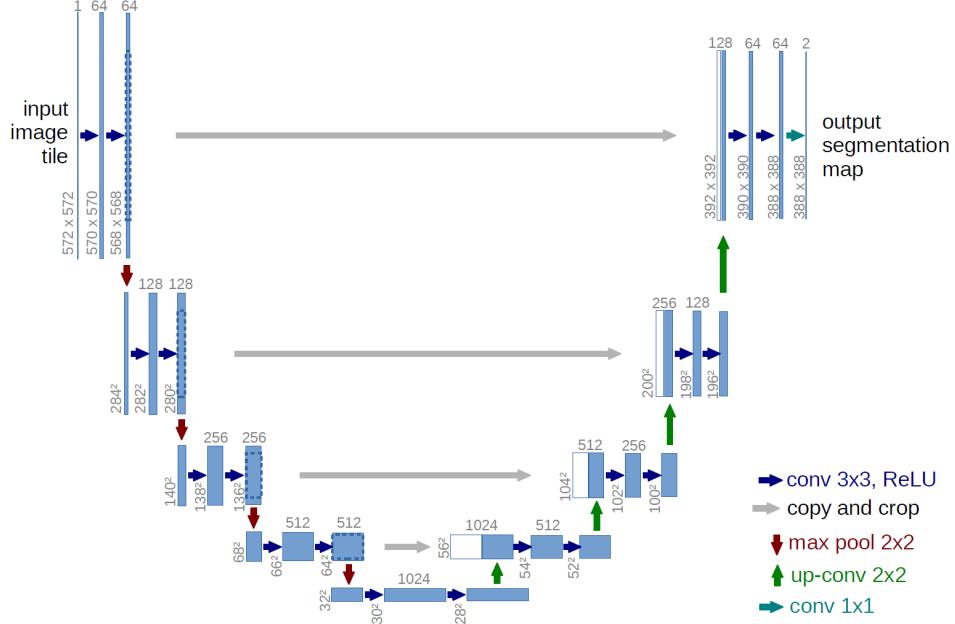


Figure 1.10: Scheme of the typical architecture of a U-net NN. This particular model was firstly proposed by Ronneberger et al. in [22].

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 de-convolution), 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 for the generation of the segmentation mask of the input image.

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 characteristics.

A good example of segmentation is the Cityscapes dataset [7], which is a large-scale

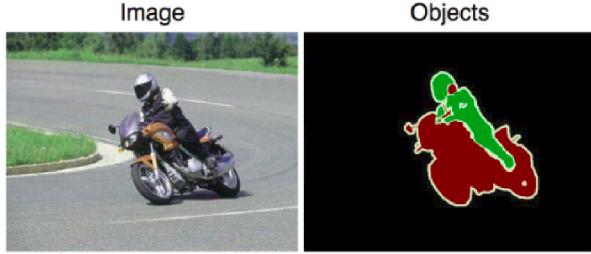


Figure 1.11: An example image from the PASCAL dataset and its corresponding segmentation mask [10].

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) [10] 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 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 proposed 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 code necessary for the work has been written in a pure `Python` environment, using several already established libraries and writing by myself the 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 [8]. 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 to 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 defines three independent *imaginary* units \mathbf{i} , \mathbf{j} , \mathbf{k} as in (2.1), which allows the general representation of a quaternion \mathbf{q} in (2.2) and its inverse \mathbf{q}^{-1} (2.3) where a, b, c, d are real numbers:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1, \quad (2.1)$$

$$\mathbf{q} = a + bi + cj + dk, \quad (2.2)$$

$$\mathbf{q}^{-1} = (a + bi + cj + dk)^{-1} = \frac{1}{a^2 + b^2 + c^2 + d^2} (a - bi - cj - dk). \quad (2.3)$$

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

$$\begin{aligned} \mathbf{i} \cdot 1 = 1 \cdot \mathbf{i} &= \mathbf{i}, & \mathbf{j} \cdot 1 = 1 \cdot \mathbf{j} &= \mathbf{j}, & \mathbf{k} \cdot 1 = 1 \cdot \mathbf{k} &= \mathbf{k} \\ \mathbf{i} \cdot \mathbf{j} &= \mathbf{k}, & \mathbf{j} \cdot \mathbf{i} &= -\mathbf{k} \\ \mathbf{k} \cdot \mathbf{i} &= \mathbf{j}, & \mathbf{i} \cdot \mathbf{k} &= -\mathbf{j} \\ \mathbf{j} \cdot \mathbf{k} &= \mathbf{i}, & \mathbf{k} \cdot \mathbf{j} &= -\mathbf{i}. \end{aligned} \quad (2.4)$$

This number system has plenty of peculiar properties and applications, but for this project, quaternions are important for their ability to represent, in a very convenient way, rotations in three dimensions. 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 $\text{SO}(3)$. 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 [4], 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 \mathbf{q} becomes (2.6):

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

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

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

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 \mathbf{q} on an arbitrary 3D vector \vec{v} should be easily expressed as:

$$\vec{v}' = \mathbf{q} \vec{v} \mathbf{q}^{-1}, \quad (2.8)$$

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 \mathbf{q} and \mathbf{p} applied on the vector \vec{v} the resulting rotated vector \vec{v}' is simply yielded as:

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

which essentially is the application of the rotation $\mathbf{r} = \mathbf{q} \mathbf{p}$ 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 property, quaternions are indeed widely used in all sorts of applications of digital 3D space design, as for simulations and videogame. 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 2.2.2, 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 [17] in 1968 by Aristid Lindenmayer. Successively, a lot of geometrical interpretations of L-systems were proposed 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:

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

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

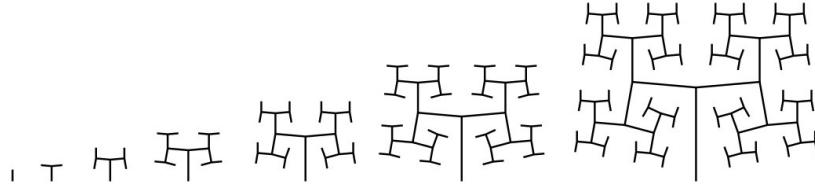


Figure 2.1: Growth pattern for the space-filling fractal-like system, used to mimic the blood vessel bifurcations in sec 2.2.2.

