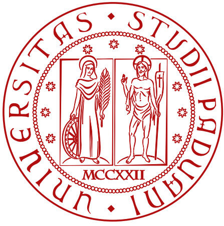
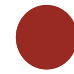


Università degli Studi di Padova



Dipartimento di Ingegneria dell'Informazione

Master Thesis in ICT for internet and multimedia

Geometric Shape Recognition Algorithm from Coaxial Vision System Images for 3D Reconstruction

Master Candidate

Dal Bello Nicola

Student ID 2004064

Supervisor

Prof. Zanuttigh Pietro

University of Padova

Co-supervisor

Passarella Luca, Cerato Davide

Sisma S.p.A.

Date
12/10/2023

Academic Year
2022/2023

Abstract

This thesis focuses on geometric shape recognition from 2D high-resolution grayscale images suitable for 3D reconstruction. The aim is to develop an accurate computer vision algorithm that is able to retrieve object geometries through deep learning techniques from images provided by a coaxial vision system of laser systems.

The main task is to assign a unique label (or category) to every single pixel in the image, which can be considered as a dense classification problem. This type of problem in computer vision domain is known as Image Segmentation. Segmentation is a key step in image understanding and is used in many different industries. Due to limited resources (GPU memory above all), image segmentation is a very challenging task, especially in high resolution images.

Nowadays, convolutional neural networks (CNNs) have shown excellent results in object recognition and have also been the first choice for dense classification problems such as image segmentation.

The research explores the strategies and techniques adopted in order to obtain an optimal solution and the problems encountered during development.

The images exploited for the machine learning algorithms were obtained from high-precision laser machines and systems, capturing objects used in the industrial and jewelery sectors, mostly metallic, of all types of shapes and sizes. A total of 700 photos were captured, which were segmented by hand to obtain ground truth.

Contents

List of Figures **xi**

List of Tables **xiii**

List of Acronyms **xix**

1 Introduction **1**
1.1 Sisma S.p.A. 2
1.1.1 Main Software: SLC^3 and 3D Scan Pack 2
1.2 Thesis goals 4
1.2.1 Image Segmentation 6
1.3 Dataset 7
1.4 Thesis structure 8

2 Image Segmentation **11**
2.1 Common Segmentation applications 11
2.2 Image Segmentation Techniques 12
2.3 Main Approaches for Segmentation 13
2.3.1 Traditional approaches 13
2.3.2 Deep learning approaches 15
2.3.3 U-Net 20
2.3.4 RefineNet 22
2.3.5 High-Resolution Refine Net (HRRNet) 23
2.4 Evaluation metrics 24

3 Main tool and libraries **31**
3.1 Programming languages 31
3.2 Main Python libraries 32

3.3 OpenCV and Emgu CV 33
3.4 TensorFlow and TensorFlow.NET 34
3.5 Tools 34

4 Data Exploration and Preprocessing for Improving the Performance of the AI Model 37
4.1 Dataset Overview 37
4.1.1 Data Augmentation 39
4.1.2 Synthetic Dataset 39
4.2 Preprocessing and Edge detectors 41
4.2.1 Sobel Edge Detection 42
4.2.2 Canny Edge Detection 43
4.2.3 Final Considerations 45
4.3 Train, validation and Test split 46

5 Image Segmentation Model Architecture 49
5.1 Rough Module 50
5.1.1 Model Training 54
5.1.2 Model Evaluation 57
5.2 Refinement Module 59
5.2.1 Model Training 62
5.2.2 Model Evaluation 63

6 Edge Refinement Interpolation 65
6.1 Image Interpolation Strategies 65
6.2 Edge Refinement Interpolation 68
6.3 Final Evaluation 73

7 SLC^3 Integration 77
7.1 Implementation details 77
7.1.1 Export and Import of the models 77
7.1.2 User Interface 78

8 Conclusions 83
8.1 Future works 84
8.2 Personal Growth and Accomplishments 85

References 87

List of Figures

1.1 Engravings types 2
1.2 Example of laser marking via 3D Scan Pack Module 3
1.3 Surfaces Acquisition 3
1.4 Axes Synchronization 4
1.5 Segmentation Example 5
1.6 Segmentation Example 6
1.7 Example Data 8

2.1 Encoder-Decoder Neural Network 17
2.2 Convolution Operation 18
2.3 Max Pooling Operation 18
2.4 Transposed Convolution Operation 20
2.5 U-Net structure [25] 21
2.6 RefineNet Architecture [14] 22
2.7 Global Process HRRNet [28] 24
2.8 Confusion Matrix 25
2.9 Dice Coefficent 28
2.10 IoU Coefficent 29

3.1 OpenCV and EmguCV logo. 33
3.2 Tensorflow logo. 34

4.1 Example Data 38
4.2 Synthetic Shapes 40
4.3 Generated Synthetic Data Examples 41
4.4 Image Derivatives. 42
4.5 X and Y direction Kernel, respectively 43
4.6 Hysteresis Threshold [26]. 44

5.1 Whole Architecture 49
5.2 Rough Module Architecture 51
5.3 Refinement Module Architecture 60
5.4 Results Comparison. 64

6.1 Bilinear Interpolation 66
6.2 Interpolation Example 69
6.3 Interpolated Results. 74

7.1 User Interface 79
7.2 Rendered 3D model 80
7.3 UV Maps 81

List of Tables

4.1 Number of samples for each set. 47

5.1 Rough Module Results 58
5.2 Refinement Module Results 63

6.1 Final Interpolation Results 75

List of Acronyms

ML Machine Learning

DL Deep Learning

AI Artificial Intelligence

ANN Artificial Neural Network

DA Data Augmentation

IT Information Technology

NN Neural Network

TPR True Positive Rate

FPR False Positive Rate

IDE Integrated Development Environment

IOU Intersection over Union

ROI Region of Interest

RU Refinement Unit

OS Original Size

Introduction

Computer vision is an area of artificial intelligence that teaches and equips machines to comprehend the visual environment. Deep learning models and digital photos can be used by computers to precisely recognize, categorize and monitor objects. As a result of recent developments in areas like artificial intelligence and computing capabilities the field of computer vision has made significant progress toward becoming more pervasive in everyday life. The goal of computer vision in AI is to create automated systems that can interpret visual data (such pictures or videos) in a similar way to how people do. The aim is to teach computers how to individually analyze and understand images.

The objective of this project is to explore the potential of Machine Learning and Deep Learning structures in the computer vision field trying to improve the handcrafted vision system, not based on machine learning, already present in the Sisma main software for laser marking $''SLC^3''$. This application has the task of facilitating the operator in interfacing with laser machines to create 2D textures or 3D engravings to enhance or customize fashion products.

Specifically, I developed an algorithm that through a convolutional neural network (CNN) for semantic segmentation is able to obtain the geometries of a random object separating it from the background. The input images are in grayscale and of variable size. Image sizes vary depending on the scanned sample and typically range from the smallest in size (1620x1620) to the largest in size (10,000x10,000).

1.1 Sisma S.p.A.

This master's thesis project is being developed in collaboration with Sisma S.p.A., which is a worldwide reference for the design and production of very high precision laser machinery and systems. It produces over 130 different models of machines for the automatic production of jewellery, extending its skills to offer a panorama of products that includes resin and metal 3D printers, laser systems for marking, engraving, welding and cutting as well as the solid sector of machines for the automatic production of gold chain. Sisma includes an internal research and development department to define tailor-made projects according to customer requests both from the hardware and software side.

The primary mission is to design and implement innovative systems and solutions, at the service of those who produce and create, using the technologies of laser precision micromechanical systems and additive manufacturing (3D).

1.1.1 Main Software: SLC^3 and 3D Scan Pack

In my internship at the company i worked to improve the functionalities of the main software "SLC^3", that provides an interface for each Sisma laser marking machine, which can easily create 2D texture and 3D engravings to enhance or customize fashion products.

SLC^3 provides tools that allows to perform:

- **2D engraving on 2D surface:** two-dimensional engraving on flat surface
- **2D engraving on 3D surface:** two-dimensional engraving on a non-planar surface
- **3D engraving on 2D surface** three-dimensional engraving on flat surface
- **3D engraving on 3D surface:** three-dimensional engraving on a non-planar surface

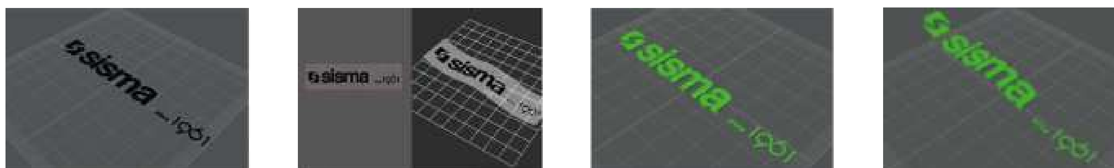
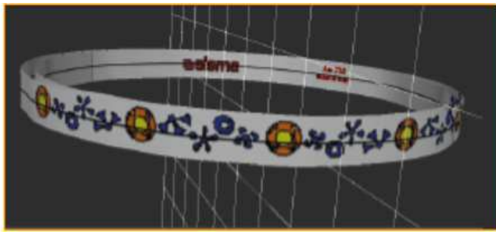


Figure 1.1: Engravings types

The software needs a 3D model of the sample in concern in order to efficiently engrave non-flat surfaces. During my internship I focused on the *3D Scan*

Pack Module which allows to automatically process objects like rings, bracelets, bangles, and any other oval or circular sample without knowing their geometry. Thanks to this, object shape is automatically recognized by a coaxial camera and then laser processed with the synchronized movement of a roto-tilting spindle. 3D Scan Pack takes a top-down profile photo of the object using a coaxial camera and a roto-tilting mandrel and uses this information to generate a 3D reconstructed geometric model.

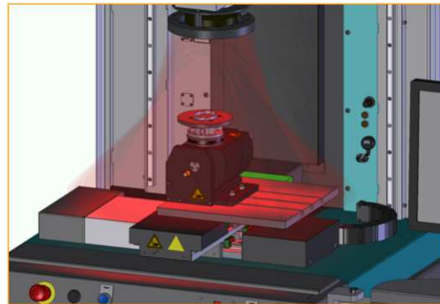


(a) Bangle 3D model software visualization

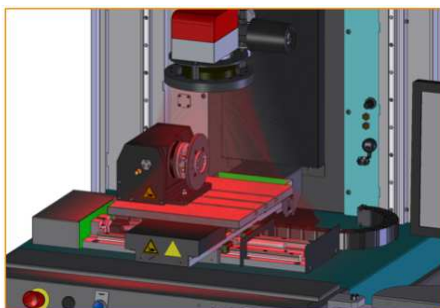
(b) Final result of a laser processed bangle

Figure 1.2: Example of laser marking via 3D Scan Pack Module

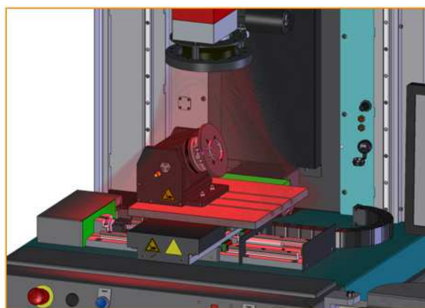
The acquisition process is illustrated by the figures 1.3.



(a) Profile



(b) External



(c) Internal

Figure 1.3: Surfaces Acquisition

First, the spindle tilts to vertical position (a) while the laser head sets at the

correct height for the coaxial camera to acquire the object profile in focus. The geometries are obtained from a shape recognition algorithm which has the task of distinguishing the pixels belonging to the background from those belonging to the region of interest in the input image. Once the geometries have been extrapolated the user will see the 3D model together with the unwrapping of the object's inner and outer surfaces. On these UV maps, the user can easily place and visualize the lasering pattern in form of vector files, texts or textures. Auto-tilting spindle makes laser processing possible on both inner and outer surfaces of the sample. With the 3D obtained object geometry, the spindle is tilted horizontally for outer (b), or to a small angle for inner (c) surface acquisition. Once entered the scanning range expressed in degrees, the mandrel rotates while the coaxial camera synchronously acquires a set of images, which are stitched and lied onto the reconstructed UV map to visualize the object surfaces. Here, lasing processing pattern can be placed either manually or automatically.

The synchronization of the axes allows to focus the camera and perform precise laser engraving.

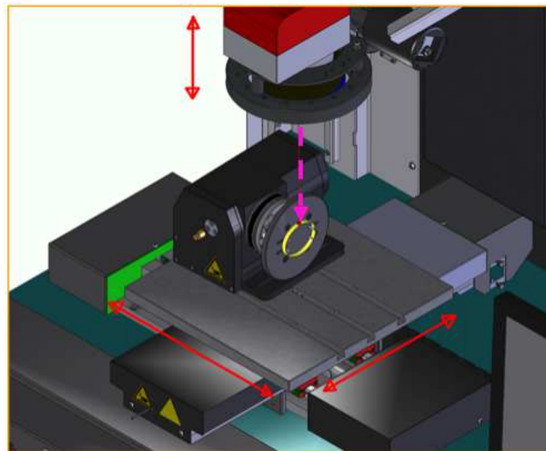
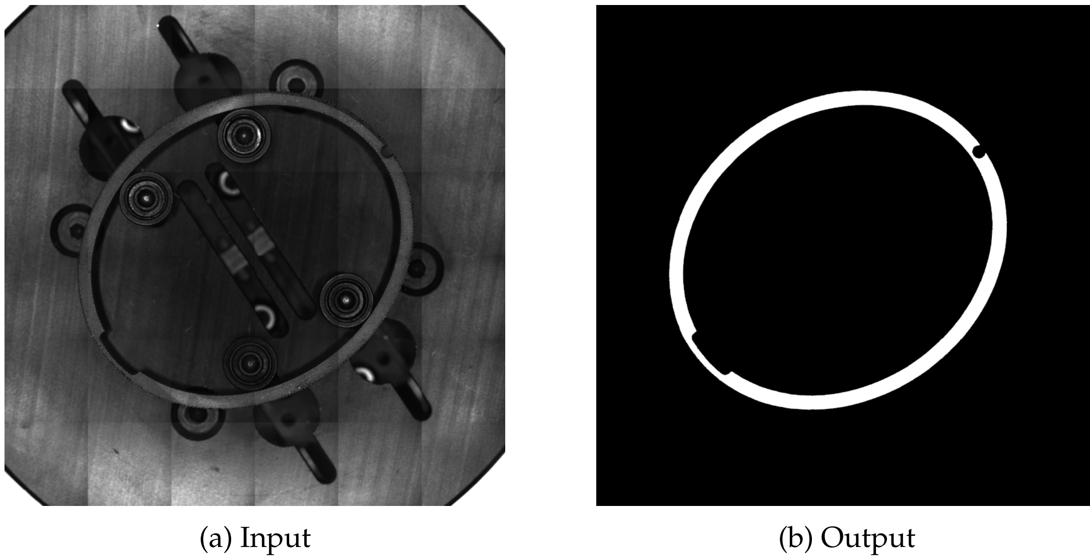


Figure 1.4: Axes Synchronization

1.2 Thesis goals

My work focused on improving the shape recognition algorithm. The shape recognition algorithm currently receives the scan of the profile as input and output a binary mask, see figure 1.5, which distinguishes the white pixels belonging to the sample under examination from the black ones belonging to the background.



(a) Input (b) Output

Figure 1.5: Segmentation Example

The previous strategy applies median filters, to remove low frequency details, and extrapolates the edges map using the *Canny* [3] method, then starting from the center, rays are projected towards the edges of the image and the coordinates where there is an edge are recorded. The *Canny* algorithm is one of the most accurate specified edge detection techniques that uses a multi-step algorithm to detect a wide range of edges in images. It offers accurate and dependable detection thanks to the application of Non-maximum Suppression and Hysteresis Thresholding operations. The final 3D model is constructed using the obtained coordinates.

This algorithm is limited to recognize only circular or oval shapes such as rings and bracelets, furthermore it cannot achieve satisfactory results when the edges of the samples are not well defined or the objects do not have a totally flat profile (figure 1.6). Moreover, reflections caused by external lights often lead to unwanted results.

Hence the idea of a deep learning solution with the aim of overcoming these shortcomings thus obtaining better results and expanding the range of recognizable shapes. However, putting Machine Learning technologies into practice can be difficult and complicated. To maximize the impact of AI we must consider various factors, such as the quality of data, which is a common issue and limitation when implementing AI models.

The fact that the images being analyzed are of various sizes and high resolution is crucial to take into account. In order to be analyzed by the algorithm that

Figure 1.6: Segmentation Example

generates the 3D model, the dimensions of the output mask must match those of the source image.

1.2.1 Image Segmentation

Image segmentation is a crucial component in image understanding, in its most basic definition, it is the division of an image into a number of regions. The pixels of these regions should typically have some qualities in common. In general, there are three groups of image segmentation tasks:

- **Semantic segmentation:** simply the task of assigning a class label to every single pixel of an input image.
- **Instance Segmentation:** it's like semantic segmentation but it distinguishes between different objects of the same class.
- **Panoptic segmentation:** combine semantic and instance segmentation.

For this thesis I focused on a sort of binary semantic segmentation with only two types of classes, since the task is to assign a label for each pixel belonging to the sample and a label for the background. Convolutional neural networks (CNNs) have proven to perform exceptionally well at object recognition and are the preferred option also for dense classification tasks. Image segmentation is a quite challenging task especially when we work with high resolution images since repeated subsampling operations like pooling or convolution striding in deep CNNs lead to a significant decrease in the initial image resolution. Multiple stages of spatial pooling and convolution strides reduce the final prediction

typically by a factor of 32, leading to a loss of much of the finer image structure and therefore, they are unable to output accurate high-resolution prediction. Low-level visual information is essential for accurate prediction on the boundaries or details. Using larger receptive fields without downscaling the image needs to perform convolutions on a large number of detailed (high-resolution) feature maps that usually have high-dimensional features, which are computational expensive and require huge GPU memory resources.

For this thesis I developed a encoder-decoder architecture that has the aim to extrapolate a low-resolution segmentation mask, therefore, a refinement module is proposed to improve the accuracy of the high-resolution segmentation. The output of the first is used as the input of the latter. Finally, given that the input images have variable dimensions, instead of using common interpolations algorithms like bilinear, bicubic or lanczos [20], I have developed an algorithm that exploits the edge information of the original high resolution image to obtain the most accurate final segmentation possible, that follows the shape of the sample.

1.3 Dataset

All the dataset images considered were captured by me using the prototypes of the Sisma coaxial vision system present in the company. About 50 objects of different shapes and sizes were scanned (including large and small jewels such as rings and bracelets, and industrial samples), obtaining a dataset of about 700 grayscale images. The dimensions of the photos produced vary according to the sample examined, typically range from the smallest in size (1620x1620) to the largest in size (10,000x10,000)

Scans are typically performed placing a dark background to facilitate capture and accentuate the contrast with metallic samples or performed directly when the sample is positioned on the mandrel gripper (also dark) ready for the laser marking process.

To diversify and expand the dataset and thus obtain more robust models the same objects were photographed several times in different positions, backgrounds, exposure times and output dimensions (see figure 1.7).

In my internship a lot of time was spent getting the ground truth by hand using free image editors like *Gimp* [30]. The ground truth had to be very accurate to get the most precise output masks possible.