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

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

This kind of tool, as will be shown also in 2.2.2, 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 lenght ratio parameter R and inclined of $\delta = \pm 85^\circ$ respect the previous branch. The axiom and the rules that produce this structure are the following:

$$\begin{aligned}
 \text{axiom} & : A & (2.10) \\
 \text{rule}_1 & : A \rightarrow F(1)[+A][-A] \\
 \text{rule}_2 & : F(s) \rightarrow F(s \cdot R)
 \end{aligned}$$

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 differently 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.

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, \dots, 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 [31].

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

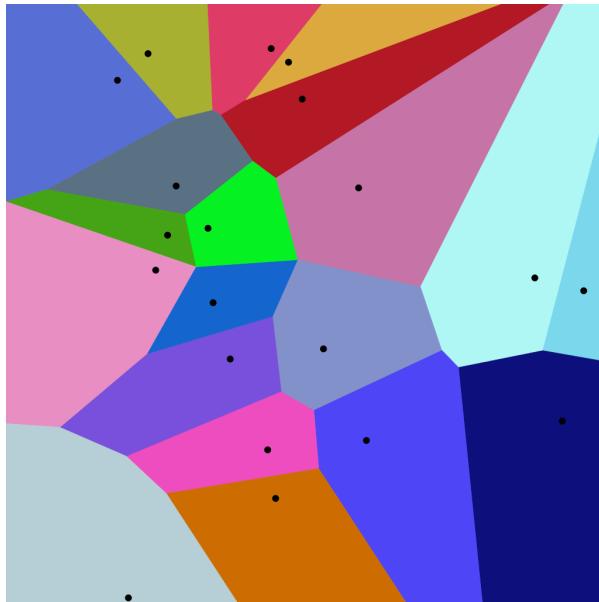


Figure 2.2: Example of a Voronoi decomposition of a plane into 20 regions corresponding to 20 starting points.

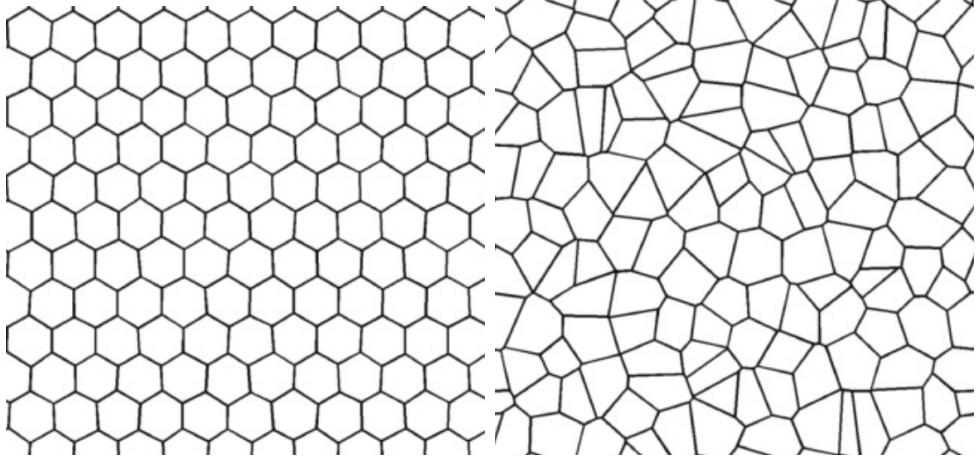


Figure 2.3: On the right an example of 2D Voronoi decomposition resulting from homogeneously distributed points in the plane. On the left the resulting decomposition obtained from randomly distributed points in the plane, from [2].

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:

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

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

In addition to the choice of the distance function, another fundamental factor is the distribution of sites in the space to be divided. If the points are chosen equally and homogeneously distributed the final distribution will appear as a simple regular lattice, while a completely 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 2.2.2, which leads to 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 2.2.2. The formal definition of Voronoi regions (2.11) ensures 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 hull¹, with except for those most external cells which are unbounded and requires special attention while using.

¹See section 2.1.5 for further details.

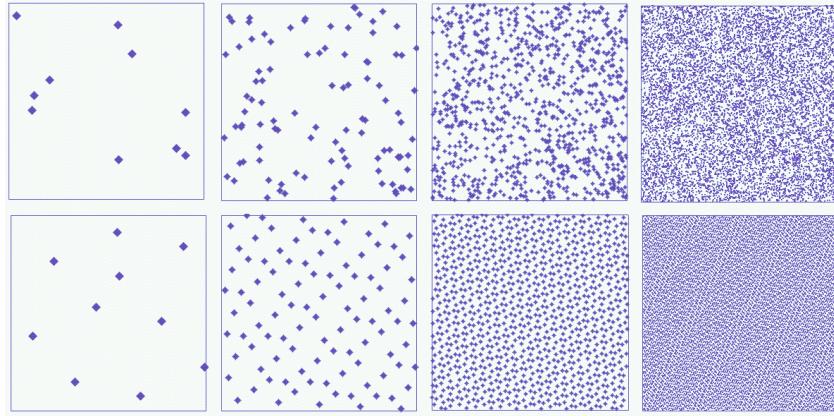


Figure 2.4: Coverage of the unit square with an additive quasirandom numbers sequence as in 2.12 (up) and for uniformly sampled random numbers (bottom). From top to bottom: 10, 100, 1000, 10000 points.

The most widespread tool for the computation of Voronoi decompositions in Python is contained in the `spatial` submodule of the famous library `SciPy` [30], 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. Unluckily this module does not easily allow to perform Voronoi decomposition with different definitions of distance functions d other than the Euclidian distance, which would have allowed interesting studies.

2.1.4 Saltelli Algorithm - Random Number Generation

As mentioned in section 2.1.3, in this project there was the need for quasi-random number generation for the production of Voronoi tessellations. Quasi-random sequences (or low-discrepancy sequences) are patterns of numbers that emulate the behavior of uniform random distributions but have a more homogeneous and quick coverage of the sampling domain, which provides an important advantage 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 stochasticity is completely deterministic in its essence even if produced by very chaotic processes². 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* and well-behaved algorithms.

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

A first good example to understand the concept of quasi-random generation could be an additive recurrence, as the following:

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

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

In the bottom line of Figure 2.4 is clearly visible the good and homogeneous coverage of the sampling domain, although it is 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* appearance of sampled points and to produce very good samples.

[THE FOLLOWING PARAGRAPH SOUNDS TOO HEAVY AND TECHNICAL]

A notorious algorithm for quasi-random number generation is the Sobol sequence, introduced by the russian mathematician Ilya M. Sobol in 1967 [29]. In its work, Sobol wanted to construct a sequence x_n of points in the s -dimensional unitary hypercube $I^s = [0, 1]^s$ such as for any integrable function f :

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

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 fulfill those requirements 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:

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

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

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:

$$\text{Card } \mathbf{P} \cap \{x_1, \dots, x_n\} = b^t \quad (2.15)$$

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 $\{x_{kb^m}, \dots, x_{(k+1)b^m-1}\}$ 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 thorough description of all the formal properties of those particular sequences could be found in [28].

In order to perform the actual sampling during the modelization, it has been used the `saltelli` module from the `SALib` library, which performs 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 [23], [24].

2.1.5 Planar Section of a Polyhedron

As will be shown in section 2.2.2 a fundamental step for the functioning of the modelization is the planar section of a three-dimensional polyhedron. It turned 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 case of a full intersection, the result of the sectioning process of a polyhedron is a polygonal surface, 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 to the model.

Given a convex polyhedron with n vertices and 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 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:

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

If any of the two subsets turns 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 numbers 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 points are drawn in dark gray.
3. Detect P , the set of all the points 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.

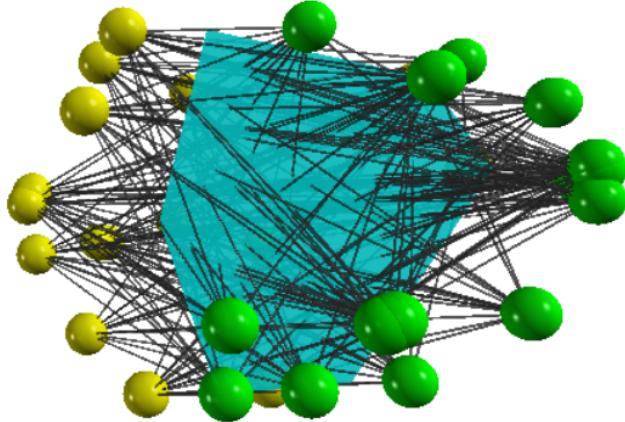


Figure 2.5: In this picture is shown an example of the application of the algorithm for the planar section on a polyhedron. The vertices are divided into two groups, with different colors yellow and green. All the possible lines between any couple of vertices picked from the two classes are drawn in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

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 ensures the convexity of any section of the solid.

The result of the algorithm is, as just stated, 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 (In Figure 2.6 is shown the so-called "rubber band effect"), and in 3 dimensions it is the smallest convex polyhedron containing a set of points in the space.

In **Python**, the most convenient way to work with convex hulls was to use the sub-module `spatial.ConvexHull` from the **SciPy** library [30]. This module allows also a convenient way for plotting images with **MatPlotLib**, which is the point of reference for plotting and image formation in **Python**.

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

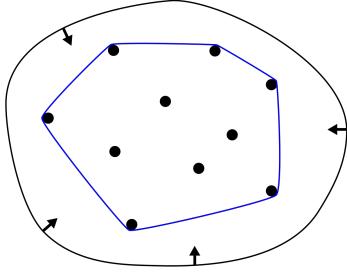


Figure 2.6: Representation of the convex hull of a bounded planar set of points. This particular enclosure goes under the name of "rubber band effect".

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} between the grid nodes 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 nearby gradient vectors and the offset vector. This operation should be thought of as made on every point in the lattice, as in Figure 2.7b, 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 products. 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 pre-computed 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 behavior that mimics really well an organic appearance, like in Figure 2.8, with fluctuation around the value 0, with amplitude $\in [0; 1]$. 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.