Figure 1.7: Example Data

Since manual acquisition and segmentation process is highly time consuming, in order to broaden the set of available data, techniques like data augmentation were applied. Furthermore, I created an algorithm for consistent synthetic data production with the aim of exploiting shapes and samples not physically present in the company and considerably reducing the manual segmentation time. Using real world data is always better but it takes a long time to manually capture and segment them.

1.4 Thesis structure

The structure of this thesis is organized to provide a comprehensive understanding of the research and implementation process. The thesis is divided into several sections, each focused on specific aspects related to the development of a machine learning solution for shape recognition for Sisma's main software.

Chapter 1 provides an introduction to Sisma S.p.A., the company where I did the internship and its main software SLC^3. The automatic marking process is also described, using the 3D scan pack module, from the generation of the 3D model to the various movements of the mandrel. In the introduction the thesis goals are outlined specifying the limitations of the previous algorithm and listing the potential benefits of a machine learning-based solution. The image segmentation task in its various forms is also introduced together with the dataset used to train my models.

The chapter 2 explores the importance of image segmentation in a business scenario. Common computer vision tasks are introduced, with particular attention to the most important image segmentation strategies. Subsequently the various factors involved in deep learning approaches are listed and discussed,

including disadvantages and benefits. This chapter presents the main architectures that I took inspiration from, in the development of my solution. At the end, evaluation metrics are exposed.

The main tools and libraries used in the whole project are discussed in the third section.

Chapter 4 concerns data exploration and preprocessing techniques to enhance the performance of both AI models implemented. Includes a description of the operations performed to expand the training data set automatically and considerations regarding class imbalance. The techniques used to extrapolate edge maps are also explored by listing their steps and discussing their properties.

Chapter 5 delves into the implementation of my solution and presents a comprehensive evaluation of these architectures. The two modules are widely illustrated through two figures that outline the set of layers and operations and by the developed code. It also focuses on the loss functions and the type of optimization algorithm chosen. The main objective of this section is to appropriately describe the developed models and evaluate their performance and effectiveness.

The chapter 6 illustrates the main interpolation strategies for upsample the masks obtained in the previous phases to the original image dimensions. An interpolation method developed by me is exposed which exploits the information extrapolated from the edge maps to obtain more accurate masks. This chapter encompasses a detailed examination of final results obtained.

Chapter 7 shows the user interface developed in C# and describes the work done to integrate the machine learning models into $SLC3$.

The thesis's final conclusion and future works are both included in the final chapter 8.

Image Segmentation

Computer vision is an interdisciplinary branch of science that studies how to make computers grasp complex information from digital photos or videos. Engineering-wise, it aims to automate activities that the human visual system can perform.

Image segmentation is a widely used method in digital image processing and analysis to divide an image into various parts or areas. Foreground and background can be distinguished in an image by segmenting it, or pixels can be grouped together based on their similarity in color or shape.

For computer vision technologies and algorithms, image segmentation is a crucial building element. It is employed in a variety of real-world contexts, including face detection and recognition, video surveillance, satellite image analysis, medical image analysis, and autonomous vehicle computer vision.

2.1 Common Segmentation applications

Image segmentation has undergone many improvements, both large and small, that have had a profound impact on our lives. Here are some examples of use cases for segmentation models.

Object detection and scene understanding

Accuracy of object detection models is improved by segmentation techniques. Scene comprehension can be enhanced by determining a scene's semantic mean-

ing, and the capacity to partition items can be applied to tracking, object recognition, and relational understanding.

For example It is used to automatically locate objects from images captured by drones or satellites, in agriculture for flood monitoring, in any intelligent video surveillance system for pedestrian detection and many others.

Medical imaging

Segmentation is used in medical image analysis to recognize and separate various organs and tissues. Diseases can be diagnosed and treatment strategies can be developed using this knowledge. Algorithms for automated dental radiography and analysis have also been developed using image segmentation techniques.

In fact, one of the most widely used algorithms in medical AI nowadays is image segmentation.

Autonomous driving

Semantic segmentation techniques are widely used on self-driving cars. We can interpret environmental data gathered from the car's sensors and cameras by assigning a predefined class to each pixel in an image. The benefit of autonomous vehicles is their ability to comprehend their surroundings, locate vehicles and other objects on the road, and plan a safe route for the vehicle.

2.2 Image Segmentation Techniques

In this thesis I focused on Binary Image Segmentation where the goal is to separate the foreground from the background, in which each pixel in an image is given a label that specifies its content. Foreground-background separation is relatively easier compared to tasks like instance or panoptic segmentation. Like for Semantic segmentation, we display identical items with the same color since it is unable to discriminate between distinct instances of the same category. On the other hand, instance segmentation can discriminate between many instances of the same category, so samples belonging to the same class are divided by various colors.

2.3 Main Approaches for Segmentation

A variety of approaches to perform image segmentation have been developed over the years using domain-specific knowledge to effectively solve segmentation problems in specific application areas.

The strategy previously adopted by the older SLC^3 version was to look for rapid changes in pixel values, which often denote the edges that define a region. Other common approaches are based on detecting similarities in regions of image like region growing, clustering and thresholding.

The assumption that items in an image or video share similar characteristics is the basis of the segmentation principle used by AI to process picture data. AI models can learn to identify objects by extracting features from data and clustering pixels based on similarity.

2.3.1 Traditional approaches

There are different techniques for image segmentation, every solution has advantages and disadvantages and depends on specific application and type of images we want to segment.

- **Threshold Method:** Thresholding divides pixels according to their intensity in relation to a predetermined value or threshold, making it the simplest approach for segmenting images. Suitable for images with high contrast between foreground and background. The threshold value can be constant in low-noise images, but in some cases is necessary to use dynamic thresholds.
- **Region Based Method:** Involves dividing an image into regions with similar characteristics. Each region can grow or shrink or merge with another region based on seed points. Good choice for images with a wide range of features but it can be slow for large images.
- **Edge Based Method:** Identifies the edges of various objects in a given image. It helps locate features of associated objects in the image using the information from the edges. Edge-based segmentation algorithms identify edges based on contrast, texture, color, and saturation variations. Good results for low-contrast images but sensitive to noise.
- **Clustering Based Method:** Clustering algorithms are unsupervised classification algorithms that help identify hidden information in images. The method separates data pieces and groups comparable elements into clusters, dividing images into groups of pixels with similar properties. Best for images with many objects but can be computationally expensive.

- **Graph Partitioning Methods:** involves the application of graph theory to construct a representation of an image in the form of a graph. In this approach, each image pixel is represented as a node, while the edges connecting the nodes represent the degree of similarity between the corresponding pixels. It can be seen as a hybrid method which combines different approaches like the edge based, region based and many others.

The most popular strategies are introduced below.

Otsu's Method

In computer vision and image processing, Otsu's method [27] is used to perform automatic image thresholding. The algorithm returns a single intensity threshold in its most basic form, dividing pixels into the foreground and background classes. The threshold is automatically computed from the histogram. The success of this operation depends on various factors like the distance between peaks, noise, illumination and relative size of the regions. The optimal global threshold is determined by minimizing intra-class intensity variance, or equivalently, by maximizing inter-class variance.

K-Means

K-Means clustering algorithm [23] is an unsupervised algorithm and it is used to separate the interest area from the background. It clusters, or partitions the given data into a fixed number K of clusters or parts based on the K centroids. It is an iterative algorithm that alternate two procedures that try to find cluster centers and point-cluster allocations that minimize the error made by approximating the points with the cluster centers. It always converges to a solution but does not guarantee that it is the optimal one. The final solution depends on the initialization of the parameters and the number of clusters must be known a priori. Furthermore, it assumes that clusters are spherical and have similar variance, which may not be suitable for complex or irregular clusters.

Mean Shift

Mean shift [6] is an unsupervised learning algorithm that is mostly used for clustering. Since it is non-parametric and doesn't require a predetermined shape of the clusters in the feature space, it is frequently utilized in real-world data analysis (such as image segmentation). In simple terms The mean shift algorithm seeks modes or local maxima of samples' density in the feature space.

In other words, Mean shift is a procedure for locating the maxima of a density function given discrete data sampled from that function. Like K-Means, it is an iterative method. It needs a single input parameter which is the window size. It is not trivial to set since inappropriate window size can cause unwanted behaviors. Differing from the K-Means algorithm, It output variable number of not assumed spherical clusters.

2.3.2 Deep learning approaches

Traditional image segmentation methods such as those presented previously can be quick and easy, but they frequently need a lot of fine-tuning to fit particular use cases, moreover they are not sufficiently accurate to use for complex images. Machine learning and Deep learning solutions try to overcome this problems increasing accuracy and flexibility.

Deep learning is a subset of machine learning that uses artificial neural networks (ANNs) to model and solve complex problems. The input data for deep learning algorithms is propagated through an input layer, several hidden layers, and finally the output layer in a layered architecture. Each layer applies a set of mathematical operations, called weights and biases, to the input data, and the output of one layer serves as the input to the next. During the training phase we reduce the error between the projected output and the true output, modifying DL model's weights and biases. This is typically done using a variant of gradient descent, an optimization algorithm that adjusts the weights and biases in the direction of the steepest decrease in the error. Deep learning can also manage huge and complex data, and it has been utilized to solve a variety of challenges achieving state-of-the-art performances.

Unfortunately these technologies have some disadvantages we have to consider:

1. **High computational cost:** Deep learning model training takes a lot of processing power, requiring powerful GPUs and lots of RAM. This can be costly and time-consuming.

2. **Overfitting:** When a model performs well on training data but underperforms on new, untrained data, it is said to overfit. Deep learning frequently encounters this issue, which can be brought on by a lack of data, a complex model, or a lack of regularization, especially when working with big neural networks.

3. **Lack of interpretability:** Deep learning models can be complicated and challenging to interpret, especially those with several layers. Some deep

learning models are referred to as "black-box" models because it can be challenging to figure out how the model makes predictions and what influences those predictions.

4. **Dependence on data quality:** The quality of the data that deep learning algorithms are trained on is crucial. The model's performance will be negatively affected if the data is noisy, incomplete, or biased.

5. **Limited to the data its trained on:** Deep learning models can only make predictions based on the data it has been trained on. They might not be able to extrapolate to new contexts or conditions that weren't included in the training data.

Deep neural network technology is especially effective for image segmentation tasks. Model training is used in machine learning-based image segmentation techniques to enhance the program's capacity to recognize significant features.

Different neural network implementations and designs are appropriate for image segmentation but they typically share the same fundamental elements (figure 2.1):

- **An Encoder:** a set of successively deeper, smaller layers used to extract visual information. If the encoder has prior experience with a comparable task (like image recognition), it may be able to use that experience to complete segmentation tasks.
- **A Decoder:** a set of layers that gradually transform the encoder output into a segmentation mask matching the pixel resolution of the input image.
- **Skip connections:** Multiple connections across long-range neural networks enable the model to recognize information at various scales, improving model accuracy.

Unlike classification, where the outcome of the network is a single value, segmentation in addition to pixel-level discrimination also requires a mechanism for projecting the discriminative features acquired throughout the encoder's many phases onto the pixel space.

One key term that comes up frequently is "receptive field" which refers to the area in the input volume that a specific feature extractor (filter) is focusing on. Receptive field (context), to put it simply, is the portion of the input image that the filter is currently covering.

The main operations typically used in encoder and decoder paths are described below.

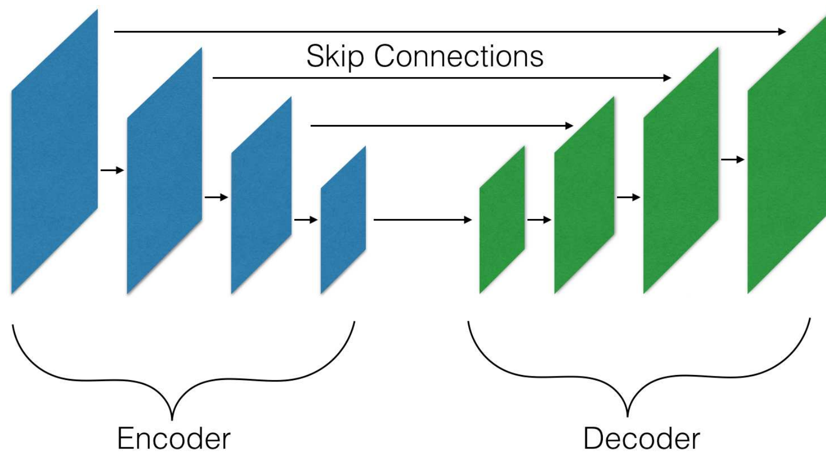


Figure 2.1: Encoder-Decoder Neural Network

Convolution Operation

the convolution operation takes in input a 3D volume (that cloud be the input image) of size (n_{in} x n_{in} x channels) and a set of 'k' filters (also called kernels) each one of size (f x f x channels). The output will be also a 3D volume (also called as output image or feature map) of size (n_{out} x n_{out} x k). The relationship between n_{in} and n_{out} is as follows:

$$\mathrm{n}_{\mathrm{out}} = \left\lfloor \frac{n_{\mathrm{in}} + 2p - k}{s} \right\rfloor + 1 \tag{2.1}$$

with k as convolution kernel size, p convolution padding size and s convolution stride size. Padding (p) represents the number of zeros padded around the original input, while stride (s) is the amount by which the kernel is shifted when sliding across the input image.

The figure 2.2 shows how a convolutional layer works as a two-step process.

Max Pooling Operation

Pooling has the purpose of shrinking the feature map in order to have fewer parameters in the network. In essence, we choose the highest pixel value from each block of the input feature map to create a pooled feature map. Strides and filter size are two crucial hyper-parameters for the this operation. The objective

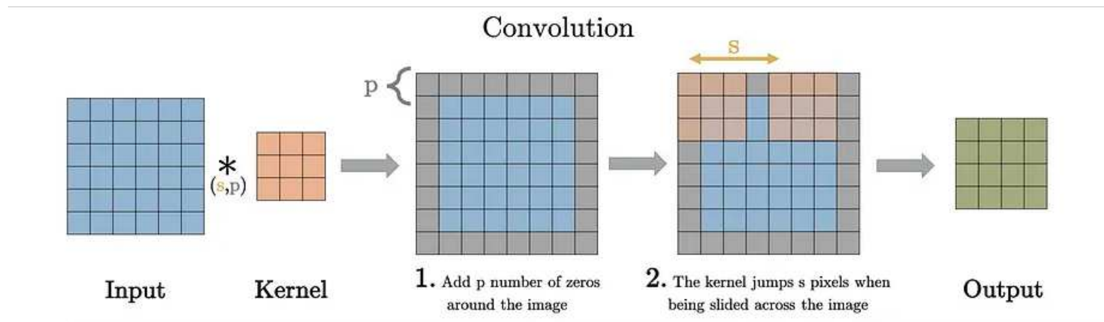


Figure 2.2: Convolution Operation

is to save only the most important features (highest valued pixels) from each region and discard the rest. By important, I mean the information that best characterizes the image's context. It is critical to note that both the convolution and the pooling operations reduce the size of the image. This is referred to as down sampling. The filters in the following layers will be able to see more context, that is, as we move deeper into the network, the size of the image decreases while the receptive field increases. However, as the number of channels/depth (number of filters applied) increases, it becomes easier to extract more complicated features from the image.

By down sampling the model gains a better understanding of "WHAT" is present in the image but it loses information about "WHERE" it is present.

In figure 2.3 the original image size is (4x4) and after pooling is reduced to (2x2).

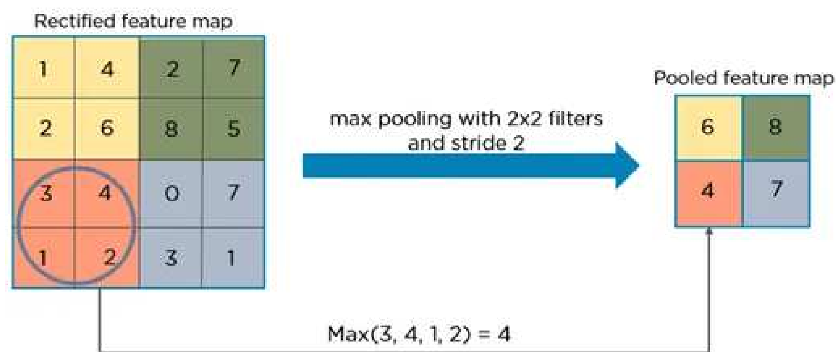


Figure 2.3: Max Pooling Operation

Transposed Convolution Operation

Image segmentation produces more than simply a class label or some bounding box parameters, it produces a whole high-resolution image in which all pixels are classified. In the case of segmentation, we require both "WHAT" and "WHERE" informations. To recover the "WHERE" information, the intermediate space of low-resolution features must be mapped into a final higher-resolution image.

There are numerous approaches in the literature for up sampling an image. Bi-linear interpolation, cubic interpolation, nearest neighbor interpolation, un-pooling, transposed convolution, and others are examples. However, for most state of the art networks, transposed convolution is the recommended method for up sampling an image.

Transposed convolution (also called deconvolution) is a method for doing up sampling of an image with learnable parameters, can be used if we want our network to learn how to up-sample as efficiently as possible, since we have weights that we learn through back-propagation. The padding and stride are the parameters that determine the transposed convolutional layer as they are for the conventional convolutional layer. These padding and stride values correspond to the hypothetical operations that were performed on the output to produce the input. In other words, if you take the output and perform a conventional convolution with defined stride and padding, the generated spatial dimension will be the same as the input.

Implementing a transposed convolutional layer is best described as a four-step procedure (see figure 2.4):

1. Calculate new parameters $z = s - 1$ and $p' = k - p - 1$
2. Insert z number of zeros between each row and column of the input. This increases the input size to $(2 * n_{\text{in}} - 1)x(2 * n_{\text{in}} - 1)$
3. Pad the modified input image with p' zeros.
4. Apply standard convolution to the image created in step 3 with a stride length of 1.

The relationship between n_{in} and n_{out} is as follows:

$$\mathrm{n}_{\text{out}} = (n_{\text{in}} - 1) * s + k - 2p \tag{2.2}$$

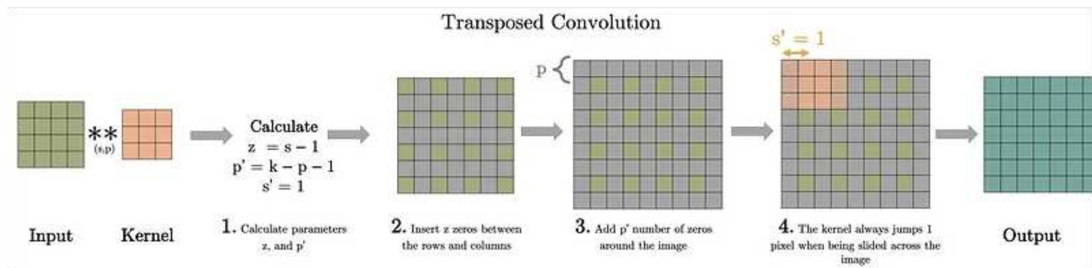


Figure 2.4: Transposed Convolution Operation

with k as convolution kernel size, p convolution padding size and s convolution stride size.