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

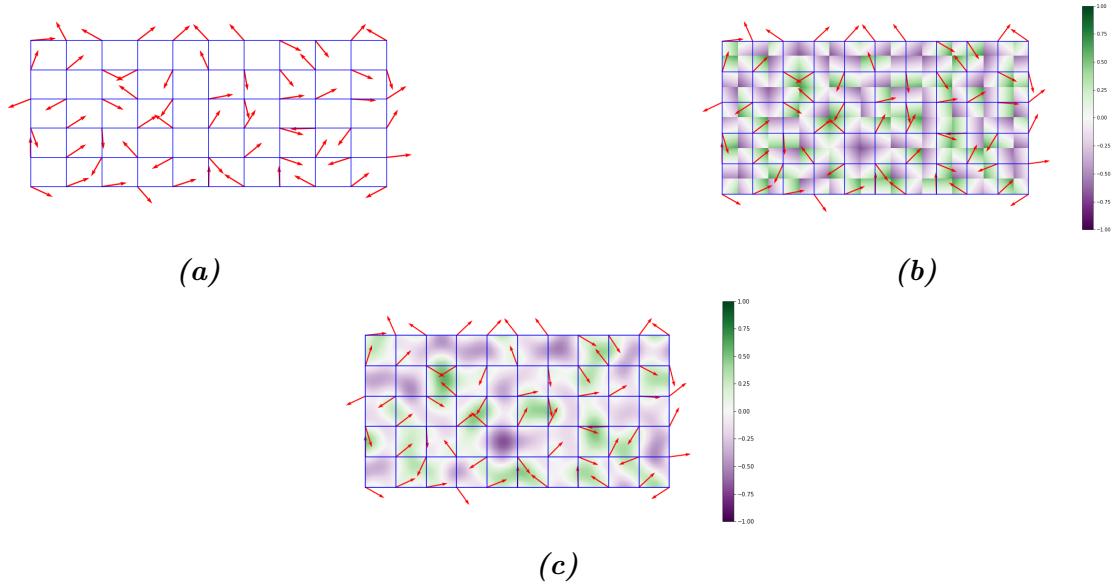


Figure 2.7: The three main steps of the algorithm to produce Perlin noise. In 2.7a the plane discretization and the assignment of a gradient vector to every node of the grid. In 2.7b the computation of the dot product with all the points inside the discretization and in 2.7c the interpolation of the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

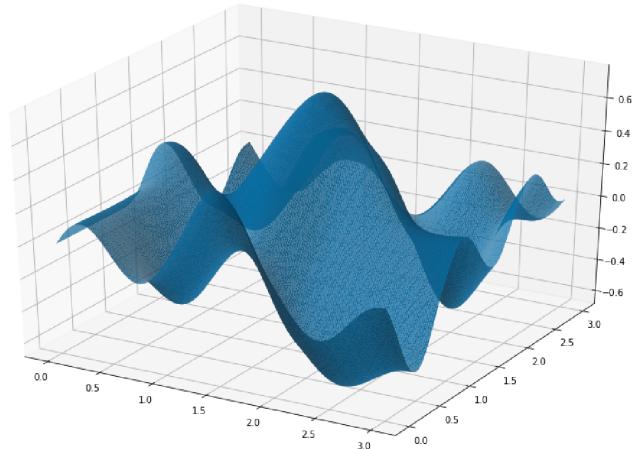


Figure 2.8: Example of Perlin noise 2D function produced while working on this project. This surface offers a smooth variation around the value 0 with amplituded $\in [0; 1]$.

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.

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 [11], using the *PyTorch* ecosystem to implement the necessary code.

The backbone of the architecture is the VGG-19 network, which is a convolutional neural network 19 layers deep, as in figure 2.10. This huge model has been pre-trained on over a million images from the ImageNet database [9], 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 that define the network and to initialize an empty module with those values. This is the perfect start for creating a style transfer network, which requires a further and briefer training phase, to completely customize



Figure 2.9: Different examples of application of a style-transfer NN on the same content image, with different style images, from [11]. The original picture depicts the Neckarfront in Tbingen, Germany (TOP-LEFT). The painting used as style image shown in the bottom left corner of each panel are in clockwise order: • The Shipwreck of the Minotaur by J.M.W. Turner, 1805 • Femme nue assise by Pablo Picasso, 1910 • Composition VII by Wassily Kandinsky, 1913 • Der Schrei by Edvard Munch, 1893 • The Starry Night by Vincent van Gogh, 1889.

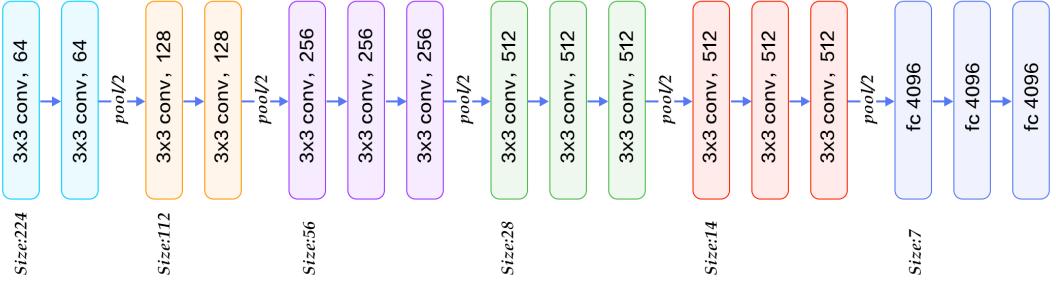


Figure 2.10: The structure of the VGG 19 network. It is for its most a convolutional NN with 224×224 input size and with some downsampling layers which reduce the first two dimensions of the tensors along with the information flux of the network. At the very end of the architecture, there are three subsequent fully connected layers, responsible for the actual classification based on the features extracted from the previous layers.

the network.

The key ingredient for finalizing the model is to insert some little but fundamental modifications and to extend the training on the pair of input images. This final training should be aware of the *concepts* of the 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 commonly used function to evaluate the preservation of content between two images is the simple Mean Squared Error as in equation (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 .

$$L_{Cont} = \|F_{XL}F_{CL}\|^2 \quad (2.17)$$

In order to evaluate this content loss, it is necessary to insert a custom transparent⁴ 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 [11]. This loss function is implemented similarly to the content loss module, as it will act as a

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

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 a 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_{XL}^k . 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} :

$$L_{Style} = \|G_{XL}G_{SL}\|^2 \quad (2.18)$$

After the appropriate insertion of the loss-function evaluator layers, one last piece for finalizing the model is the right choice of the gradient descent optimizer. As in [11] and according to the Deep Learning community the optimizer which suite best this role is the Limited Memory-BFGS [5],[25]. L-BFGS is an iterative algorithm in the family of quasi-Newton methods that approximates the Broyden-Fletcher-Goldfarb-Shanno algorithm (BFGS) using a limited amount of computer memory, and it is a popular choice when estimating parameters of a non-linear differentiable scalar function.

The final phase of the training process thus makes run the optimizer for hundreds of epochs on X , C , and S , and reduce the two loss functions values acting on the network's parameters. After the fine-tuning of the weights, the hybrid image is produced, as in Figure 2.9.