2.3.3 U-Net

Olaf Ronneberger et al. designed the *U-Net* architecture for Bio Medical Image Segmentation [25]. U-Net is a very popular end-to-end U-shaped encoder-decoder network for semantic segmentation which consists of four encoder blocks and four decoder blocks that are connected via a bridge. It has two main parts:

- The contraction path (also known as the encoder) is simply a standard stack of convolutional and max pooling layers that is used to capture the context in the image encoding the input image into feature representations at multiple different levels.
- The symmetric expanding path (also known as the decoder) is utilized to achieve precise localisation via transposed convolutions.

This is what gives the architecture a symmetric U-shape, hence the name U-Net. It only has Convolutional layers and none of them are Dense. U-Net structure as in the original paper is described in figure 2.5.

In the original paper the size of the input image is (572x572x3). In the left hand side (the contracting path) the main block of operations comprehends two 3x3 convolution layers, each followed by a rectified linear unit (ReLU), and 2x2 max pooling producing feature maps with half the spatial dimensions and double the number of filters. These operations are repeated leading to gradually reducing size of the input image while increasing gradually the depth, finally features of size (32x32x512) are obtained. This essentially means that the network learns the ”WHAT” information in the image but loses the ”WHERE” information.

In the right hand side (the expansion path) is repeatedly applied transposed convolutions along with regular convolutions, , each followed by a ReLU, that

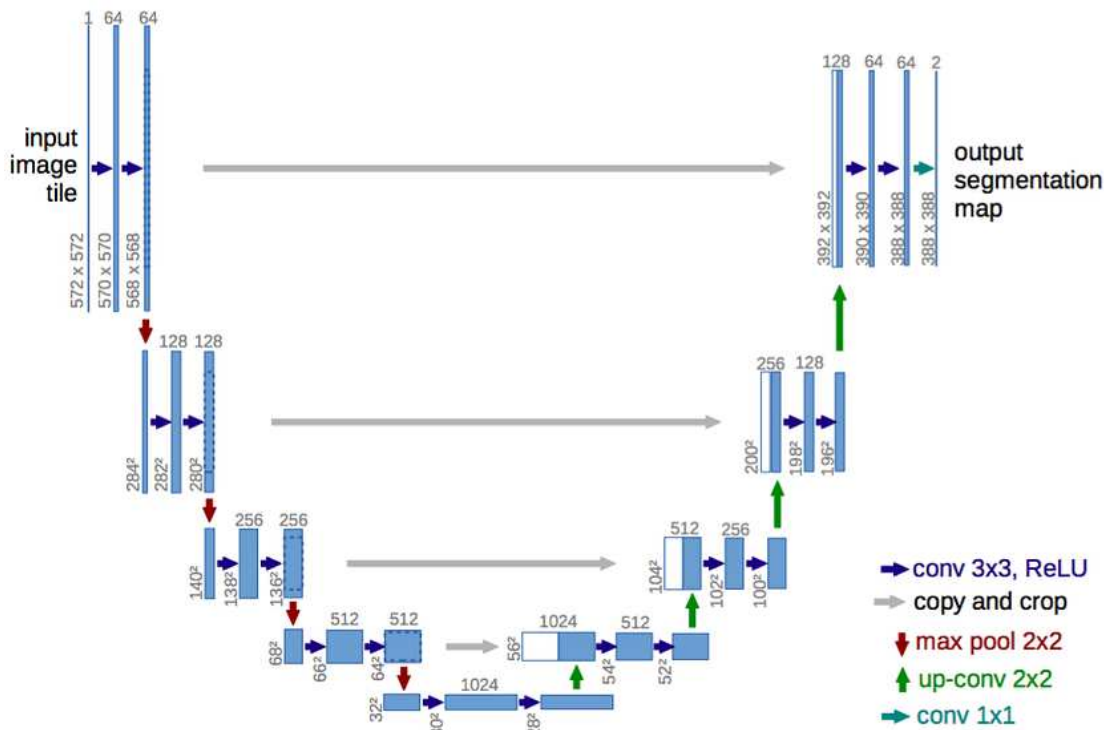


Figure 2.5: U-Net structure [25]

doubles the spatial dimensions and half the number of feature channels. In this part the size of the image gradually increases while the depth gradually decreases, starting from size (28x8x512) to output size (338x338x2). The Decoder intuitively recovers the "WHERE" information (exact location) by gradually applying up-sampling.

Since upsampling is a sparse operation we need a good prior from earlier stages to better represent the localization, for this purpose we use skip connections at each step of the decoder by concatenating the output of the transposed convolution layers with the feature maps from the Encoder at the same level.

We need to notice that the output dimensions (388 x 388) differ from the original input dimensions (572 x 572). If we wish to maintain the dimensions consistent, we have to exploit padded convolutions.

Image segmentation has been attempted by other neural networks in the past, but U-Net outperforms them by being less computationally expensive and minimizing information loss.

2.3.4 RefineNet

RefineNet is a generic multi-path refinement network that explicitly exploits all the information available along the down-sampling process to enable high-resolution prediction using long-range residual connections [14]. This allows the fine-grained features from previous convolutions to directly improve the deeper layers that capture high-level semantic data. Low-level visual details that are lost during the downsampling procedure in the convolution forward stage not always can be recovered by deconvolution processes leading to output an inaccurate prediction. For precise boundary or detail prediction, low-level visual data is essential.

RefineNet exploits Residual Net (ResNet) [19] network, pre-trained on ImageNet, as fundamental building block for semantic segmantation.

For achieving standard multi-path architecture, ResNet is divided into 4 main blocks according to the resolutions of the feature maps, and employ a 4-cascaded architecture with 4 RefineNet units, each of which directly connects to the output of one ResNet block as well as to the preceding RefineNet block in the cascade.

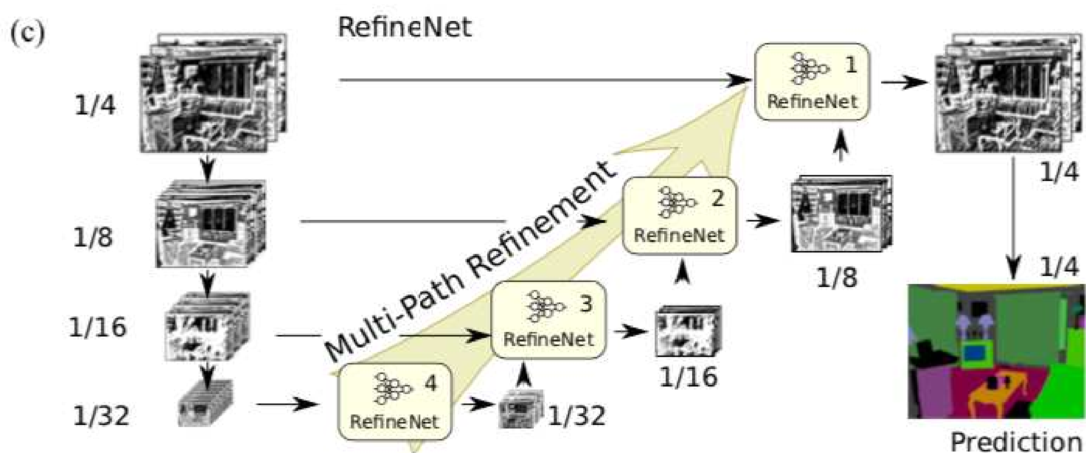


Figure 2.6: RefineNet Architecture [14]

As shown in figure 2.6 RefineNet does not require the maintenance of very big intermediate feature maps since it takes advantage of varying levels of detail at various stages of convolutions and merges them to provide a high-resolution output.

In practice, one convolutional layer is applied to each ResNet output to adjust the dimensionality.

Although each RefineNet Unit has the same basic structures, there are no ties

between its parameters, allowing for a more flexible adaption for different levels of detail.

2.3.5 High-Resolution Refine Net (HRRNet)

The network *HRRNet* proposed by Qiming Li and Chengcheng Chen [28] is divided into a rough segmentation module and a refinement module.

The first obtain rough segmentation results, exploiting DeepLabV3+ [21] architecture, the output of which is used as the input of the second. In the refinement module the input image's global context information is first gathered by a global procedure. In order to extract local information in a local process, each patch of the high-resolution image is processed separately. Both local and global processes use multiple refinement units (RU) and multi-scale inputs, such as RefineNet [14], to adaptively fuse the features of different scale feature maps and maximize the information obtained. Finally, the improved patches are stitched together based on the context information of the global process to produce the refined segmentation result of the entire high-resolution image.

In order to generate fine segmentation, the RU block in the original paper takes as input the original image and three segmentation images of the same size at various scales. RU module has an encoder-decoder architecture and uses a modified version of Deeplabv3+ as the segmentation network and ResNet50 for feature extraction. The decoder block output segmentation results at different scales: $1/8$, $1/4$ or equal to the original input spatial dimensions (OS).

Global process of the refinement module, which I took as inspiration for my solution, is discussed more in details below.

Global process

The refinement module takes the original image and the rough segmentation image of the original image size as input. The structure of the three-stage cascade RU is shown in Figure 2.7.

The arrows marked with "Up" represent upsampling operations since the input must have a consistent size in order to be analyzed by a neural network. The design goal of this input is to more effectively collect specific features and data at various scales in order to produce more accurate segmentation results.

Each RU is given as input the original image and three segmentation results and outputs masks of different sizes:

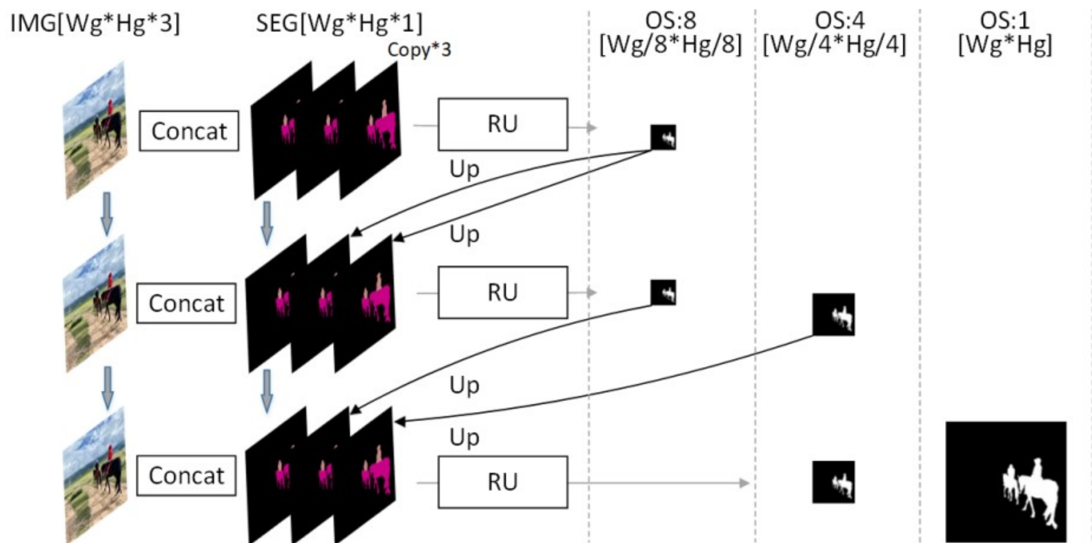


Figure 2.7: Global Process HRRNet [28]

- The first RU input consists of a concatenation of the original image and three times the rough segmentation result, upsampled to match the input size. In this case the input will be analyzed only from the first decoding block since only the low resolution segmentation result will return, with size $1/8$ OS.
- The input for the second RU is the same as the first replacing the last two rough segmentation results by the upsampled output segmentation map of OS8 obtained by the first RU. The output will consists in 2 segmentation maps in different scales: OS8 and OS4. In this case the input was analyzed by two blocks of the decoder.
- Finally, input image with rough segmentation result are concatenated with upsampled segmentation maps OS8 and OS4 obtained in previous RU, to obtain the input for the last RU block. In this case the set of masks are processed by the entire structure obtaining a mask of the same size as the input one.

The local process is very similar to the global one using only two cascaded RUs, exploiting the masks obtained in the global part. The overall loss function of the network is constructed by adding a combination of cross-entropy, L1, and L2 loss functions of different scales of the network.

2.4 Evaluation metrics

The effectiveness of a statistical or machine learning model is measured using evaluation metrics. Evaluating machine learning models or algorithms is

crucial for every project. To test a model a wide variety of evaluation metrics are available, the best known include classification accuracy. When we use the word accuracy, we usually indicate classification accuracy, which is the ratio of the number of accurate predictions to the total number of input samples. Testing our ML models involves using combination of different individual evaluation metrics.

It is crucial to analyze our model using a variety of evaluation metrics since a model may perform well when one metric is used, but poorly when another is applied. In order to make sure that our model is working properly and ideally, evaluation metrics are essential.

Before listing the adopted metrics, it is important to state the concept of confusion matrix. A confusion matrix produces a matrix that summarizes the model's overall performance. Each column of the matrix represents the instances in the real class while each row of the matrix represents the occurrences in the predicted class, or vice versa (figure 2.8).

Predicted Values \ Actual Values	Positive (1)	Negative (0)
Positive (1)	TP	FP
Negative (0)	FN	TN

Figure 2.8: Confusion Matrix

For tasks involving image segmentation, we predict a mask, representing the location where the object of interest is present. Since we assign 0 for background pixels and 1 for the object of interest, we are referring to binary segmentation, so:

- **TP (True Positive):** represents the number of true sample pixels that have been properly classified as sample (1).
- **FP (False Positive):** represents the number of background pixels being misclassified as sample (1)

- **FN (False Negative):** represents the number of sample pixels being misclassified as background (0)
- **TN (True Negative):** represents the number of background pixels that have been properly classified as background (0)

The metrics used for this project are listed below, all of them are based on the computation of a confusion matrix for a binary segmentation mask and are bounded between 0 (worst result) and 1 (better result).

Pixel Accuracy

Pixel Accuracy represents the ratio of correctly classified pixels in the final output mask. It consists of correct positive and negative predictions divided by the total number of predictions.

$$Accuracy = \frac{TP + TN}{TP + TN + FN + FP} \tag{2.3}$$

Even though it is simple to understand, this metric is far from ideal: in our specific case the background is the vast majority of the image and the object to predict represents only a tiny percentage of the whole amount of pixels. In our ground truth dataset white pixel, representing the objects of interest, account for about 10 percent. For example analyzing images of dimension 448x448 with a total of 200704 pixels, of these 20070 are white the rest belong to the background. If we now have the same model predict all background pixels, we will get an accuracy of: (200704-20070) / 200704 = 0.90 or 90%. That's a very high accuracy for a very bad model. This situation's problem is caused by the dataset's high imbalance. The data accuracy is not sufficient to describe the complete real situation when working with a class-imbalanced dataset, such as this one, where there is a noticeable disparity between the number of positive and negative labels.

Precision

Precision, as its name suggests, is a measure of how accurate our predictions are, in other words, it is how accurate the model is. Precision effectively describes the purity of our positive detections relative to the ground truth. This score is calculated dividing the number of true positive results by the number of all positive results.

$$Precision = \frac{TP}{TP + FP} \tag{2.4}$$

For instance, the precision is 1 if we correctly predict a single pixel. This strategy becomes ineffective when predicting very few pixels from the object class, in this case, the fewer predictions one makes, the lower the error.

Recall

Recall, also known as Sensitivity, effectively describes the completeness of our positive predictions relative to the ground truth. Recall score, is the number of true positive results divided by the number of all samples that should have been identified as positive.

$$Recall = \frac{TP}{TP + FN} \tag{2.5}$$

So, the more pixels we predict as the true class, the better will be the recall. If Precision can be compared to the quality of the model, Recall refers to quantity, or how many of the real labels were predicted.

Dice Coefficient (F1-Score)

It is one of the most widespread scores for performance measuring in computer vision and in Medical Image Segmentation. Dice coefficient is a "harmonious" balance between precision and recall, sometimes using only this metrics separately isn't enough. It evaluate the overlap between the predicted segmentation and the ground truth, additionally, it penalizes false positives, which are a frequent occurrence in datasets with highly class imbalance, like the one obtained for this project. As shown in figure 2.9 the Dice coefficent, also known as F1 score, is obtained multiplying by 2 the intersection between prediction and ground truth divided by the the total number of pixels.

Here is how it is defined when applied to two sets of pixels A and B, where A is the set of true pixels and B the set of predicted ones:

$$\text{Dice} = \frac{2\ X\ Area\ of\ overlap}{Total\ area} = \frac{2\ \text{x}\ \text{(Prediction} \cap \text{Ground truth)}}{\text{Prediction} + \text{Ground truth}}$$

Figure 2.9: Dice Coefficent

$$Dice(A, B) = \frac{2|A \cap B|}{|A| + |B|} \tag{2.6}$$

In terms of the confusion matrix, the metric can be rephrased in terms of true/false positives/negatives:

$$Dice = \frac{2TP}{2TP + FP + FN} \tag{2.7}$$

In the formula, we can see one limitation of the F1 score: true negatives aren't used directly.

The F1 score formula can be rephrased as a combination of Precision and Recall, as shown below:

$$Dice = \frac{2(Precision * Recall)}{Precision + Recall} \tag{2.8}$$

Jaccard Index (Intersection-over-Union)

Jaccard index, also known as Intersection-over-Union (IoU), is the area of the intersection over union of the predicted segmentation and the ground truth (figure 2.10).

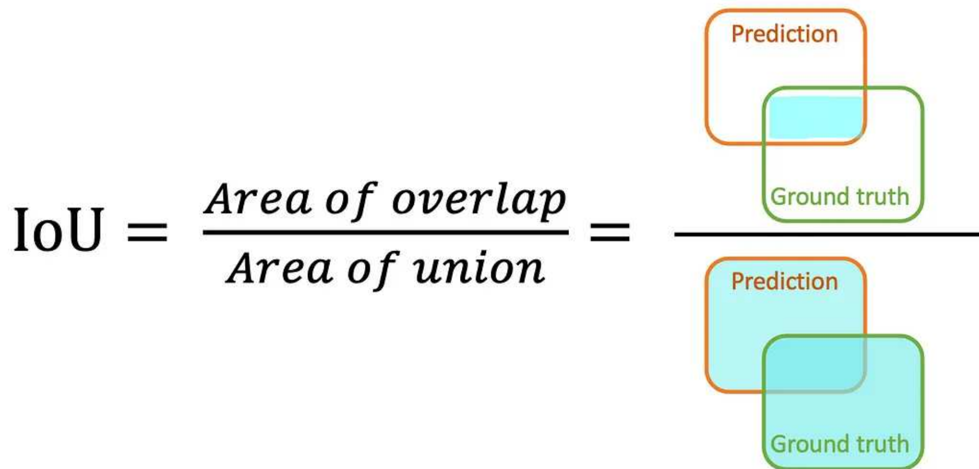


Figure 2.10: IoU Coefficent

In other words is essentially a method to quantify the percent overlap between the target mask and our prediction output. Considering for example A the ground truth ad B the predicted mask, the IoU metric measures the number of pixels common between the target A and prediction masks B divided by the total number of pixels present across both masks.

$$\mathrm{I}oU(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.9}$$

In terms of the confusion matrix:

$$\mathrm{I}oU = \frac{TP}{TP + FP + FN} \tag{2.10}$$

The metric mIoU is often used when dealing with multiclass problems, this can be calculated simply computing the IoU coefficient for each class and then take the mean. Often in problems like this where background is the major part, it is better to exclude the background class from the mIoU computation.