2.2 Model Development

The main goal of the present work, as stated before, is to recreate a three-dimensional virtual model of histological tissue as faithfully as possible and then, to perform planar sectioning on it to emulate virtually the traditional histological specimen preparation procedure. The creation of a model of such complex structures is definitely a high-level problem, and it has required a careful designing, made of subsequent stages of improvements. In this work, I will report only two specific attempts of modelization: the first aiming to represent pancreatic tissue, and the second oriented toward dermal tissue.

2.2.1 Pancreatic Tissue Model

The Pancreas is an internal organ of the human body, part of both the digestive system and the endocrine system. It acts as a gland with both endocrine and exocrine functions, and it is located in the abdomen behind the stomach. Its main endocrine duty is the regulation of sugar levels in the blood and the secretion of hormones, like insulin and glucagon. While, as a part of the digestive system it acts as an exocrine gland secreting pancreatic juice. The majority of pancreatic tissue has a digestive role, and the cells with this role form clusters (*acini*) around the small pancreatic ducts, and are arranged in lobes. The acinus secrete inactive digestive enzymes called zymogens into the small intercalated ducts which they surround, and then in the pancreatic blood vessels system [18]. In Figure 2.11 is shown a picture of the pancreas, with its structure and its placement in the human body.

All the tissue is actually rich in other important elements as the islets of Langerhans, and sporadic connective tissue all over the structure, which are clearly visible in the

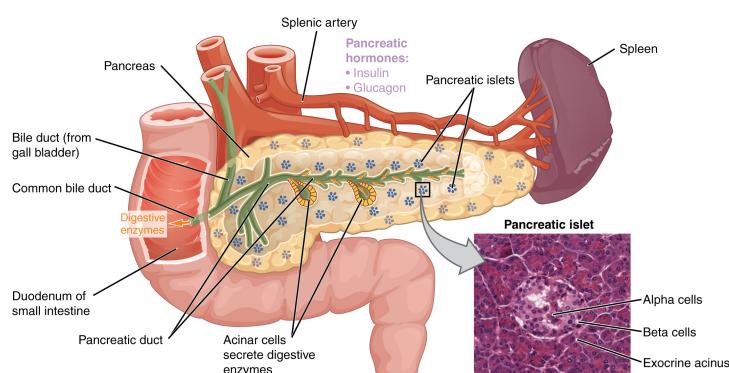


Figure 2.11: A picture of pancreas' structure in its phisiologiacl context. In this picture is clearly visible the macroscopic structure and the galndular organization at microscopic level, and how it reflects in the histological sample.

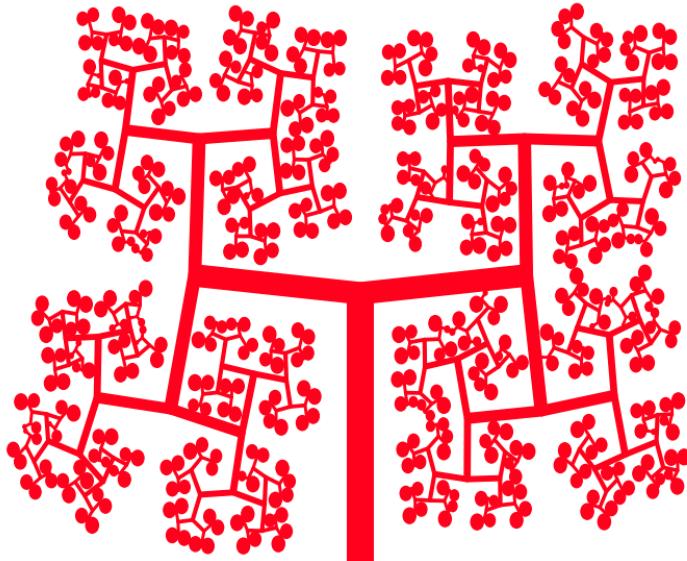


Figure 2.12: The development of the simple curve in Figure 2.1, with some features to give it a more realistic look, like progressive thickness, angular noise in bifurcation and spheres at free ends of the ramification. The image is made using the tools exposed in section 2.1.2.

traditional histological specimens. In this first attempt of modelization from scratch this second layer of complexity has not been already considered, and the main focus has been to reflect only the main structural features on the virtual specimens. Given pancreatic tissue's organization the first features I decided to put emphasis on were: 1) The iterative (with a fractal-like behavior) ramification of blood vessels for the irrigation of glandular acinus, 2) The space-filling distribution of acinus in the tissue, in fact, we expect a homogeneous density in the organ and to not see *holes* at all inside it. In this section I will describe step by step all the process I followed to create the model of a portion of pancreatic tissue, and all the interesting pitfalls I overcame.

1) 2D Ramification

The first step was took in two dimensions, and it was the choice of the right *structure* to emulate the ramification of blood vessels in pancreatic tissue. The choice fell on a particular parametric L-system, as the one shown in Figure 2.1, in section 2.1. This structure is made of an iterative bifurcation of gradually shorter segments, with an angle of $\pm 85^\circ$ respect the main direction. For a start I added some features to give a more realistic look to the structure, which are all well represented in Figure 2.12:

- A progressive thickness of the bifurcation's segments, starting from a thick

main branch that dwindle every junction. The idea is that the main blood vessel becomes gradually smaller becoming capillaries for single-cell irrigation.

- A progressive randomness in the angular deflection at every fork. Perfectly repeated angles are almost nonexistent in nature, so I decided to introduce an increasing indetermination in the angle of bifurcation from the main branch to the free ends of the structure's branches.
- Spheres at the ends of each branch, which acts as glandular acini. The maximum radius is comparable to the length of the final segments.
- A mechanism to avoid self-superimposition between branches and spheres. After the insertion of noise, the cumulative effect on the final segments might lead to different branches to intersect. This is clearly a paradoxical situation, as real tissues while growing naturally occupy the space in a gradual way.