In this project the accuracy values are marginally analysed since the dataset has highly imbalanced classes between Regions of Interest (ROIs) and background. In other words, the ROIs in our images typically just take up a small portion of the total image, the remaining is background. Accuracy score includes

true negative results, thus leading to not meaningful high scoring. In contrast, dice coefficient and IoU are the most commonly used metrics for image segmentation because both metrics penalize false positives. However, choose between dice coefficient and IoU or vice versa depends on the task's individual use cases.

3

Main tool and libraries

I will briefly describe the primary tools that I have used to conduct the analysis and development in this section before getting into the specifics of implementation.

3.1 Programming languages

For the study of the optimal solution and the development of the model, I used *Python* [13], versions 3.9, one of the most popular programming languages in the field of artificial intelligence, especially in machine learning and image processing. There are many programming languages utilized in the building of AI models, It is the best option for processing massive datasets and creating complicated models thanks to its extensive collection of open-source modules and tools.

For the integration part in SLC^3, Sisma's software, I used the .NET framework [12] and in particular the C# [24] language. .NET is a free, cross-platform, open-source developer platform for building many types of applications. Apps and libraries are compiled from source code and a project file, using an integrated development environment (IDE) such as Visual Studio. C# language has its roots in the C family, It is a modern, object-oriented, and type-safe programming language.

3.2 Main Python libraries

One of the key benefits of adopting Python is the huge number of open-source libraries. In this section I'll outline the key Python libraries utilized in this project.

Numpy

Numpy [15] is a popular open-source package that Python programmers use for numerical computation and data analysis. It provides powerful tools for working with matrices, vectors and other data structures. For engineers and data scientists who work with massive amounts of data, Numpy is a crucial tool.

The key benefits of Numpy are interoperability with other libraries and tools, like as Pandas and Matplotlib, making it simple to use, and broadcasting, a useful feature that permits operations between elements in arrays with diverse sizes and forms.

Matplotlib

The most popular Python library for creating static, animated, and interactive visualizations is *Matplotlib* [16]. It allows users to make a wide variety of plots, including line plots, scatter plots, bar graphs, histograms, and more. It also enables users to display RGB and grayscale images.

Scikit-learn

Scikit-learn [9] library is used for machine learning tasks, It offers easy-to-use tools for data mining and data analysis. It was mostly used in this thesis for splitting the dataset into training, validation, and test, and also to calculate the evaluation metrics on the final results.

Albumentations

Albumentations is a Python library for fast and flexible image augmentations [1]. It is a computer vision tool that boosts the performance of deep convolutional neural networks. It provides a clear but effective image augmentation interface for a number of computer vision tasks, such as object classification,

segmentation, and detection. Albumentations implements a wide range of image transform operations that are optimized for performance. In this thesis it was used above all for the creation of synthetic data.

3.3 OpenCV and Emgu CV

OpenCV (Open Source Computer Vision Library) [18] is an open source computer vision and machine learning software library. OpenCV was developed to facilitate the use of machine perception in commercial products and to provide a common infrastructure for computer vision applications. More than 2500 optimized algorithms are available in the library, including a wide range of both traditional and state-of-the-art computer vision and machine learning techniques. It supports Windows, Linux, Android, and Mac OS, it is natively written in C++ but provides interfaces also for Python, Java, and MATLAB.

Emgu CV [7] is a cross platform .Net wrapper to the OpenCV image processing library. It allows call OpenCV functions from .NET compatible languages like C#. This wrapper is compiled by Visual Studio and it can run on Windows, Linux, Mac OS, iOS and Android.

I have extensively used OpenCV and Emgu CV, in Python and C# respectively, for numerous image operations including interpolations, scaling, extrapolation of edge maps and division into patches.

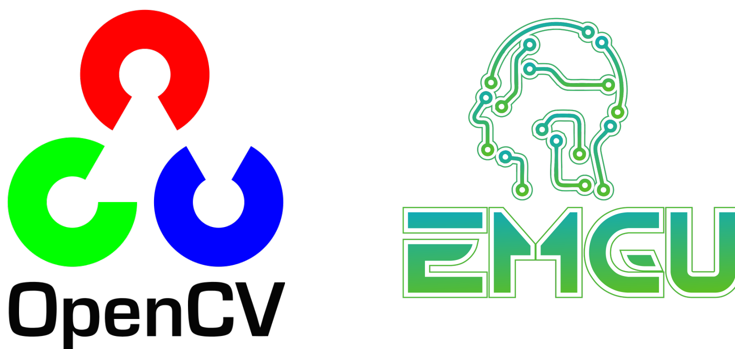


Figure 3.1: OpenCV and EmguCV logo.

3.4 TensorFlow and TensorFlow.NET

TensorFlow [2] is another open-source machine learning framework developed by Google Brain commonly used. It offers a full range of tools for creating and training machine learning models, including neural networks, decision trees and regression models. The native APIs in the library are developed in Python, C, Java, Go and Rust languages. However, the use of the Python language is recommended because it allows you to implement and call the TensorFlow libraries more quickly. TensorFlow therefore allows you to specify the architecture of a model, including the number of layers, the type of activation function and the optimizer to use, and then provides the tools to train the model on a dataset. *Keras* [5] is the high-level API of the TensorFlow platform.

TensorFlow.NET [4] is a library that provides a .NET standard binding for TensorFlow, allowing developers to design, train and implement neural networks in the .NET environment. This library allowed me to import the model, already trained in Python, into C# and integrate it into the Sisma application.

Figure 3.2: Tensorflow logo.

3.5 Tools

It is essential to have access to a variety of software tools and services that can simplify the project development process. In this section I will review some of the main software services and development tools used to create the complete project.

Anaconda and Jupyter Notebook

Anaconda [17] is a distribution of the Python and R programming languages for scientific computing, that aims to simplify package management and deployment. Over 250 data science packages are included in the distribution, and over 7,500 additional open source packages suitable for Windows, Linux, and macOS can be installed. One of these includes Jupyter Notebook.

Jupyter Notebook [11] is an open source web application that allows you to create and share interactive text documents, containing objects such as equations, graphs and executable source code. Jupyter notebooks are composed of a number of "cells" that are organized in a linear fashion. Any cell can contain text or a code.

Microsoft Visual Studio

Microsoft Visual Studio (or more commonly Visual Studio) [8] is an integrated development environment (IDE) developed by Microsoft. Visual Studio support various languages, including C#, Visual Basic .Net, C++, Java and Javascript, and the creation of projects for various platforms, including Mobile and Console. Visual Studio integrates IntelliSense technology which allows you to correct any syntactic errors, and even some logical ones, without compiling the application. Unlike classic compilers, the one available with the .NET Framework converts the source code (Visual Basic, C#, etc.) into IL (Intermediate Language) code.

Git

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency [22]. An impressive number of software projects rely on Git for version control, including commercial and open source projects. Thanks to Git, programmers can record every change that is made to the code over time, compare different versions of the same file, collaborate simultaneously on the same program to implement new features without interfering with the work of other team members.

GIMP

GIMP [30] is a cross-platform image editor available for GNU/Linux, macOS, Windows and more operating systems. GIMP a is free open-source software that offers many features, such as high-quality photo manipulation, original artwork creation and graphic design. Thanks to its selection tools it allowed me to obtain the ground truth segmentation masks in a simpler and faster way.

4

Data Exploration and Preprocessing for Improving the Performance of the AI Model

In this chapter I will introduce the structure and properties of the dataset used in this project and the preprocessing operations performed in order to improve prediction results. One of the most crucial elements affecting the performance of the AI models is the quality of the data, for this purpose preprocessing operations can significantly impact the quality of the model's output. We will have the input data prepared for the development/training phase by the conclusion of this chapter.

4.1 Dataset Overview

All images used for training and testing the DL model were captured manually through the use of the vision system of Sisma's machinery present in the company and the SLC^3 application for laser marking. I have been provided by the company with samples of shapes similar to those used by customers of different sizes and geometries. The objects supplied to me were photographed with dark backgrounds in order to highlight the contrast or positioned directly in the gripper of the mandrel ready for processing. From about 50 objects I obtained about 700 photos by varying positions and exposure times of the camera. Depending on the sample under examination, the generated images di-

mensions differ; they normally span from the smallest (1620x1620) to the largest (10,000x10,000).

Training a segmentation model requires a dataset which contains images segmentation masks as ground truth of the same size as the original photos, they express the class to which each pixel of the respective photo belongs. The binary masks representing the ground truth were obtained using Gimp a free and open source image editor application capable of facilitating the precise manual segmentation procedure. The process of capture and manual segmentation of the images obtained, also given the considerable resolution, it is quite time consuming.

The more images dataset contains, the better the model will train, because it will see more examples during training process, excellent results are achieved with 5000 or more images varied as possible. Data augmentation techniques and generation of synthetic data were applied with the aim of expanding the set of images for training. It must be taken into account that the captured images often had reflections and shadows caused by sometimes unavoidable external lights.

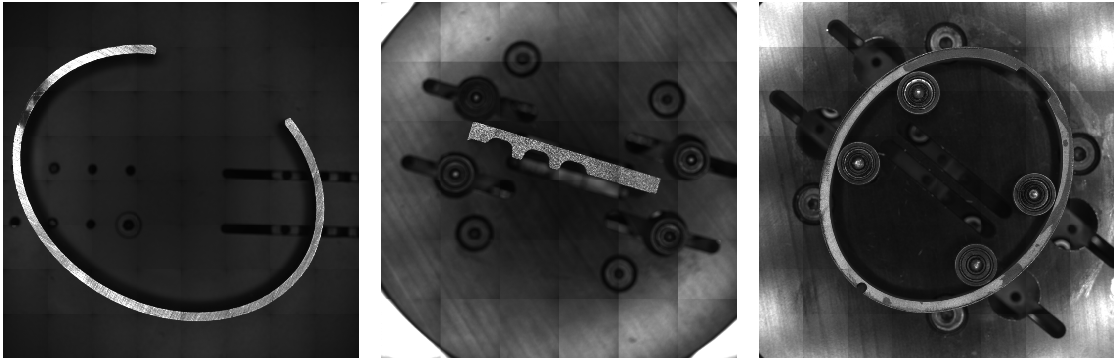


Figure 4.1: Example Data

Label distribution

The ground truth is formed by binary masks, the value 0 represents a position occupied by the background while 1 the object of interest. By analyzing the dataset and the ground truth, we can state that the distribution of binary values is highly unbalanced, with background values that prevail over ROI values by a percentage of about 90 of the total number of pixels. This issue leads us to focus on metrics and loss functions more suited to the type of data we have.

4.1.1 Data Augmentation

In my thesis I made use of data augmentation techniques since the set of images obtained is limited. Overfitting is a term used in machine learning (ML) to describe the circumstance in which a model does not generalize well from training data to unseen data. It is one of the most difficult challenges in applied machine learning. Acquiring and classifying additional data will produce superior outcomes in many circumstances, but it is typically time-consuming and costly in terms of labour. That is where data augmentation (DA) comes in.

Data augmentation is a method for producing altered data from existing data in order to artificially increase the size of a training set. Using DA is a useful technique if you want to avoid overfitting, if the initial dataset is too small to train on, or if you want to squeeze more performance out of your model.

For this thesis I made use of the following DA techniques:

- **Geometric transformation:** include for example random flip, rotation, shear, resize, zoom.
- **Color space transformations:** random change greyscale values (altering brightness and contrast).
- **Linear filters:** sharpen or blur an image.

Unfortunately the biases in the original dataset persist in the augmented data and finding an effective approach for data augmentation can be difficult.

4.1.2 Synthetic Dataset

In computer vision, the term "synthetic data" refers to images that are created by algorithms rather than acquired by a camera. These photos are typically used to train artificial intelligence (AI) models. Using synthetic data offers various advantages over using real data:

- It is easier to obtain a large amount of synthetic data rather than real data.
- Synthetic data can be generated with specific properties that are difficult to find in real data, for example it is possible to create images of partially hidden objects (helpful for developing object identification models). In this case the images generated will not have the problem of having to manage reflections and shadows.
- For instance, the color of the objects in the photos can be altered to systematically change the generated data. This is useful for developing models that must be resistant to changes in appearance.

- Synthetic data can be generated with labels, in this case segmentation masks are generated automatically.

When compared to a human procedure, an automated process allows us to build datasets significantly faster.

For this purpose I have considered 2 different approaches: in the first case I used about 50 cropped photos of real samples in various positions (especially in the jewelry sector) as objects of interest, in the second case I used random abstract shapes to which grayscale textures were applied. Since we are working with mostly metallic samples, we first estimate the gray value distribution of the samples in the real photos, then this distribution is applied to the pixels of the synthetic images to obtain an image with a real metal-like appearance (see figure 4.2).

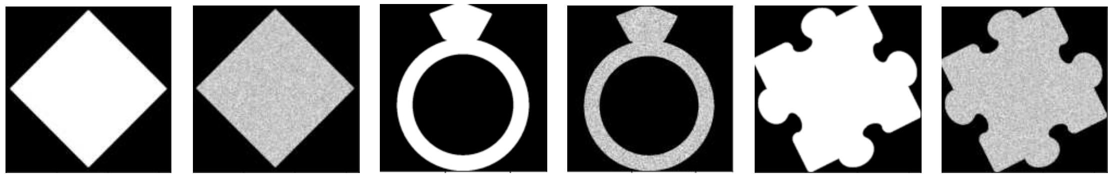


Figure 4.2: Synthetic Shapes

More than 40 background images obtained from Sisma laser systems and about 100 cropped synthetic shapes were used in the creation of the synthetic dataset.

Here are the steps I followed to generate a synthetic scene:

1. Randomly choose a background image
2. Randomly pick an object of interest
3. Synthetic cropped sample is randomly edited in order to obtain different shapes exploiting *Albumentations* [1], a powerful python library that allows transformations (like rotation, flip, blur, change colors, distortion contrast and brightness) both to the input image and the output mask. Thanks to this step we are able to distort the shape taken as input and obtain different shapes at each iteration.
4. Edited object of interest is added to background image in a random position and It is retrieved relative mask composition (object area is filled with white color, background with black).
5. Since we want to have dataset with backgrounds as varied as possible I finally add some Gaussian random background noise.

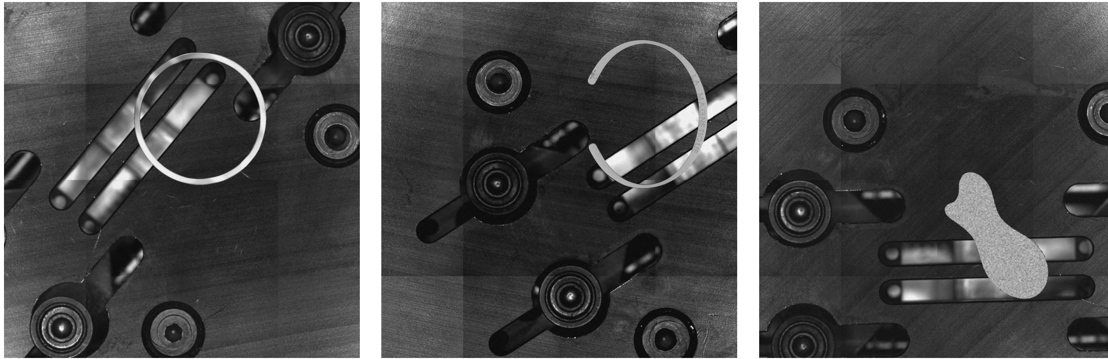


Figure 4.3: Generated Synthetic Data Examples

In figure 4.3 we can see some of the synthetic final scenes obtained.

Working with real images is always better than using artificial datasets, the background noise generated by a camera is always complicated to emulate, furthermore, unlike real data, these do not suffer from shadow and reflection problems.

4.2 Preprocessing and Edge detectors

Deep learning requires a lot of processing power. There are more than one hundred thousand parameters in even a simple neural network model, training large neural networks is highly expensive in terms of computer resources. As a result, training a network using original resolution images is prohibitive in terms of available hardware. I decided to reduce the size of the input images to (448x448), a fair compromise to be able to manage all the images in memory and train the developed model within acceptable times. Furthermore, although not mandatory, most standard CNNs are designed for a fixed-size input, because they contain elements of their architecture that don't generalize well to other sizes.

To help the network recognize the geometries and obtain the most precise result possible for each image, the edge maps were obtained, which were linked to the downsampled image and then given as input to the deep learning model. This choice proved successful, improving the final results.

By edge in an image we refer to a place of rapid change in the image intensity function. Edge detection is a widely used and popular image processing technique required for most computer vision applications, is used to identify the boundaries (edges) of objects or regions within an image. While maintaining

the image's structure, it lowers the amount of noise and the number of features in the image.

We must identify image discontinuities in order to detect edges, and computing the derivative in image intensities will help us do this.

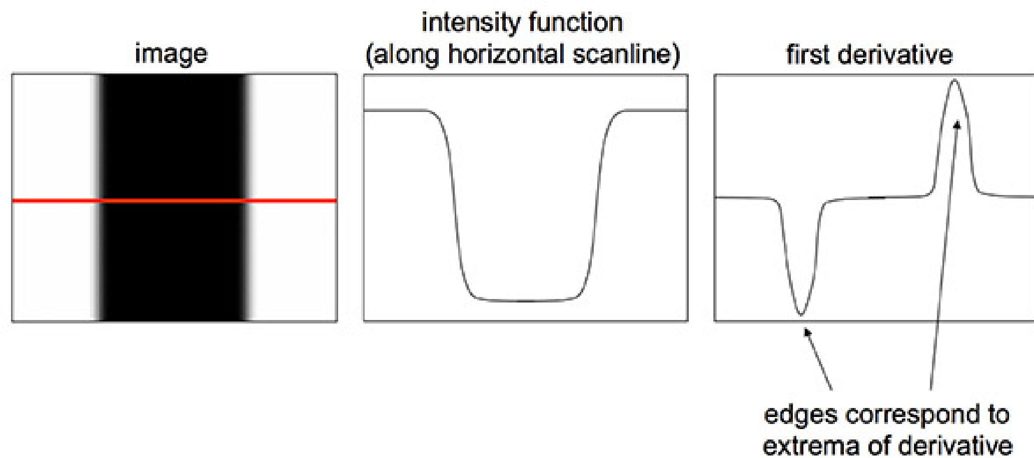


Figure 4.4: Image Derivatives.

As shown in the figure above 4.4, the edges corresponds to the derivative peak values. Derivates, however, are also impacted by noise, hence, it is advised to first smooth the image before taking the derivative.

Fortunately, OpenCV [18] package allows to perform edge detection with different techniques, the most important, considered for this thesis, are described below.

Finally a normalization procedure is applied to all the samples of the dataset in order to obtain pixel values between 0 and 1. This will help training the neural networks speeding up convergence.

4.2.1 Sobel Edge Detection

One of the most used edge detection methods is *Sobel* [29] Edge Detection, it detects edges marked by sudden changes in pixel intensity in a grayscale image. By analyzing the first derivative of the intensity function, the increase in intensity becomes much more apparent, so edges are detected in areas where the gradient is greater than a particular threshold value. Sobel method use one kernel to detect sudden changes in pixel intensity in the X direction and another in the Y direction, therefore approximating the first derivative using 2 different

(3x3) kernels.

$$A = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \quad B = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Figure 4.5: X and Y direction Kernel, respectively

The intensity gradient in the x and y directions are denoted as G_x and G_y, X and Y kernels defined above as A and B, respectively. G_x and G_y are obtained through convolution operator with the original input image I:

$$\mathrm{G}_x = A \circledast I \tag{4.1}$$

$$\mathrm{G}_y = B \circledast I \tag{4.2}$$

The final approximation of the gradient magnitude G can be computed as:

$$\mathrm{G} = \sqrt{G_x^2 + G_y^2} \tag{4.3}$$

And the orientation of the gradient can be approximated as:

$$\Theta = tan^{-1}(\frac{G_y}{G_x}) \tag{4.4}$$

To compute edge maps with less noise some filters like the Gaussian are often performed on original images resulting in blurred images. It must be taken into consideration that this method suffers from poor localization: It implies that multiple edges could be extrapolated where there should only be one.

4.2.2 Canny Edge Detection

Because it is so reliable and adaptable, *Canny* [3] Edge Detection is one of the most widely used edge-detection techniques today. It is a multi-stage algorithm made up of 4 different stages:

1. **Noise Reduction**: Since edge detection is susceptible to noise in the image noise must be reduced. A 5x5 Gaussian blur filter is mainly utilized in Canny Edge Detection to remove or minimize unwanted information that could result in undesired edges.
2. **Calculate Intensity Gradient**: The image is filtered using a Sobel kernel both horizontally and vertically after it has been smoothed. The intensity gradient magnitude (G) and direction (Θ) for each pixel are then determined as previously explained.
3. **Edges Suppression**: Non-maximum suppression of edges is a method that the algorithm in this phase use to remove undesirable pixels that correspond to unwanted edges. Each pixel is compared to its neighbors in both the positive and negative gradient directions: the magnitude of the analyzed pixel remains unaffected if it is greater than of its surrounding pixels. Otherwise, the current pixel's magnitude is set to zero. As result we will get a binary image with "thin edges".
4. **Hysteresis Thresholding**: The gradient magnitudes are compared with two threshold values, one smaller than the other:
   - If the gradient magnitude value exceeds the greater threshold value the pixels are associated with 'solid' edges and are included in the final edge map.
   - If the gradient magnitude values are less than the smaller threshold value, the pixels are suppressed and left out of the final edge map.
   - Pixels whose gradient magnitude lies between these two thresholds are classified as "weak", this pixels are represented in the final edge map if they are related to those corresponding to solid edges.

See figure 4.6 [26].

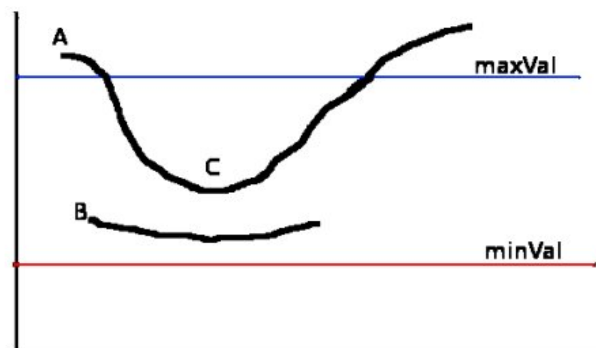


Figure 4.6: Hysteresis Threshold [26].

The edge A is a "sure-edge" because it is above the maximum value. The whole curve is obtained even if point C is below maxVal since it connects to edge A. Although edge B is in the same region as edge C and is above minVal, it is not related to any other "sure-edges" and is therefore discarded.

4.2.3 Final Considerations

The Sobel operator's key benefits are that it is less complicated and takes less time, the edges, though, are rough. On the other hand, the non-maxima suppression and the hysteresis threshold used in the Canny technique result in finer edges. The Canny algorithm has the drawback of being more complex and less time-efficient than Sobel.
In my specific case I used Sobel for the images analyzed by the Rough model, that works with low resolution images, given that TensorFlow provides a method for calculating gradients in the pipeline dedicated to the creation of TensorFlow datasets. This operation allowed me to save memory and upload more images for training. For the refinement module, however, we need the concatenated edge map to be as precise as possible, which is why Canny method is used in this case.
OpenCV library [18] provides useful function that performs both Sobel and Canny edge detection in sigle functions. In the Python code example below, the `Sobel()` and `Canny()` function implements the methodologies described in previous chapters.

```
1 import cv2
2
3 sobelxy_edgemap = cv2.Sobel(image, ddepth=cv2.CV_64F, dx=1, dy=1,
     ksize=3)
4 canny_edgemap = cv2.Canny(image, threshold1=minVal, threshold2=maxVal
     )
```

Listing 4.1: OpenCV Sobel and Canny Edge Detection

For the Sobel function it is important to specify:

- The parameter `ddepth`, that means the desired depth of the destination image in this case we receive a single channel grayscale image with float pixel values.
- I set `dx = 1` and `dy = 1` in order to compute the first derivative in both directions.
- We can also change the size of the original 3x3 kernel by setting the argument `ksize`

For the second, OpenCV takes care of all the technical details while I provide the two thresholds that the Canny Edge Detection method uses.

4.3 Train, validation and Test split

The division of the dataset in train, validation, and test is essential for the creation of strong and reliable machine learning models. By ensuring that the datasets used to train the model and to evaluate it are separate, you are better able to evaluate the performance of the model.
Given that our dataset is made up of multiple images representing the same sample, with different positions or exposure times, when dividing the samples into the various sets, care must be taken to assign a single set to a given sample in order to obtain unbiased evaluations.
The following are the key characteristics of each set:

- **Train set:** The training set is the portion of the dataset reserved to fit the model. It consists of labeled examples that the model uses to learn the underlying patterns and relationships between input features and target outputs. The model sees and adjusts its parameters based on the training data in order to minimize a loss function and enhance its predictive capabilities. The training set must be large enough to avoid overfitting. To maximize model performance, I considered about 80% of total samples.
- **Validation set:** During training process, the validation set is utilized to fine-tune a machine learning model. Thanks to the validation set we can know the ability of a trained model to generalize to new data. Potential problems like overfitting are identified by this assessment. It is also essential for hyperparameter tuning. Hyperparameters such as learning rate or regularization are settings that control the behavior and also the training of a ML model. We can determine the ideal combination of hyperparameters that produces the best results by experimenting with different hyperparameter values, training the model on the training set, and evaluating its performance using the validation set. About 10% of total samples are used for validation.
- **Test set:** The test set is used to evaluate the final performance of the trained model. It serves as an unbiased measure of how well the model can generalize to unseen data. The test set should be representative of the real-world data the model will encounter. About 10% of total samples are used for testing.

The following table shows the number of samples assigned to each set. Images generated through data augmentation and synthetic data were not considered.

Training Set	**Validation Set**	**Test Set**
568	64	61

Table 4.1: Number of samples for each set.

5

Image Segmentation Model Architecture

This section provides an in-depth explanation of the machine learning model that was designed and developed which led to the optimal solution. As shown in Figure 5.1, the network model proposed in this thesis is composed by two main components: a rough segmentation module and a refinement module, like the *HRRNet* [28] network.

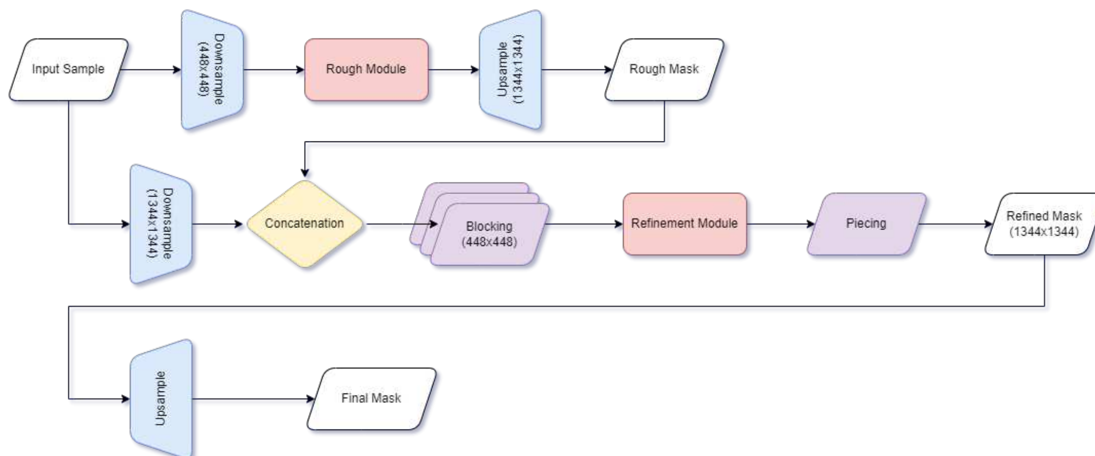


Figure 5.1: Whole Architecture

For the rough segmentation module I got inspired from *U-Net* [25] architecture and processes the input image in low resolution. This first module aims to obtain a first approximate segmentation mask at low resolution. The high-resolution

input image, together with the edge map, is then first reduced to the dimensions 448x448x2 in order to be analyzed by the first network, obtaining a binary mask with spatial dimensions equal to the input (448x448). Subsequently the result obtained from the previous module is interpolated with a bilinear method, until a mask measuring 1344x1344 is obtained.
In the refinement module, the approximate segmentation result, the original image and its edge map are taken into consideration together. The input is divided into 9 patches, each patch is segmented separately and refined to extract local details information of the image. This strategy is similar to the one presented by Anton Milan and Guosheng Lin [14] which uses multiscale inputs to maximize the information obtained during the previous layers.
The refined patches are then pieced together to generate a high-resolution refined segmentation result.
Finally, the mask needs to be interpolated again until the dimensions match the original input ones: various interpolation techniques have been tried, I have also developed a method that takes advantage of information from the edges and assigns pixel intensity values accordingly .
Below both modules are described in detail together with the evaluation metrics calculated on the test set.

5.1 Rough Module

For the development of this architecture I took inspiration from the U-Net architecture presented in 2015 by Olaf Ronneberger, Philipp Fischer and Thomas Brox [25].
The original structure was used originally in the medical field, specifically for tumor detection, but over the years it has been exploited in multiple fields achieving excellent results.
The model's goal as an image segmentation tool is to assign each pixel to one of the output classes, producing as output a single channel mask. The output mask will have the same number of columns and rows as the output image, each element will be between 0 and 1. These values can be interpreted as the probability that the pixels of the original image, present in the same position, belong to class 1 (region of interest). On the other hand, the probability that a given pixel belongs to class 0 (background) can be calculated as $1 - predicted\ value$.
It is important to underline that this network doesn't have any fully connected

layer. The developed architecture is described by the figure 5.2.

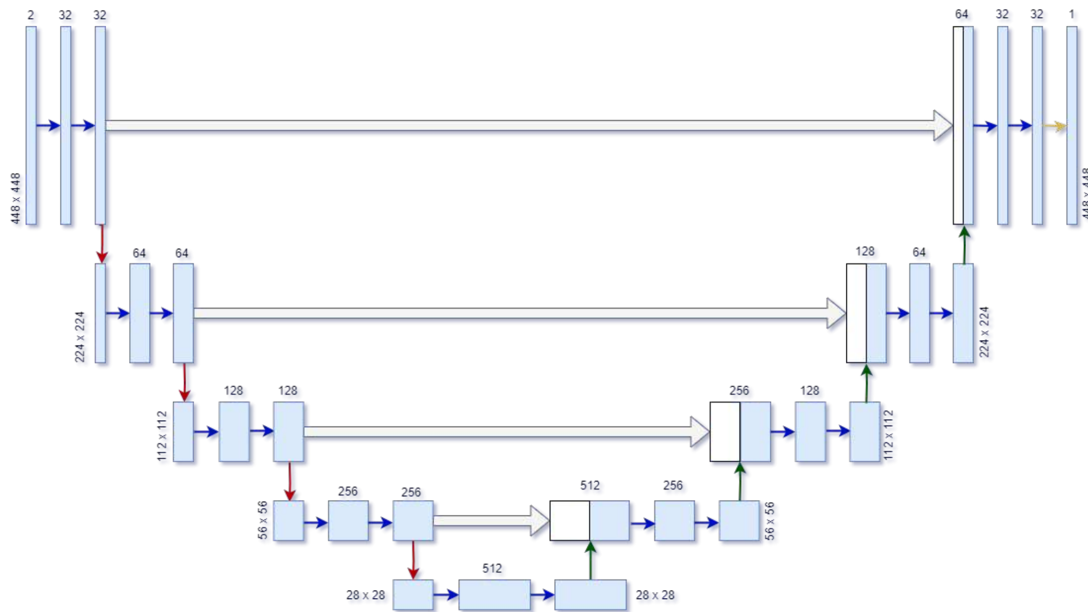


Figure 5.2: Rough Module Architecture

As in the original paper the architecture is composed of two main parts: encoder and decoder. The contracting path, or encoder, reduces (down-sample) the input image into a feature map, and through pooling layers extracts the key elements. The second path is the symmetric expansion path, also known as the decoder, it allows for accurate localisation through transposed convolution. At each step transposed convolution double the size of feature maps while the number of channels is halved. The decoder amplifies (up sample) the feature map to obtain a single channel image, using the deconvolution layers, also through skip connection techniques.
Each operation represented by the arrows is described below:

- Each blue arrow applies a convolutional layer with a kernel of size 3x3. Padding values were placed in the outlines of the maps before the convolution operation in order to avoid map downsampling. Furthermore, batch normalization layers and Leaky ReLU activation functions are applied after each convolutional layer.

- The red arrows represent max pooling operators. This calculates the maximum value applying a filter to non-overlapping subregions of size 2x2 and obtains a downsampled feature map with half sizes. It is commonly used to reduce the number of parameters and increase computational speed but also because it allows you to analyze features at different scales.

- Transposed convolution layers are represented by green arrows. This phase involves two steps: add padding to each pixel in input feature map, then apply convolution with kernel size 3x3 with stride 2 in order to match same level encoder feature maps sizes.
- The gray connections between encoder and decoder maps refer to concatenation operations. Skip Connections copies the image matrix from the earlier layers and uses it as a part of the later layers. This enables the model to preserve information from a richer matrix and prevent information loss.
- The last operation represented in yellow indicates a last convolutional layer, with kernel size equal to 1, that returns a single channel. In this case the activation function performed is the sigmoid in order to obtain values between 0 and 1.

The developed code is shown below: all layers used are standard layers provided by *Keras* [5]. To build a model layers need to be assembled specifying the calculation order from the input to the output. Since convolution operations are frequently performed in both contracting and expansive paths, to avoid code repetition I grouped them into a single function `conv2d_block()`.

```
import tensorflow as tf

def conv2d_block(input_tensor, n_filters):

    # first layer
    x = tf.keras.layers.Conv2D(filters = n_filters, kernel_size = (3,
     3), kernel_initializer = 'he_normal', padding = 'same')(
    input_tensor)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.LeakyReLU(alpha=0.01)(x)

    # second layer
    x = tf.keras.layers.Conv2D(filters = n_filters, kernel_size = (3,
     3), kernel_initializer = 'he_normal', padding = 'same')(x)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.LeakyReLU(alpha=0.01)(x)

    return x
```

Listing 5.1: Convolutional Block

This function receives the current layer and the depth of the feature map it should produce. 2D convolutional layers use 3x3 kernel sizes, adding padding values to the edges of the maps in order to avoid downsampling. Initial random weights are set following a truncated normal distribution centered on 0.

Batch normalization layers apply a transformation that maintains the mean output close to 0 and the output standard deviation close to 1. This has the effect of stabilizing the learning process and reducing the number of training epochs required to train deep networks.
Leaky Rectified Linear Units, also known as Leaky ReLUs, are activation functions that are based on ReLU but have a small slope for negative values instead of a flat slope. The slope coefficient is not learned during training but is a constant entered as input.
The following code shows the entire structure of the rough module and how it is initialized.

```
def get_rough_module(dropout = 0.1):
    input_layer = tf.keras.layers.Input(shape=[448, 448, 2])

    # Contracting Path
    c1 = conv2d_block(input_layer, 32)
    p1 = tf.keras.layers.MaxPooling2D((2, 2))(c1)
    p1 = tf.keras.layers.Dropout(dropout)(p1)

    c2 = conv2d_block(p1, 64)
    p2 = tf.keras.layers.MaxPooling2D((2, 2))(c2)
    p2 = tf.keras.layers.Dropout(dropout)(p2)

    c3 = conv2d_block(p2, 128)
    p3 = tf.keras.layers.MaxPooling2D((2, 2))(c3)
    p3 = tf.keras.layers.Dropout(dropout)(p3)

    c4 = conv2d_block(p3, 256)
    p4 = tf.keras.layers.MaxPooling2D((2, 2))(c4)
    p4 = tf.keras.layers.Dropout(dropout)(p4)

    c5 = conv2d_block(p4, n_filters = 512)

    # Expansive Path
    u6 = tf.keras.layers.Conv2DTranspose(256, (3, 3), strides = (2,
   2), padding = 'same')(c5)
    u6 = tf.keras.layers.concatenate([u6, c4])
    u6 = tf.keras.layers.Dropout(dropout)(u6)
    c6 = conv2d_block(u6, 256)

    u7 = tf.keras.layers.Conv2DTranspose(128, (3, 3), strides = (2,
   2), padding = 'same')(c6)
```

```
30    u7 = tf.keras.layers.concatenate([u7, c3])
31    u7 = tf.keras.layers.Dropout(dropout)(u7)
32    c7 = conv2d_block(u7, 128)
33
34    u8 = tf.keras.layers.Conv2DTranspose(64, (3, 3), strides = (2, 2)
      , padding = 'same')(c7)
35    u8 = tf.keras.layers.concatenate([u8, c2])
36    u8 = tf.keras.layers.Dropout(dropout)(u8)
37    c8 = conv2d_block(u8, 64)
38
39    u9 = tf.keras.layers.Conv2DTranspose(32, (3, 3), strides = (2, 2)
      , padding = 'same')(c8)
40    u9 = tf.keras.layers.concatenate([u9, c1])
41    u9 = tf.keras.layers.Dropout(dropout)(u9)
42    c9 = conv2d_block(u9, 32)
43
44    output_layer = tf.keras.layers.Conv2D(1, (1, 1), activation='
     sigmoid')(c9)
45    model = tf.keras.Model(inputs=[input_layer], outputs=[
     output_layer])
46    return model
```

Listing 5.2: Solution Code

The model currently takes as input the image together with its edge map reduced to 448x448 spatial dimensions.
The function `tf.keras.layers.concatenate ([l1 , l2, ... ])` takes as input a list of layers, all of the same shape except for the concatenation axis, and returns a single tensor that is the concatenation of all inputs.
In order to avoid overfitting, the Dropout layer randomly sets input units to 0 with a frequency of a given rate at each step during training. Inputs not set to 0 are scaled up by $1/(1-rate)$ such that the sum over all inputs is unchanged [32].

5.1.1 Model Training

It is necessary to compile the model before training. When compiling the model is important to specify:

- **Loss Function**: a method of measuring how far predictions are from the intended result. With neural networks, we often aim to reduce the error. The cost or loss function has a significant task since it must accurately distill all the model's components into a single number such that improvements in that number are a sign of a better model. Choosing the right loss function

can be a challenging problem since it must capture the characteristics of the issue and be motivated by concerns that are important to the project and data considered. Importantly, the choice of loss function is directly related to the activation function used in the output layer.