To produce the specific image in Figure 2.12 I used a particular setting of the tools described in section 2.1.2, which have a greatly wider range of customization and could be used to create many other different structures to the need.

2) Expansion to 3D

The successive step I followed was to expand this structure in three dimensions and fill the space in each of the three directions. The idea to evolve the structure in Figure 2.12 is simply to twist of 90° the ramification at every junction point, in such a way to exit the previous belonging plane. However, putting into practice this development has not been easy. The organization of the structure in a 3D space requires an appropriate system of reference for handling subsequent rotations in three dimensions. The best option for handling relative 3D rotations, often used in computer graphics and every kind of 3D modelization, are quaternions, as shown in section 2.1.1.

In this new structure, segments are replaced with cylinders, and circles are replaced with spheres. At every bifurcation to every cylinder are applied the following transformations:

- a contraction in its extensions, regulated by an adjustable parameter R .
- the usual deviation of $\pm 85^\circ$ respect to the direction of the parent branch.
- a 90° specific rotation along the axis of its parent branch.

The result of this procedure is a 3D ramification like the one in Figure 2.13, in which we can recognize a good coverage of the space defined by the structure's boundaries and immediate relation with the 2D structure in Figure 2.12. It should be noted that, in the further refinements of the model from now on, there won't be present the progressive angular indetermination on the direction of branches. Although it

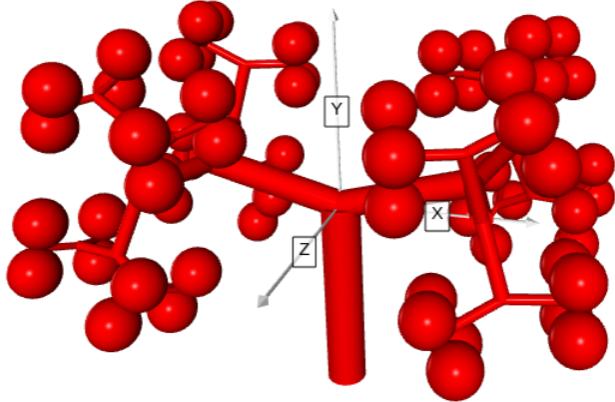


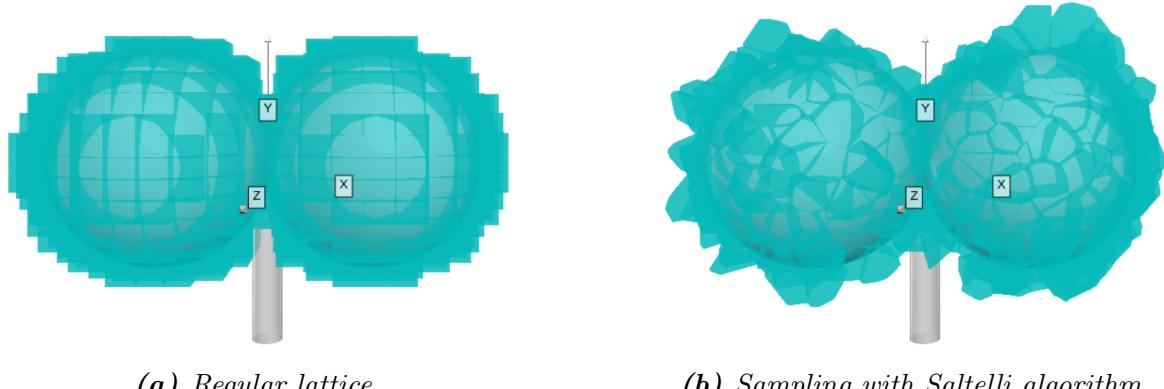
Figure 2.13: The three-dimensional expansion of the 2D ramification in Figure 2.12.

is a feature already implemented and working, it requires efficient control to avoid reciprocal overlapping between elements to produce a realistic structure. This second element has not been already developed and it would certainly enrich the representative power of the model.

As for the 2D ramification the production of this structure has required the implementation of a tool for the 3D generation with a greatly wider power, able to produce almost any type of three-dimensional iterative structure after the right adjustment, and with a high degree of customization. It is necessary to mention the fundamental tool which allowed me to accomplish this step of the development, which is the **Python** library **VPython**: a library for 3D graphics visualization. This library allows a convenient and powerful interface to draw many types of objects and to move them around in space, which has been priceless to orient my self in three dimensions while developing the model and to produce all the 3D images visible in this work.

3) Subdivision in Cells

Once the 3D backbone of the pancreatic tissue blood vessels ramification system has taken shape, the next step was to embed all this structure in a spatial partitioning process, to create the subdivision into single cells. To perform this important task I used a 3D Voronoi decomposition, as shown in section 2.1.3. Depending on the choice of the starting points, the Voronoi tessellation could be an excellent item to recreate individual cells because it could guarantee some important properties: all the regions are convex, adjacent, with similar size and volume, with different shapes, and without holes. These have been chosen as the most significant properties to be reflected in the first modelization of cells.



(a) Regular lattice.

(b) Sampling with Saltelli algorithm.

Figure 2.14: Comparison between two Voronoi decompositions. The first (left) is created from a regular lattice of starting points, and every piece is exactly equal to all the others, creating a regular subdivision of the space. The second (right) is created instead from a sampling made following the Saltelli quasi-random algorithm. The pieces are all different in shape, but they all have similar sizes and volumes. In this pictures in particular have been shown only the pieces of the tessellation which lie in correspondence to the boundaries of the spheres underneath. While watching this picture one should imagine the decomposition extended similarly in all the space around the ramification, within certain boundaries, which loosely contains the structure. This representation choice has been done just for enhance the interpretability of the image.