- **Optimizer Function**: The algorithm performed to adjust network internal trainable parameters. Optimizers minimize an error function(loss function), can be seen as mathematical functions which are dependent on model's learnable parameters. They determine how to change weights and learning rate of a neural network.

The most commonly used loss function for the task of image segmentation is a pixel-wise cross entropy loss. This loss examines each pixel individually, comparing the class predictions with the ground truth:

$$\mathrm{L}_{CE} = -\frac{1}{N}\sum_{i}^{N}\sum_{j}^{M} y_{ij} log(p_{ij}) \tag{5.1}$$

With N the number of samples (in this case all pixels), M the number of classes (binary case $M = 2$), y_{ij} the ground truth value and p_{ij} the corresponding prediction result.

This way we are imposing the same learning for each pixel in the image because the cross entropy loss analyzes the class predictions for each pixel vector individually before averaging across all pixels. If the distribution of several classes in the image ground truth is not balanced choose this type of loss function could be an issue because the most common class might dominate training. In the dataset I worked with, the pixels classified as background occupy a large part of the final mask resulting in a highly unbalanced dataset.

To counteract this problem TensorFlow allows you to weight the loss function for each output channel using a representation of the final mask in one hot encoding, but other functions based on dice or Jaccard coefficients are commonly used.

For both modules I tried implementing both solutions and got better results using a custom loss function based on the *Jaccard index*, also known as Intersection over Union (IoU).

Jaccard index is the area of the intersection over union of the predicted segmentation B and the ground truth A.

$$L_{jaccard}(A, B) = -\frac{|A \cap B|}{|A \cup B|} \tag{5.2}$$

where $|A \cap B|$ represents the total common elements between sets A and B, and $|A \cup B|$ represents the number of elements which are in A, in B, or in both A and B.
In other words it is essentially a measure of overlap between two samples. This measurement has a range of 0 to 1, with a coefficient of 1 signifying perfect and total overlap.
Since output mask is binary, I approximate the intersection $|A \cap B|$ as the element-wise multiplication between the prediction and target mask, and then sum the resulting matrix.
In order to quantify $|A \cup B|$ I simple calculate the sum for both sets A and B and then I subtract the intersection calculated previously.
In order to formulate a loss function which can be minimized, I simply multiplied the IoU coefficient by -1.

$$L_{jaccard} = -\frac{\sum_i^N y_i p_i}{\sum_i^N y_i + \sum_i^N p_i - \sum_i^N y_i p_i} \tag{5.3}$$

The loss function's source code is shown below.

```
1  from keras import backend as K
2
3  def jaccard_coef(y_true, y_pred):
4      y_true = tf.cast(y_true, tf.float32)
5      y_pred = tf.cast(y_pred, tf.float32)
6
7      y_true_f = K.flatten(y_true)
8      y_pred_f = K.flatten(y_pred)
9      intersection = K.sum(y_true_f * y_pred_f)
10     return (intersection + 1.0) / (K.sum(y_true_f) + K.sum(y_pred_f)
    - intersection + 1.0)
11
12 def jaccard_coef_loss(y_true, y_pred):
13     return -jaccard_coef(y_true, y_pred)
```

Listing 5.3: Jaccard Loss function

A value equal to 1 was added to the denominator and numerator in order to avoid the possible error of division by zero.
As regards the optimization algorithm, *Adam* [10] (Adaptive Moment Estimation) was chosen. The Adam optimizer is one of the most widely utilized gradient descent optimization methods. Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. According to Kingma et al [10], the method is "computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters". The Adam optimizer works by computing an exponential moving average of the gradient and the squared gradient. The Adam optimizer maintains two moving averages: the first moment (mean) and the second moment (uncentered variance) of the gradient. The first moment is calculated as an exponentially decaying average of the gradient, and the second moment is calculated as an exponentially decaying average of the squared gradient [31].
TensorFlow allows to set Adam parameters such as:

- **Learning Rate**: the parameter that determines the step size at each iteration while moving toward a minimum of a loss function.
- **Beta1 and Beta2**: are the decay rates for the first and second moments.
- **Epsilon**: A small constant for numerical stability.

For this project I kept Adam's default values: `beta1=0.9`, `beta2=0.999`, `learning_rate=0.001`, and `epsilon=1e-7`.
After compiling, the training phase can be started thanks to the method `model.fit()`. The model was trained for 200 epochs, by epoch I mean an iteration over the entire training data provided.

5.1.2 Model Evaluation

In this subsection, I will present the results of the tests conducted on the dataset considering only the first Rough Module.
`sklearn` library in python allows to compute the confusion matrix in order to evaluate the accuracy of a classification. In the following code the metrics described in the previous chapters are obtained by exploiting the number of true positive, true negative, false positive and false negative values obtained from the confusion matrix between predicted masks and ground truth.

```
from sklearn.metrics import confusion_matrix

def show_evaluation_metrics(y_true, y_pred):

    tn, fp, fn, tp = confusion_matrix(y_true.flatten(), y_pred.flatten()).ravel()
    print('True Negative:', tn, ', False Positive:', fp, ', False Negative:', fn, ', True Positive:', tp)

    print('Pixel Accuracy:', (tp + tn)/(tp + tn + fn + fp))
    print('Precision:', tp / (tp + fp))
    print('Recall:', tp / (tp + fn))

    f1_score = 2 * tp / (2 * tp + fp + fn)
    print('Dice Coefficent (F1 Score): ', f1_score)

    iou = tp / (tp + fp + fn)
    print('Intersection-Over-Union IoU Coefficent(Jaccard Index): ', iou)
```

Listing 5.4: Evaluation Metrics

In the first row of the table it is possible to observe the value of the relevant metrics evaluated in the test set taking into consideration predicted masks and the ground truth downsampled to dimensions 448x448. In the second line instead we take into consideration the ground truth with original dimensions and predicted masks interpolated with a bilinear method.

	Pixel Accuracy	**Precision**	**Recall**	**Dice Coeff.**	**IoU Coeff.**
448 x 448	0.997	0.991	0.977	0.984	0.969
Original Sizes	0.996	0.996	0.968	0.982	0.964

Table 5.1: Rough Module Results

These results were achieved by applying data augmentation techniques to the training and validation samples, obtaining a total of 4544 and 512 respectively. For each image 7 different images are generated by applying random transformations such as rotations, shears, zooms and flips. Furthermore, 2000 synthetic images were imported, obtaining a total training set composed of a total of 6544

units.

5.2 Refinement Module

For this module I took inspiration from the *RefineNet* [14] architecture designed by Guosheng Lin et al. and the paper "A robust and high-precision edge segmentation and refinement method for high-resolution images" [28].
This module aims to improve the result obtained in the previous part and at the same time increase the size of the mask from 448x448 to 1344x1344. It tries to maximize the features obtained in various scales fusing features maps at different levels.
This architecture works with original images, resized to 1344x1344, relative Canny edge map and segmentation result obtained by the rough network. The latter must be resized with bilinear interpolation in order to be compatible. In this case the edge map is calculated using the Canny method in order to obtain more precise traces thanks to the application of non-maximum suppression and hysteresis thresholding techniques.
First of all, each input is divided into 9 patches measuring 448x448 using *patchify* [33], a Python library that allows to divide the input images into overlapping patches, giving the desired size, and merge them to regain the original image. Each patch is analyzed separately from the model.
As can be seen from the descriptive image 5.3 the structure makes use of three cascade refinement units (RU). In the left part is represented the corresponding input, while on the right side the output. Each RU works with 5-channel inputs, three of them are standard to every block: the original patch, relative edge map and rough segmentation output.
The RU is a encoder-decoder structure, like the U-Net, that returns segmentation results at different scale. As for the global process described in the *HRRNet* paper, RUs has the following properties:

- The original image patch, the relative edge map and the relative crop of the rough segmentation result concatenated three times make up the first RU input. It will output a low resolution segmentation result with size 1/8 OS.
- The input for the second RU is identical to the first, with the upsampled output segmentation map of OS8 acquired by the first RU replacing the last two rough segmentation outputs. Two segmentation maps on two different scales, OS8 and OS4, will be the final outcome.

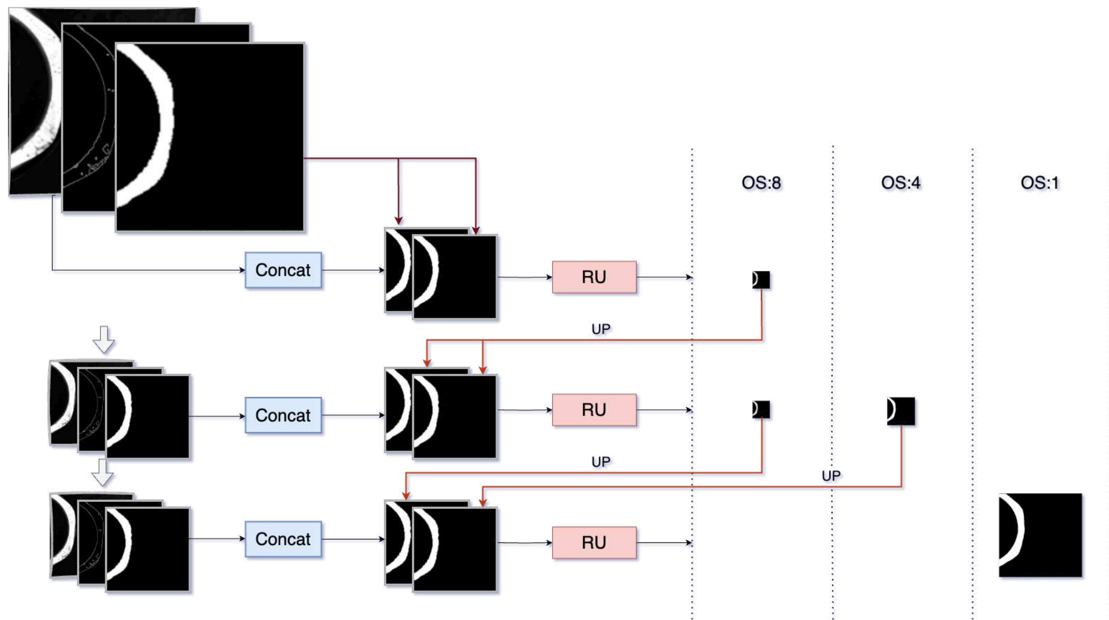


Figure 5.3: Refinement Module Architecture

- The input for the last RU block is created by concatenating the image's rough segmentation results with the upsampled segmentation maps OS8 and OS4 from the previous RU. The input is processed by the complete structure in this instance, yielding a mask with the same size as the input one.

For greater clarity the code developed for creating the model is shown below.

```
def get_refine_module(n_filters):

    input_layer = tf.keras.layers.Input(shape=[448, 448, 3])
    rough_mask = tf.expand_dims(input_layer[:,:,:,2], -1)

    input_layer_ru1 = tf.keras.layers.concatenate([input_layer,
   rough_mask, rough_mask])
    _, _, os8_1 = ru_block(input_layer_rm1, n_filters)

    os8_1 = Resize()([os8_1, [448, 448]])
    input_layer_rm2 = tf.keras.layers.concatenate([input_layer, os8_1
   , os8_1])
    _, os4, os8_2 = ru_block(input_layer_rm2, n_filters)

    os8_2 = Resize()([os8_2, [448, 448]])
    os4 = Resize()([os4, [448, 448]])
    input_layer_rm3 = tf.keras.layers.concatenate([input_layer, os4,
   os8_2])
```

```
16     os1, _, _ = ru_block(input_layer_rm3, n_filters)
17 
18     #label layer
19     label_layer = tf.keras.layers.Input(shape=[448, 448])
20 
21     ref_model = tf.keras.Model(inputs=[input_layer, label_layer],
    outputs=[os1])
22 
23     loss = - (jacard_coef(os8_2, label_layer) + jacard_coef(os4,
    label_layer) + jacard_coef(os1, label_layer)) / 3
24     ref_model.add_loss(loss)
25 
26     return ref_model
```

Listing 5.5: Refinement Module

`Resize()` is a custom layer that I created that interpolates the input image according with the data given using a bilinear approach.
A RU block is applied calling `ru_block()` function specifying the number of filters performed. It returns three outputs that are segmentation results at different scales at different levels of the decoder.
Refinement Unit structure is explained by following code.

```
1 def ru_block(input_tensor, n_filters, dropout = 0.1):
2 
3     # Encoder, Contracting Path
4     c1 = conv2d_block(input_tensor, n_filters)
5     p1 = tf.keras.layers.MaxPooling2D((2, 2))(c1)
6     p1 = tf.keras.layers.Dropout(dropout)(p1)
7 
8     c2 = conv2d_block(p1, n_filters*2)
9     p2 = tf.keras.layers.MaxPooling2D((2, 2))(c2)
10    p2 = tf.keras.layers.Dropout(dropout)(p2)
11 
12    c3 = conv2d_block(p2, n_filters*4)
13    p3 = tf.keras.layers.MaxPooling2D((2, 2))(c3)
14    p3 = tf.keras.layers.Dropout(dropout)(p3)
15 
16    c4 = conv2d_block(p3, n_filters*8)
17 
18    # Decoder, Expansive Path
19    rcu1 = conv2d_block(c4, n_filters*4)
20    os8 = tf.keras.layers.Conv2D(1, (1, 1), activation='sigmoid')(
   rcu1)
```

```
    u5 = tf.keras.layers.Conv2DTranspose(n_filters*4, (3, 3), strides
     = (2, 2), padding = 'same')(rcu1)
    u5 = tf.keras.layers.concatenate([u5, c3])
    u5 = tf.keras.layers.Dropout(dropout)(u5)

    rcu2 = conv2d_block(u5, n_filters*2)
    os4 = tf.keras.layers.Conv2D(1, (1, 1), activation='sigmoid')(
   rcu2)
    u6 = tf.keras.layers.Conv2DTranspose(n_filters*2, (3, 3), strides
     = (2, 2), padding = 'same')(rcu2)
    u6 = tf.keras.layers.concatenate([u6, c2])
    u6 = tf.keras.layers.Dropout(dropout)(u6)

    rcu3 = conv2d_block(u6, n_filters)
    u7 = tf.keras.layers.Conv2DTranspose(n_filters, (3, 3), strides =
     (2, 2), padding = 'same')(rcu3)
    u7 = tf.keras.layers.concatenate([u7, c1])
    u7 = tf.keras.layers.Dropout(dropout)(u7)
    os1 = tf.keras.layers.Conv2D(1, (1, 1), activation='sigmoid')(u7)

    return os1, os4, os8
```

Listing 5.6: Refinement Unit

This architecture is similar to U-Net but more simplified with only 3 blocks in the contraction part and 3 in the expansion part, furthermore the number of initial filters has decreased from 32 to 16. It uses the same function that applies 2 convolutional layers, batch normalization and leaky ReLU to extrapolate the context of the feature maps.
In the decoder the convolutional block this time is applied before the upsampling operation via a transposed convolutional layer.
The outputs at each scale (OS1, OS4, OS8) are obtained by applying a single-channel convolutional layer and a sigmoid as the activation function.

5.2.1 Model Training

The model was trained for 100 epochs using Adam as optimization algorithm and a loss function based on jaccard index as in the rough module. The custom loss function is embedded in the code that returns the refinement model 5.5.
The overall loss function of the network in this case is constructed by adding the Intersection over Union coefficient of different scales of the network, that is:

$$L = \frac{L^{OS8}_{Jaccard} + L^{OS4}_{Jaccard} + L^{OS1}_{Jaccard}}{3} \tag{5.4}$$

$L_{Jaccard}$ is an approximation of the Jaccard index calculated as 5.3. $L^{OS8}_{Jaccard}$ and $L^{OS4}_{Jaccard}$ are calculated on the resized output of the second Refinement Unit, $L^{OS1}_{Jaccard}$ is instead referred at the final output. In this case the loss function is not specified when compiling the model but since it needs to analyze intermediate layers the `add_loss()` method is applied once the model has been initialized.
The network was trained with Adam's default parameters.

5.2.2 Model Evaluation

In this subsection are presented the results of the tests conducted on the test set considering also the Refinement Module. The original image is first analyzed by the first model which has the task of extrapolating an approximate mask, the result is then used as input for the second which has the task of refining and increasing the dimensions of the final mask.
The calculation of the metrics was performed by calling the same function described in the chapter for the evaluation of the rough module.
The results achieved are shown in the table 5.2.

	Pixel Accuracy	**Precision**	**Recall**	**Dice Coeff.**	**IoU Coeff.**
1344 x 1344	0.998	0.988	0.984	0.986	0.972
Original Sizes	0.997	0.995	0.979	0.987	0.975

Table 5.2: Refinement Module Results

In the first row you can see the value of the metrics that were evaluated in the test set by taking the predicted masks and the ground truth downsampled to 1344x1344. Instead, for the second line, we consider the ground truth with the original dimensions and the predicted masks interpolated using a bilinear technique.
Given the computational and memory limitations, this model was trained by importing only the original data, without applying data augmentation or import-

ing synthetic data. Patches measuring 448x448 were obtained from 568 training data resized to 1344x 1344. 9 patches were extrapolated from each image for a total of 5112.

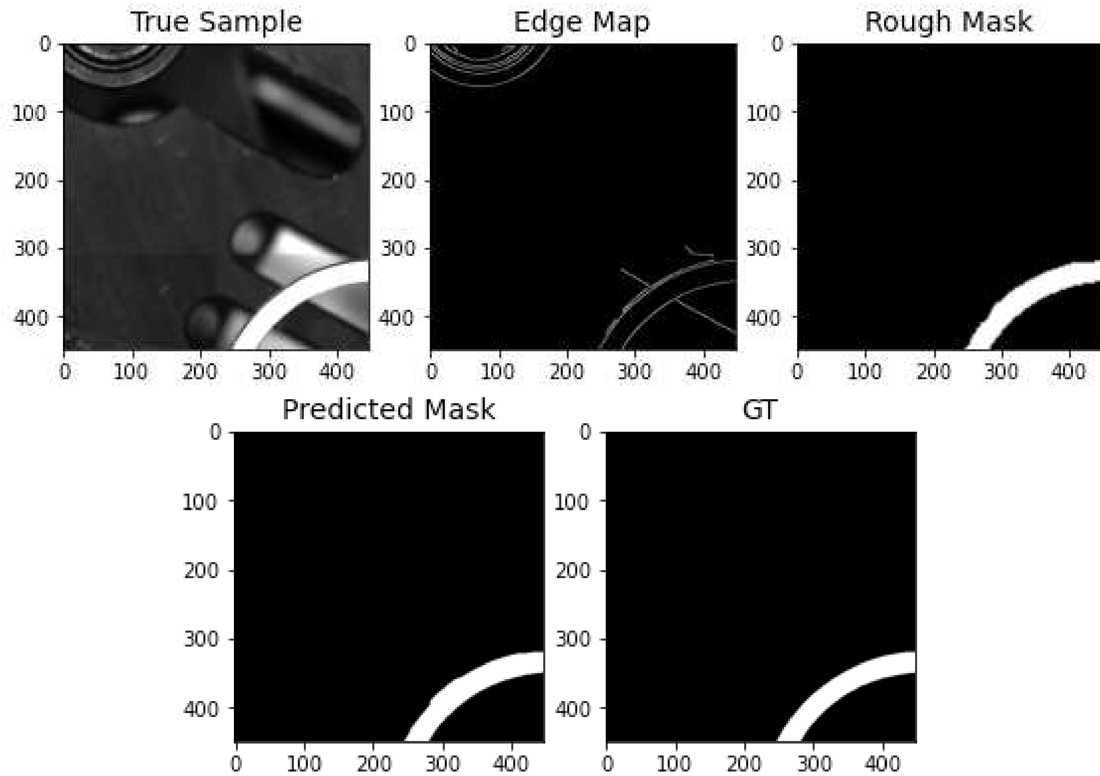


Figure 5.4: Results Comparison.

Edge Refinement Interpolation

Once the images have been processed we obtain a mask measuring 1344x1344, these must finally be interpolated to obtain dimensions equal to the original images in order to generate the 3D model. The input images depend on the size of the sample taken into consideration, typically larger jewelery such as bracelets and bangles produce 100 MP resolution images.
In this section, different interpolation strategies are discussed, listing their characteristics, advantages and disadvantages. Furthermore, I developed an interpolation strategy that aims to align the boundaries of the low resolution masks with the edges obtained from the original high resolution image.
Finally, in the last subchapter the results obtained by applying the different techniques are shown and discussed.

6.1 Image Interpolation Strategies

Interpolation strategies allow to enlarge an image estimating the value of the function describing the image in locations different from the sampling ones. It is pertinent also for other operations like rotations and geometric transforms. Depending on the interpolation algorithm, the outcomes can differ significantly even when the same image is resized. Every time interpolation is used, an image will always lose some quality because it basically estimates new data sampled in positions not present in the original image. Image interpolation aims to achieve the best approximation of a pixel's intensity and color depending on the values of the neighboring pixels. It operates in two directions (x,y).

All the strategies described below are applicable via a simple function `cv2.resize()` in OpenCV specifying the new size and the interpolation method.

```
import cv2
cv2.resize(input_mask, size, interpolation = cv2.INTER_LANCZOS4)
```

Listing 6.1: OpenCV Resize Function

Nearest-neighbor interpolation

Nearest neighbor is the most basic and simplest interpolation scheme. It only considers one pixel: it just select the value of the closest available point at each location. In other word, it has the effect of simply making each pixel bigger. It requires low processing time but achieves poor results.

Bilinear interpolation

The nearest 2x2 neighborhood of known pixel values around the unknown pixel is taken into account in this case. The final interpolated value is then obtained by taking a weighted average of these 4 pixels:

$$\mathrm{p}(x, y) = (1-a)(1-b)I_1 + a(1-b)I_2 + (1-a)bI_3 + abI_4 \tag{6.1}$$

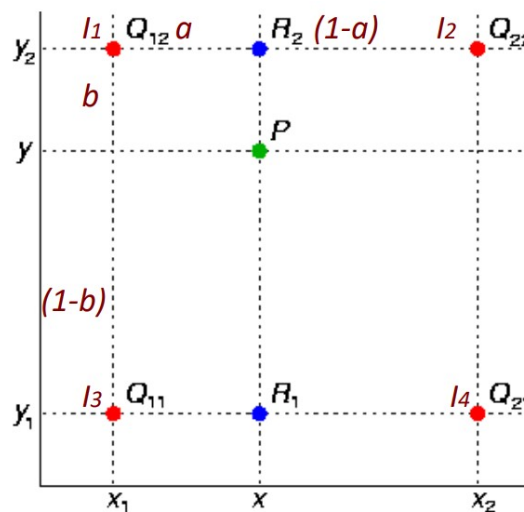


Figure 6.1: Bilinear Interpolation

As seen from the figure 6.1, the coefficients a and b depend on the distance of the point from the 4 close samples.
This solution represents a good compromise between speed and complexity.

Bicubic interpolation

Bicubic goes a step further than bilinear by taking the nearest 4x4 neighborhood of known pixels (a total of 16) into account. As the previous, interpolated values are obtained by taking a weighted average, since proximity pixels are at various distances from the value to calculate, closer pixels are given a higher weighting in the calculation.

$$\mathrm{p}(x, y) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_{i,j} x^i y^j \tag{6.2}$$

Coefficients $a_{i,j}$ are computed from various constraints including sample positions, first order derivatives and cross derivatives.
Bicubic produces noticeably sharper images than the previous two methods but is much slower.

Lanczos interpolation

Lanczos interpolation is one of the most popular methods to resize images, together with bilinear and bicubic interpolation.
It can be used as a low-pass filter or used to smoothly interpolate the value of a digital signal between its samples. It maps each sample of the given signal to a translated and scaled copy of the Lanczos kernel, which is a windowed $sinc$ function [20].
The Lanczos kernel $L(x)$ determines how each input sample affects the interpolated values.

$$L(x) = \begin{cases} sinc(\pi x) sinc(\pi x/a) & \text{if } -a < x < a, \\ 0 & \text{otherwise} \end{cases} \tag{6.3}$$

As can be seen from the equation 6.3, it is the normalized $sinc$, windowed (multiplied) by the Lanczos window, or sinc window, which is the central lobe of

a horizontally stretched $sinc$ function. The parameter a is a positive integer, typically 2 or 3, which determines the size of the kernel.
Given a two-dimensional signal with samples s_{ij}, the value $S(x, y)$ interpolated is obtained by the discrete convolution of those samples with the Lanczos kernel:

$$\mathrm{S}(x, y) = \sum_{i=\lfloor x \rfloor - a + 1}^{\lfloor x \rfloor + a} \sum_{j=\lfloor y \rfloor - a + 1}^{\lfloor y \rfloor + a} s_{ij} L(x - i) L(y - j) \tag{6.4}$$

where a is the filter size parameter, and $\lfloor x \rfloor$ is the floor function. This method is more computationally heavy but obtains good results, which is why it is used by many image processing applications like Photoshop and Gimp.

6.2 Edge Refinement Interpolation

In this section I present a solution that I developed to interpolate the mask obtained in the previous steps to original dimensions by exploiting the information obtained from the edges map. The algorithm works with the 2x2 nearest neighborhood, like the bilinear method, and the unknown values are established depending on their intensity.
This method combines two different approaches:

- Pixel values are assigned depending on the proximity intensity values and the traces given by the edge map calculated on the original high resolution image if an edge exists within the considered window.
- Otherwise the values are assigned using the bilinear method described in the previous chapters.

Now let's look at the code and discuss it. First of all we need to generate a suitable edge map.

```
1 def generate_blurred_edge_map(image):
2 
3     image = np.uint8(image * 255)
4     blurred = cv2.medianBlur(image, 15)
5 
6     otsu_thresh_val = cv2.threshold(blurred, 0, 255, cv2.
    THRESH_BINARY | cv2.THRESH_OTSU)[0]
7 
```

```
8       edgemap = cv2.Canny(blurred, otsu_thresh_val * 0.35,
       otsu_thresh_val)
9
10      return edgemap/255
```

Listing 6.2: Edge Map Generation Function

Figure 6.2: Interpolation Example

A blurring filter via median blur is applied to the original image. The Canny method was chosen for the best precision, then I looked for an automatic method to establish the threshold values, to achieve this I used Otsu's [27] method. This threshold is automatically computed from the image histogram, the optimal global threshold is computed maximizing the interclass variance.
Once the edges map has been calculated, the values of the predicted mask are positioned relatively far apart inside a numpy array of size equal to the original image, with default values equal to zero. Sampled values are equally spaced based on the dimensions of the original image. This interpolation strategy tries to obtain the best approximation of the label of a pixel based on the values of the 4 surrounding pixels and the edges relating to the portion of the image considered. The algorithm cycles through sampling points and if it finds a value assigned to the region of interest extrapolates the relevant window in the edge map: if there are values the following `edge_refinement_interpolation ()` function is performed, otherwise the bilinear method is chosen.
In the following code we are considering a square window (nxn) where: `p1` represents the pixel at the upper left corner, `p2` the upper right pixel, `p3` the lower right pixel and `p4` the lower left pixel.

```
def edge_refinement_interpolation(pixel_values, edge_map_patch):

    p1 = pixel_values[0]
    p2 = pixel_values[1]
    p3 = pixel_values[2]
    p4 = pixel_values[3]

    mask = np.zeros (edge_map_patch.shape, dtype = np.float64)

    if (p1 == p2 == p3 == 0 and p4 == 1) :
        ## start from left / down
        var_r, var_c = variance_on_axis(edge_map_patch)

        if var_r > var_c:
            # left
            mask = fill_mask(edge_map_patch, column_iteration = False
, start_fill= True)
        else :
            # down
            mask = fill_mask(edge_map_patch, column_iteration = True,
 start_fill= False)

    elif (p1 == p3 == p4 == 1 and p2 == 0) :
        ## start from left / down
        var_r, var_c = variance_on_axis(edge_map_patch)

        if var_r > var_c:
            # left
            mask = fill_mask(edge_map_patch, column_iteration = False
, start_fill= True, no_edge_val = 1)
        else :
            # down
            mask = fill_mask(edge_map_patch, column_iteration = True,
 start_fill= False, no_edge_val = 1)

    elif (p1 == p2 == p4 == 0 and p3 == 1):
        ## start from right / down
        var_r, var_c = variance_on_axis(edge_map_patch)
        if var_r > var_c:
            # right
            mask = fill_mask(edge_map_patch, column_iteration = False
, start_fill= False)

```

```
39         else :
40             # down
41             mask = fill_mask(edge_map_patch, column_iteration = True,
     start_fill= False)
42
43     elif (p1 == 0 and p2 == p3 == p4 == 1):
44         ## start from right / down
45         var_r, var_c = variance_on_axis(edge_map_patch)
46         if var_r > var_c:
47             # right
48             mask = fill_mask(edge_map_patch, column_iteration = False
    , start_fill= False, no_edge_val = 1)
49
50         else :
51             # down
52             mask = fill_mask(edge_map_patch, column_iteration = True,
     start_fill= False, no_edge_val = 1)
53
54     elif (p2 == 1 and p1 == p3 == p4 == 0):
55         ## start from right / up
56         var_r, var_c = variance_on_axis(edge_map_patch)
57         if var_r > var_c:
58             # right
59             mask = fill_mask(edge_map_patch, column_iteration = False
    , start_fill= False)
60
61         else :
62             # up
63             mask = fill_mask(edge_map_patch, column_iteration = True,
     start_fill= True)
64
65     elif (p1 == p2 == p3 == 1 and p4 == 0):
66         ## start from right / up
67         var_r, var_c = variance_on_axis(edge_map_patch)
68         if var_r > var_c:
69             # right
70             mask = fill_mask(edge_map_patch, column_iteration = False
    , start_fill= False, no_edge_val = 1)
71
72         else :
73             # up
74             mask = fill_mask(edge_map_patch, column_iteration = True,
     start_fill= True, no_edge_val = 1)
```

```
75
76      elif (p1 == 1 and p2 == p3 == p4 == 0):
77          ## start from left / up
78          var_r, var_c = variance_on_axis(edge_map_patch)
79          if var_r > var_c:
80              # left
81              mask = fill_mask(edge_map_patch, column_iteration = False
    , start_fill= True)
82
83          else :
84              # up
85              mask = fill_mask(edge_map_patch, column_iteration = True,
     start_fill= True)
86
87      elif  (p1 == p2 == p4 == 1 and p3 == 0):
88          ## start from left / up
89          var_r, var_c = variance_on_axis(edge_map_patch)
90          if var_r > var_c:
91              # left
92              mask = fill_mask(edge_map_patch, column_iteration = False
    , start_fill= True, no_edge_val = 1)
93
94          else :
95              # up
96              mask = fill_mask(edge_map_patch, column_iteration = True,
     start_fill= True, no_edge_val = 1)
97
98      elif (p3 == p4 == 1 and p1 == p2 == 0):
99          ## start from down
100         mask = fill_mask(edge_map_patch, column_iteration = True,
    start_fill= False)
101
102     elif (p2 == p3 ==1 and p1 == p4 == 0):
103         ## start from right
104         mask = fill_mask(edge_map_patch, column_iteration = False,
    start_fill= False)
105
106     elif (p1 == p4 == 1 and p2 == p3 == 0):
107         ## start from left
108         mask = fill_mask(edge_map_patch, column_iteration = False,
    start_fill= True)
109
110     elif (p1 == p2 == 1 and p3 == p4 == 0):
```

```
111            ## start from up
112            mask = fill_mask(edge_map_patch, column_iteration = True,
       start_fill= True)
113
114        elif (p1 == p3 and p2 == p4):
115            ## segmentation error
116            print('segmentation error')
117
118        return mask
```

Listing 6.3: Edge Refinement Interpolation

The values of 1 (ROI) are assigned depending on the proximity pixels present at the corners of the windows taken into consideration. `fill_mask()` function returns the interpolated mask by filling the area outlined by the edges. By setting the flag `start_fill = True` the function will position the ROI values to the left or from the top of the trace depending on the value of the second `column_iteration` parameter which establishes whether the iteration is performed first on the columns or on the rows. The third parameter `no_edge_val` allows you to set the values assigned to the column or row in case there is no trace in the edge map.
In some cases when the information from the 4 corners is not enough to understand the type of filling that needs to be performed, this decision is made by calculating the variance of the indices in which an edge appears in the x and y axis. Let's take for example a window where the pixel at the top left is white while the other three are black, in this case there are two possibilities: fill the mask starting from the left or from the top. By calculating the variance on the indices of both axes we can estimate in which direction the edge extends the most and make a decision accordingly.
The images below 6.3 show how this strategy performs well when the edges are clearly defined. The masks obtained were superimposed on the original images by altering the RGB channel corresponding to red.

6.3 Final Evaluation

This chapter lists the statistics obtained once the mask obtained from the two networks has been upsampled to the dimensions of the original images.
As can be seen from the table the results are almost the same. This is given by the fact of working with very high resolution images and therefore the pixels

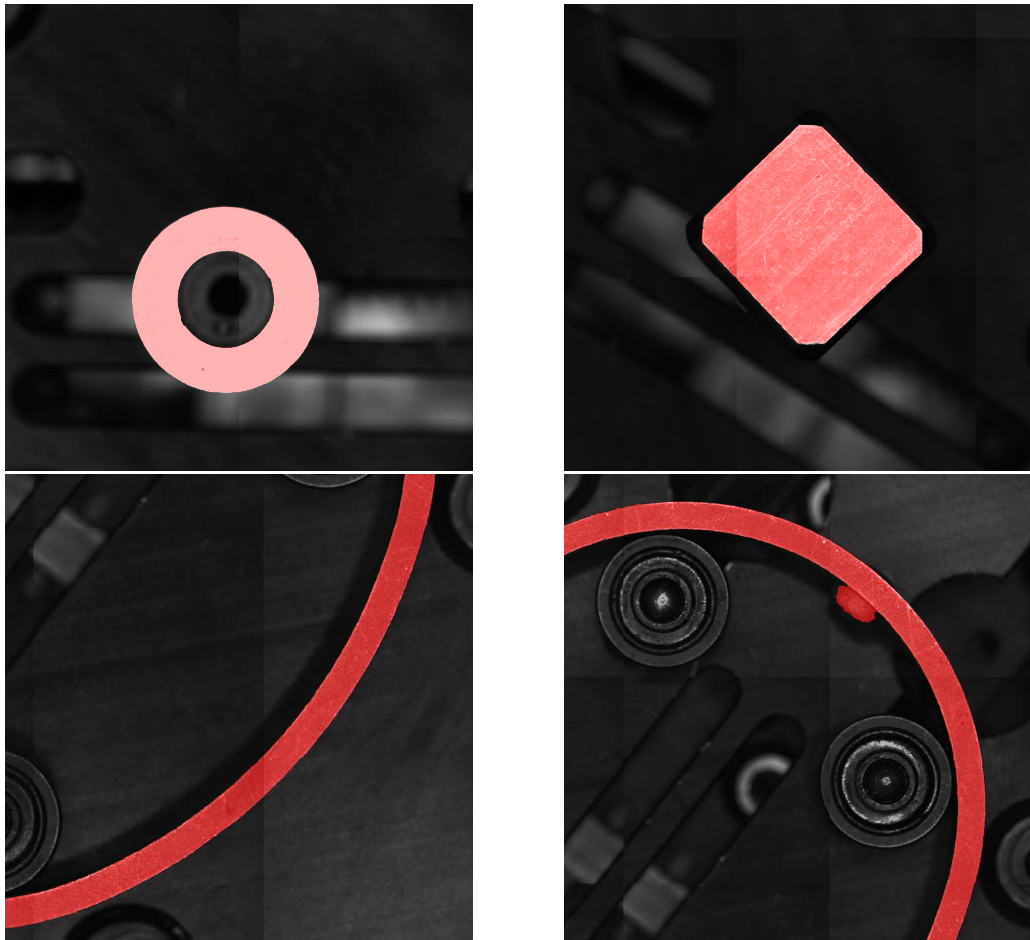


Figure 6.3: Interpolated Results.

belonging to the edges, the region of the image where a more complex strategy should produce more accurate results, represent a small percentage of the total.

Interpolation Strategy	**Pixel Accuracy**	**Precision**	**Recall**	**Dice Coeff.**	**IoU Coeff.**
Nearest-Neighbor	0.997	0.995	0.979	0.987	0.974
Bilinear	0.997	0.995	0.979	0.987	0.975
Bicubic	0.997	0.995	0.979	0.987	0.975
Lanczos4	0.997	0.995	0.979	0.987	0.975
Edge Refinement	0.997	0.992	0.981	0.987	0.974

Table 6.1: Final Interpolation Results

By analyzing the results obtained from the interpolation via edges developed by me in comparison to the other strategies we can draw information. We can observe a lower value regarding Precision while a higher Recall. This could be due to an increase in the number of false positives (pixels belonging to the background classified as ROI) and a decrease in false negatives (pixels belonging to ROI classified as background) and therefore a general increase in white pixels. The algorithm can still be improved by considering several aspects:

- I did not consider the case in which there is more than one edge in the considered window. Textures belonging to the object of interest or the background can generate unwanted edges which can lead to falsified results.
- The case where there is a clear segmentation fault is not handled. A method can be developed to assign a truth index, based on neighbors, which, if an established threshold is exceeded, this pixel is considered reliable.
- In the developed approach 2x2 windows are considered, by using larger windows, as is done for example in the bicubic approach, information can be obtained at larger scales.

SLC^3 Integration

This chapter summarizes the work carried out in C# to incorporate the results of the Python study into the main Sisma SLC^3 software's marking process. This work was carried out together and in conjunction with Dr. Cerato Davide who helped me in the development part. The models were exported as two different frozen graphs to allow the end user to analyze the image from only the rough network, when this is sufficient, in order to speed up the marking process. Instead the edge refinement interpolation algorithm has been completely rewritten in C#. To align the code with the other team members, *git* [22] was used, a tool incorporated into the functions of *Microsoft Visual Studio* [8].

7.1 Implementation details

Both models have been incorporated into the latest version of SLC^3 for the shape recognition task. Many of Sisma's customers operate in international contexts, necessitating the need for a user interface that can handle multiple languages. The models have been extrapolated into a format that assembles architecture and weights in a single file, facilitating distribution, and also minimizes size by removing unnecessary metadata for the inference process.