As shown in section 2.1.3, the decomposition strongly depends on the choice of the starting point. Points spread uniformly on a 3D regular lattice will produce a series of parallelepipeds repeated in the space. An example of uniform tessellation is shown in Figure 2.14a. On the other side, a decomposition based on a quasi-random generated point can present all the good properties we mentioned before, including the diversity in shapes. In Figure 2.14b is shown an example of a Voronoi decomposition based on points sampled in a 3D with the Saltelli algorithm, in reference to section 2.1.4. Regardless of the points sampling technique, the boundaries of the sampling 3D box have been chosen to loosely contain the ramification.

There are tough some delicate considerations to be highlighted about the decomposition procedure. The first regards the most external pieces of the decomposition. Whilst the internal pieces are neatly bounded and defined, the most external layer instead is made on unbounded regions, which extend themselves to infinity. Those regions have clearly to be rejected, as it would be absurd for a cell to have an infinity volume. Typically those unbounded regions are resized in order to adhere

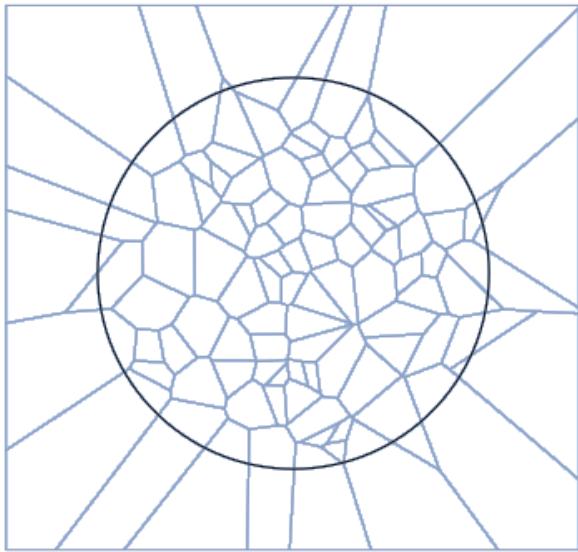


Figure 2.15: Example of circular cropping in a 2D Voronoi decomposition: all the regions which intersect the circumference have to be resized.

to some limiting boundaries, with an operation known as *cropping*. In Figure 2.15 is shown an example of circular cropping in a 2D Voronoi decomposition: all the regions which intersect the circumference have to be resized.

The cropping operation in 3D is extremely complex, tough. Thus, a more simple and efficient, yet less elegant, technique has been used. Instead of resizing the regions which lie on the boundaries of the sampling region, those regions have directly been rejected. This process is really fast and it does not lead to any danger of representativity loss if the boundaries are loose enough and if the density of sampling is not too low.

The other important consideration regards the density of sampling points. Increasing the number of points to be extracted from the same volume automatically the number of cells in the box will rise, and in contrast, their relative dimension will decrease. This is a key element of the model: a too rarified decomposition would not be able to reflect the complexity of the structure underneath, but a too crowd decomposition on the other side would lead to an unrealistic dimension of the cells in the tissue. Furthermore, this parameter has a huge influence on the computing time necessary to generate the model and to process it for the sectioning as will be shown in section 2.2.3. In almost all the applications so far, the density parameter has been tuned by eye, with a trial and error procedure. Although, a more rigorous way to adjust this parameter would be to consider the average dimension of the cells and make some microanatomical considerations to define the correct

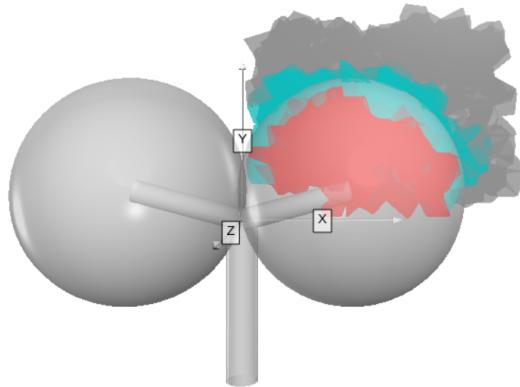


Figure 2.16: Portion of the complete Voronoi decomposition, showing the three different classes of cell in three different colors: the internal cells in red, those on the boundaries in turquoise and the external cells in gray.

relative dimensions. The measure of the volume⁵ of the decomposition's regions is an accessible parameter, thus an easy way to estimate the average linear dimension of the cells can be to approximate all the cells to cuboid seeing their volumes as $V \approx L^3$. Averaging all the L measures an estimate \hat{L} can be done. This average length may be compared to the length of the blood vessel ramification, allowing a good reference tool.

4) Cells Identity Assignment

The great power of creating all the models virtually is to know exactly the identity of every point in the structure. Although, This identity has to be reflected at the cellular level, assigning to every region a label. Imagining the Voronoi decomposition represented in Figure 2.14b extended to the entire box containing the ramification, good discrimination would distinguish three classes of cells: those which lie completely inside a sphere, those which lie completely outside a sphere, and those which lie on the boundaries of a sphere. In Figure 2.16 is shown a portion of the complete decomposition where the three classes of cells are reported with different colors: the internal cells in red, those on the boundaries in turquoise, and the external cells in gray.

In this particular case to find the relative position between every sphere in the structure and each cell it has been used a test on the proximity between the centers and the vertices of every polyhedral cell. If all the vertices of a region lie within a distance lower than the radius from the center of the same sphere then that region can be said to be an internal one. If none of the vertices lie within the radius

⁵The volume is expressed in the same arbitrary length unit of measures used during the ramification structure. This allows a coherent reference tool.

distance from any center then that region is said to be external. In any other case, the region is said to be on the boundaries of some sphere, and this third label is assigned to it. As could be imagined the number of cells inside the volume can grow very quickly, and in the more rich ramifications also the number of spheres could be high. If we think that any polyhedron has a number of vertices of the order of 20/30 then it is clear that the number of distance evaluations could grow very quickly, requiring some relevant computational power in the more extended simulations. In order to optimize this computation, I decided to use a python implementation of a K-dimensional Tree, which is a space-partitioning data structure especially suited for fast and optimized computation of distances [3]. A K-d Tree is an algorithm which iteratively binarily splits the space: every node of the tree could be thought as a splitting $(k - 1)$ -hyperplane dividing the space into two semi-hyperspace. The result is an optimized algorithm for repeated distance evaluations. As for many other tools, in my code I used an excellent pre-implemented module `KDTree` from the `Scipy` library.