7.1.1 Export and Import of the models

The two models were developed and trained with the standard TensorFlow libraries in Python [2]. Unfortunately there is no official TensorFlow package for

the .NET environment but there are various third-party libraries that allow the import of models already trained in other languages such as TensorFlow.NET [4].
There are several ways to save a model and its weights, the standard methods maintain several files that contain the weights and the network architecture with all the metadata to be able to retrain the network. However, when we want to use a model in production, we only need our model and its weights to be organized in one file, we don't need any other metadata to complicate our files.
For this reasons I decided to convert them into frozen graphs. When a graph is frozen in TensorFlow, the description of the graph and its variables are combined into a single file `.pb` that can be quickly deployed in a real-world setting. Basically, freezing a graph involves removing all nodes not required for inference and changing all variables in the graph to constants. Freezing a graph is important because it allows us to deploy our models without having to include the entire TensorFlow framework. This can greatly lower the size of the deployment package.
To obtain this type of file it is sufficient to first export the model in the `SavedModel` format and then convert it to `GraphDef` using the `convert_variables_to_constants_v2` TensorFlow function. Finally, The frozen graph must then be saved to a `.pb` file using the `tf.io.write_graph` function. Making inferences in the .NET environment is very easy, first the graph is loaded via the `Import()` method. A `Session` object allows to obtain the final mask.

7.1.2 User Interface

The developed interface 7.1 allows the user to select the file paths of the models and the image to be analyzed.
Using the Refinement module is optional, good results are obtained even just using the first model. In general, C# is a statically-typed and compiled language, while Python is a dynamically-typed and interpreted language. This means that C# is generally faster and more efficient but Python has numerous libraries optimized for working with large images and matrices, which makes it more performant in this case. Interpolation with edges has been implemented as in Python and can be enabled by clicking on the corresponding checkbox, otherwise it uses the bilinear method.

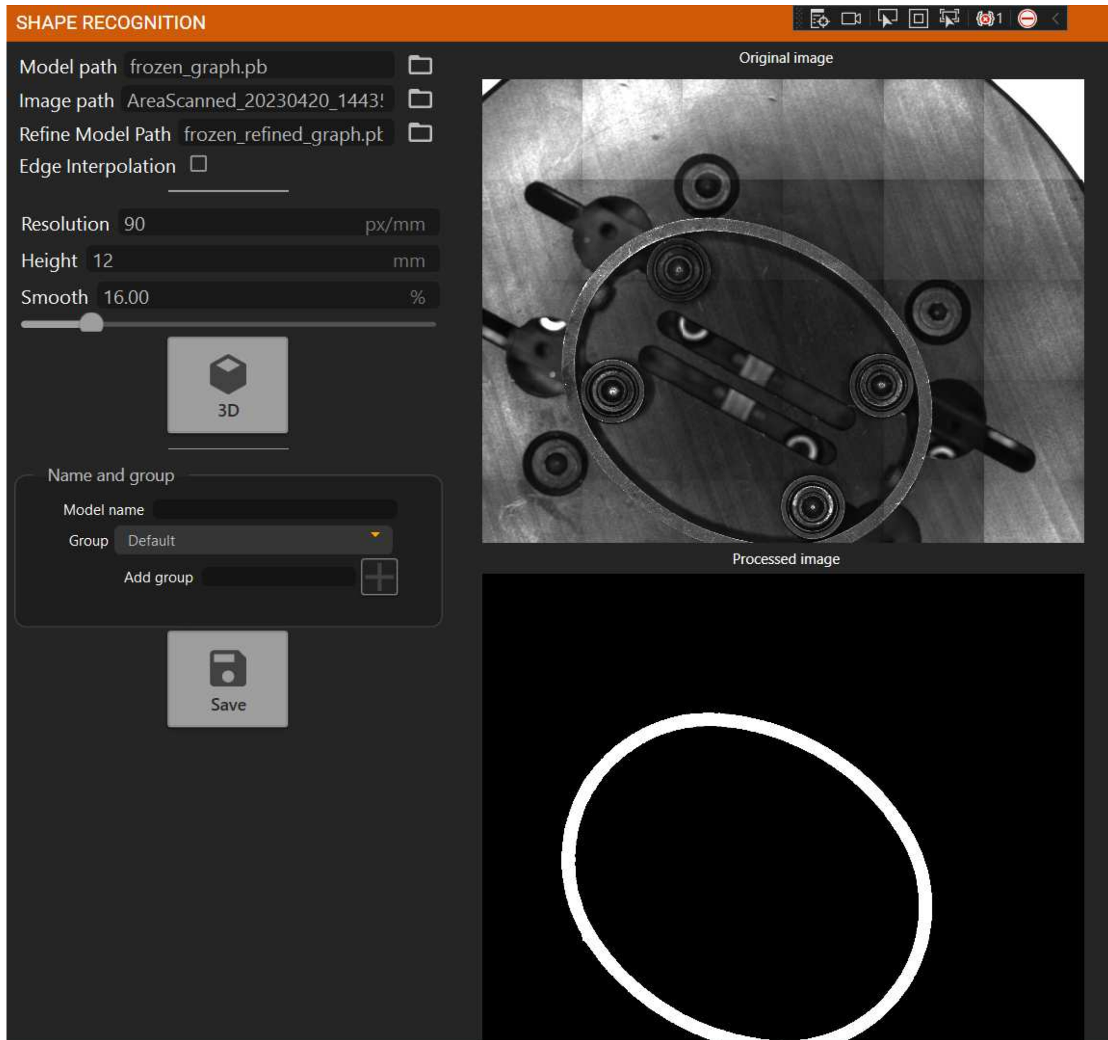


Figure 7.1: User Interface

Before starting the recognition algorithm it is important to set three main parameters:

- The **Resolution** of the image (pixel/millimeter), usually set automatically, is used to obtain a 3D scale model.
- The **Height** of the object, we assume that the sample is flat with a constant height at all points, even if this constraint does not exist during the marking phase
- The **Smooth** percentage is an operation that is done postprocessing, it aims to carry out a denoising operation on the final mask, it tries to obtain more ideal shapes.

Once the binary mask has been obtained, the 3D model is generated in a Sisma proprietary file format and rendered, see fig. 7.2.

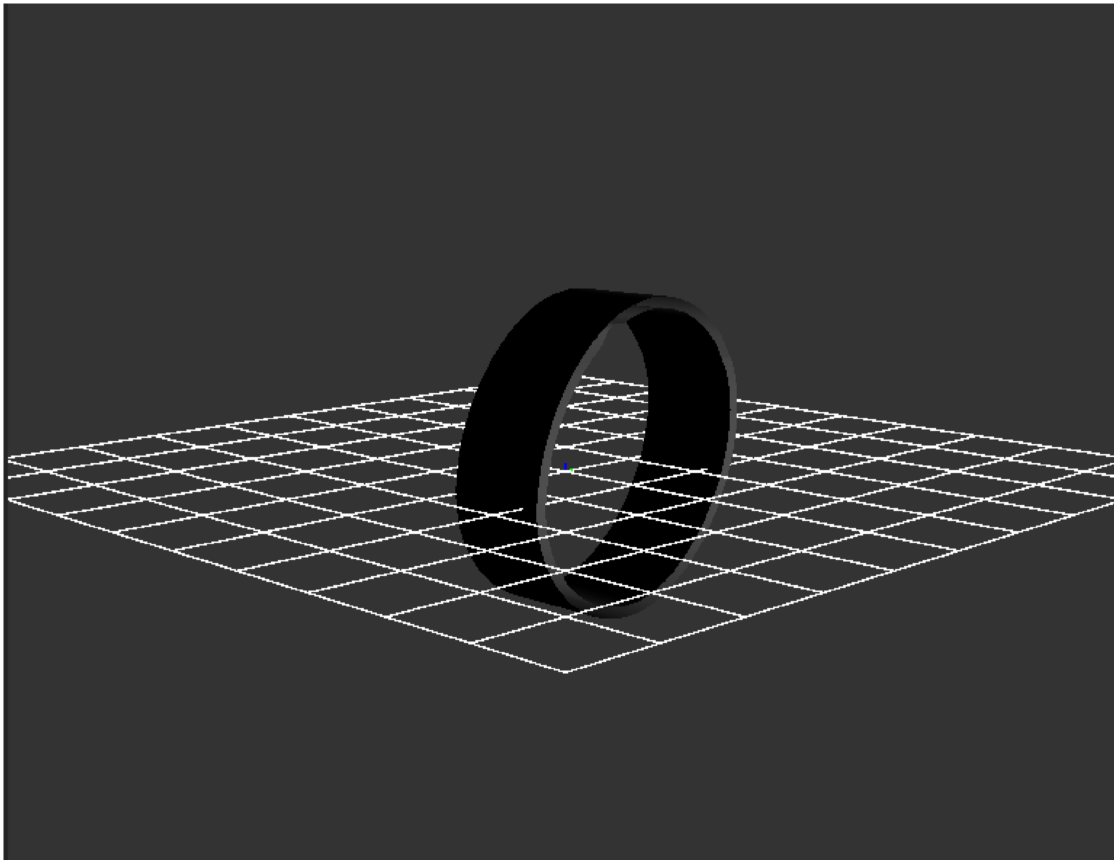


Figure 7.2: Rendered 3D model

Taking into consideration a ring as in the figure 7.3, upon confirmation of the 3D model, a new project will be created where it is possible to observe the UV mapping of the external development at the top, the development of the edge

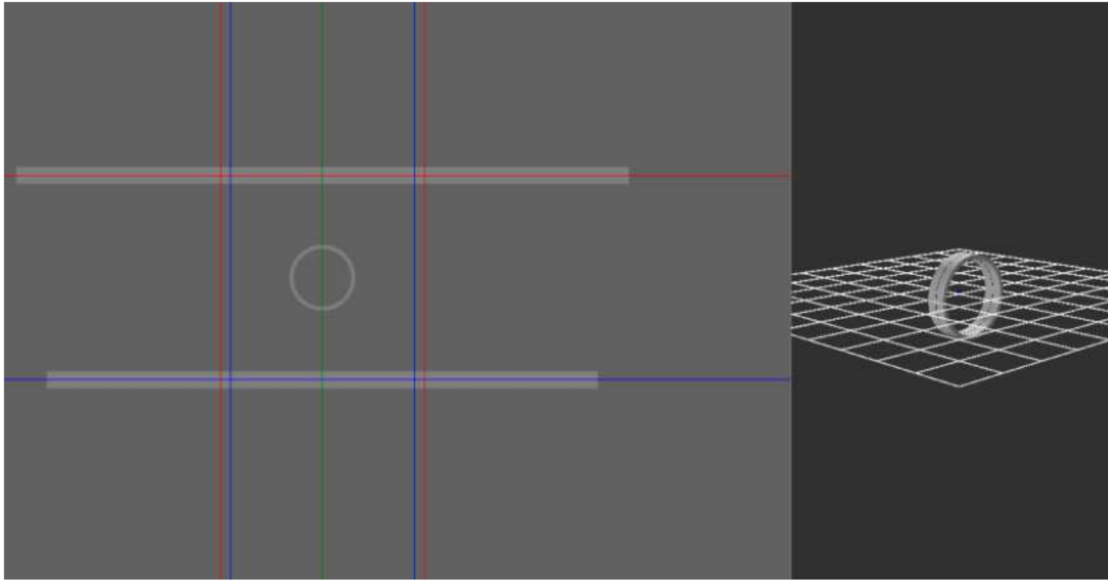


Figure 7.3: UV Maps

(vertical mandrel) in the center, and finally the UV mapping of the internal development.

Is possible to achieve automatic internal and external scanning by tilting the mandrel at 0 and 15 degrees respectively. Using the editor it is possible to position the engraving to be made in the desired position on the maps. Finally, the generated file containing the geometries of the scanned object can be saved and imported for subsequent processing.

Conclusions

The primary objective of this master's thesis was to improve and expand the capabilities of the shape recognition algorithm in the 3D scan package of the Sisma SLC^3 software. This thesis describes the entire strategy adopted in order to obtain an artificial intelligence model capable of segmenting the input image and dividing it between background and foreground pixels.
In the image segmentation domain, I worked with very high resolution black and white images captured and manually segmented to obtain the ground truth. The main goal was to obtain the most accurate binary mask possible given that these machines work for high precision laser marking. To expand the data set, data augmentation techniques were applied and synthetic images were also artificially generated.
Various algorithms were implemented to obtain an optimal final solution.
In the preprocessing phase, it was effective to incorporate information obtained from edge maps to help the network process more accurate masks. The main structure was inspired by the U-Net architecture [25], proving to achieve high performance even with limited datasets. Subsequently, a refinement network refines the result by aligning the edges of the mask with those obtained from techniques such as the Canny algorithm.
To evaluate the performance of the models, a comprehensive examination of the detection results was performed. The performance of each model was evaluated based on overall accuracy, precision, recall, F1 score and Jaccard index.
An interpolation technique that exploits the edge maps obtained from the original size images was developed. Furthermore, different interpolation strategies

were analyzed and the results compared.
The solutions were finally implemented in C# and integrated into the marking process pipeline. By implementing these solutions, the software is able to obtain better final masks compared to the previous approach and directly contributes to making more precise automatic laser engravings. Furthermore, this solution expands the range of recognizable shapes, allowing to engrave more types of objects.
Additionally, the prediction times, as well as resource consumption, were taken into consideration. Execution times are longer in the .NET environment compared to Python. The precision obtained through the refinement module can be sacrificed to obtain more immediate results. In this case only the original downsampled image is analyzed and not the 9 patches for the second module.

8.1 Future works

Artificial intelligence is already being used in numerous sectors, particularly computer vision, and is expected to continue to have a significant impact on a variety of elements of daily life. For this reasons researches in this field are in continuous evolution.
Although it took me a lot of time to capture real images and manually segment the binary masks (approximately 50 different samples were analyzed) the resulting dataset is still limited and I would need to obtain other data for better performance. Furthermore, the developed solution can be improved by focusing on minimizing execution times, the implementation carried out in C# must certainly be optimized.
Taking advantage of transfer learning, fine-tuning and more complex architectures can improve the final results but at the same time increase execution times and computational cost due to their larger size. Transfer learning implies the use of pretrained models, these usually are advanced deep learning models that have been trained on very large datasets using supervised learning techniques.
Moreover the developed interpolation algorithm needs to be improved considering the cases where there are multiple edges within the considered window and the cases where there is a clear segmentation error, as explained in the related chapter.

8.2 Personal Growth and Accomplishments

The effective conduct of this research was made possible by the theoretical foundations of machine learning and algorithms, as well as the practical and critical thinking skills developed during the course of my studies. Throughout the development and the writing of this thesis, I came across a real-world scenario that allowed me to effectively address and achieve the objectives established in this research. Thanks to the years of diligent study and global skills I have developed throughout my academic career, I have been able to complete these goals successfully. My involvement in this project not only increased my technical knowledge but also helped me improve my project management, problem-solving, and critical thinking abilities. Both from my perspective and that of the concerned Sisma technicians, the outcomes achieved are very satisfactory.

References

[1] Alex Parinov Alexander Buslaev et al. *Albumentations*. 2018. URL: https://albumentations.ai/.

[2] Google Brain. *TensorFlow*. 2015. URL: https://www.tensorflow.org/.

[3] John Francis Canny. “A Computational Approach To Edge Detection”. In: *IEEE Xplore* (1986).

[4] Haiping Chen et al. *TensorFlow.NET*. 2018. URL: https://scisharp.github.io/tensorflow-net-docs/.

[5] François Chollet. *Keras*. 2015. URL: https://keras.io/.

[6] Dorin Comaniciu and Peter Meer. “Mean Shift: A Robust Approach Towards Feature Space Analysis”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol.24* (2002).

[7] Emgu Corporation. *Emgu CV*. 2008. URL: https://www.emgu.com/.

[8] Microsoft Corporation. *Microsoft Visual Studio*. 1997. URL: https://visualstudio.microsoft.com/.

[9] Olivier Grisel David Cournapeau et al. *Scikit-learn: Machine Learning in Python*. 2009. URL: https://scikit-learn.org/.

[10] Jimmy Ba Diederik P. Kingma. “Adam: A Method for Stochastic Optimization”. In: *arXiv preprint arXiv:1412.6980v9* (2015).

[11] Brian Granger Fernando Pérez. *Jupyter Notebook*. 2014. URL: https://jupyter.org/.

[12] .NET Foundation. *.NET*. URL: https://learn.microsoft.com/it-it/dotnet/.

[13] Python Software Foundation. *Python*. 2001. URL: https://www.python.org/.

[14] Anton Milan Guosheng Lin et al. "RefineNet: Multi-Path Refinement Networks for High-Resolution Semantic Segmentation". In: *arXiv preprint arXiv:1611.06612v3* (2016).

[15] Charles R. Harris et al. *NumPy: array processing for numbers, strings, records, and objects*. 2006. URL: https://numpy.org/.

[16] J. D. Hunter. *Matplotlib: A 2D Graphics Environment*. 2007. URL: http://matplotlib.org.

[17] Anaconda Inc. *Anaconda*. 2012. URL: https://www.anaconda.com/.

[18] Willow Garage Intel. *OpenCV*. 2020. URL: https://opencv.org/.

[19] Xiangyu Zhang Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *arXiv preprint arXiv:1512.03385* (2015).

[20] *Lanczos resempling*. URL: https://en.wikipedia.org/wiki/Lanczos_resampling (visited on 2023).

[21] Yukun Zhu Liang-Chieh Chen et al. "Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation". In: *arXiv preprint arXiv:1802.02611v3* (2018).

[22] Junio Hamano Linus Torvalds. *Git*. 2005. URL: https://git-scm.com/.

[23] S. Lloyd. "K-Means Clustering". In: (1957).

[24] Microsoft. *C#*. URL: https://learn.microsoft.com/it-it/dotnet/csharp/.

[25] Thomas Brox Olaf Ronneberger Philipp Fischer. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *arXiv preprint arXiv:1505.04597v1* (2015).

[26] *OpenCV Canny Edge Detection*. URL: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html (visited on 2023).

[27] Nobuyuki Otsu. "A Threshold Selection Method from Gray-Level Histograms". In: *IEEE Transactions on Systems, Man, and Cybernetics* (1979).

[28] Chengcheng Chen Qiming Li. "A robust and high-precision edge segmentation and refinement method for high-resolution images". In: *AIMS, Mathematical Biosciences and Engineering* (2022).

[29] Irwin Sobel and Gary M. Feldman. *Sobel operator*. 1968.

[30] Peter Mattis Spencer Kimball. *GIMP*. 1996. URL: https://www.gimp.org/.

[31] *Tensorflow Confusion regarding the Adam optimizer*. URL: https://saturncloud.io/blog/tensorflow-confusion-regarding-the-adam-optimizer/#:~:text=To%20use%20the%20Adam%20optimizer%20effectively%2C%20it%20is%20important%20to,and%20epsilon%3D1e%2D7%20. (visited on 2023).

[32] *Tensorflow Dropout Layer*. URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout (visited on 2023).

[33] Weiyuan Wu. *Patchify*. 2021. URL: https://pypi.org/project/patchify/.