This procedure of labeling the regions is completely customizable, and it should be adapted to the specific application. By the way, the principle will always be to perform some sort of spatial consideration respect to the primary structure and assign all the interesting labels accordingly to the cells in the volume.

After labeling the cells in the decomposition the model is considered complete. Every enrichment to the structure should be reflected in some type of label for the cells, which are chosen as the fundamental unit in the model. As we will see in section 2.2.3 during the sectioning process in the produced image will be printed mainly the identity of the cells, hence any detail on a finer scale in the model would not be conveyed properly on the final image.

2.2.2 Dermal Tissue Model

The dermis or corium is a layer of skin between the epidermis (with which it makes up the cutis) and subcutaneous tissues, that primarily consists of dense irregular connective tissue and cushions the body from stress and strain. It is divided into two layers, the superficial area adjacent to the epidermis called the papillary region and a deep thicker area known as the reticular dermis.[1] The dermis is tightly connected to the epidermis through a basement membrane. Structural components of the dermis are collagen, elastic fibers, and extrafibrillar matrix.[2] It also contains mechanoreceptors that provide the sense of touch and thermoreceptors that provide the sense of heat. In addition, hair follicles, sweat glands, sebaceous glands (oil glands), apocrine glands, lymphatic vessels, nerves and blood vessels are present in the dermis. Those blood vessels provide nourishment and waste removal for both dermal and epidermal cells.

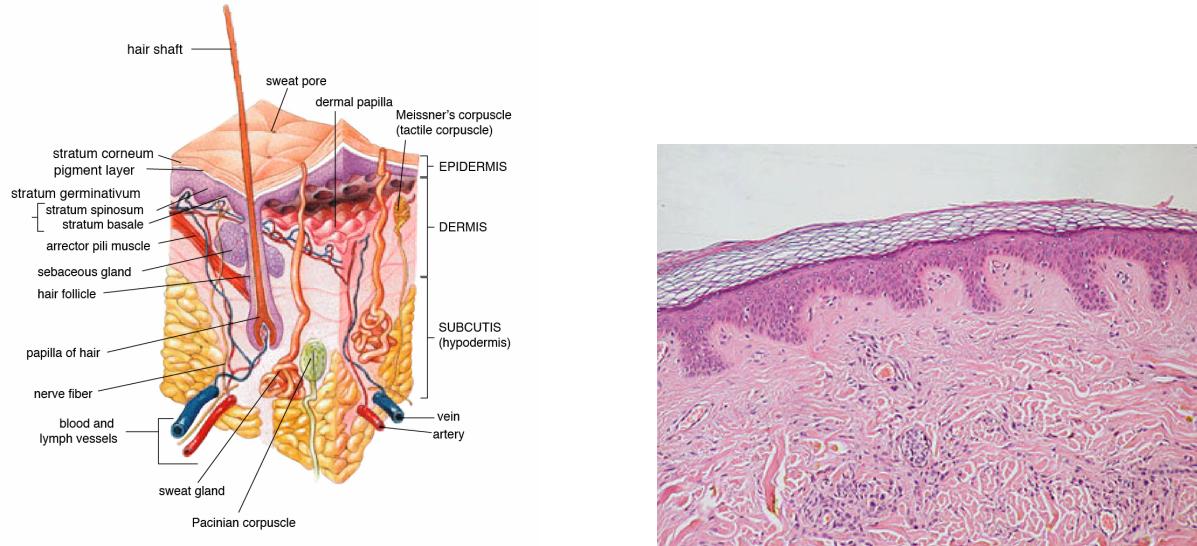


Figure 2.17: Microanatomical description of a region of dermal tissue and all the interesting elements present in cutis, and subcutaneous layer. An actual histological specimen from a sample of dermal tissue.

2.2.3 Sectioning Process - Synthetic Images

SECTIONING AND IMAGE PRODUCTION (random direction) -; SEGMENTATION MASK

ADJUSTMENTS: PALETTE, NOISE, STYLE TRANSFER

PIPELINE TO AUTOMATIZATION
ALTERNATIVE METHODS???

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] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509517, September 1975.
- [4] 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.
- [5] C. G. BROYDEN. The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 03 1970.
- [6] Siddharth Singh Chouhan, Ajay Kaul, and Uday Pratap Singh. Image segmentation using computational intelligence techniques: Review. *Archives of Computational Methods in Engineering*, 26(3):533–596, February 2018.
- [7] 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.
- [8] Alessandro d'Agostino. Dataset generation for the training of neural networks oriented toward histological image segmentation, april 2020.
- [9] 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.
- [10] 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.

- [11] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style, 2015.
- [12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [13] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks, 2016.
- [14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.
- [15] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.
- [16] 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.
- [17] 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.
- [18] Daniel. Longnecker. Anatomy and histology of the pancreas, 2014.
- [19] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey, 2020.
- [20] 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.
- [21] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta Numerica*, 8:143195, 1999.
- [22] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.
- [23] Andrea Saltelli. Making best use of model evaluations to compute sensitivity indices. *Computer Physics Communications*, 145(2):280 – 297, 2002.
- [24] 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.

- [25] David F Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of computation*, 24(111):647–656, 1970.
- [26] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.
- [27] Sandro Skansi. *Introduction to Deep Learning: From Logical Calculus to Artificial Intelligence*. Springer Publishing Company, Incorporated, 1st edition, 2018.
- [28] I.M. Sobol. Uniformly distributed sequences with an additional uniform property. *USSR Computational Mathematics and Mathematical Physics*, 16(5):236 – 242, 1976.
- [29] 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.
- [30] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.
- [31] 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.